Provably Efficient Algorithms for Numerical Tensor Algebra

by

Edgar Solomonik

A dissertation submitted in partial satisfaction of the
requirements for the degree of
Doctor of Philosophy

in

Computer Science
and the Designated Emphasis

in

Computational Science and Engineering

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Professor James Demmel, Chair
Professor Katherine Yelick
Professor Martin Head-Gordon

Fall 2014

Provably Efficient Algorithms for Numerical Tensor Algebra

Abstract

Provably Efficient Algorithms for Numerical Tensor Algebra

by

Edgar Solomonik

Doctor of Philosophy in Computer Science
and the Designated Emphasis

in

Computational Science and Engineering

University of California, Berkeley

Professor James Demmel, Chair

This thesis targets the design of parallelizable algorithms and communication-efficient parallel schedules for numerical linear algebra as well as computations with higher-order tensors. Communication is a growing bottleneck in the execution of most algorithms on parallel computers, which manifests itself as data movement both through the network connecting different processors and through the memory hierarchy of each processor as well as synchronization between processors. We provide a rigorous theoretical model of communication and derive lower bounds as well as algorithms in this model. Our analysis concerns two broad areas of linear algebra and of tensor contractions. We demonstrate the practical quality of the new theoretically-improved algorithms by presenting results which show that our implementations outperform standard libraries and traditional algorithms.

We model the costs associated with local computation, interprocessor communication and synchronization, as well as memory to cache data transfers of a parallel schedule based on the most expensive execution path in the schedule. We introduce a new technique for deriving lower bounds on tradeoffs between these costs and apply them to algorithms in both dense and sparse linear algebra as well as graph algorithms. These lower bounds are attained by what we refer to as 2.5D algorithms, which we give for matrix multiplication, Gaussian elimination, QR factorization, the symmetric eigenvalue problem, and the Floyd-Warshall all-pairs shortest-paths algorithm. 2.5D algorithms achieve lower interprocessor bandwidth cost by exploiting auxiliary memory. Algorithms employing this technique are well known for matrix multiplication, and have been derived in the BSP model for LU and QR factorization, as well as the Floyd-Warshall algorithm. We introduce alternate versions of LU and QR algorithms which have measurable performance improvements over their BSP counterparts, and we give the first evaluations of their performance. We also explore network-topology-aware mapping on torus networks for matrix multiplication

and LU, showing how 2.5D algorithms can efficiently exploit collective communication, as well as introducing an adaptation of Cannon's matrix multiplication algorithm that is better suited for torus networks with dimension larger than two. For the symmetric eigenvalue problem, we give the first 2.5D algorithms, additionally solving challenges with memory-bandwidth efficiency that arise for this problem. We also give a new memory-bandwidth efficient algorithm for Krylov subspace methods (repeated multiplication of a vector by a sparse-matrix), which is motivated by the application of our lower bound techniques to this problem.

The latter half of the thesis contains algorithms for higher-order tensors, in particular tensor contractions. The motivating application for this work is the family of coupled-cluster methods, which solve the many-body Schrödinger equation to provide a chemically-accurate model of the electronic structure of molecules and chemical reactions where electron correlation plays a significant role. The numerical computation of these methods is dominated in cost by contraction of antisymmetric tensors. We introduce Cyclops Tensor Framework, which provides an automated mechanism for network-topology-aware decomposition and redistribution of tensor data. It leverages 2.5D matrix multiplication to perform tensor contractions communication-efficiently. The framework is capable of exploiting symmetry and antisymmetry in tensors and utilizes a distributed packed-symmetric storage format. Finally, we consider a theoretically novel technique for exploiting tensor symmetry to lower the number of multiplications necessary to perform a contraction via computing some redundant terms that allow preservation of symmetry and then cancelling them out with low-order cost. We analyze the numerical stability and communication efficiency of this technique and give adaptations to antisymmetric and Hermitian matrices. This technique has promising potential for accelerating coupled-cluster methods both in terms of computation and communication cost, and additionally provides a potential improvement for BLAS routines on complex matrices.

Contents

Acknowledgments

I'd like to firstly acknowledge the guidance and feedback I received from my adviser, Jim Demmel, which has had a great influence on this work. I've learned a lot from Jim, in particular with respect to rigorous theoretical analysis of algorithms. Additionally, on a number of occasions Jim has corrected critical oversights in my analysis and often these corrections created pathways for further development of the research work. I would also like to thank my committee members Kathy Yelick and Martin Head-Gordon, who have had a direct positive influence on this work and from whom I have also learned a lot.

I'd like to also especially acknowledge Grey Ballard, Nicholas Knight, and Devin Matthews all of whom contributed to research in multiple chapters of this thesis. Grey, Nick, and Devin were always open for research discussions from which I have gained a lot of knowledge, particularly on numerical linear algebra from Grey, on numerical methods from Nick, and on electronic structure algorithms from Devin. Additionally, I would like to thank Jeff Hammond and Erin Carson both for contributions to the work and numerous interesting research discussions. I would also like to thank a number of other collaborators who have contributed to this work: Abhinav Bhatele, Aydin Buluc, Laura Grigori, Mathias Jacquelin, Hong Diep Nguyen, and John F. Stanton. Further, I would like to thank Erik Draeger, Michael Driscoll, Todd Gamblin, Evangelos Georganas, Penporn Koanantakool, Richard Lin, Benjamin Lipshitz, Satish Rao, Oded Schwartz, Harsha Simhadri, Brian Van Straalen, and Sam Williams with whom I also worked and had many useful discussions throughout my time in Berkeley. Also, I'd like to thank Andrew Waterman for help and useful discussions on a number of topics and especially his insight on computer architecture. I look forward to and hope to do more collaborative work with everyone mentioned above.

I also thank my family and friends for positivity and support in my research endeavours and beyond. Lastly, I want to thank Krell Institute for the Department of Energy Computational Science Graduate Fellowship, the support of which I have enjoyed over the past four years and which has opened many doors for me.

Chapter 1

Introduction

A pervasive architectural evolutionary trend is the growth of parallelism and associated relative growth of the cost of communication of data and synchronization among processors with respect to the cost of execution of local operations on the data. This changing nature of architecture makes the design of efficient numerical applications a moving target. Numerical linear algebra provides a layer of abstraction, which is leveraged by optimized matrix libraries. Tensors raise the level of abstraction and allow expression of more complex numerical methods on top of optimized library primitives. Further, symmetries in the structure of data are expressible via the tensor representation and can be used to lower the algorithmic cost of tensor operations. This thesis analyzes new algorithms for numerical linear algebra computations as well as symmetric tensor computations, deriving communication-optimal schedules and benchmarking implementations of them on distributed-memory supercomputers.

Tensors, most commonly vectors and matrices, are the predominant data abstractions in numerical methods and computations. Non-numerical algorithms, such as shortest-paths computations on graphs may also be formulated via algebras over other semirings as matrix (or, for hypergraphs, tensor) operations. Frequently, the most computationally-expensive operations in numerical methods are expressible as algebraic operations on tensors. Such a high-level representation of data and its operation allows for an interface between the numerical method developer and the algorithms which perform the tensor algebra. This abstraction attracts the development of efficient algorithms and computational frameworks for tensor algebra.

Chapter 2 formally defines the theoretical execution and cost model, which takes into account not only the number of operations performed by any processor, but also the amount of data transferred between processors (interprocessor/horizontal communication), the amount of data transferred between main memory and cache (vertical communication), and the number of interprocessor synchronizations. The model is based on asynchronous messages whose progress must be guaranteed by synchronizations of groups of processors. This algorithmic model has similarities to the BSP model [162] and the LogP model [40], in fact we show that the algorithms from both BSP and LogP may be efficiently simulated in our model (under some assumptions on the relative value of costs in the LogP model). We also specifically consider collective communication, which is a key communication primitive in matrix and tensor parallel algorithm designs. We derive

a LogP performance model for the communication costs of communication collectives tuned for multidimensional torus networks [57] and compare them to topology-oblivious (binomial) collectives [57, 129, 155] at the end of Chapter 2. Figure 1.1 [143] motivates our network-topology-aware collective communication analysis by showing that the performance of a multicast varies dramatically depending on whether it exploits the topology of the underlying interconnect network. The bandwidth of a 1 MB multicast *drops* by a factor of as much as 30x as the number of processors grows from 2 to 4096 on a Cray XE6 (Hopper), but *grows* by a factor of 4.3x on the Intrepid Blue Gene/P (BG/P). This contrast demonstrates the advantage of topology-aware collective communication over topology-oblivious binomial collectives.

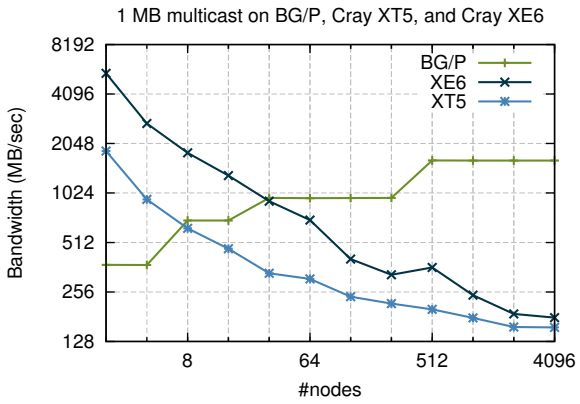


Figure 1.1: BG/P uses topology aware multicasts to utilize a high fraction of node bandwidth.

We express the parallelization or schedule of an algorithm as given by the sequence of communication, synchronization, or computation tasks performed on each processor and we measure the execution time as the most expensive sequence of dependent tasks, which may not necessarily all be done on one processor. This new critical path execution cost model (detailed in Chapter 2.1) allows us to effectively analyze the parallelization potential of algorithms by considering their dependency structure. Chapter 3 introduces techniques for dependency graph analysis that will allow us to lower bound the communication and synchronization costs which are achievable by any possible schedule for an algorithm, based on the dependency structure of the algorithm. The process of determining lower bounds (limits on best achievable costs), will allow us to evaluate the parallelization potential of algorithms and to certify whether a given schedule is optimal in its communication and synchronization costs. Figure 1.2 demonstrates the development flow which lower bound analysis enables. While Figure 1.2 considers budget constrained algorithm and schedule development, we will usually seek the algorithm with the lowest cost. So, we will seek to lower bound the communication costs of all algorithms, for which we define schedules. In a number of cases, we will present new or modified algorithms, which are able to attain a lower communication cost than existing alternatives, which is something that lower bounds on the existing algorithm allow us to prove. Chapter 3 will introduce techniques for deriving lower bounds on communication costs based on min-cut characterizations of hypergraphs, as well as techniques for deriving

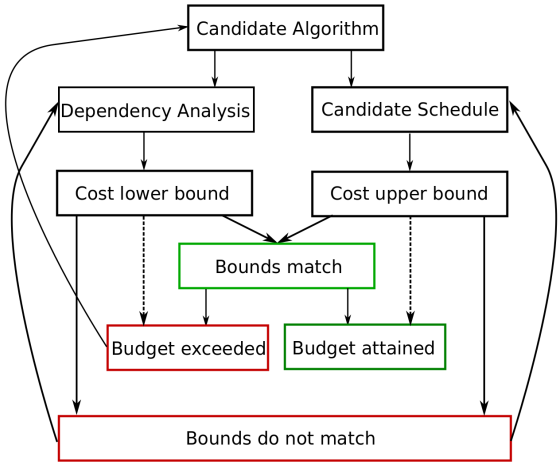


Figure 1.2: How do lower bounds facilitate the design of parallel algorithms?

tradeoff lower bounds between synchronization cost and computation as well as communication costs. This general lower bound infrastructure will allow us to derive useful lower bounds for dense matrix and tensor computations, graph algorithms, and sparse linear algebra algorithms. The infrastructure builds on previous lower bound techniques based on pebbling games [90], geometric inequalities [12, 159, 156] (most notably the Loomis-Whitney inequality [111]), as well as graph expansion [11]. Specifically, we will use our lower bound techniques to prove novel lower bounds on tradeoffs between communication and synchronization for Cholesky factorization in Chapter 5 and for Krylov subspace methods in Chapter 8.

In Chapter 4, we start specific derivation and analysis of algorithms with dense matrix multiplication, which will be a building block for the analysis of many of the later algorithms. Our analysis will be limited to classical (non-Strassen-like [152]) matrix multiplication algorithms (we will also not use Strassen-like algorithms in our analysis of other dense linear algebra problems). We derive a lower bound with tight constants on the bandwidth cost of matrix multiplication, and give parallel algorithms that attain the lower bounds for any amount of available memory. Previously, such '3D' algorithms that use extra memory have been analyzed by [1, 2, 23, 43, 93, 113], and "2.5D algorithms" (ones that use a tunable amount of extra memory) have been given by [113]. We also study the effects of topology-aware mapping strategies for matrix multiplication which arrange the interprocessor data transfers so as to use all available network links may be saturated in a torus network topology. Combining topology-awareness with asymptotic reduction in communication cost, our 2.5D matrix multiplication algorithm achieves up to a 12X speed-up over the standard '2D' algorithm on a BlueGene/P supercomputer. We study how this network utilization is achievable using pipelined collective communication [57] and also present an adaptation of Cannon's algorithm, which can explicitly utilize all torus links without the use of pipelining, achieving a 1.5X speed-up over regular Cannon's algorithm.

Chapter 5 considers the problem of solving dense linear systems of equations via dense LU or

Cholesky factorization and a triangular solve. The triangular solve and Gaussian elimination algorithms have a more intertwined dependency structure then matrix multiplication, and our lower bound analysis shows that their parallelization requires higher synchronization cost. We refer to existing and give our own versions of schedules that achieve these costs. We specifically study the potential of asymptotically reducing communication cost by utilizing additional available memory in "2.5D LU and Cholesky algorithms", which is not exploited by current '2D' parallel numerical linear algebra libraries. Such approaches have been previously been explored for LU by Irony and Toledo [87] (who minimized communication volume rather than critical path communication cost) and by Tiskin [158] for BSP. We present a more practical approach and study different pivoting algorithms for 2.5D LU factorization (partial pivoting necessitates a high synchronization cost); in contrast to Tiskin [158] who uses pairwise pivoting [149], we employ tournament pivoting [69]. The applications of our lower bounds demonstrates the optimality of the communication and synchronization cost of these parallelizations, answering an open question posted by [158]. Our method of utilizing auxiliary memory for these algorithms enables improved strong scaling (faster time to solution of problems using more processors), which we highlight in Figure 1.3.

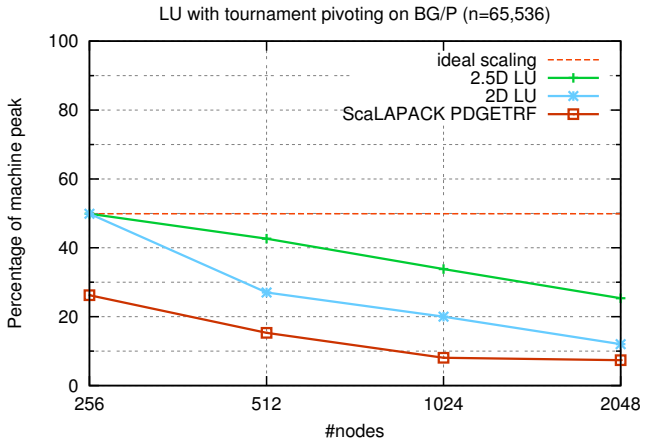


Figure 1.3: Performance of 2.5D LU (communication-avoiding algorithm) with respect to 2D LU (standard approach) on BlueGene/P supercomputer.

We take the step from LU to QR factorization in Chapter 6, where we try to attain the same communication costs in parallel algorithms for QR factorization as we did for LU factorization. One challenge posed in formulating a 2.5D QR factorization algorithm is the desire to use aggregated trailing matrix updates to achieve good bandwidth efficiency, but to avoid using Householder QR, which requires high synchronization cost. Previous work in BSP by Tiskin [158] relied on Givens rotations. We give a technique for reconstructing the Householder vectors from an implicit form given by block Givens rotations (TSQR [47]). This technique is based on a similar idea of Yamamato [169]. Our approach involves forming a panel of the orthogonal matrix and computing an LU factorization of this orthogonal matrix added to a diagonal sign matrix. We also improve the alternative approach, by showing how the implicit form may be applied to the trailing matrix more

communication-efficiently. We study the performance of our algorithms on a distributed memory supercomputer. We end the chapter by providing a parallel QR algorithm that can efficiently exploit extra available memory and yields the standard Householder representation of the orthogonal matrix. Our approach is simpler and has practical advantages with respect to [158]. Our approach of doing two-level aggregation, which may be combined with TSQR via our Householder reconstruction technique, achieved a 1.4X speed-up on a Cray XE6 supercomputer.

QR factorization itself serves as a building block for algorithms for the symmetric eigenvalue problem, which we study in Chapter 7. We focus on approaches that reduce the symmetric matrix to a tridiagonal matrix by an orthogonal transformation (so the tridiagonal matrix retains the eigenvalues of the full symmetric matrix). Finding communication-efficient algorithms for this tridiagonalization problem is more difficult as aggregating the trailing matrix update to achieve low memory-bandwidth cost necessitates reducing the symmetric matrix to successively thinner intermediate band widths [13, 6]. We give an 2.5D symmetric eigensolve algorithm that uses a minimal number of such successive reduction stages to achieve optimal memory bandwidth and interprocessor communication cost for any given cache size. We also give an alternative 2.5D successive band-reduction approach, which uses only one intermediate matrix band width and achieves minimal interprocessor communication cost but not necessarily optimal memory-bandwidth cost.

We transition from dense to sparse numerical linear algebra in Chapter 8. We specifically consider Krylov subspace computations, which rely on repeated multiplication of a vector by a sparse matrix (a ubiquitous kernel in sparse-matrix methods). The memory-bandwidth cost of multiplication of a sparse-matrix by a vector is often high, due to the necessity to read in the whole matrix from main memory into cache and the fact that no data reuse is achieved for each matrix entry. Further, the interprocessor synchronization cost incurred for each sparse-matrix vector multiplication may be a bottleneck. There are well known methods (s-step methods) which achieve better cache reuse and perform fewer synchronizations for Krylov subspace methods via performing blocking across multiple sparse-matrix-by-vector multiplications [90, 109, 49, 154]. However, the interprocessor communication cost (amount of data transferred) requirements of these methods is higher for most sparse matrices, albeit by a low order term when the problem size per processor is large enough. By application of our lower bound methods, we demonstrate that there is a tradeoff between synchronization cost and interprocessor communication cost, which is attained by s-step methods. This lower bound motivates a new algorithm, which does not attempt to lower synchronization cost, avoiding the tradeoff and keeping interprocessor communication low, but still succeeds in lowering the memory-bandwidth cost. This algorithm promises the most potential benefit, when executing problems that do not fit wholly into cache, but are also not many times larger than the cache. One such scenario occurs in the context of heterogeneous architectures, which have a local workspace that is smaller than the host-processors memory capacity and cannot fit the whole problem at once. In this scenario, the theoretical memory-bandwidth improvements achieved by the algorithm would translate to lowering the amount of communication between the host-processor and accelerator.

Chapter 9 explores problems outside the domain of numerical linear algebra, namely shortest-paths algorithms for graphs. In fact, many shortest-paths algorithms have similar structure to numerical linear algebra algorithms and can be formulated as matrix computations over the tropical

semiring [117]. We show that our lower bound technique applies to the Bellman-Ford single-source shortest paths algorithm [20, 62], which is similar in structure to Krylov subspace computations. Further, the lower bounds apply to the Floyd-Warshall all-pairs shortest-paths algorithm [59, 167], which is analogous to Gaussian elimination. We study communication-efficient algorithms and their performance for Floyd-Warshall in detail. We demonstrate that using data replication in the parallelization of Floyd-Warshall improves performance by a factor of up to 2X for large problems and up to 6.2X for small problems on a Cray XE6 supercomputer.

The last two major chapters consider algorithms for tensors, extending algorithmic ideas from matrix computations to higher-order tensors. We focus on tensor contractions, with the primary motivation of accelerating applications in quantum chemistry, most notably coupled cluster [165, 17, 39], which employs tensors of order 4, 6, and 8. Chapter 10 studies the problem of mapping and redistributing the data of a tensor on a distributed-memory parallel computer and introduces Cyclops Tensor Framework, a library which solves this problem in a general fashion. The framework uses distributed packed symmetric data layouts to store symmetric tensors and is capable of migrating the data between many decompositions, including mappings onto different processor grids and mappings where data is replicated. The framework performs tensor contractions by selecting the best choice of parallel schedule and data mapping based on a performance model, redistributing the data, then executing the schedule. The contractions done by the framework are capable of exploiting symmetry that is preserved in the contraction equation and leverages matrix multiplication to achieve high performance locally. We demonstrate the scalability of Cyclops Tensor Framework on the BlueGene/Q and Cray XC30 architectures, where we benchmark two methods for coupled-cluster: CCSD and CCSDT [165, 17, 39]. We highlight the weak scaling flop rate results (largest water problem with cc-pVDZ basis set on each number of processors) achieved by the framework in Figure 1.4 (see Chapter 10 for details). The performance of the framework is generally faster than NWChem [31], the most widely-used distributed memory application solving the same problem, which automates contraction equation derivation via the Tensor Contraction Engine [79, 19, 64] and parallelization via Global Arrays [120].

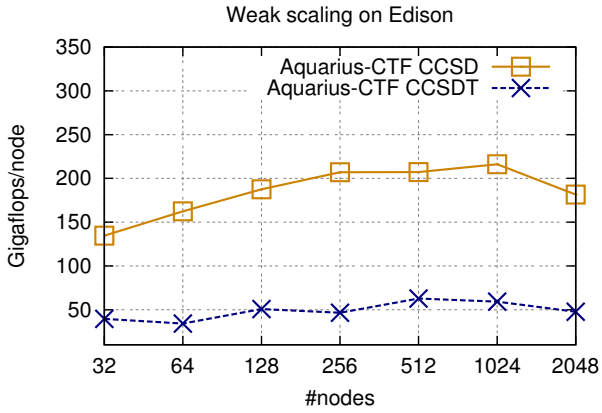


Figure 1.4: CCSD and CCSDT weak scaling with on Edison (Cray XC30).

The framework presented in Chapter 10 follows previous approaches in exploiting only tensor symmetries that are preserved in the contraction to lower computation cost. Chapter 11 demonstrates that it is possible to exploit symmetry more effectively in cases where the contraction 'breaks' the symmetry. For instance, in symmetric-matrix-by-vector multiplication, the entries of the symmetric matrix are multiplied by different vector entries, so typically n^2 multiplications are done if the matrix dimension is n. Our method is capable of performing symmetric-matrix-by-vector multiplication with $n^2/2$ scalar multiplications to leading order, but a larger number of additions. We give a general form of the approach, which can handle arbitrary order symmetric contractions, and lowers the number of multiplications by a factor of up to $\omega!$ when ω is the number of different tensor indices involved in the contraction. This method is similar to Gauss's trick for complex number multiplication, Strassen's matrix multiplication algorithm [152], and the simultaneous matrix multiplication algorithm described by Victor Pan [123], as it computes redundant terms and then subtracts them out, lowering the number of multiplications but increasing the relative number of additions. We show how the method can be adapted to handle antisymmetric and Hermitian tensors. The applications of the approach range from lowering the leading order number of total operations needed for some complex BLAS routines by a factor of $4/3$ to achieving higher speed-ups (we give examples with 2X, 4X, and 9X speed-ups) for coupled-cluster contractions, where the algorithm is applied in a nested fashion over multiple antisymmetric index groups of partially-antisymmetric tensors.

Chapter 2

Theoretical Performance Model

We model a parallel machine as a homogeneous network of p processors, each with a local main memory of size M and a cache of size $\hat{M}$, which communicate via asynchronous point-to-point messages and collective (though not necessarily global) synchronizations (collective communication algorithms are derived on top of the point-to-point messaging model). This model has four basic architectural parameters,

- α = network latency, time for a synchronization between two or more processors,
- β = time to inject a word of data into (or extract it from) the network,
- γ = time to perform a floating point operation on local data,
- ν = time to perform a transfer of a word of data between cache and main memory.

which are associated with four algorithmic costs,

- S = number of synchronizations (network latency cost),
- W = number of words of data moved (interprocessor bandwidth cost / communication cost),
- F = number of local floating point operations performed (computational cost),
- $\hat{W}$ = number of words of data moved between main memory and cache (memory bandwidth cost).

Each of these quantities is in our scheduling model accumulated along some path of dependent tasks in the schedule. The longest path weighted by the given cost of each task in the schedule is

Parts of this chapter are based on joint work with Erin Carson and Nicholas Knight [145] and Abhinav Bhatele [143].

In this chapter and later we denote a discrete integer interval $\{1, \ldots, n\}$ by $[1, n]$ and let discrete and contiguous intervals be disambiguated by context.

the critical path cost. The sequence of execution tasks done locally by any processor corresponds to one such path in the schedule, so our costs are at least as large as those incurred by any single processor during the execution of the schedule. The parallel execution time of the schedule is closely proportional to these four quantities, namely,

$$\max(\alpha \cdot S, \beta \cdot W, \gamma \cdot F, \nu \cdot \hat{W}) \leq \text{execution time} \leq \alpha \cdot S + \beta \cdot W + \gamma \cdot F + \nu \cdot \hat{W}.$$

We do not consider overlap between communication and computation and so are able to measure the four quantities separately and add them, yielding the above upper bound on execution time, which is at most 4X the lower bound. We detail our scheduling model and show how these costs may be formally derived in Section 2.1. We demonstrate that if an algorithm has a certain LogP cost, it can be done with similar cost in our scheduling model, assuming that the latency parameters in LogP are not very different. Similarly, we demonstrate that our scheduling model can simulate BSP algorithms. These reductions imply that lower bounds for communication costs of dependency graphs in our model hold as lower bounds on communication cost of any BSP/LogP parallelization and that all algorithms in BSP and LogP are valid algorithms in our model with similar upper bounds on costs.

In Section 2.2, we additionally derive more detailed performance models for collective communication, which employ the LogP messaging model. These collective communication models will be instrumental for predicting and modelling the performance of parallel algorithms for dense linear algebra and tensor computations. Techniques from efficient collective algorithms will also later find their place in the design of communication-efficient numerical algorithms in Chapter 4 on matrix multiplication and Chapter 6 on QR factorization.

We analyze on binomial collectives, which are usually used in a network-oblivious manner, and rectangular collectives, which are specially-designed for torus network architectures. We contrast the scalability between these two types of collectives on current computer architectures and measure the precision of our performance models. We then give use our performance models to predict relative performance of the collectives on a future supercomputer architecture based on projected parameters. Our results suggest that on future architectures topology-aware collectives will be many times (>10X) faster.

The rest of this chapter is organized as follows,

- Section 2.1 gives our main scheduling and cost model,
- Section 2.2 derives our performance model of communication collectives.

2.1 Scheduling Cost Model

The **dependency graph** of an algorithm is a directed acyclic graph (DAG), $G = (V, E)$. The vertices $V = I \cup Z \cup O$ correspond to either input values I (the vertices with indegree zero), or the results of (distinct) operations, in which case they are either temporary (or intermediate) values Z, or outputs O (including all vertices with outdegree zero). This dependency graph corresponds

to an execution of an algorithm for a specific problem and is a static representation that is not parameterized by the values assigned to the vertices I. There is an edge $(u, v) \in E \subset V \times (Z \cup O)$ if v is computed from u (so a k-ary operation resulting in v would correspond to a vertex with indegree k). These edges represent data dependencies, and impose limits on the parallelism available within the computation. For instance, if the dependency graph $G = (V, E)$ is a line graph with $V = \{v_1, \ldots, v_n\}$ and $E = \{(v_1, v_2), \ldots, (v_{n-1}, v_n)\}$, the computation is entirely sequential, and a lower bound on the execution time is the time it takes a single processor to compute $F = n - 1$ operations. Using graph expansion and hypergraph analysis, we will derive lower bounds on the computation and communication cost of any execution of dependency graphs with certain properties. In the following subsections, we develop a formal model of a parallel schedule in detail and show that our construction generalizes the BSP and the LogP models.

2.1.1 Parallel Execution Model

A **parallelization** of an algorithm corresponds to a coloring of its dependency graph $G = (V, E)$, i.e., a partition of the vertices into p disjoint sets $C_1, \ldots, C_p$ where $V = \bigcup_{i=1}^{p} C_i$and processor i computes $C_i \cap (Z \cup O)$. We require that in any parallel execution among p processors, at least two processors compute at least $\lfloor |Z \cup O|/p \rfloor$ elements; this assumption is necessary to avoid the case of a single processor computing the whole problem sequentially (without parallel communication). We will later make further problem-specific restrictions that require that no processor computes more than some (problem-dependent) constant fraction of $Z \cup O$. Any vertex v of color i ($v \in C_i$) must be communicated to a different processor j if there is an edge from v to a vertex in C_j, though there need not necessarily be a message going directly between processor i and j, as the data can move through intermediate processors. We define each processor's **communicated set** as

$$T_i = \{u : (u, w) \in [(C_i \times (V \setminus C_i)) \cup ((V \setminus C_i) \times C_i)] \cap E\}.$$

We note that each T_i is a vertex separator in G between $C_i \setminus T_i$ and $V \setminus (C_i \cup T_i)$.

We define a **(parallel) schedule** of an algorithm with dependency graph $G = (V, E)$ as a DAG $\bar{G} = (\bar{V}, \bar{E})$, which consists of a set of p edge-disjoint paths, Π, where the vertices in each $\pi \in \Pi$ correspond to the tasks executed by a certain processor. The edges along each processor's path carry the state of the cache and main memory of that processor. In our notation, functions that have domain $\bar{V}$ will have hats and functions that have domain $\bar{E}$ will have tildes. Each vertex $\bar{v} \in \bar{V}$ corresponds to a unique type within the following types of tasks:

$\bar{v} \in \bar{V}_{\text{comp}}$ the computation of $\hat{f}(\bar{v}) \subset V$,

$\bar{v} \in \bar{V}_{\text{sync}}$ a synchronization point,

$\bar{v} \in \bar{V}_{\text{send}}$ the sending of a message $\hat{s}(\bar{v}) \subset V$

$\bar{v} \in \bar{V}_{\text{recv}}$ the reception of a message $\hat{r}(\bar{v}) \subset V$

$\bar{v} \in \bar{V}_{\text{trsf}}$ the transfer of a set of data $\hat{h}(\bar{v}) \subset V$ between cache and main memory

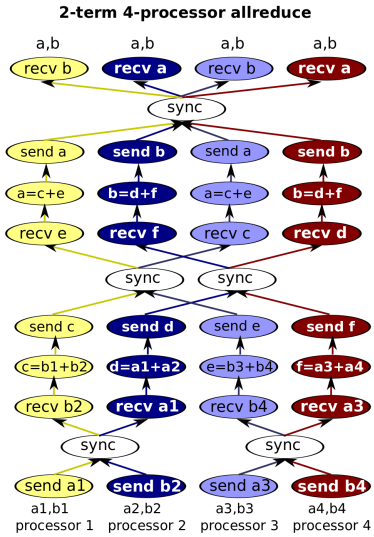


Figure 2.1: Depiction of a sample 4-processor execution schedule in our model for an all-reduction, which computes $a = \sum_i a_i$ and $b = \sum_i b_i$. Vertices for transfer of data between memory and cache would also be needed between each send or receive task and a computation task.

We assign a unique color (processor number) i using function $\tilde{c}$ to the edges along each path π, so that for every edge $\bar{e} \in (\pi \times \pi) \cap \bar{E}$, $\tilde{c}(\bar{e}) = i$. Each vertex $\bar{u} \in \bar{V}_{\text{comp}} \cup \bar{V}_{\text{send}} \cup \bar{V}_{\text{recv}}$ should be adjacent to at most one incoming edge $\bar{e}_i$ and at most one outgoing edge $\bar{e}_o$. We assign $\bar{u}$ the same color $\hat{c}(\bar{u}) = \tilde{c}(\bar{e}_i) = \tilde{c}(\bar{e}_o)$ as the processor which executes it (the single path that goes through it). Each vertex $\bar{v} \in \bar{V}_{\text{sync}}$ corresponds to a synchronization of $k \in [2, p]$ processors and should have k incoming as well as k outgoing edges. With every edge $\bar{e} \in \bar{E}$, we associate the data kept by processor $\tilde{c}(\bar{e})$ in memory as $\tilde{m}(\bar{e}) \subset V$, the data a processor keeps in cache between tasks as $\tilde{h}(\bar{e}) \subset V$, the data kept in the send buffer as a collection of sets $\tilde{s}(\bar{e}) = \{W_1, W_2, \ldots\}$ where each $W_i \subset V$ denotes the contents of a sent point-to-point message posted at some node $\bar{u}$ with $\hat{s}(\bar{u}) = W_i$, and the data kept in the receive buffer as a collection of sets $\tilde{r}(\bar{e}) = \{W'_1, W'_2, \ldots\}$ where each $W'_i \subset V$ denotes the contents of a point-to-point message received at some node $\bar{v}$ with $\hat{r}(\bar{v}) = W'_i$. The schedule has p indegree-zero vertices, corresponding to startup tasks $\bar{I} \subset \bar{V}_{\text{comp}}$, i.e., 'no-op' computations which we assume have no cost. The single outgoing edge of each startup task is assigned a disjoint part of the input data, so that $\bigcup_{\bar{e} \in (\bar{I} \times \bar{V}) \cap \bar{E}} \tilde{m}(\bar{e}) = I$. We assume that the cache starts out empty, so $\tilde{h}(\bar{e}) = \emptyset$ for any $\bar{e} \in (\bar{I} \times \bar{V}) \cap \bar{E}$.

In our model, each message is point-to-point in the sense that it has a single originating and destination processor. However, multiple messages may be transferred via the same synchronization vertex. We say the schedule $\bar{G}$ is **point-to-point** if each synchronization vertex has no more than two paths going through it. Each message is sent asynchronously, but a synchronization is required before the communicated data may change control. We illustrate an example schedule, which demonstrates the messaging behavior within our model in Figure 2.1. Transfers (reads and writes) between memory must be done to move data into cache prior to computation tasks, and to

read computed data from cache to memory prior to interprocessor communication.

We enforce the validity of the schedule via the following constraints. For every task $\bar{v}$ executed by processor i, with incoming edge $\bar{e}_{\text{in}}$ and outgoing edge $\bar{e}_{\text{out}}$,

- if $\bar{v} \in \bar{V}_{\text{comp}} \setminus \bar{I}$ (for all computation tasks),

  1. $(\forall f \in \hat{f}(\bar{v}))(\forall (g,f) \in E)$, $g \in \hat{f}(\bar{v})$ or $g \in \tilde{h}(\bar{e}_{\text{in}})$ (meaning the dependencies must be in cache at the start of the computation or computed during the computation task),
  2. $\tilde{h}(\bar{e}_{\text{out}}) \subset \hat{f}(\bar{v}) \cup \tilde{h}(\bar{e}_{\text{in}})$ (meaning the cache can carry newly computed data), and
  3. $\tilde{m}(\bar{e}_{\text{out}}) = \tilde{m}(\bar{e}_{\text{in}})$ and $\tilde{h}(\bar{e}_{\text{out}}) = \tilde{h}(\bar{e}_{\text{in}})$ (meaning the contents of the memory and cache stay the same);

- if $\bar{v} \in \bar{V}_{\text{send}}$ (for all tasks which send messages),

  1. $\exists \bar{w} \in \bar{V}_{\text{recv}}$ with $\hat{s}(\bar{v}) = \hat{r}(\bar{w})$ and a path from $\bar{v}$ to $\bar{w}$ in $\bar{G}$ (meaning the task that sends the message is connected by a path to the task that receives the message),
  2. $\hat{s}(\bar{v}) \subset \tilde{m}(\bar{e}_{\text{in}})$ and $\tilde{s}(\bar{e}_{\text{out}}) = \tilde{s}(\bar{e}_{\text{in}}) \cup \{\hat{s}(\bar{v})\}$ (meaning what is sent must be a subset of the memory at the start of the task and in the send buffer after the task),
  3. $\tilde{m}(\bar{e}_{\text{out}}) \subset \tilde{m}(\bar{e}_{\text{in}})$ (meaning what is contained in the memory after the send task must be a subset of what was in memory at its start), and
  4. $\tilde{h}(\bar{e}_{\text{out}}) = \tilde{h}(\bar{e}_{\text{in}})$ (meaning the contents of the cache stay the same);

- if $\bar{v} \in \bar{V}_{\text{recv}}$ (for all tasks that receive messages),

  1. $\hat{r}(\bar{v}) \in \tilde{r}(\bar{e}_{\text{in}})$ and $\tilde{r}(\bar{e}_{\text{out}}) = \tilde{r}(\bar{e}_{\text{in}}) \setminus \{\hat{r}(\bar{v})\}$ (meaning the receive buffer at the start of the task contained the data received by the task and does not contain it thereafter),
  2. $\tilde{m}(\bar{e}_{\text{out}}) \subset \tilde{m}(\bar{e}_{\text{in}}) \cup \hat{r}(\bar{v})$ (meaning the memory contents after the task may contain the data received by the task), and
  3. $\tilde{h}(\bar{e}_{\text{out}}) = \tilde{h}(\bar{e}_{\text{in}})$ (meaning the contents of the cache stay the same); and,

- if $\bar{v} \in \bar{V}_{\text{trsf}}$ (for all tasks which transfer data between cache and memory),

  1. $\hat{h}(\bar{v}) \subset \tilde{m}(\bar{e}_{\text{in}}) \cup \tilde{h}(\bar{e}_{\text{in}})$ (meaning the transferred data is either inside the incoming main memory or inside the cache),

2. $\tilde{m}(\bar{e}_{\text{out}}) \subset \tilde{m}(\bar{e}_{\text{in}}) \cup \hat{h}(\bar{v})$ (meaning the outgoing main memory contents may gain any elements from the transfer buffer),
3. $\tilde{h}(\bar{e}_{\text{out}}) \subset \tilde{h}(\bar{e}_{\text{in}}) \cup \hat{h}(\bar{v})$ (meaning the cache may gain any elements from the transfer buffer).

For all synchronization tasks $\bar{u} \in \bar{V}_{\text{sync}}$ messages may pass from send to receive buffers under the following rules,

1. if processor i posted a send $\bar{v}_i \in \bar{V}_{\text{send}}$ and processor j posted the matching receive $\bar{v}_j \in \bar{V}_{\text{recv}}$, i.e., $\hat{s}(\bar{v}_i) = \hat{r}(\bar{v}_j)$, and if there exist paths $\pi_i = \{\bar{v}_i, \ldots, \bar{w}_i, \bar{u}\}$ and $\pi_j = \{\bar{u}, \bar{w}_j, \ldots, \bar{v}_j\}$ in $\bar{G}$, they may exchange the message during task $\bar{u}$ by moving the data $\hat{s}(\bar{v}_i)$ from $\tilde{s}((\bar{w}_i, \bar{u}))$ to $\tilde{r}((\bar{u}, \bar{w}_j))$,
2. $\bigcup_{(\bar{v},\bar{u})\in\bar{E}} \tilde{s}((\bar{v},\bar{u})) \setminus \bigcup_{(\bar{u},\bar{w})\in\bar{E}} \tilde{s}((\bar{u},\bar{w})) = \bigcup_{(\bar{u},\bar{w})\in\bar{E}} \tilde{r}((\bar{u},\bar{w})) \setminus \bigcup_{(\bar{v},\bar{u})\in\bar{E}} \tilde{r}((\bar{v},\bar{u}))$ (meaning that if the data on one processor's path is exchanged in a synchronization node, it should be removed from the send buffer after the task),
3. $\tilde{c}((\bar{v},\bar{u})) = \tilde{c}((\bar{u},\bar{w})) \Rightarrow \tilde{m}((\bar{v},\bar{u})) = \tilde{m}((\bar{u},\bar{w}))$ and $\tilde{h}((\bar{v},\bar{u})) = \tilde{h}((\bar{u},\bar{w}))$ (meaning that the memory and cache contents on each processor are unmodified during a synchronization task).

For all edges $\bar{e} \in \bar{E}$, the cache contents must not exceed the cache size, $|\tilde{h}(\bar{e})| \leq \hat{M}$, and the contents of the memory must not exceed the main memory size $|m(\bar{e})| \leq M$.

The runtime of the schedule,

$$T(\bar{G}) = \max_{\pi\in\Pi} \sum_{\bar{v}\in\pi} \hat{t}(\bar{v}),$$

is maximum total weight of any path in the schedule, where each vertex is weighted according to its task's cost,

$$\hat{t}(\bar{v}) = \begin{cases} \gamma \cdot |\hat{f}(\bar{v})| & \bar{v} \in \bar{V}_{\text{comp}} \\ \alpha & \bar{v} \in \bar{V}_{\text{sync}} \\ (\nu+\beta) \cdot |\hat{s}(\bar{v})| & \bar{v} \in \bar{V}_{\text{send}} \\ (\nu+\beta) \cdot |\hat{r}(\bar{v})| & \bar{v} \in \bar{V}_{\text{recv}} \\ \nu \cdot |\hat{h}(\bar{v})| & \bar{v} \in \bar{V}_{\text{trsf}} \end{cases}.$$

The runtime $T(\bar{G})$ is by construction at least the cost incurred by any individual processor, since each processor's workload is a path through the schedule. We force all sent and received data to incur a cost of $\nu+\beta$ per element of the message transferred from memory to the send buffer or from the receive buffer to memory. Since all data from the receive buffer arrives from remote processors and all data from the sent buffer is forwarded to remote processors, at least β network cost is incurred by the sending and receiving processor per word of data in the message. Whether this data also needs to go through the memory hierarchy depends on the particular network card and

memory architecture, but in general we can expect that $\beta \gg \nu$, i.e., interprocessor communication is more expensive than memory traffic. Therefore, the $\nu+\beta$ cost does not significantly overestimate the β cost, and allows us to make the simplifying assumption that $\hat{W} \geq W$. So, from here-on we specifically consider $\hat{W}$ only when $\hat{W} > W$. Also, note that our construction allows us to ignore idle time, since a processor can only be idle at a message node if there exists a costlier path to that node.

2.1.2 Relation to Existing Models

Our theoretical model is closest to, and can efficiently simulate, the LogP model [40], which differs from our model most notably with its three hardware parameters,

- L = interprocessor latency cost incurred on network,
- o = messaging overhead incurred by sending and receiving processes,
- g = inverse bandwidth (per byte cost) of a message.

Both models are asynchronous and measure the cost of an algorithm along the longest execution path in a given schedule, so a close relationship is expected. Since our model only has a single latency parameter α, while the LogP model considers overhead o, injection rate g, and interprocessor latency L, we will consider the special case where $L = o$, i.e., the sequential overhead of sending a message is equivalent to the network latency of message delivery.

Theorem 2.1.1. *If there exists a LogP algorithm for computation G with cost $L \cdot S + g \cdot W$, there exists a point-to-point parallel schedule $\bar{G}$ with synchronization cost $O(\alpha \cdot S)$ and bandwidth cost $O(\beta \cdot W)$.*

Proof. Consider the given LogP algorithm for G, which encodes a timeline of LogP actions for each processor. We now construct an equivalent schedule $\bar{G} = (\bar{V}, \bar{E})$. Consider the kth (LogP) action of the ith processor, which is either a local computation, a sent message, or a received message. For each computation, add a sequence of computation vertices to $\bar{V}_{\text{comp}} \subset \bar{V}$ and memory transfer vertices (the cost of which was not bounded by the LogP schedule and so is also not bounded in $\bar{G}$) as necessary to move dependencies between cache and main memory. If the kth action of the ith processor is a send of dataset U which is received as the lth action of processor j, add nodes $\bar{v}_1$, $\bar{s}$, and $\bar{v}_2$ to $\bar{V}$ where

- $\bar{v}_1 \subset \bar{V}_{\text{send}}$, $\hat{s}(\bar{v}_1) = U$, $\hat{c}(v_1) = i$, and $\bar{v}_1$ succeeds the $(k-1)$th action of processor i and precedes $\bar{s}$,
- $\bar{s} \subset \bar{V}_{\text{sync}}$ and $\bar{s}$ has an incoming edge from $\bar{v}_1$ as well as from the $(l-1)$th action of processor j, and outgoing edges to $\bar{v}_2$ and the $(k+1)$th action of processor i,
- $\bar{v}_2 \subset \bar{V}_{\text{recv}}$, $\hat{r}(\bar{v}_2) = U$, and $\bar{v}_2$ is of color j with an incoming edge from $\bar{s}$.

Since each synchronization node has two paths through it, $\bar{G}$ is a point-to-point schedule. A runtime bound on the LogP algorithm provides us with a bound on the cost of any path in $\bar{G}$, since any path in the LogP algorithm exists in $\bar{G}$ and vice versa. Every adjacent pair of actions on this path will be done consecutively on a single processor or the path will follow some message of size $\bar{M}$. In the former case, the cost of the LogP message on the path is $o + g \cdot \bar{M}$, while the cost of this portion of the path in $\bar{G}$ is $\alpha + \beta \cdot \bar{M}$. In the latter case, the cost of the LogP message is $o \cdot 2 + L + g \cdot \bar{M}$, while a path in $\bar{G}$ which goes from the sending to the receiving node will incur cost $\alpha + \beta \cdot 2\bar{M}$. Since we limit our analysis to $L = o$, this means each LogP interprocessor path message cost of $O(L + g \cdot \bar{M})$ and cost incurred by a single processor for sending or receiving a message, $O(o + g \cdot \bar{M})$, translates to a cost of $O(\alpha + \beta \cdot \bar{M})$ in the constructed schedule. Since we know that for any path in the LogP schedule, the bandwidth cost is bounded by $O(\beta \cdot W)$, and since $\bar{G}$ has the same paths, the bandwidth cost of any path in $\bar{G}$ is also bounded by $O(\beta \cdot W)$ Since the cost of the LogP schedule is bounded by S, the longest path will be of length S and latency cost $O(\alpha \cdot S)$, giving a total cost bound of $O(\alpha \cdot S + \beta \cdot W)$. □

Our theoretical model is also a generalization of the BSP model [162], which allows global synchronization in unit synchronization cost, with each such synchronization defining a BSP timestep. At every timestep/synchronization, processors may communicate arbitrary datasets between each other, which is referred to as an h-relation, where h is the largest amount of data sent or received by any processor at the synchronization point.

We show a reduction from any BSP algorithm to an algorithm in our model by adding a global synchronization vertex for every BSP timestep, which follows the local computations and sends posted at each timestep and precedes the receives of these sends at the beginning of the next BSP timestep.

Theorem 2.1.2. *Given a BSP algorithm for computation G with S synchronizations and W words communicated, there exists a parallel schedule $\bar{G}$ with synchronization cost $O(\alpha \cdot S)$ and bandwidth cost $O(\beta \cdot W)$.*

Proof. We construct $\bar{G} = (\bar{V}, \bar{E})$ from the BSP schedule by adding a single synchronization vertex to $\bar{V}_{\text{sync}}$ for each (BSP) timestep and connecting all processors' execution paths to this vertex. So, for the jth of the S timesteps, we define a global synchronization vertex $\bar{s}_j$, as well as sequences of vertices $\bar{F}_{1j}, \ldots, \bar{F}_{pj}$ in $\bar{V}_{\text{comp}}$ corresponding to the local computations and memory transfers done by each processor during that BSP timestep (BSP only bounds the computation cost, so we have no bound on memory transfers). For the kth message sent by processor i during timestep j, we add a vertex $\bar{w}_{ijk}$ to $\bar{V}_{\text{send}}$ and for the lth message received by processor i, we add a vertex $\bar{r}_{ijl}$ to $\bar{V}_{\text{recv}}$. The execution path of processor i will then take the form

$$\pi_i = \{\ldots, \bar{F}_{ij}, \bar{w}_{ij1}, \bar{w}_{ij2}, \ldots, \bar{s}_j, \bar{r}_{ij1}, \bar{r}_{ij2}, \ldots, \bar{F}_{i,j+1}, \ldots\}.$$

For every BSP timestep, which executes an h-relation with cost $\beta \cdot h$ in BSP, the cost of $\bar{G}$ includes not only all paths which correspond to the data sent and received by some processor, but also paths which correspond to the data sent on one processor (no greater than h) and received on another

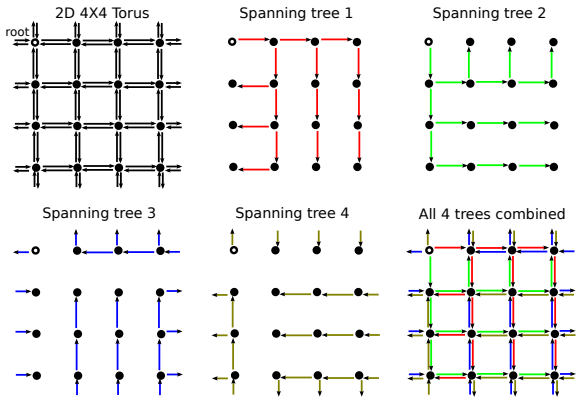


Figure 2.2: The four disjoint spanning trees used by a rectangular algorithm on a 4x4 processor grid

processor (also no greater than h), for a total cost of at most $2h$. The total communication cost of $\bar{G}$ constructed in this way will therefore be at least $\alpha \cdot S + \beta \cdot W$ but no more than $\alpha \cdot S + \beta \cdot 2W$, where W, the total bandwidth cost of the BSP algorithm, is the sum of h over all S BSP timesteps. □

2.2 Performance Models for Communication Collectives

Generic multicast (broadcast of a message to a set of processors) and reduction (e.g. summation of a set of contributions from each processor into one set) algorithms typically propagate a message down a spanning tree (e.g. binary or binomial) of nodes. To improve bandwidth and network utilization, large messages are pipelined down multiple spanning trees. Each tree spans the full list of nodes but in a different order [15, 94, 114]. However, if any of the tree edges pass through the same physical links, network contention creates a bottleneck in bandwidth utilization.

A topology aware rectangular multicast on a torus network topology can utilize all the links on the network without any contention. On a mesh of dimension d, rectangular protocols form d edge-disjoint spanning trees (each having a different 'color'). On a torus of dimension d, rectangular protocols form $2d$ colors/spanning trees. For each tree, the root sends unique packets down one of $2d$ combinations of dimension and network direction. On a ring of processors, the two edge-disjoint spanning trees are simply the two directions of the bidirectional network. On a 2D torus of processors, four edge-disjoint spanning trees are formed by routing in different dimensional order ($x \rightarrow y$, $y \rightarrow x$) and in different directions ($-x \rightarrow -y$, $-y \rightarrow -x$). These four trees are displayed in Figure 2.2. For higher dimensional meshes, rectangular algorithms form d edge-disjoint dimensional orders by performing $d-1$ circular shifts on some initial ordering ($D_1 \rightarrow D_2 \rightarrow \ldots D_d$, $D_2 \rightarrow D_3 \rightarrow \ldots D_d \rightarrow D_1$, etc.).

These topology aware multicasts are valid only if the partition is a cuboid, meaning the nodes and network allocated themselves constitute a mesh or atorus. This condition requires not only that the machine scheduler allocates cuboid partitions but also that algorithms and applications perform

multicasts on cuboid partitions. We evaluate the utility of topology aware collectives by designing and implementing algorithms that can exploit these collectives. We also derive new performance models to shed light on the scaling characteristics of these collectives and our new algorithms.

Rectangular multicasts and reductions can reduce communication time in dense linear algebra operations significantly on current supercomputers. We model the performance of rectangular collectives in order to evaluate their scaling behavior.

2.2.1 Basic Assumptions and Terminology

We base our assumptions to reflect current architectures and an idealized messaging implementation.

1. We assume the LogP performance model for messaging.
2. We assume m, the message size, is big and gear our analysis towards understanding bandwidth cost of collectives. Bandwidth cost is more relevant for dense linear algebra. Latency costs are heavily dependent on network architecture and implementation.
3. We restrict our analysis to multicasts and reductions on k-ary d-cube networks (so the torus has d dimensions and each one of the torus edges has k processors).
4. We assume the networks are bidirectional tori (there is one link in each of two directions for each adjacent pair of nodes and wrap-around links) for brevity. However, our models can be modified to meshes (without wrap-around links).
5. We assume that a DMA device and wormhole routing are used for messaging.

We build a model from the following parameters:

- $\mathbf{L}$ (seconds) is the physical network latency cost as defined by the LogP model.
- $\mathbf{o}$ (seconds) is the overhead of sending and receiving a message as defined by the LogP model.
- $\mathbf{d}$ (integer) is the number of dimensions of the k-ary d-cube network.
- $\mathbf{g}$ (seconds/bytes) is the reciprocal of the bandwidth achieved by a single message. We assume a single message achieves at best the unidirectional bandwidth of a single link.
- $\mathbf{P}$ (integer) is the number of processors.
- $\mathbf{m}$ (bytes) is the message size in bytes.
- $\boldsymbol{\gamma}$ (operations/second) is the flop rate of a node.
- $\boldsymbol{\beta}$ (bytes/second) is the memory bandwidth of a node.

2.2.2 Rectangular Multicast Model

Rectangular multicasts function by pipelining packets of a message down multiple edge-disjoint spanning trees. Each spanning tree traverses the network in a different dimensional order. As shown in figure 2.2, all sends are near-neighbor and no contention occurs.

A rendezvous send is the protocol of choice for large message sizes on modern networks. Rendezvous messaging establishes a handshake by sending a small *eager send* to exchange buffer information. Once the handshake is established, the receiver pulls the data with a one-sided *get* operation. The latency cost of an eager send under the LogP model is $2o + L$. A one-sided get requires an extra hop to start the transaction but does not incur overhead on the sender side. Therefore, the latency cost of a get is $o + 2L$. The cost of sending a rendezvous message of size m_r incurs both the eager send and get latency costs,

$$t_r = m_r \cdot g + 3o + 3L \tag{2.2.1}$$

A naive version of a rectangular multicast would synchronously send a $m_r = \frac{m}{2d}$ sized message down both directions of each dimension. Such a protocol would achieve single link bandwidth at best. A more aggressive rectangular protocol can overlap the sends in each of the $2d$ directions/dimensions. This rectangular protocol overlaps the network latency (L) and bandwidth (g) costs of each direction. However, the sequential overhead suffered by the sender grows in proportion to the number of trees. Therefore, the start-up cost of a rectangular multicast (the time it takes the root to send off the entire message) is

$$\begin{aligned} t_s &= (m/2d) \cdot g + (2d - 1) \cdot o + (3o + 3L) \\ &= (m/2d) \cdot g + 2(d + 1) \cdot o + 3L \end{aligned} \tag{2.2.2}$$

The multicast does not complete until all nodes receive the entire message. So the multicast finishes when the last packet travels all the way to the farthest node of its spanning tree. The last packet leaves the root at time t_s (Eq. 2.2.2). To get to the farthest node of any dimensional tree takes $d \cdot P^{1/d}$ hops. A rendezvous transaction handshake (Eager Send) must be established with the next node at each hop, so the cost per hop is $2o + L$. Over all hops, the overhead of the path of the packet is

$$t_p = d \cdot P^{1/d} \cdot (2o + L) \tag{2.2.3}$$

Combining t_s (Eq. 2.2.2) and t_p (Eq. 2.2.3) gives us an estimate of the time it takes to complete a multicast

$$\begin{aligned} t_{rect} &= t_s + t_p \\ &= \frac{m}{2d} \cdot g + 2(d + 1) \cdot o + 3L + d \cdot P^{1/d} \cdot (2o + L) \end{aligned} \tag{2.2.4}$$

To review, our model of the cost of a multicast (Eq. 2.2.4) is composed of

1. The bandwidth term, $(m/2d) \cdot g$ – the time it takes to send the full message out from the root.
2. The start-up overhead, $2(d + 1) \cdot o + 3L$ – the overhead of setting up the multicasts in all dimensions.
3. The per hop overhead, $d \cdot P^{1/d} \cdot (2o + L)$ – the time it takes for a packet to get from the root to the farthest destination node.

2.2.3 Rectangular Reduction Model

Reductions behave similarly to multicasts. A multicast tree can be inverted to produce a valid reduction tree. However, at every node of this tree, it is now necessary to apply some operator (do computational work) on the incoming data. We assume that the reduction operator is the same single flop operation applied to each element of the same index (e.g. sum-reduction). We assume that the elements are double-precision values. The packet size (size of pipelined chunks) must be larger than the value size (size of values on which to apply the reduction operator) to pipeline the reduction. We will assume in this analysis that the value size is 8 bytes.

We adopt the reduction model from the multicast model. However, we need to account for the extra computation and access to memory involved in the application of the reduction operator. The amount of computational work done in each spanning tree node differs depending on whether the node is a leaf or an internal node of the spanning tree. However, summing over all trees, each node receives at most m bytes of data and sends at most m bytes. Therefore, the total amount of computational work on a node is simply the reduction operator applied once on two arrays of size m bytes.

Applying the operator on 2 arrays of m bytes, requires reading $2m$ bytes, writing m bytes, and performing $m/8$ flops (8 bytes per double-precision value). The bandwidth and computational costs are effectively overlapped on modern processors. Given a DMA device, the computation can also be overlapped with the network bandwidth. So, we adjust the start-up time t_s (Eq. 2.2.2) to

$$t_{sr} = \max\left(\frac{m}{8\gamma}, \frac{3m}{\beta}, \left(\frac{m}{2d} \cdot g\right)\right) + 2(d+1) \cdot o + 3L \tag{2.2.5}$$

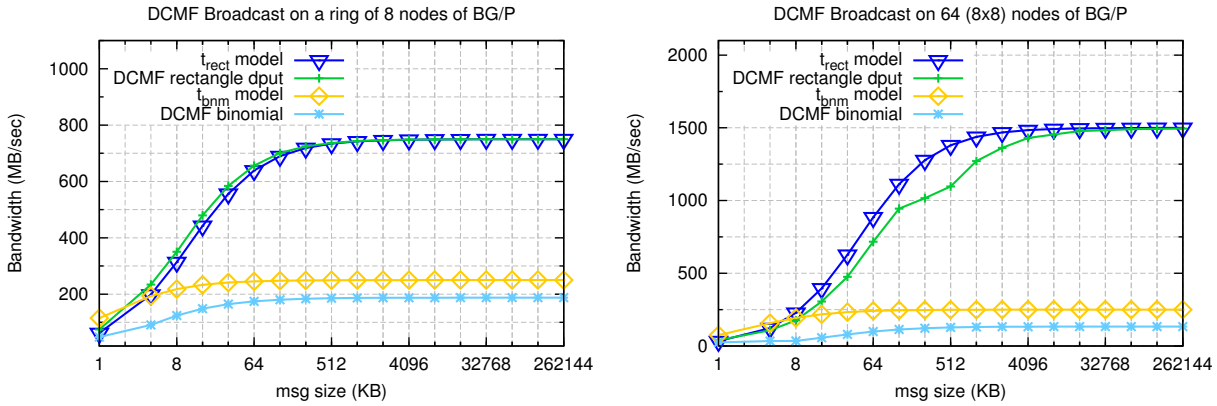
The per hop cost t_p (Eq. 2.2.3) should be the same for a reduction as a multicast. We build a reduction model by combining t_p and the new start-up cost t_{sr} (Eq. 2.2.5),

$$\begin{aligned} t_{red} &= t_{sr} + t_p \\ &= \max\left(\frac{m}{8\gamma}, \frac{3m}{\beta}, \left(\frac{m}{2d} \cdot g\right)\right) + 2(d+1) \cdot o + 3L \\ &\quad + d \cdot P^{1/d} \cdot (2o + L) \end{aligned} \tag{2.2.6}$$

2.2.4 Binomial Tree Multicast Model

Binomial tree multicasts are commonly used as generic algorithms for multicasts and reductions [57, 129, 155]. Binomial collectives models have been written for the LogGP model [4] and for the Hockney model [80] in [155]. Here, we construct a slightly modified binomial tree multicast model under the LogP model. Our model reflects the DCMF binomial multicast implementation on BG/P [57].

A binomial tree multicast has $\log_2(P)$ stages. In each stage, the multicast root sends the message to a node in a distinct sub-partition. Each sub-partition recursively applies the algorithm on a smaller subtree. A message can be pipelined into these stages to exploit multiple links simultaneously. We assume that there is no network contention in the tree. Modeling contention is non-trivial and out of the scope of this thesis.



(a) 1D multicast (b) 2D multicast

Figure 2.3: Multicast on an 8-node and a n8-by-8 node toroidal grid of processors

The binomial tree is unbalanced. The first established sub-tree is the deepest. Therefore, the overhead of the first packet traversing the deepest sub-tree dominates the overhead of the root establishing handshakes with each of the sub-trees. So the cost of a binomial tree multicast is

$$t_{bnm} = \log_2(P) \cdot ((m/2d) \cdot g + 2o + L) \tag{2.2.7}$$

2.2.5 Validation of Multicast Model

Our rectangular collectives models are based purely on architectural parameters. We can validate the quality of our assumptions and analysis by comparing our performance prediction with an actual implementation of rectangular algorithms on a Blue Gene/P machine. We implemented our benchmarks using DCMF [103]. We restrict our analysis to multicasts and not reductions, due to DCMF reduction performance being lower than expected (see [143] for details). Subtracting the performance of DCMF Get (cost: $o + L = 1.2\,\mu$s) from DCMF Put (cost: $o + 2L = 2.0\,\mu$s) as reported in [103], we get $L = .8\,\mu$s, $o = .4\,\mu$s. The inverted achievable link bandwidth is known to be $g = 1/375$ s/MB.

We validate performance for rectangular algorithms of tori of different dimension. This analysis is important since it justifies the portability of the model among k-ary d-cubes of different dimension d. In particular, algorithms that perform subset multicasts (e.g. line multicasts/reductions in dense linear algebra), operate on lower dimensional partitions. We benchmark collectives on processor partitions of dimension $d \in \{1, 2, 3\}$. Lower dimensional toroidal partitions are generated by extracting sub-partitions of a BG/P 3D torus partition.

Figure 2.3(a) details the performance of rectangular and binomial DCMF Broadcast on a ring of 8 BG/P nodes. Our performance model (t_{rect}) matches the performance of the rectangular DCMF algorithms (DCMF rectangle dput) closely. Our binomial performance model (t_{bnm}) overestimates

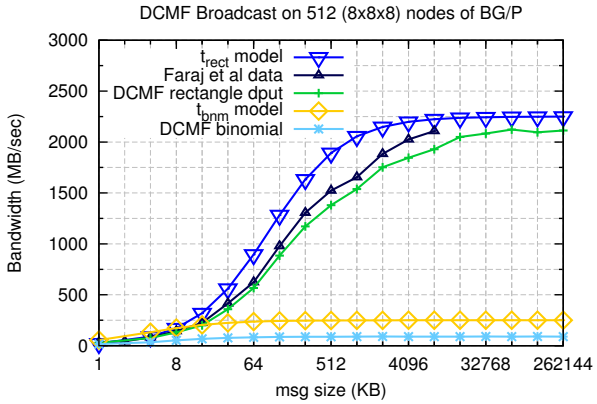


Figure 2.4: Multicast on a 8-by-8-by-8 node toroidal grid of processors

the actual achieved bandwidth (DCMF binomial) by a factor of about 1.4X, which is probably due to network contention that is ignored by our model.

The (predicted and observed) bandwidth of the binomial multicast peaks at exactly half of link bandwidth. On eight nodes, a binomial algorithm must form three trees and has two available links (directions). Our model does not consider contention and assumes the three trees share all available node bandwidth. Since two links are available, the model predicts a peak of two thirds of the link bandwidth. However, two of the trees get mapped to the same physical link of the root, creating contention over the link. As a result, the binomial protocol achieves only half of the link bandwidth.

On a 2D toroidal sub-partition, the available bandwidth as well as the overheads increase. Figure 2.3(b) shows that our model (t_{rect}) matches the performance of the rectangular, one-sided DCMF protocol (DCMF rectangle dput) fairly well. The observed data seems to take a dip in performance growth for messages larger than 128 KB. This may be due to the buffer exceeding the size of the L1 cache, which is 64 KB. The binomial model (t_{bnm}) overestimates performance again due to contention.

On a 3D partition (Figure 2.4), a similar overhead is even more pronounced than on a 2D partition. However, we see that the performance data from Faraj et al. [57] achieves higher bandwidth than the best data we were able to collect (DCMF rectangle dput). Perhaps the difference is due to our usage of DCMF or the use of a lower level interface with less overhead by Faraj et al. [57]. We also found that the performance of multicasts varied substantially depending on the buffer-size of the receive FIFO buffer. We sampled data over a few different buffer-sizes and selected the best performing data-points. Perhaps the data in [57] was collected with more intensive control for such parameters.

Comparisons between the idealized rectangular performance and actual observed data show that implementation overhead grows as a function of dimension. Higher dimensional rectangular algorithms stress the DMA implementation by utilizing multiple links. The hardware and software have to manage communication through multiple links simultaneously.

Total flop rate (γ)	10^{18} flop/s
Total memory	32 PB
Node count (P)	262,144
Node interconnect bandwidth (g)	.06 s/GB
Latency overhead (o)	250 ns
Network latency (L)	250 ns
Topology	3D Torus
Size of dimensions	$64 \times 64 \times 64$

Table 2.2.1: Predicted architecture characteristics of an Exaflop/s machine

The binomial multicast performance suffers severely from network contention. It is particularly difficult to build a general contention model since it is dependent on the semantics of spanning tree construction. Rectangular algorithms have *no contention* when implemented on a torus network. This feature improves their scalability with comparison to binomial and other tree-based generic algorithms.

2.2.6 Predicted Performance of Communication Collectives at Exascale

The performance of binomial and tree-based collectives deteriorates with increased partition size due to increased tree depth and contention. In particular, contention and branch factor of trees bounds the peak achievable bandwidth. The only cost of rectangular collectives that grows with partition size is the increased depth of spanning trees. To examine the scaling of collectives, we model performance of raw collectives and dense linear algebra algorithms on a potential exascale architecture.

2.2.7 Exascale Architecture

Table 2.2.1 details the parameters we use for an exascale machine. These parameters are derived from a report written at a recent exascale workshop [160].

We assume the exascale machine is a 3D torus. Our analysis and conclusions would not change significantly for a torus of slightly higher dimension. However, rectangular collectives are not applicable to a switched network. In particular, the Dragonfly network architecture [98] seems to be a suitable exascale switched network topology.

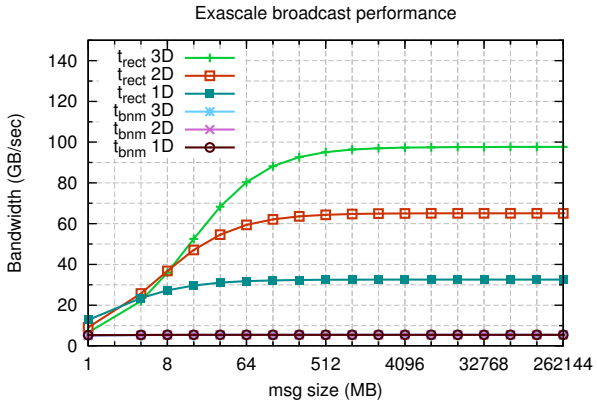


Figure 2.5: Performance of rectangular and binomial multicasts on a potential exascale torus architecture. Lines signifying performance of binomial collectives overlap, since our binomial collective model is independent of network architecture.

2.2.8 Performance of Collectives at Exascale

We model the performance of collectives on 1D, 2D, and 3D sub-partitions of the exascale machine. Figure 2.5 details the performance of rectangular and binomial multicasts for these partitions. The peak bandwidth of binomial multicast stays the same for each partition since the number of trees increases proportionally to the number of available links. In practice, this peak bandwidth would actually deteriorate on larger partitions due to increased network contention.

Figure 2.5 also demonstrates that to achieve peak bandwidth, large messages are required (1 GB for a 3D partition). We assume that intra-node concurrency is utilized hierarchically. A single process per node performs all inter-node communication.

Chapter 3

Communication Lower Bound Techniques

Our lower bound techniques build on methods first introduced by Hong and Kung [90] for sequential algorithms and also extended to the parallel case by Irony et al. [88]. Our lower bounds are extensions to those obtained by Ballard et al. [12]. We discuss and introduce more related lower bounds work in detail in Section 3.1. Like [88] and [12], we will employ volumetric inequalities to derive communication lower bounds. Section 3.2 states the Loomis-Whitney lower bound, a generalization of it given by Tiskin [156], and introduces a specialization (tighter version of the bound) of the latter to normally-ordered sets.

This specialized inequality will allow us to establish a lower bound on the minimum cut size of any hypergraph lattice in Section 3.3. These hypergraph lattices will be characterizations of dependency structures that are present in many of the algorithms we discuss in later chapters. Obtaining a lower bound on the cut of hypergraph lattices will then allow us to obtain a lower bound on the communication cost necessary to execute a computation with such a lattice dependency structure. In Section 3.3, we introduce and study (m, r)-lattice hypergraphs where the set of vertices has dimension m and is connected by a set of hyperedges of dimension r. For the linear algebra applications we consider, it will be the case that $r = m - 1$, however, in Chapter 11, we will use bounds on lattice hypergraph cuts for arbitrary $r < m$.

Knowing the expansion and minimum cut parameters of certain dependency graphs, we will show how these expansion parameters yield lower bounds on tradeoffs between synchronization cost S, interprocessor bandwidth cost W and computation cost F in Section 3.4. Most of the applications of our lower bounds apply to computations which have $\Omega(n^d)$ vertices, with a d-dimensional lattice dependency structure (where $d = m$ and $r = d - 1$ for the corresponding (m, r)-lattice hypergraph as defined in Section 3.3), and take the form

$$F \cdot S^{d-1} = \Omega\left(n^d\right), \quad W \cdot S^{d-2} = \Omega\left(n^{d-1}\right).$$

These bounds indicate that a growing amount of local computation, communication, and synchronization must be done to solve a larger global problem. Thus, the bounds are important because

This chapter is based on joint work with Erin Carson and Nicholas Knight [145].

they highlight a scalability bottleneck dependent only on local processor/network speed and independent of the number of processors involved in the computation. The lower bounds are motivated by tradeoffs between computation and synchronization costs demonstrated by Papadimitriou and Ullman [124] and are are a generalization of the result.

The tradeoff infrastructure presented in this chapter will allow us to prove the following bounds in later chapters:

1. for solving a dense n-by-n triangular system by substitution (TRSV),

$$F_{\mathrm{TR}} \cdot S_{\mathrm{TR}} = \Omega\left(n^2\right),$$

   which is attainable as discussed in Section 5.2,

2. for Cholesky of a dense symmetric n-by-n matrix,

$$F_{\mathrm{Ch}} \cdot S_{\mathrm{Ch}}^2 = \Omega\left(n^3\right), \qquad W_{\mathrm{Ch}} \cdot S_{\mathrm{Ch}} = \Omega\left(n^2\right),$$

   which are also attainable as shown in Section 5.3, although they are provably not attainable by conventional algorithms such as LU with partial pivoting, and

3. for computing an s-step Krylov subspace basis with a $(2m+1)^d$-point stencil (defined in Section 8.1),

$$F_{\mathrm{Kr}} \cdot S_{\mathrm{Kr}}^d = \Omega\left(m^d \cdot s^{d+1}\right), \qquad W_{\mathrm{Kr}} \cdot S_{\mathrm{Kr}}^{d-1} = \Omega\left(m^d \cdot s^d\right),$$

   which are again attainable (Section 8.3).

The Cholesky lower bounds which we derive in Chapter 5 suggest the communication optimality of the parallel algorithms for LU factorization given by [113] and [146]. The parallel schedules for LU and QR in these papers are parameterized and exhibit a trade-off between synchronization and communication bandwidth cost. We come close to answering an open question posed by [158], showing that it is not possible to achieve an optimal bandwidth cost without an associated increase in synchronization cost for Cholesky (the question was posed for LU factorization and we assume that no recomputation is performed).

The rest of the chapter is organized as follows,

- Section 3.1 discusses previous and related work on communication lower bounds,
- Section 3.2 states the Loomis-Whitney inequality as well as a generalization of it then gives a new specialization to normally-ordered sets,
- Section 3.3 gives a definition of lattice hypergraphs and proves a lower bound on their minimum cut,
- Section 3.4 uses characterizations of the path-expansion of dependency graphs to derive lowers bounds on tradeoffs between costs.

3.1 Previous Work

Theoretical lower bounds on communication volume and synchronization are often parameterized by the size of the cache, $\hat{M}$, in the sequential setting, or the size of the main memory M, in the parallel setting. Most previous work has considered the total sequential or parallel communication volume Q, which corresponds to the amount of data movement across the network (by all processors), or through the memory hierarchy. Hong and Kung [90] introduced sequential communication volume lower bounds for computations including n-by-n matrix multiplication, $Q_{\mathrm{MM}} = \Omega(n^3/\sqrt{\hat{M}})$, the n-point FFT, $Q_{\mathrm{FFT}} = \Omega(n\log(n)/\log(\hat{M}))$, and the d-dimensional diamond DAG (a Cartesian product of line graphs of length n), $Q_{\mathrm{dmd}} = \Omega(n^d/\hat{M}^{1/(d-1)})$. Irony et al. [88] extended this approach to distributed-memory matrix multiplication on p processors, obtaining the bound $W_{\mathrm{MM}} = \Omega(n^3/(p\sqrt{M}))$. Aggarwal et al. [2] proved a version of the memory-independent lower bound $W_{\mathrm{MM}} = \Omega(n^3/p^{2/3})$, and Ballard et al. [9] explored the relationship between these memory-dependent and memory-independent lower bounds. Ballard et al. [12] extended the results for matrix multiplication to Gaussian elimination of n-by-n matrices and many other matrix algorithms with similar structure, finding

$$W_{\mathrm{Ch}} = \Omega\left(\frac{n^3}{p\sqrt{\min(M, n^2/p^{2/3})}}\right).$$

Bender et al. [21] extended the sequential communication lower bounds introduced in [90] to sparse matrix vector multiplication. This lower bound is relevant to our analysis of Krylov subspace methods, which essentially perform repeated sparse matrix vector multiplications. However, [21] used a sequential memory hierarchy model and established bounds in terms of memory size and track (cacheline) size, while we focus on interprocessor communication.

Papadimitriou and Ullman [124] demonstrated tradeoffs for the 2-dimensional diamond DAG (a slight variant of that considered in [90]). They proved that the amount of computational work F_{dmd} along some execution path (in their terminology, execution time) is related to the communication volume Q_{dmd} and synchronization cost S_{dmd} as

$$F_{\mathrm{dmd}} \cdot Q_{\mathrm{dmd}} = \Omega\left(n^3\right) \text{ and } F_{\mathrm{dmd}} \cdot S_{\mathrm{dmd}} = \Omega\left(n^2\right).$$

These tradeoffs imply that in order to decrease the amount of computation done along the critical path of execution, more communication and synchronization must be performed. For instance, if an algorithm has an 'execution time' cost of $F_{\mathrm{dmd}} = \Omega(nb)$, it requires $S_{\mathrm{dmd}} = \Omega(n/b)$ synchronizations and a communication volume of $Q_{\mathrm{dmd}} = \Omega(n^2/b)$. The lower bound on $F_{\mathrm{dmd}} \cdot S_{\mathrm{dmd}}$ is a special case of the d-dimensional bubble latency lower bound tradeoff we derive in the next section, with $d = 2$. These diamond DAG tradeoffs were also demonstrated by Tiskin [156].

Bampis et al. [14] considered finding the optimal schedule (and number of processors) for computing d-dimensional grid graphs, similar in structure to those we consider in Section 3.3. Their work was motivated by [124] and took into account dependency graph structure and communication, modeling the cost of sending a word between processors as equal to the cost of a

computation. Tradeoffs in parallel schedules have also been studied in the context of data locality on mesh network topologies by [24].

We will introduce lower bounds that relate synchronization to computation and data movement along sequences of dependent execution tasks. Our work is most similar to the approach in [124]; however, we attain bounds on W (the parallel communication volume along some dependency path), rather than Q (the total communication volume). While bounds on Q translate to bounds on the energy necessary to perform the computation, bounds on W translate to bounds on execution time. Our theory also obtains tradeoff lower bounds for a more general set of dependency graphs which allows us to develop lower bounds for a wider set of computations.

3.2 Volumetric Inequalities

A key step in the lower bound proofs in [88] and [12] was the use of an inequality introduced by [111]. We state this inequality in Theorem 3.2.1.

Theorem 3.2.1 (Loomis-Whitney Inequality)**.** *Let V be a finite, nonempty set of d-tuples $(i_1, \ldots, i_d) \in [1, n]^d$;*

$$|V| \leq \left(\prod_{j=1}^{d} |\pi_j(V)| \right)^{1/(d-1)},$$

where, for $j \in [1, d]$, $\pi_j \colon [1, n]^d \to [1, n]^{d-1}$ is the projection

$$\pi_j(i_1, \ldots, i_d) = (i_1, \ldots, i_{j-1}, i_{j+1}, \ldots, i_d).$$

We use a generalized discrete version of the inequality, which was given by [156] and is a special case of the theory in [37]. We state this generalized inequality in Theorem 3.2.2.

Theorem 3.2.2 (Generalized Discrete Loomis-Whitney Inequality)**.** *Let V be a finite, nonempty set of m-tuples $(i_1, \ldots, i_m) \in [1, n]^m$. Consider $\binom{m}{r}$ projections*

$$\pi_{s_1,\ldots,s_r}(i_1, \ldots i_m) = (i_{s_1}, \ldots, i_{s_r})$$

$\forall (s_1, \ldots, s_r)$ such that $1 \leq s_1 < \ldots < s_r \leq m$,

$$|V| \leq \left(\prod_{1 \leq s_1 < \ldots < s_r \leq m} |\pi_{s_1,\ldots,s_r}(V)| \right)^{1/\binom{m-1}{r-1}}.$$

We derive a version of the Loomis-Whitney inequality for sets of normally-ordered (i.e., increasing) tuples, in order to obtain a tighter bound for our scenario. We state the inequality in Theorem 3.2.3 and prove its correctness.

Theorem 3.2.3. *Let V be a finite, nonempty set of normally-ordered m-tuples $(i_1, \ldots, i_m) \in [1, n]^m$, $i_1 < \ldots < i_m$. Consider $\binom{m}{r}$ projections*

$$\pi_{s_1,\ldots,s_r}(i_1, \ldots i_m) = (i_{s_1}, \ldots, i_{s_r})$$

$\forall (s_1, \ldots, s_r)$ such that $1 \leq s_1 < \ldots < s_r \leq m$. Define the union of these projections of V as

$$\bar{\Pi} = \bigcup_{1 \leq s_1 < \ldots < s_r \leq m} \pi_{s_1,\ldots,s_r}(V).$$

Now the size of V may be bound as a function of the number of projections in $\bar{\Pi}$,

$$|V| \leq \left(r! \cdot |\bar{\Pi}|\right)^{m/r} / m!.$$

Proof. For any V, consider the projected sets $\pi_{s_1,\ldots,s_r}(V)$ for $1 \leq s_1 < \ldots < s_r \leq m$. We now construct set $\bar{V}$, with $V \subset \bar{V}$ and obtain an upper bound on the size of $\bar{V}$, yielding an upper bound on the size of V. We consider the union of all the projections $\bar{\Pi}$ and expand this set to construct a larger set $\hat{\Pi}$, where for any r-tuple permutation p, if $w \in \bar{\Pi}$, $p(w) \in \hat{\Pi}$, and $\bar{\Pi} \subset \hat{\Pi}$. Since $\bar{\Pi}$ consisted only of r-tuples whose indices are normally-ordered (since they are ordered projections of normally-ordered d-tuples),

$$|\hat{\Pi}| = r! \cdot |\bar{\Pi}|.$$

We now construct the set $\hat{V}$ of all (not necessarily normally-ordered) tuples whose projections are in $\hat{\Pi}$, that is

$$\hat{V} = \{(w_1, \ldots, w_m) \in [1, n]^m : \text{ such that } \forall \{q_1, \ldots, q_r\} \subset \{w_1, \ldots, w_m\}, \ (q_1, \ldots, q_r) \in \hat{\Pi}\}.$$

We note that the original m-tuple set is a subset of the new one, $V \subset \hat{V}$, since

$$\forall \{q_1, \ldots, q_r\} \in \pi_{s_1,\ldots,s_r}(V) \subset \hat{\Pi}, \ \exists \{v_1, \ldots, v_m\} \in V \text{ such that } \{q_1, \ldots, q_r\} \subset \{v_1, \ldots, v_m\}.$$

Further, $\hat{V}$ includes all m-tuples such that all arbitrarily ordered r-sized subsets of these tuples is in $\hat{\Pi}$. Since subsets of all possible orderings are considered, the ordering of the m-tuple does not matter, so $\hat{V}$ should contain all orderings of each tuple, and is therefore a symmetric subset of $[1, n]^m$.

Now by application of the Loomis-Whitney inequality (Theorem 3.2.1), we know that the size of $\hat{V}$ is

$$|\hat{V}| \leq |\hat{\Pi}|^{\binom{m}{r}/\binom{m-1}{r-1}}.$$

We now let $\bar{V}$ be the set of all normally-ordered tuples inside $\hat{V}$. We have that $V \subset \bar{V}$ since, $V \subset \hat{V}$ and V is normally-ordered. Since there are $m!$ symmetrically equivalent tuples to each normally-ordered tuples inside $\hat{V}$, $|\bar{V}| = |\hat{V}|/m!$. Combining this bound with the bound on Loomis-Whitney bound on W, we obtain

$$\begin{aligned} |V| \leq |\bar{V}| \leq |\hat{V}|/m! &\leq |\hat{\Pi}|^{\binom{m}{r}/\binom{m-1}{r-1}}/m! \\ &= \left(r! \cdot |\bar{\Pi}|\right)^{\binom{m}{r}/\binom{m-1}{r-1}} / m! = \left(r! \cdot |\bar{\Pi}|\right)^{m/r} / m! \end{aligned}$$

□

3.3 Lower Bounds on Lattice Hypergraph Cuts

A hypergraph $H = (V, E)$, is defined by a set of vertices V and a set of hyperedges E, where each hyperedge $e_i \in E$ is a subset of V, $e_i \subseteq V$. We say two vertices in a hypergraph $v_1, v_2 \in V$ are connected if there exists a path between v_1 and v_2, namely if there exist a sequence of hyperedges $\mathcal{P} = \{e_1, \ldots, e_k\}$ such that $v_1 \in e_1$, $v_2 \in e_k$, and $e_i \cap e_{i+1} \neq \emptyset$ for all $i \in [1, k-1]$. We say a hyperedge $e \in E$ is **internal** to some $V' \subset V$ if $e \subset V'$. If no $e \in E$ is adjacent to (i.e., contains) a $v \in V' \subset V$, then we say V' is **disconnected** from the rest of H. A $\frac{1}{q}$-$\frac{1}{x}$**-balanced hyperedge cut** of a hypergraph is a subset of E whose removal from H partitions $V = V_1 \cup V_2$ with $\lfloor |V|/q \rfloor \leq \min(|V_1|, |V_2|) \leq \lfloor |V|/x \rfloor$ such that all remaining (uncut) hyperedges are internal to one of the two parts.

For any $q \geq x \geq 2$, a $\frac{1}{q}$-$\frac{1}{x}$**-balanced vertex separator** of vertex set $\hat{V} \subset V$ in a hypergraph $H = (V, E)$ (if $\hat{V}$ is unspecified then $\hat{V} = V$) is a set of vertices $Q \subset V$, whose removal from V and from the hyperedges E in which the vertices appear splits the vertices V in H into two disconnected parts $V \backslash Q = V_1 \cup V_2$. Further, the subsets of these vertices inside $\hat{V}$ must be balanced, meaning that for $\hat{V}_1 = \hat{V} \cap V_1$ and $\hat{V}_2 = \hat{V} \cap V_2$, $\lfloor |\hat{V}|/q \rfloor - |Q| \leq \min(|\hat{V}_1|, |\hat{V}_2|) \leq \lfloor |\hat{V}|/x \rfloor$. The smaller partition may be empty if the separator Q contains at least $\lfloor |\hat{V}|/q \rfloor$ vertices.

3.3.1 Lattice Hypergraphs

For any $m > r > 0$, we define a (m, r)**-lattice hypergraph** $H = (V, E)$ of breadth n, with $|V| = \binom{n}{m}$ vertices and $|E| = \binom{n}{r}$ hyperedges. Each vertex is represented as $v_{i_1,\ldots,i_m} = (i_1, \ldots, i_m)$ for $\{i_1, \ldots, i_m\} \in [1, n]^m$ with $i_1 < \cdots < i_m$. Each hyperedge connects all vertices which share r indices, that is $e_{j_1,\ldots,j_r}$ for $\{j_1, \ldots, j_r\} \in [1, n]^r$ with $j_1 < \cdots < j_r$ includes all vertices $v_{i_1,\ldots,i_m}$ for which $\{j_1, \ldots, j_r\} \subset \{i_1, \ldots, i_m\}$. Each vertex appears in $\binom{m}{r}$ hyperedges and therefore each hyperedge contains $\binom{n}{m}\binom{m}{r}\binom{n}{r}^{-1} = \binom{n-r}{m-r}$ vertices.

Theorem 3.3.1. *For $1 \leq r < m \ll n$ and $2 \leq x \leq q \ll n$, the minimum $\frac{1}{q}$-$\frac{1}{x}$-balanced hyperedge cut of a (m, r)-lattice hypergraph $H = (V, E)$ of breadth n is of size $\epsilon_q(H) = \Omega(n^r/q^{r/m})$.*

Proof. Consider any $\frac{1}{q}$-$\frac{1}{x}$-balanced hyperedge cut $Q \subset E$. Since all hyperedges which contain at least one vertex in both V_1 and V_2 must be part of the cut Q, all vertices are either disconnected by the cut or remain in hyperedges which are all internal to either V_1 or V_2. Let $U_1 \subset V_1$ be the vertices contained in a hyperedge internal to V_1 and let $U_2 \subset V_2$ be the vertices contained in a hyperedge internal to V_2. Since both V_1 and V_2 contain at least $\lfloor |V|/q \rfloor$ vertices, either $\lfloor |V|/(2q) \rfloor$ vertices must be in internal hyperedges within both V_1 as well as V_2, that is,

$$\text{case (i): } |U_1| \geq \lfloor |V|/(2q) \rfloor \text{ and } |U_2| \geq \lfloor |V|/(2q) \rfloor,$$

or there must be $\lfloor |V|/(2q) \rfloor$ vertices that are disconnected by the cut,

$$\text{case (ii): } |(V_1 \setminus U_1) \cup (V_2 \setminus U_2)| \geq \lfloor |V|/(2q) \rfloor.$$

First we note that when $r \leq \lfloor m/2 \rfloor$, every hyperedge intersects with every other hyperedge at some vertex since for any two r-tuples (hyperedges), the union of their entries has no more than m entries and so must correspond to at least one vertex. Since we assume $q < n$, case (i) implies that U_1 and U_2 each have an internal hyperedge. This is not possible for $r \leq \lfloor m/2 \rfloor$, since any pair of hyperedges intersects at some vertex. We note that this precludes case (i) for $r = 1, m = 2$ (the most basic version of the (m, r)-lattice hypergraph) and furthermore it precludes any scenario with $r = 1$.

In case (i), for $r \geq 2$ and $r > m/2$ (the other cases were ruled out in the previous paragraph), we start by assuming without loss of generality that $|U_1| \leq |U_2|$, therefore $|U_1| \leq \lfloor |V|/x \rfloor \leq \lfloor |V|/2 \rfloor$. We consider the hyperedges W_1 internal to U_1. Each vertex may appear in at most $\binom{m}{r}$ hyperedges and each hyperedge has $\binom{n-r}{m-r}$ vertices, so the number of hyperedges is at most

$$|W_1| \leq \frac{\binom{m}{r}|U_1|}{\binom{n-r}{m-r}} \leq \frac{\binom{m}{r}\binom{n}{m}}{2\binom{n-r}{m-r}} = \frac{1}{2}\binom{n}{r}. \tag{3.3.1}$$

Since both U_1 and U_2 have at least $\lfloor |V|/(2q) \rfloor$ vertices and each hyperedge has $\binom{n-r}{m-r}$ vertices, the number of hyperedges is at least

$$|W_1| \geq \frac{|U_1|}{\binom{n-r}{m-r}} \geq \left\lfloor \frac{\binom{n}{m}}{2q} \right\rfloor \cdot \binom{n-r}{m-r}^{-1} \geq \left\lfloor \frac{\binom{n}{r}}{2q\binom{m}{r}} \right\rfloor. \tag{3.3.2}$$

We define a **fiber** $f_{k_1,\ldots,k_{m-1}}$ for each $\{k_1, \ldots, k_{m-1}\} \in [1, n]^{m-1}$ with $k_1 < \cdots < k_{m-1}$ as the set of vertices in the hypergraph H which satisfy $\{k_1, \ldots, k_{m-1}\} \subset \{j_1, \ldots, j_m\}$. We note that each fiber is internal to at least one hyperedge (which is defined by $r \leq m - 1$ indices). We define a **hyperplane** $x_{k_1,\ldots,k_{r-1}}$ for each $\{k_1, \ldots, k_{r-1}\} \in [1, n]^{r-1}$ with $k_1 < \cdots < k_{r-1}$ as the set of all hyperedges $e_{j_1,\ldots,j_r}$ which satisfy $\{k_1, \ldots, k_{r-1}\} \subset \{j_1, \ldots, j_r\}$. Thus, each of the $|X| = \binom{n}{r-1}$ hyperplanes contains $n - (r - 1)$ hyperedges, and each hyperedge is in r hyperplanes. Note that each hyperplane shares a unique hyperedge with $(r - 1)(n - (r - 1))$ other hyperplanes.

We now obtain a bound on the number of hyperplanes to which the hyperedges W_1 are adjacent. The number of fibers internal to each hyperedge is $\binom{n-r}{m-r-1}$ (all ways to choose the other $m-r-1$ indices of the $m - 1$ indices in the fiber, r of which come from the hyperedge). We denote the fibers internal to W_1 as F_1, and determine the number of hyperplanes adjacent to these fibers. The total number of fibers in the set W_1 is then $|F_1| = \binom{n-r}{m-r-1}|W_1|$. We now apply the generalized modified Loomis-Whitney inequality (Theorem 3.2.3) for $\binom{m-1}{r-1}$ projections of a set of $(m - 1)$-tuples (fibers) onto $(r - 1)$-tuples (hyperplanes),

$$\pi_{s_1,\ldots,s_{r-1}}(e_{i_1,\ldots,i_{m-1}}) = (i_{s_1}, \ldots, i_{s_{r-1}}),$$

for $0 < s_1 < s_2 < \ldots < s_{r-1} < m$ corresponding to each of $\binom{m-1}{r-1}$ hyperplanes adjacent to each fiber. The inequality (Theorem 3.2.3) yields the following lower bound on the product of the size of the projections with respect to the union of the projections $Z_1 = \cup_j \pi_j(F_1)$, that is,

$$|F_1| \leq \Big((r-1)! \cdot |Z_1|\Big)^{(m-1)/(r-1)} / (m-1)!.$$

The cardinality of the union of the projections, $|Z_1|$ is the number of unique hyperplanes adjacent to F_1, which we seek to lower bound, so we reverse the above inequality,

$$|Z_1| \geq ((m-1)! \cdot |F_1|)^{(r-1)/(m-1)} / (r-1)! \equiv L(|F_1|). \tag{3.3.3}$$

We now argue that when $|W_1|$ (and therefore $|F_1|$) is at its maximum, $|F_1| = w \equiv \frac{1}{2}\binom{n-r}{m-r-1}\binom{n}{r}$ from equation 3.3.1 and the fact that $|F_1| = \binom{n-r}{m-r-1}|W_1|$, the right side of the inequality equation 3.3.3 becomes $L(w) \geq \frac{1}{2^{(r-1)/(m-1)}}\binom{n}{r-1}$ (with the assumption that $n \gg m, r$) since

$$\begin{aligned}
L(w) &= ((m-1)! \cdot w)^{(r-1)/(m-1)} / (r-1)! \\
&= \left((m-1)! \cdot \frac{1}{2}\binom{n-r}{m-r-1}\binom{n}{r}\right)^{(r-1)/(m-1)} / (r-1)! \\
&\geq \left(\frac{1}{2}n!/(n-(m-1))!\right)^{(r-1)/(m-1)} / (r-1)! \\
&= \frac{1}{2^{(r-1)/(m-1)}} \left(n!/(n-(m-1))!\right)^{(r-1)/(m-1)} / (r-1)!
\end{aligned}$$

we now note that $k = n!/(n-(m-1))!$ is a product of $m-1$ terms the largest $m-r$ of are the product $n!/(n-(m-r))!$. This implies that we can lower bound $k^{(r-1)/(m-1)}$ as the product of the lowest $r-1$ terms, $k^{(r-1)/(m-1)} \geq (n-(m-r))!/(n-(m-1))!$. This implies

$$\begin{aligned}
L(w) &= \frac{1}{2^{(r-1)/(m-1)}} k^{(r-1)/(m-1)} / (r-1)! \\
&\geq \frac{1}{2^{(r-1)/(m-1)}} [(n-(m-r))!/(n-(m-1))!] / (r-1)! \\
&= \frac{1}{2^{(r-1)/(m-1)}} \binom{n-(m-r)}{r-1}.
\end{aligned}$$

Applying Pascal's rule repeatedly we lower bound the binomial coefficient as,

$$\begin{aligned}
\binom{n-(m-r)}{r-1} &= \binom{n}{r-1} - \binom{n-1}{r-2} - \binom{n-2}{r-2} - \ldots - \binom{n-(m-r)}{r-2} \\
&\geq (1-(m-r)(r-1)/n) \cdot \binom{n}{r-1}
\end{aligned}$$

Plugging this back into $L(w)$ we obtain,

$$L(w) \geq \frac{1}{2^{(r-1)/(m-1)}} \left[\left(1 - \frac{(m-r)(r-1)}{n}\right)\binom{n}{r-1}\right].$$

Now applying our assumption that $n \gg m, r$, we approximate the inequality

$$L(w) \geq \frac{1}{2^{(r-1)/(m-1)}} \binom{n}{r-1}.$$

Now, the number of hyperedges R_1 contained inside the hyperplanes Z_1 is at least

$$|R_1| \geq \frac{(n-(r-1))|Z_1|}{r},$$

since each hyperplane contains $n-(r-1)$ hyperedges and each hyperedge can appear in at most r hyperplanes. Noting that each hyperedge is adjacent to every other hyperedge within any of its hyperplanes, we can conclude that Z_1 contains either hyperedges that are in W_1 or hyperedges that are cut (it cannot contain hyperedges that are internal to V_2). Therefore, we can lower bound the number of hyperedges in the cut as

$$\begin{aligned} |Q| &\geq |R_1| - |W_1| \geq \frac{(n-(r-1))|Z_1|}{r} - |W_1| \\ &\equiv T_1 - T_2. \end{aligned}$$

We would like to show that $T_1/T_2 > 1$ and drop $T_2 = |W_1|$. To do so, we first recall our $|F_1|$-dependent bound of $|Z_1|$ in equation 3.3.3 and simplify its form (to what the generalized Loomis-Whitney inequality would yield, which is weaker),

$$\begin{aligned} |Z_1| &\geq \left((m-1)! \cdot |F_1|\right)^{(r-1)/(m-1)} /(r-1)! \\ &\geq |F_1|^{(r-1)/(m-1)}. \end{aligned}$$

Since $T_1 = \frac{n-(r-1)}{r}|Z_1|$ has a fractional (less than one) power of $|F_1|$ while $T_2 = |W_1|$, the ratio of T_1/T_2 is minimized when $|W_1|$ is maximized, namely when $|W_1| = \frac{1}{2}\binom{n}{r}$. In this case, we have shown that $|Z_1| \geq L(w) \geq \frac{1}{2^{(r-1)/(m-1)}}\binom{n-r}{m-r-1}\binom{n}{r-1}$ (with the assumption that $n \gg m, r$), so we see that the ratio is at least

$$\begin{aligned} T_1/T_2 &= \frac{(n-(r-1))|Z_1|}{r} \cdot \frac{1}{|W_1|} \\ &\geq \frac{(n-(r-1))\frac{1}{2^{(r-1)/(m-1)}}\binom{n}{r-1}}{r} \cdot \frac{1}{\frac{1}{2}\binom{n}{r}} \\ &= \frac{\binom{n}{r}}{2^{(r-1)/(m-1)}} \cdot \frac{2}{\binom{n}{r}} \\ &= 2^{1-(r-1)/(m-1)} > 1. \end{aligned}$$

This ratio is greater than one by a small factor. Our approximation on $L(w)$ which used $n \gg m, r$ implies that the ratio is greater than one in the limit when $n \gg m, r$ and both m and r are small constants (in the applications of our algorithms m never exceeds 4). This lower bound on the ratio implies that

$$\begin{aligned} |Q| \geq T_1 - T_2 &\geq \left(1 - \frac{1}{2^{1-(r-1)/(m-1)}}\right)T_1 \\ &= \left(1 - \frac{1}{2^{1-(r-1)/(m-1)}}\right)\frac{(n-(r-1))|Z_1|}{r}. \end{aligned}$$

Remembering that we have $|Z_1| \geq |F_1|^{(r-1)/(m-1)}$ and that $|F_1| = \binom{n-r}{m-r-1}|W_1| \geq \binom{n-r}{m-r-1}\left\lfloor \frac{\binom{n}{r}}{2q\binom{m}{r}} \right\rfloor$ from equation 3.3.2, we obtain

$$\begin{aligned}
|Q| &\geq \left(1 - \frac{1}{2^{1-(r-1)/(m-1)}}\right)\frac{(n-(r-1))|Z_1|}{r} \\
&\geq \left(1 - \frac{1}{2^{1-(r-1)/(m-1)}}\right)\frac{(n-(r-1))\left[\binom{n-r}{m-r-1}\left\lfloor \frac{\binom{n}{r}}{2q\binom{m}{r}} \right\rfloor\right]^{(r-1)/(m-1)}}{r} \\
&= \Omega\left(n \cdot \left[n^{m-r-1} \cdot \frac{n^r}{q}\right]^{(r-1)/(m-1)}\right) = \Omega(n^r/q^{(r-1)/(m-1)}),
\end{aligned}$$

which is greater (and so better) than the desired bound $|Q| = \Omega(n^r/q^{r/m})$ since $r/m > (r-1)/(m-1)$.

In case (ii), we know that $\lfloor |V|/(2q) \rfloor = \lfloor \binom{n}{m}/(2q) \rfloor$ vertices $\bar{U} \subset V$ are disconnected by the cut (before the cut, every vertex was adjacent to d hyperedges). We define $\binom{m}{r}$ projections,

$$\pi_{s_1,\ldots,s_r}(e_{i_1,\ldots,i_m}) = (i_{s_1}, \ldots, i_{s_r}),$$

for $0 < s_1 < s_2 < \ldots < s_r \leq m$ corresponding to each of $\binom{m}{r}$ hyperedges adjacent to $v_{i_1,\ldots,i_m}$. In particular the set of hyperedges adjacent to $\bar{U}$ is

$$\bar{\Pi} = \bigcup_{1 \leq s_1 < \ldots < s_r \leq m} \pi_{s_1,\ldots,s_r}(\bar{U}).$$

We apply the generalized Loomis-Whitney inequality (Theorem 3.2.3) to obtain a lower bound on the the size of the projections $|\bar{\Pi}|$,

$$\begin{aligned}
|\bar{U}| &\leq \left(r! \cdot |\bar{\Pi}|\right)^{m/r}/m! \\
|\bar{\Pi}| &\geq (m! \cdot |\bar{U}|)^{r/m}/r!
\end{aligned}$$

Plugging in the lower bound on $|\bar{U}|$ corresponding to case (ii) and discarding asymptotically small factors, we obtain the desired lower bound on the minimum number of hyperedges that must be cut,

$$\begin{aligned}
\epsilon_q(H) &\geq \left(m! \cdot \left\lfloor \binom{n}{m}/(2q) \right\rfloor\right)^{r/m}/r! \\
&= \Omega\left(n^r/q^{r/m}\right).
\end{aligned}$$

□

3.3.2 Parent Hypergraphs

In order to apply our lower bounds to applications, we consider how a given hypergraph can be transformed without increasing the size of the minimum hyperedge cut. We employ the idea that it is possible to merge sets of hyperedges which define connected components without increasing the size of any cut. Additionally, we employ the fact that discarding vertices in a hypergraph cannot increase the size of the minimum hyperedge cut that is balanced. Given a hypergraph $H = (V, E)$, we can define a **parent hypergraph** $\hat{H} = (\hat{V}, \hat{E})$, where $\hat{V} \subset V$, and each hyperedge in $\hat{E}$ is associated with an element of R, a collection of sub-hypergraphs of H induced by a partition of E. In particular, each hyperedge $e \in \hat{E}$ is associated with a unique sub-hypergraph $(W, Y) \in R$ that corresponds to the subset of hyperedges $Y \subset E$ of the original graph that are adjacent to $W \subset V$, such that $e \subset W$ and the sub-hypergraph (W, Y) is connected. We derive parent hypergraphs for directed graphs by treating the directed edges as if they were undirected, which means we use the notion of weak connectivity in directed graphs. This is consistent with our definitions of vertex separators, which do not allow edges in any direction between the two disconnected parts V_1 and V_2, meaning $E \cap (V_1 \times V_2) = \emptyset$.

We obtain lower bounds on the vertex separator size of a directed or undirected graph by constructing a parent hypergraph in which every vertex is adjacent to a constant number of hyperedges. The following theorem states this result for vertex separators of (undirected) hypergraphs, which immediately generalizes the case of undirected and directed graphs.

Theorem 3.3.2. *Given a hypergraph $H = (V, E)$, consider any parent hypergraph $\hat{H} = (\hat{V}, \hat{E})$ defined by a collection of sub-hypergraphs R as defined above. If the degree of each vertex in $\hat{H}$ is at most k, the minimum $\frac{1}{q}$-$\frac{1}{x}$-balanced hyperedge cut of $\hat{H}$, where $q \geq x \geq 4$, is no larger than k times the minimum size of a $\frac{1}{q}$-$\frac{1}{x}$-balanced vertex separator of $\hat{V}$ in hypergraph H.*

Proof. Consider any $\frac{1}{q}$-$\frac{1}{x}$-balanced vertex separator $S \subset V$ of $\hat{V}$ inside H, for $q \geq x \geq 2$, that yields two disjoint vertex sets $\hat{V}_1$ and $\hat{V}_2$. We construct a hyperedge cut $\hat{X}$ of $\hat{H}$ consisting of all hyperedges corresponding to parts (elements of R) in which a vertex in the separator S appears. $\hat{X}$ is a cut of $\hat{H}$ since for any path consisting of hyperedges $\mathcal{P} = \{y_1, y_2, \ldots\}$ in $\hat{H}$ corresponding to parts $\{r_1, r_2, \ldots\}$, there exists a path in H consisting of hyperedges $\mathcal{Q} = \{e_{11}, e_{12}, \ldots\} \cup \{e_{21}, e_{22}, \ldots\} \cup \cdots$, where for each i, $\{e_{i1}, e_{i2}, \ldots\} \subset r_i$. Therefore, since S disconnects every path through H between $\hat{V}_1$ and $\hat{V}_2$, the cut $\hat{X}$ must disconnect all such paths also. Assuming without loss of generality that $|\hat{V}_1| \leq |\hat{V}_2|$, we let the vertex parts defined by the cut $\hat{X}$ of $\hat{H}$ be $\hat{V}_1 \cup S$ and $\hat{V}_2$, since $\hat{X}$ disconnects all hyperedges connected to S, so S can be included in either part. Making this choice, we ensure that the size of the smaller part

$$|\hat{V}_1 \cup S| = |\hat{V}_1| + |S| \geq \lfloor |\hat{V}|/q \rfloor - |S| + |S| \geq \lfloor |\hat{V}|/q \rfloor,$$

which ensures the balance of the partition created by the cut $\hat{X}$. We can assert that $|\hat{V}_1 \cup S| \leq |\hat{V}_2|$, since we have assumed $x \geq 4$, which implies that $|\hat{V}_1| \leq \lfloor |\hat{V}|/4 \rfloor$ and $|S| \leq \lfloor |\hat{V}|/4 \rfloor$. The bound on separator size, $|S| \leq \lfloor |\hat{V}|/4 \rfloor$, holds because any quarter of the vertices would lead to a valid vertex separator of any H by choosing S to be any subset of $\lfloor |\hat{V}|/4 \rfloor$ vertices and $\hat{V}_1 = \emptyset$. In this

case, removing the hyperedges in Q leaves $S \cup \hat{V}_1 = S$ disconnected. Further, since the maximum degree of any vertex in $\hat{H}$ is k and each hyperedge in H appears in at most one part in R, which corresponds to a unique hyperedge in $\hat{H}$, the cut $\hat{X}$ of $\hat{H}$ is of size at most $k \cdot |S|$. □

3.4 Lower Bounds on Cost Tradeoffs Based on Dependency Path Expansion

In this section, we introduce the concept of dependency bubbles and their expansion. Bubbles represent sets of interdependent computations, and their expansion allows us to analyze the cost of computation and communication for any parallelization and communication schedule. We will show that if a dependency graph has a path along which bubbles expand as some function of the length of the path, any parallelization of this dependency graph must sacrifice synchronization or, alternatively, incur higher computational and data volume costs, which scale with the total size and cross-section size (minimum cut size) of the bubbles, respectively.

3.4.1 Bubble Expansion

Given a directed graph $G = (V, E)$, we say that for any pair of vertices $v_1, v_n \in V$, v_n **depends on** v_1 if and only if there is a path $\mathcal{P} \subset V$ connecting v_1 to v_n, i.e., $\mathcal{P} = \{v_1, \ldots, v_n\}$ such that $\{(v_1, v_2), \ldots, (v_{n-1}, v_n)\} \subset E$. We denote a sequence of (not necessarily adjacent) vertices $\{w_1, \ldots, w_n\}$ a **dependency path**, if for $i \in [1, n-1]$, w_{i+1} depends on w_i. Any consecutive subsequence of a dependency path is again a dependency path, called simply a **subpath**, which context will disambiguate from its usual definition.

The (**dependency**) **bubble** around a dependency path $\mathcal{P}$ connecting v_1 to v_n is a subgraph $\zeta(G, \mathcal{P}) = (V_\zeta, E_\zeta)$ where $V_\zeta \subset V$, each vertex $u \in V_\zeta$ lies on a dependency path $\{v_1, \ldots, u, \ldots, v_n\}$ in G, and $E_\zeta = (V_\zeta \times V_\zeta) \cap E$. If G is a dependency graph, this bubble corresponds to all vertices which must be computed between the start and end of the path. Equivalently, the bubble may be defined as the union of all paths between v_1 and v_n.

3.4.2 Lower Bounds Based on Bubble Expansion

For any $q \geq x \geq 2$, a $\frac{1}{q}$-$\frac{1}{x}$**-balanced vertex separator** of $\hat{V} \subset V$ in a directed graph $G = (V, E)$ (if $\hat{V}$ is unspecified then $\hat{V} = V$) is a set of vertices $Q \subset V$, which splits the vertices V in G into two disconnected partitions, $V \setminus Q = V_1 \cup V_2$ so that $E \subset (V_1 \times V_1) \cup (V_2 \times V_2) \cup (Q \times V) \cup (V \times Q)$. Further, the subsets of these vertices inside $\hat{V}$ must be balanced, meaning that for $\hat{V}_1 = \hat{V} \cap V_1$ and $\hat{V}_2 = \hat{V} \cap V_2$, $\lfloor |\hat{V}|/q \rfloor - |Q| \leq \min(|\hat{V}_1|, |\hat{V}_2|) \leq \lfloor |\hat{V}|/x \rfloor$. We denote the minimum size $|Q|$ of a $\frac{1}{q}$-$\frac{1}{x}$-balanced vertex separator Q of G as $\chi_{q,x}(G)$. The lower bound, $\lfloor |\hat{V}|/q \rfloor - |Q|$, can be zero, e.g., when $|Q| = \lfloor |\hat{V}|/x \rfloor$, but cannot be negative, since any $Q' \subset \hat{V}$ of size $\lfloor |\hat{V}|/q \rfloor$ is a $\frac{1}{q}$-$\frac{1}{x}$-balanced vertex separator with $V_1 = \hat{V}_1 = \emptyset$, so $\chi_{q,x}(G) \leq \lfloor |\hat{V}|/q \rfloor$. For most graphs we

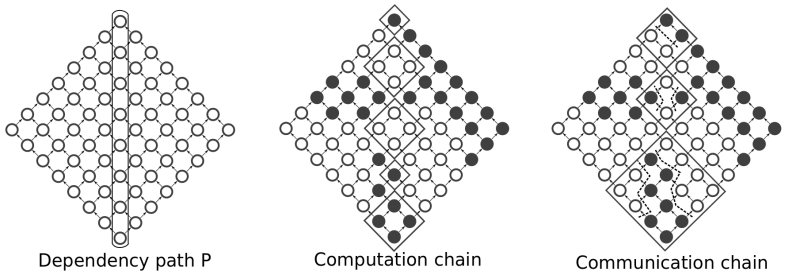


Figure 3.1: Illustration of the construction in the proof of Theorem 3.4.1 in the case of a diamond DAG (e.g., [124]), depicting a dependency path, and communication and computation chains about that path, for a 2-processor parallelization.

will consider, $\chi_{q,x}(G) < \lfloor |\hat{V}|/q \rfloor$. If $\zeta(G, \mathcal{P})$ is the bubble around dependency path $\mathcal{P}$, we say $\chi_{q,x}(\zeta(G, \mathcal{P}))$ is its **cross-section expansion**.

We now introduce the notion of a (ϵ, σ)**-path-expander** graph, which is the key characterization of dependency graphs to which our analysis is applicable. In inexact terms, such a graph needs to have a dependency path such that any bubble around a subpath $\mathcal{R}$ of this path is of size $|\zeta(G, \mathcal{R})| = \Theta(\sigma(|\mathcal{R}|))$ and has cross-section size $\chi_{q,x}(\zeta(G, \mathcal{R})) = \Omega(\epsilon(|\mathcal{R}|))$ for some q, x. Now we define this notion more precisely. We call a directed graph G a (ϵ, σ)**-path-expander** if there exists a dependency path $\mathcal{P}$ in G, where $\zeta(G, \mathcal{P}) = G$, and positive real constants $k, c_\epsilon, c_\sigma, \hat{c}_\sigma, q, x \ll |\mathcal{P}|$, where $k \in \mathbb{Z}$, $c_\epsilon, c_\sigma > 1$, $x \geq 2$, and $q = \max(x\hat{c}_\sigma^2 c_\sigma + 1, \hat{c}_\sigma \cdot \sigma(k))$, such that

- every subpath $\mathcal{R} \subset \mathcal{P}$ of length $|\mathcal{R}| \geq k$ has bubble $\zeta(G, \mathcal{R}) = (V_\zeta, E_\zeta)$
  1. with size $|V_\zeta| = \Theta(\sigma(|\mathcal{R}|)$ and further $\sigma(|\mathcal{R}|)/\hat{c}_\sigma \leq |V_\zeta| \leq \hat{c}_\sigma \cdot \sigma(|\mathcal{R}|)$,
  2. with cross-section expansion $\chi_{q,x}(\zeta(G, \mathcal{R})) = \Omega(\epsilon(|\mathcal{R}|))$,
- where the given real-valued functions ϵ, σ are
  1. positive and convex,
  2. increasing with $\epsilon(b+1) \leq c_\epsilon \epsilon(b)$ and $\sigma(b+1) \leq c_\sigma \sigma(b)$ for all real numbers $b \geq k$.

Theorem 3.4.1 (General Bubble Lower Bounds)**.** *Suppose a dependency graph G is a (ϵ, σ)-path-expander about dependency path $\mathcal{P}$. Then, for any schedule of G corresponding to a parallelization (of G) in which no processor computes more than $\frac{1}{x}$ of the vertices of $\zeta(G, \mathcal{P})$, there exists an integer $b \in [k, |\mathcal{P}|]$ such that the computation (F), bandwidth (W), and latency (S) costs incurred are*

$$F = \Omega\left(\sigma(b) \cdot |\mathcal{P}|/b\right), \quad W = \Omega\left(\epsilon(b) \cdot |\mathcal{P}|/b\right), \quad S = \Omega\left(|\mathcal{P}|/b\right).$$

Proof. We consider any possible parallelization, which implies a coloring of the vertices of $G = (V, E)$, $V = \bigcup_{i=1}^{p} C_i$, and show that any schedule $\bar{G} = (\bar{V}, \bar{E})$, as defined in Section 2.1, incurs

the desired computation, bandwidth, and latency costs. Our proof technique works by defining a chain of bubbles within G in a way that allows us to accumulate the costs along the chain.

The tradeoff between work and synchronization, $F = \Omega(\sigma(b) \cdot |\mathcal{P}|/b)$ and $S = \Omega(|\mathcal{P}|/b)$, can be derived by considering a computation chain: a sequence of monochrome bubbles along $\mathcal{P}$, each corresponding to a set of computations performed sequentially by some processor (see computation chain in Figure 3.1). However, to obtain the bandwidth lower bound, we must instead show that there exists a sequence of bubbles in which some processor computes a constant fraction of each bubble; we then sum the bandwidth costs incurred by each bubble in the sequence. We show a communication chain (a sequence of multicolored bubbles) for a diamond DAG in Figure 3.1.

By hypothesis, there exists $k \ll |\mathcal{P}|$ such that every subpath $\mathcal{R} \subset \mathcal{P}$ of length $|\mathcal{R}| \geq k$ induces a bubble $\zeta(G, \mathcal{R}) = (V_\zeta, E_\zeta)$ of size

$$\sigma(|\mathcal{R}|)/\hat{c}_\sigma \leq |V_\zeta| \leq \hat{c}_\sigma \cdot \sigma(|\mathcal{R}|),$$

for all $|\mathcal{R}| \geq k$.

We define the bubbles via the following procedure, which partitions the dependency path $\mathcal{P}$ into subpaths by iteratively removing leading subpaths. In this manner, assume we have defined subpaths $\mathcal{R}_j$ for $j \in [1, i-1]$. Let the tail (remaining trailing subpath) of the original dependency path be

$$\mathcal{T}_i = \mathcal{P} \setminus \bigcup_{j=1}^{i-1} \mathcal{R}_j = \left\{t_1, \ldots, t_{|\mathcal{T}_i|}\right\}.$$

Our procedure defines the next leading subpath of length r, $\mathcal{R}_i = \{t_1, \ldots, t_r\}, k \leq r \leq |\mathcal{T}_i|$, with bubble $\zeta(G, \mathcal{R}_i) = (V_i, E_i)$. Suppose processor l computes t_1. The procedure picks the shortest leading subpath of $\mathcal{T}_i$ of length $r \geq k$ which satisfies the following two conditions, and terminates if no such path can be defined.

Condition 1: The subset of the bubble that processor l computes, $C_l \cap V_i$, is of size

$$|C_l \cap V_i| \geq \lfloor |V_i|/q \rfloor.$$

Condition 2: The subset of the bubble that processor l does not compute, $V_i \setminus C_l$, is of size

$$|V_i \setminus C_l| \geq |V_i| - \lfloor |V_i|/x \rfloor.$$

Let c be the number of subpaths the procedure outputs and $\mathcal{T} = \mathcal{P} \setminus \bigcup_{j=1}^{c} \mathcal{R}_j = \{t_1, \ldots, t_{|\mathcal{T}|}\}$ be the tail remaining at the end of the procedure. We consider the two cases: $|\mathcal{T}| \geq |\mathcal{P}|/2$ and $\sum_j |\mathcal{R}_j| > |\mathcal{P}|/2$, at least one of which must hold. We show that in either case, the theorem is true for some value of b.

Case (i): If $|\mathcal{T}| \geq |\mathcal{P}|/2$ (the tail is long), we show that Condition 1 must be satisfied for any leading subpath of $\mathcal{T}$. Our proof works by induction on the length of the leading subpath $k \leq r \leq |\mathcal{T}|$, with the subpath given by $\mathcal{K}_r = \{t_1, \ldots, t_r\}$. We define the bubble about $\mathcal{K}_r$ as

$\zeta(G, \mathcal{K}_r) = (V_r, E_r)$. When $r = k$, Condition 1 is satisfied because $|V_r|/q \leq \hat{c}_\sigma \cdot \sigma(k)/q \leq 1$ and processor l computes at least one element, t_1.

For $r > k$, we have

$$|V_r| \leq \hat{c}_\sigma \cdot \sigma(r) \leq \hat{c}_\sigma c_\sigma \cdot \sigma(r-1) \leq \frac{(q-1)}{x\hat{c}_\sigma} \cdot \sigma(r-1).$$

Further, by induction, Condition 1 was satisfied for $\mathcal{K}_{r-1}$ which implies Condition 2 was not satisfied for $\mathcal{K}_{r-1}$ (otherwise the procedure would have terminated with a subpath of length $r-1$). Now, using bounds on bubble growth, we show that since Condition 2 was not satisfied for $\mathcal{K}_{r-1}$, Condition 1 has to be satisfied for the subsequent bubble, $\mathcal{K}_r$,

$$|C_l \cap V_r| \geq |C_l \cap V_{r-1}| \geq |V_{r-1}| - (|V_{r-1}| - \lfloor |V_{r-1}|/x \rfloor) = \lfloor |V_{r-1}|/x \rfloor;$$

applying the lower bound on bubble size $|V_{r-1}| \geq \sigma(r-1)/\hat{c}_\sigma$,

$$|C_l \cap V_r| \geq \lfloor \sigma(r-1)/(x\hat{c}_\sigma) \rfloor;$$

and applying the bound $|V_r| \leq \sigma(r-1)\cdot(q-1)/(\hat{c}_\sigma x)$, which implies $\sigma(r-1) \geq |V_r|\cdot(x\hat{c}_\sigma)/(q-1)$, we obtain

$$|C_l \cap V_r| \geq \lfloor (|V_r| \cdot (x\hat{c}_\sigma)/(q-1))/(x\hat{c}_\sigma) \rfloor = \lfloor |V_r|/(q-1) \rfloor \geq \lfloor |V_r|/q \rfloor,$$

so Condition 1 holds for $\mathcal{K}_r$ for $r \in [k, |\mathcal{T}|]$. Due to Condition 1, processor l must compute $F \geq \left\lfloor |V_{\zeta(G,\mathcal{T})}|/q \right\rfloor = \Omega(\sigma(|\mathcal{T}|))$ vertices. Since, by assumption, no processor can compute more than $\frac{1}{x}$ of the vertices of $\zeta(G, \mathcal{P})$, we claim there exists a subpath $\mathcal{Q}$ of $\mathcal{P}$, $\mathcal{T} \subset \mathcal{Q} \subset \mathcal{P}$, where processor l computes $\left\lfloor |V_{\zeta(G,\mathcal{Q})}|/q \right\rfloor$ vertices (Condition 1) and does not compute $|V_{\zeta(G,\mathcal{Q})}| - \left\lfloor |V_{\zeta(G,\mathcal{Q})}|/x \right\rfloor$ vertices (Condition 2). The subpath $\mathcal{Q}$ may always be found to satisfy these two conditions simultaneously, since we can grow $\mathcal{Q}$ backward from $\mathcal{T}$ until Condition 2 is satisfied, i.e., processor l does not compute at least $|V_{\zeta(G,\mathcal{Q})}| - \left\lfloor |V_{\zeta(G,\mathcal{Q})}|/x \right\rfloor$ vertices, which must occur since we have assumed that no processor computes more than $\lfloor |V|/x \rfloor = \left\lfloor |V_{\zeta(G,\mathcal{P})}|/x \right\rfloor$ vertices. Further, we will not violate the first condition that $|C_l \cap V_{\zeta(G,\mathcal{Q})}| \geq \lfloor |V_{\zeta(G,\mathcal{Q})}|/q \rfloor$, which holds for $\mathcal{Q} = \mathcal{T}$, due to bounds on growth of $|V_{\zeta(G,\mathcal{Q})}|$. The proof of this assertion is the same as the inductive proof above which showed that Condition 1 holds on $\mathcal{K}_r$, since the bubble growth bounds are invariant to the orientation of the path we are growing. So, the size of processor l's communicated set is at least the size of a $\frac{1}{q}$-$\frac{1}{x}$-balanced vertex separator of $\zeta(G, \mathcal{Q})$, namely $\chi_{q,x}(\zeta(G, \mathcal{Q})) = \Omega(\epsilon(|\mathcal{Q}|))$, since the communicated set

$$R = \left[u : (u, w) \in \left(\left(V_{\zeta(G,\mathcal{Q})} \cap C_l\right) \times \left(V_{\zeta(G,\mathcal{Q})} \setminus C_l\right)\right) \cup \left(\left(V_{\zeta(G,\mathcal{Q})} \setminus C_l\right) \times \left(V_{\zeta(G,\mathcal{Q})} \cap C_l\right)\right) \cap E\right]$$

separates two partitions, one of size at least

$$|(V_{\zeta(G,\mathcal{Q})} \cap C_l) \setminus R| \geq \lfloor |V_{\zeta(G,\mathcal{Q})}|/q \rfloor - |R|$$

due to Condition 1 and at most

$$|V_{\zeta(G,\mathcal{Q})} \cap C_l| \leq \lfloor |V_{\zeta(G,\mathcal{Q})}|/x \rfloor$$

due to Condition 2 on $\mathcal{Q}$, as well as another partition of size at least

$$|(V_{\zeta(G,\mathcal{Q})} \setminus C_l) \setminus R| \geq |V_{\zeta(G,\mathcal{Q})}| - \lfloor |V_{\zeta(G,\mathcal{Q})}|/x \rfloor - |R| \geq \lfloor |V_{\zeta(G,\mathcal{Q})}|/x \rfloor - |R| \geq \lfloor |V_{\zeta(G,\mathcal{Q})}|/q \rfloor - |R|$$

due to Condition 2 on $\mathcal{Q}$ and the fact that $q \geq x \geq 2$. Applying the assumed lower bound on the $\frac{1}{q}$-$\frac{1}{x}$-balanced vertex separator of $\zeta(G, \mathcal{Q})$, we obtain a lower bound on the size of the communicated set R, which is also a lower bound on interprocessor communication cost, $W = \Omega(\epsilon(|\mathcal{Q}|)) = \Omega(\epsilon(|\mathcal{T}|))$ Since these costs are incurred along a path in the schedule consisting of the work and communication done only by processor l, the bounds hold for $b = |\mathcal{T}|$; note that $\Omega(|\mathcal{P}|/|\mathcal{T}|) = \Omega(1)$, because (since in this case the tail is long) $|\mathcal{T}| \geq |\mathcal{P}|/2$.

Case (ii): $\sum_j |\mathcal{R}_j| > |\mathcal{P}|/2$ (the tail is short), the procedure generates subpaths with a total size proportional to the size of $\mathcal{P}$. For each $i \in [1, c]$, consider

- the task $\bar{u} \in \bar{V}$ during which processor l computed the first vertex t_1 on the path $\mathcal{R}_i$, i.e., $t_1 \in \hat{f}(\bar{u})$,
- the task $\bar{v} \in \bar{V}$ during which processor l computed its last vertex within the ith bubble, so $\hat{f}(\bar{v}) \cap V_i \cap C_l \neq \emptyset$, and
- the task $\bar{w} \in \bar{V}$ during which the last vertex on the subpath $\mathcal{R}_i$, t_r, was computed, i.e., $t_r \subset \hat{f}(\bar{w})$.

Since t_r depends on all other vertices in V_i, there will be an execution path $\pi_i = \{\bar{u}, \ldots, \bar{v}, \ldots \bar{w}\} \subset \bar{V}$ in the schedule $\bar{G}$. Since $\hat{f}(\bar{u})$ contains the first vertex of $\mathcal{R}_i$, all communication necessary to satisfy the dependencies of processor l (dependencies of $V_i \cap C_l$) within bubble V_i must be incurred along π_i. This communicated set is given by

$$\hat{T}_{il} = \{u : (u, w) \in [((V_i \cap C_l) \times (V_i \setminus C_l)) \cup ((V_i \setminus C_l) \times (V_i \cap C_l))] \cap E\},$$

which is a separator of $\zeta(G, \mathcal{R}_i)$ and is $\frac{1}{q}$-$\frac{1}{x}$-balanced by the same argument as given above in case (i) for $|V_{\zeta(G,\mathcal{Q})}|$, since in both cases Conditions 1 and 2 both hold. Any separator of $\zeta(G, \mathcal{R}_i)$ must create two partitions, one of size at least

$$|(V_{\zeta(G,\mathcal{R}_i)} \cap C_l) \setminus T_{il}| \geq \lfloor |V_{\zeta(G,\mathcal{R}_i)}|/q \rfloor - |T_{il}|$$

due to Condition 1 and at most

$$|V_{\zeta(G,\mathcal{R}_i)} \cap C_l| \leq \lfloor |V_{\zeta(G,\mathcal{R}_i)}|/x \rfloor$$

due to Condition 2 on $\mathcal{R}_i$ as well as another partition of size at least

$$\begin{aligned} |(V_{\zeta(G,\mathcal{R}_i)} \setminus C_l) \setminus T_{il}| &\geq |V_{\zeta(G,\mathcal{R}_i)}| - \lfloor |V_{\zeta(G,\mathcal{R}_i)}|/x \rfloor - |T_{il}| \\ &\geq \lfloor |V_{\zeta(G,\mathcal{R}_i)}|/x \rfloor - |T_{il}| \geq \lfloor |V_{\zeta(G,\mathcal{R}_i)}|/q \rfloor - |T_{il}| \end{aligned}$$

due to Condition 2 on $\mathcal{R}_i$ and the fact that $q \geq x \geq 2$.

We use the lower bound on the minimum separator of a bubble to obtain a lower bound on the size of the communicated set for processor l in the ith bubble,

$$|\hat{T}_{il}| \geq \chi_{q,x}(G, \mathcal{R}_i) = \Omega\left(\epsilon(|\mathcal{R}_i|)\right),$$

where we are able to bound the cross-section expansion of $\zeta(G, \mathcal{R}_i)$, since $|\mathcal{R}_i| \geq k$. There exists a dependency path between the last element of $\mathcal{R}_i$ and the first of $\mathcal{R}_{i+1}$ since they are subpaths of $\mathcal{P}$, so every bubble $\zeta(G, \mathcal{R}_i)$ must be computed entirely before any members of $\zeta(G, \mathcal{R}_{i+1})$ are computed. Therefore, there is an execution path π_{critical} in the schedule $\bar{G}$ which contains $\pi_i \subset \pi_{\text{critical}}$ as a subpath for every $i \in [1, c]$. The execution cost of $\bar{G}$ is bounded below by the cost of π_{critical},

$$T(\bar{G}) \geq \sum_{\bar{v} \in \pi_{\text{critical}}} \hat{t}(\bar{v}) = \alpha \cdot S + \beta \cdot W + \gamma \cdot F,$$

which we can bound below component-wise,

$$\begin{aligned}
F &= \sum_{\bar{v} \in \pi_{\text{critical}} \cap \bar{V}_{\text{comp}}} |\hat{f}(\bar{v})| \geq \sum_{i=1}^{c} \frac{1}{q} |\zeta(G, \mathcal{R}_i)| = \Omega\left(\sum_{i=1}^{c} \sigma(|\mathcal{R}_i|)\right), \\
W &= \sum_{\bar{v} \in \pi_{\text{critical}} \cap \bar{V}_{\text{send}}} |\hat{s}(\bar{v})| \; + \sum_{\bar{v} \in \pi_{\text{critical}} \cap \bar{V}_{\text{recv}}} |\hat{r}(\bar{v})| \geq \sum_{i=1}^{c} \chi_{q,x}(\zeta(G, \mathcal{R}_i)) \\
&= \Omega\left(\sum_{i=1}^{c} \epsilon(|\mathcal{R}_i|)\right).
\end{aligned}$$

Further, since each bubble contains vertices computed by multiple processors, between the first and last vertex on the subpath forming each bubble, each π_i must go through at least one synchronization vertex, therefore, we also have a lower bound on latency cost,

$$S \geq \sum_{\bar{v} \in \pi_{\text{critical}} \cap \bar{V}_{\text{sync}}} 1 \geq c.$$

Because $|\mathcal{R}_i| \geq k$, σ as well as ϵ are assumed convex, and the sum of all the lengths of the subpaths is bounded, $\sum_i |\mathcal{R}_i| \leq |\mathcal{P}|$, the above lower bounds for F and W are minimized when all values $|\mathcal{R}_i|$ are equal[1]. Thus, we can replace $|\mathcal{R}_i|$ for each i by $b = \lfloor \sum_j |\mathcal{R}_j|/c \rfloor = \Theta(|\mathcal{P}|/c)$ (since the tail is short), simplifying the bounds to obtain the conclusion. □

Corollary 3.4.2 (d-dimensional bubble lower bounds)**.** *Suppose there exists a dependency path $\mathcal{P}$ in a dependency graph G and integer constants $2 \leq d \ll k \ll |\mathcal{P}|$ such that every subpath $\mathcal{R} \subset \mathcal{P}$ of length $|\mathcal{R}| \geq k$ has bubble $\zeta(G, \mathcal{R}) = (V_\zeta, E_\zeta)$ with cross-section expansion $\chi_{q,x}(\zeta(G, \mathcal{R})) = \Omega(|\mathcal{R}|^{d-1}/q^{(d-1)/d})$ for all real numbers $k \leq q \ll |\mathcal{P}|$, and has bubble size $|V_\zeta| = \Theta(|\mathcal{R}|^d)$.*

[1]This mathematical relation can be demonstrated by a basic application of convexity.

Then, the computation (F), bandwidth (W), and latency (S) costs incurred by any schedule of G corresponding to a parallelization (of G) in which no processor computes more than $\frac{1}{x}$ of the vertices of $\zeta(G, \mathcal{P})$ (for $2 \leq x \ll |\mathcal{P}|$) must be at least

$$F = \Omega\left(b^{d-1} \cdot |\mathcal{P}|\right), \quad W = \Omega\left(b^{d-2} \cdot |\mathcal{P}|\right), \quad S = \Omega\left(|\mathcal{P}|/b\right).$$

for some $b \in [k, |\mathcal{P}|]$, which implies the costs need to obey the following tradeoffs:

$$F \cdot S^{d-1} = \Omega\left(|\mathcal{P}|^d\right), \quad W \cdot S^{d-2} = \Omega\left(|\mathcal{P}|^{d-1}\right).$$

Proof. This is an application of Theorem 3.4.1 since $\zeta(G, \mathcal{P})$ is a (ϵ, σ)-path-expander graph with $\epsilon(b) = b^{d-1}$ and $\sigma(b) = b^d$. The particular constants defining the path-expander graph being k and x as given, as well as $c_\sigma = \frac{(k+1)^d}{k^d}$, $c_\epsilon = \frac{(k+1)^{d-1}}{k^{d-1}}$, and since $|V_\zeta| = \Theta(|\mathcal{R}|^d) = \Theta(\sigma(|\mathcal{R}|))$ there exists a constant $\hat{c}_\sigma$ such that $\sigma(|\mathcal{R}|)/\hat{c}_\sigma \leq |V_\zeta| \leq \hat{c}_\sigma \cdot \sigma(|\mathcal{R}|)$. These constants all make up q, which is $O(x\hat{c}_\sigma^2 k^d) = O(1)$, so we can disregard the factor of $q^{(d-1)/d}$ inside ϵ and obtain the desired lower bounds,

$$F = \Omega\left(b^{d-1} \cdot |\mathcal{P}|\right), \quad W = \Omega\left(b^{d-2} \cdot |\mathcal{P}|\right), \quad S = \Omega\left(|\mathcal{P}|/b\right).$$

These equations can be manipulated algebraically to obtain the conclusion. □

Chapter 4

Matrix Multiplication

Matrix multiplication is a model dense linear algebra problem and a key primitive in all other dense linear algebra algorithms that the later chapters will analyze. We review a number of matrix multiplication algorithms in Section 4.1. Some of the important for us will be Cannon's algorithm [32], the SUMMA algorithm [163, 1], and 3D matrix multiplication algorithms [43, 1, 2, 23, 93]. We do not consider Strassen-like matrix multiplication algorithms. Lower bounds on communication complexity of matrix multiplication were previously studied by Hong and Kung [90] and Irony et al. [88], as well as by [12]. We give a lower bound derivation, which provides a good constant factor for the scenarios we will consider in Section 4.2.

We will then in Section 4.3 give an algorithm for 2.5D matrix multiplication that is a memory-efficient adaptation of 3D matrix multiplication, similar to the algorithm by McColl and Tiskin [113]. Both of these algorithms attain the communication lower bound for any amount of available local memory. We also study the performance of an implementation of the 2.5D matrix multiplication algorithm in Section 4.3. On the BlueGene/P architecture, the implementation achieves up to a 12X speed-up over a standard 2D SUMMA implementation. This speed-up is in part due to the ability of the 2.5D implementation to map to the 3D network topology of the architecture, and utilize the network-optimized communication collectives. Our analysis and benchmarking of 2.5D matrix multiplication will consider only square matrices, however, in Section 4.4, we will refer to related work to obtain bounds on cost of rectangular matrix multiplication, as these will be needed in future dense linear algebra chapters (in particular for 2.5D QR in Chapter 6).

Cannon's algorithm does not employ communication collectives, so it cannot utilize rectangular collectives. However, Cannon's algorithm sends a low number of messages with respect to a pipelined algorithm like SUMMA. We design a generalization of Cannon's algorithm, Split-Dimensional Cannon's algorithm (SD-Cannon), that explicitly sends data in all dimensions of the network at once with fewer messages than SUMMA. This algorithm does not need topology-aware collectives and retains all the positive features of the classical Cannon's algorithm. However, like Cannon's algorithm, SD-Cannon is difficult to generalize to non-square processor grids. We get

Parts of this chapter were results of joint work with Grey Ballard.

around this challenge by using a virtualization framework, Charm++ [95]. Our performance results on BlueGene/P (Intrepid, located at Argonne National Lab) demonstrate that SD-Cannon outperforms Cannon's algorithm (up to 1.5X on BlueGene/P) and can match the performance of SUMMA with rectangular collectives. The virtualized version of Cannon's algorithm does not incur a high overhead but our Charm++ implementation is unable to saturate all networks links at once.

The rest of the chapter is structured as follows,

- Section 4.1 overview the needed relevant work on the communication cost of matrix multiplication,
- Section 4.2 derives a lower bound on the communication cost of sequential matrix multiplication with tight constants,
- Section 4.3 gives the 2.5D matrix multiplication algorithm and analyzes its performance relative to algorithms used in numerical libraries,
- Section 4.4 references extensions of the 2.5D matrix multiplication algorithm to rectangular matrices,
- Section 4.5 introduces and benchmarks an adaptation of Cannon's algorithm that can achieve higher utilization efficiency on torus networks.

4.1 Previous Work

In this section, we detail the motivating work for our algorithms. First, we recall linear algebra communication lower bounds that are parameterized by memory size. We also detail the main motivating algorithm for this work, 3D matrix multiplication, which uses maximal extra memory and performs less communication. The communication complexity of this algorithm serves as a matching upper-bound for our memory-independent lower bound.

4.1.1 Communication Lower Bounds for Linear Algebra

Recently, a generalized communication lower bound for linear algebra has been shown to apply for a large class of matrix-multiplication-like problems [12]. The lower bound applies to either sequential or parallel distributed memory, and either dense or sparse algorithms. Under some assumptions on load balance, the general lower bound states that for a fast memory of size $\hat{M}$ (e.g. cache size or size of memory space local to processor) the communication bandwidth cost (W) and memory latency cost ($\hat{S}$) must be at least

$$\hat{W} = \Omega\left(\frac{\#arithmetic\ operations}{\sqrt{\hat{M}}}\right), \quad \hat{S} = \Omega\left(\frac{\#arithmetic\ operations}{\hat{M}^{3/2}}\right)$$

words and messages, respectively. In particular, these lower bounds apply to matrix multiplication, TRSM, as well as LU, Cholesky, and QR factorizations of dense matrices. On a parallel machine with p processors and a local processor memory of size M, this yields the following lower bounds for communication costs of matrix multiplication of two dense n-by-n matrices as well as LU factorization of a dense n-by-n matrix:

$$W = \Omega\left(\frac{n^3/p}{\sqrt{M}}\right), \quad S = \Omega\left(\frac{n^3/p}{M^{3/2}}\right). \tag{4.1.1}$$

These lower bounds are valid for $\frac{n^2}{p} \lesssim M \lesssim \frac{n^2}{p^{2/3}}$ and suggest that algorithms can reduce their communication cost by utilizing more memory. If $M < \frac{n^2}{p}$, the input matrices won't fit in memory. As explained in [12], conventional algorithms, for example those in ScaLAPACK [28], mostly do not attain both these lower bounds, so it is of interest to find new algorithms that do.

4.1.2 SUMMA Algorithm

Algorithm 4.1.1 [C] = SUMMA(A, B, C, n, m, k, Π^{2D})

Require: $m \times k$ matrix A, $k \times n$ matrix B distributed so that $\Pi^{2D}[i,j]$ owns $\frac{m}{\sqrt{p}} \times \frac{k}{\sqrt{p}}$ sub-matrix $A[i,j]$ and $\frac{k}{\sqrt{p}} \times \frac{n}{\sqrt{p}}$ sub-matrix $B[i,j]$, for each $i,j \in [0, \sqrt{p}-1]$

1:
2: *% In parallel with all processors*
3: **for all** $i,j \in [0, \sqrt{p}-1]$ **do**
4: **for** $t = 1$ to $t = \sqrt{p}$ **do** Multicast $A[i,t]$ along rows of Π^{2D}
5: Multicast $B[t,j]$ along columns of Π^{2D}
6: $C[i,j] := C[i,j] + A[i,t] \cdot B[t,j]$

Ensure: square $m \times n$ matrix $C = A \cdot B$ distributed so that $\Pi^{2D}[i,j]$ owns $\frac{m}{\sqrt{p}} \times \frac{n}{\sqrt{p}}$ block sub-matrix $C[i,j]$, for each $i,j \in [0, \sqrt{p}-1]$

The SUMMA algorithm [1, 163] (Algorithm 4.1.1), utilizes row and column multicasts to performs parallel matrix multiplication. The algorithm is formulated on a 2D grid, with each process owning a block of the matrices A, B, and C. At each step, the algorithm performs an outer product of parts of A and B. While SUMMA can be done with k rank-one outer products, latency can be reduced by performing $\sqrt{p}$ rank-$(k/\sqrt{p})$ outer-products. The latter case yields an algorithm where at each step every process in a given column of the processor grid multicasts its block of A to all processors in its row. Similarly, a row of processors multicasts B along columns.

4.1.3 Cannon's Algorithm

Cannon's algorithm is a parallel matrix multiplication algorithm that uses shifts blocks among columns and rows of a processor grid. The algorithm starts by staggering the blocks of A and

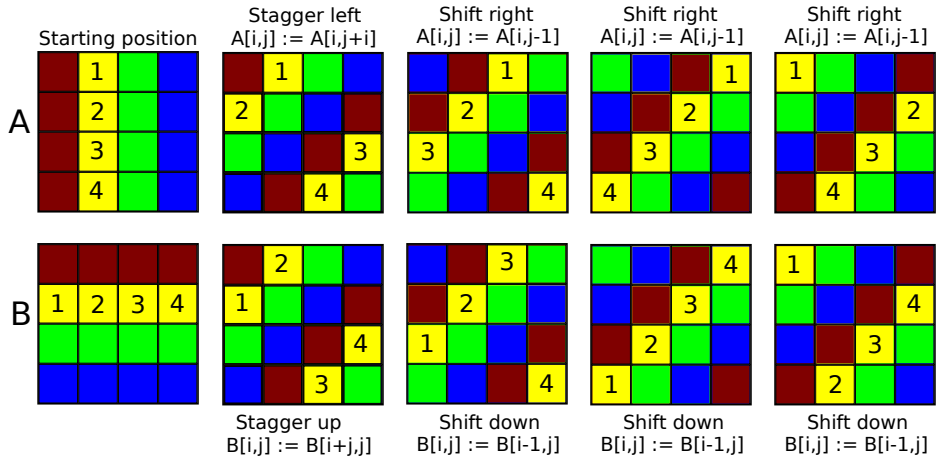


Figure 4.1: Cannon's algorithm, stagger and shift. A and B blocks of the same color must be multiplied together. Notice that the colors (blocks that need to be multiplied) align after each shift.

Algorithm 4.1.2 [C] = Cannon(A, B, C, n, m, k, p, Π^{2D})

Require: $m \times k$ matrix A, $k \times n$ matrix B distributed so that $\Pi^{2D}[i,j]$ owns $\frac{m}{\sqrt{p}} \times \frac{k}{\sqrt{p}}$ sub-matrix $A[i,j]$ and $\frac{k}{\sqrt{p}} \times \frac{n}{\sqrt{p}}$ sub-matrix $B[i,j]$, for each $i,j \in [0, \sqrt{p}-1]$

1: *% In parallel with all processors*
2: **for all** $i,j \in [0, \sqrt{p}-1]$ **do**
3: **for** $t = 1$ to $\sqrt{p}-1$ **do**
4: **if** $t \leq i$ **then**
5: % stagger A
6: $A[i,j] \leftarrow A[i, ((j+1) \mod \sqrt{p})]$
7: **if** $t \leq j$ **then**
8: % stagger B
9: $B[i,j] \leftarrow B[((i+1) \mod \sqrt{p}), j]$
10: $C[i,j] := A[i,j] \cdot B[i,j]$
11: **for** $t = 1$ to $\sqrt{p}-1$ **do**
12: % shift A rightwards
13: $A[i,j] \leftarrow A[i, ((j-1) \mod \sqrt{p})]$
14: % shift B downwards
15: $B[i,j] \leftarrow B[((i-1) \mod \sqrt{p}), j]$
16: $C[i,j] := C[i,j] + A[i,j] \cdot B[i,j]$

Ensure: square $m \times n$ matrix $C = A \cdot B$ distributed so that $\Pi^{2D}[i,j]$ owns $\frac{m}{\sqrt{p}} \times \frac{n}{\sqrt{p}}$ block sub-matrix $C[i,j]$, for each $i,j \in [0, \sqrt{p}-1]$

B leftwards and upwards, respectively. Then the A and B blocks are shifted rightwards and downwards, respectively. We describe Cannon's algorithm on a $\sqrt{p}$-by-$\sqrt{p}$ grid (Π^{2D}) (Algorithm 4.1.2). The procedure is demonstrated in Figure 4.1, where each color corresponds to an outer product. One of the outer products (the yellow blocks) is numbered, and we see that after each shift, different blocks are multiplied, and overall all sixteen distinct block multiplies are performed for that outer product (this also holds for the other 3 outer products).

Once we embed the dD grid onto a 2D grid, we can run Cannon's algorithm with the matrix distribution according to the ordered 2D processor grid. However, in this embedded network, Cannon's algorithm will only utilize $1/d$ of the links, since two messages are sent at a time by each processor and there are $2d$ links per node.

4.1.4 3D Matrix Multiplication

Given unlimited memory, we can still get a communication lower bound, by enforcing the starting data layout to contain only one copy of the data, spread in a load balanced fashion. Now, assuming load-balance, any data replication has a bandwidth cost that can be no smaller than the memory utilized,

$$W = \Omega(M). \tag{4.1.2}$$

This lower-bound is proven and analyzed in more detail in [9]. Therefore, setting $M = O(n^2/p^{2/3})$, minimizes the two communication lower bounds (Eq. 4.1.1 and Eq. 4.1.2),

$$W_{\text{3D}}(n,p) = \Omega\left(\frac{n^2}{p^{2/3}}\right), \quad S_{\text{3D}}(n,p) = \Omega\left(1\right).$$

This lower bound is also proven in [2, 87]. We call these lower bounds 3D, because they lower bound the amount of data processors can communicate, if given a 3D block of computation to perform. To achieve this lower bound it is natural to formulate algorithms in a 3D processor topology. These lower bounds are achieved by the 3D matrix multiplication algorithm [43, 1, 2, 23, 93]. Algorithm 4.1.4 is a description of the 3D MM algorithm on a cubic grid of processors Π.

The 3D matrix multiplication algorithm performs only two broadcasts and one reduction per processor, all of size $O(n^2/p^{2/3})$ words. Given efficiently pipelined collectives, this algorithm moves $W_{\text{3D}}^{\text{MM}}(n,p) = O\left(\frac{n^2}{p^{2/3}}\right)$ words. The latency cost of one reduction under our model is $S_{\text{3D}}^{\text{MM}}(p) = O\left(\log p\right)$. This cost is optimal when we consider that information from a block row or block column can only be propagated to one processor with $\Omega(\log p)$ messages.

4.1.5 2.5D Asymptotic Communication Lower Bounds

The general communication lower bounds are valid for a range of M in which 2D and 3D algorithms hit the extremes. 2.5D algorithms are parameterized to be able to achieve the communication lower bounds for any valid M. Let $c \in \{1, 2, \ldots, \lfloor p^{1/3} \rfloor\}$ be the number of replicated copies

Algorithm 4.1.3 [C] = 3D-MM(A,B,$\Pi[1 : p^{1/3}, 1 : p^{1/3}, 1 : p^{1/3}]$,$n$,$p$)

Require: On cubic processor grid Π, n-by-n matrix A is spread over $\Pi[:, 1, :]$, n-by-n matrix B^T is spread over $\Pi[1, :, :]$

1: Replicate $A[i, k]$ on each $\Pi[i, j, k]$, for $j \in [1, p^{1/3}]$
2: Replicate $B[k, j]$ on each $\Pi[i, j, k]$, for $i \in [1, p^{1/3}]$
3: % *Perform local block multiplies on every processor:*
4: $C_k[i, j] := A[i, k] \cdot B[k, j]$ on $\Pi[i, j, k]$, for each i, j, k
5: % *Compute C via a sum reduction:*
6: $C[i, j] := \sum_{k=1}^{p^{1/3}} C_k[i, j]$

Ensure: n-by-n matrix $C = A \cdot B$ spread over $\Pi[:, :, 1]$

of the input matrix. So, each processor has local memory size $M = \Omega\left(cn^2/p\right)$. The general lower bounds on communication are

$$W_{\text{2.5D}}(n, p, c) = \Omega\left(\frac{n^2}{\sqrt{cp}}\right), \quad S_{\text{2.5D}}(n, p, c) = \Omega\left(\frac{p^{1/2}}{c^{3/2}}\right).$$

The lower bound in Section 6 of [12] is valid while $1 \leq c \leq p^{1/3}$. In the special cases of $c = 1$ and $c = p^{1/3}$ we get,

$$W_{\text{2.5D}}(n, p, 1) = W_{\text{2D}}(n, p), \quad S_{\text{2.5D}}(n, p, 1) = S_{\text{2D}}(n, p)$$
$$W_{\text{2.5D}}(n, p, p^{1/3}) = W_{\text{3D}}(n, p), \quad S_{\text{2.5D}}(n, p, p^{1/3}) = S_{\text{3D}}(n, p)$$

Using $c < 1$ copies of the matrices is impossible without loss of data, and using $c > p^{1/3}$ copies of the matrices cannot be useful since the unlimited memory case is still bound by the 3D communication lower bounds.

4.1.6 Memory-Efficient Matrix Multiplication Under the BSP Model

McColl and Tiskin [113] presented a memory efficient variation on the 3D matrix multiplication algorithm under a BSP model. They partition the 3D computation graph to pipeline the work and therefore use memory to reduce communication in a tunable fashion. This algorithm achieves the bandwidth lower bound (Equation 4.1.1). This work is closest to the 2.5D matrix multiplication algorithm we give, although we additionally provide a performance evaluation.

4.2 Communication Lower Bounds

We prove the following theorem which yields a lower bound on the communication cost of matrix multiplication. This lower bound result is not new from an asymptotic stand-point, but constitutes an improvement of the constant factor on the lower bound with respect to the best bound we are aware of. In particular, the first term in the bound is a factor of 16 higher than the lower bound given by [12], although our derivation technique closely resembles the one used in [12].

Theorem 4.2.1. *Any matrix multiplication algorithm of m-by-k matrix A with k-by-n matrix B into m-by-n matrix C with a fast memory (cache) of size $\hat{M}$ assuming no operands start in cache and all outputs are written to memory has communication cost, i.e. the number of words moved between cache and memory,*

$$\bar{W}(n, m, k, \hat{M}) \geq \max \left[\frac{2mnk}{\sqrt{\hat{M}}}, mk + kn + mn \right].$$

Proof. The term $mk + kn + mn$ arises from the need to read the inputs into cache and write the output from cache to main memory.

Consider the mnk scalar multiplications done by any schedule which executes the matrix multiplication algorithm. If the schedule computes a contribution to C more than once, we only count the the contribution which gets written to the output in memory, and ignore any other redundantly computed and discarded computation. Subdivide them into f chunks of size $\hat{M}^{3/2}$ for $f = \lceil mnk/\hat{M}^{3/2} \rceil$, with the last 'remainder' chunk being of size less than $\hat{M}^{3/2}$ (we assume that $\hat{M}^{3/2}$ does not divide evenly into mnk, so $mnk/\hat{M}^{3/2} - \lfloor mnk/\hat{M}^{3/2} \rfloor > 0$, the other case is strictly simpler). We can label each scalar multiplication $A_{il} \cdot B_{lj}$ with a tuple (i, j, l). We then define three projections $\pi_1(i, j, l) = (j, l)$, $\pi_2(i, j, l) = (i, l)$, and $\pi_3(i, j, l) = (i, j)$, which define the two operands needed by the (i, j, l) multiplication and the entry of the output C to which the multiplication contributes. Given any set S of tuples of size three, and projecting them onto a set of tuples of size two using the three projections π_m, gives us sets $P_m = \{p : \exists s \in S, p \in \pi_m(s)\}, \forall m \in [1, 3]$. By the Loomis-Whitney inequality [111] (very commonly used in such lower bound proofs), we obtain that $(|P_1| \cdot |P_2| \cdot |P_3|)^{1/2} \geq |S|$, which implies that $\sum_m |P_m| \geq 3|S|^{2/3}$. This means that for each chunk $i \in [1, f]$, which contains $t_i \leq \hat{M}^{3/2}$ multiplications, at least $3t_i^{2/3}$ total operands of A and B plus contributions to C are necessary.

Now, let x_i be the number of operands present in cache prior to execution of chunk i (by assumption $x_1 = 0$). The number of operands (elements of A and B) present in cache at the end of execution of chunk i should be equal to the number of operands available for chunk $i + 1$, x_{i+1}. Let the number of contributions to C (outputs), which remain in cache (are not written to memory) at the end of chunk i, be y_i (by assumption $y_f = 0$), the rest of the outputs produced by chunk i must be written to memory. In total, the amount of reads from memory and writes to memory done during the execution of chunk i with t_i multiplications is then at least $w_i \geq 3t_i^{2/3} - x_i - y_i, \forall i \in [1, f]$. Now, since the operands and outputs which are kept in cache at the end of chunk i must fit in cache, we know that for $i \in [1, f - 1]$, $x_{i+1} + y_i \leq \hat{M}$. Substituting this in, we obtain $w_i \geq 3t_i^{2/3} - \hat{M} - x_i + x_{i+1}, \forall i \in [1, f - 1]$. Summing over all chunks yields a lower bound on

the total communication cost:

$$\begin{aligned}
\bar{W}(n,m,k,\hat{M}) &\geq \sum_{i=1}^{f} w_i \geq 3t_f^{2/3} - x_f + \sum_{i=1}^{f-1}(3t_i^{2/3} - \hat{M} - x_i + x_{i+1}) \\
&= -(f-1)\hat{M} + 3t_f^{2/3} + \sum_{i=1}^{f-1} 3t_i^{2/3} \\
&= -(f-1)\hat{M} + 3(mnk - (f-1)\hat{M}^{3/2})^{2/3} + \sum_{i=1}^{f-1} 3\hat{M} \\
&= 2(f-1)\hat{M} + 3(mnk - (f-1)\hat{M}^{3/2})^{2/3} \\
&= 2\hat{M} \cdot \lfloor mnk/\hat{M}^{3/2} \rfloor + 3(mnk - \lfloor mnk/\hat{M}^{3/2} \rfloor \cdot \hat{M}^{3/2})^{2/3} \\
&= 2mnk/\hat{M}^{1/2} - 2\hat{M} \cdot (mnk/\hat{M}^{3/2} - \lfloor mnk/\hat{M}^{3/2} \rfloor) + 3\hat{M} \cdot (mnk/\hat{M}^{3/2} - \lfloor mnk/\hat{M}^{3/2} \rfloor)^{2/3} \\
&\geq 2mnk/\hat{M}^{1/2} + 2\hat{M} \cdot \left[(mnk/\hat{M}^{3/2} - \lfloor mnk/\hat{M}^{3/2} \rfloor)^{2/3} - (mnk/\hat{M}^{3/2} - \lfloor mnk/\hat{M}^{3/2} \rfloor) \right] \\
&\geq 2mnk/\hat{M}^{1/2}
\end{aligned}$$

Therefore, we have

$$\bar{W}(n,m,k,\hat{M}) \geq \max \left[\frac{2mnk}{\sqrt{\hat{M}}}, mk + kn + mn \right].$$

□

4.3 2.5D Matrix Multiplication

Consider the 2.5D processor grid of dimensions $\sqrt{p/c}$-by-$\sqrt{p/c}$-by-c (indexed as $\Pi[i,j,k]$) Using this processor grid for matrix multiplication, Algorithm 4.3 achieves the 2.5D bandwidth lower bound and gets within a factor of $O(\log(p))$ of the 2.5D latency lower bound. Algorithm 4.3 generalizes the 2D SUMMA algorithm (set $c = 1$). At a high level, our 2.5D algorithm does an outer product of sub-matrices of A and B on each layer, then combines the results. 2.5D Cannon's algorithm can be done by adjusting the initial shift to be different for each set of copies of matrices A and B [147].

Our 2.5D algorithm also generalizes the 3D algorithm (Algorithm 4.1.4). The two algorithms differ only in the initial layout of matrices. However, after the initial replication step, the subsequent broadcasts and computation are equivalent. Further, the 2.5D algorithm has the nice property that C ends up spread over the same processor layer that both A and B started on. The algorithm moves $W_{\text{2.5D}}^{\text{MM}}(n,p,c) = O\left(\frac{n^2}{\sqrt{cp}}\right)$ words. If this 2.5D algorithm uses 2.5D SUMMA, the latency cost is $S_{2.5D}(n,p,c) = O\left(\sqrt{p/c^3}\log(p)\right)$ messages. The extra $\log(p)$ factor in the latency cost

Algorithm 4.3.1 $[C]$ = 2.5D-MM(A,B,$\Pi[1:(p/c)^{1/2}, 1:(p/c)^{1/2}, 1:c]$,$n$,$m$,$l$,$c$,$p$)

Require: On a $\sqrt{p/c}$-by-$\sqrt{p/c}$-by-c processor grid Π, m-by-n matrix A and n-by-l matrix B, are each spread over $\Pi[:, :, 1]$

1: Replicate A and B on each $\Pi[:, :, k]$, for $k \in [1, c]$
2: *% Perform an outer product on each processor layer* $\Pi[:, :, k]$ *in parallel:*
3: **for** $k = 1$ to $k = c$ **do**
4: $[C_k]$ = 2D-MM($A[:, (k-1) \cdot n/c : k \cdot n/c]$,
5: $B[(k-1) \cdot n/c : k \cdot n/c, :]$,
6: $\Pi[:, :, k]$, n/c, m, l)
7: *% Compute* C *via a sum reduction:*
8: $C := \sum_{k=1}^{c} C_k$

Ensure: square m-by-l matrix $C = A \cdot B$ spread over $\Pi[:, :, 1]$

can be eliminated by performing part of Cannon's algorithm rather than part of SUMMA on each layer [147]. Ignoring $\log(p)$ factors, this cost is optimal according to the general communication lower bound. The derivations of these costs are in Appendix A in [147].

If the latency cost is dominated by the intra-layer communication ($\Theta(\sqrt{p/c^3})$ messages), rather than the $\Theta(\log(c))$ cost of the initial broadcast and final reduction the 2.5D matrix multiplication algorithm can achieve perfect strong scaling in certain regimes. Suppose we want to multiply $n \times n$ matrices, and the maximum memory available per processor is $M_{\max}$. Then we need to use at least $p_{\min} = \Theta(n^2/M_{\max})$ processors to store one copy of the matrices. The 2D algorithm uses only one copy of the matrix and has a bandwidth cost of

$$W_{\text{2D}}^{\text{MM}}(n, p_{\min}) = W_{\text{2.5D}}^{\text{MM}}(n, p_{\min}, 1) = O(n^2/\sqrt{p_{\min}})$$

words and latency cost of

$$S_{\text{2D}}^{\text{MM}}(n, p_{\min}) = S_{\text{2.5D}}^{\text{MM}}(n, p_{\min}, 1) = O(\sqrt{p_{\min}})$$

messages. If we use $p = c \cdot p_{\min}$ processors, the total available memory is $p \cdot M_{\max} = c \cdot p_{\min} \cdot M_{\max}$, so we can afford to have c copies of the matrices. The 2.5D algorithm can store a matrix copy on each of c layers of the p processors. The 2.5D algorithm would have a bandwidth cost of

$$W_{\text{2.5D}}^{\text{MM}}(n, p, c) = O(n^2/\sqrt{cp}) = O(n^2/(c\sqrt{p_{\min}})) = O(W_{\text{2.5D}}^{\text{MM}}(n, p_{\min}, 1)/c)$$

words, and a latency cost of

$$S_{\text{2.5D}}^{\text{MM}}(n, p, c) = O(\sqrt{p/c^3}) = O(\sqrt{p_{\min}}/c) = O(S_{\text{2.5D}}^{\text{MM}}(n, p_{\min}, 1)/c)$$

messages. This strong scaling is perfect because all three costs (flops, bandwidth and latency) fall by a factor of c (up to a factor of $c = p^{1/3}$, and ignoring the $\log(c)$ latency term).

4.3.1 Performance benchmarking configuration

We implemented 2.5D matrix multiplication MPI [71] for inter-processor communication. We perform most of the sequential work using BLAS routines: DGEMM for matrix multiplication, DGETRF, DTRSM, DGEMM, for LU. We found it was fastest to use provided multi-threaded BLAS libraries rather than our own threading. All the results presented in this chapteruse threaded ESSL routines.

We benchmarked our implementations on a Blue Gene/P (BG/P) machine located at Argonne National Laboratory (Intrepid). We chose BG/P as our target platform because it uses few cores per node (four 850 MHz PowerPC processors) and relies heavily on its interconnect (a bidirectional 3D torus with 375 MB/sec of achievable bandwidth per link). On this platform, reducing inter-node communication is vital for performance.

BG/P also provides topology-aware partitions, which 2.5D algorithms are able to exploit. For node counts larger than 16, BG/P allocates 3D cuboid partitions. Since 2.5D algorithms have a parameterized 3D virtual topology, a careful choice of c allows them to map precisely to the allocated partitions (provided enough memory).

Topology-aware mapping can be very beneficial since all communication is constrained to one of three dimensions of the 2.5D virtual topology. Therefore, network contention is minimized or completely eliminated. Topology-aware mapping also allows 2.5D algorithms to utilize optimized line multicast and line reduction collectives provided by the DCMF communication layer [57, 103].

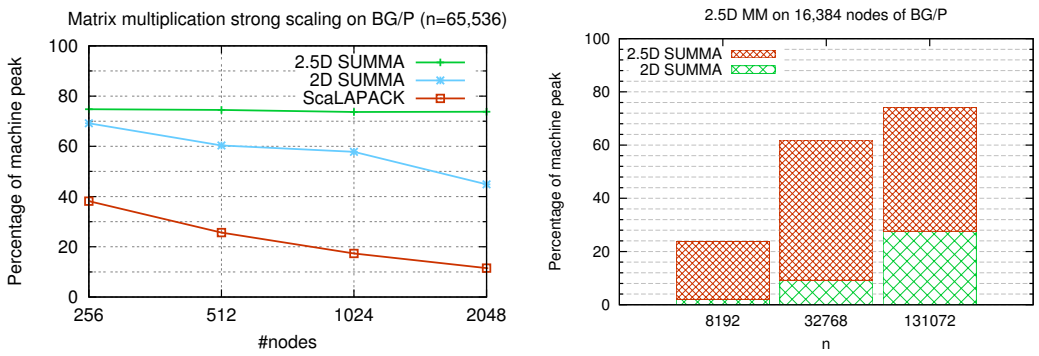
We study the strong scaling performance of 2.5D algorithms on a 2048 node partition (Figures 4.2(a), 5.5(a)). The 2048 node partition is arranged in a 8-by-8-by-32 torus. In order to form square layers, our implementation uses 4 processes per node (1 process per core) and folds these processes into the X dimension. Now, each XZ virtual plane is 32-by-32. We strongly scale 2.5D algorithms from 256 nodes $c = Y = 1$ to 2048 nodes $c = Y = 8$. For ScaLAPACK we use smp or dual mode and form a square grid.

We also compare performance of 2.5D and 2D algorithms on 16,384 nodes (65,536 cores) of BG/P (Figures 4.2(b), 5.5(b)). The 16,384 node partition is a 16-by-32-by-32 torus. We run both 2D and 2.5D algorithms in SMP mode. For 2.5D algorithms, we use $c = 16$ YZ processor layers.

We perform our analysis on a square grid and analyze only square matrices. Oddly shaped matrices of processor grids can be overcome through the use of virtualization. Each processor can be made responsible for the work of a block of processors of another virtual processor grid. So it is possible to decompose any problem onto a nicely shaped (e.g. cubic) virtual processor grid and then map it onto a smaller processor mesh.

4.3.2 2.5D matrix multiplication performance

Figure 4.2(a) demonstrates that 2.5D matrix multiplication achieves better strong scaling than its 2D counter-part. However, both run at high efficiency (over 50%) for this problem size, so the benefit is minimal. The performance of the more general ScaLAPACK implementation lags behind the performance of our code by a factor of 2-5X.



(a) MM strong scaling (b) Matrix multiplication on 16,384 nodes

Figure 4.2: Performance of 2.5D MM on BG/P

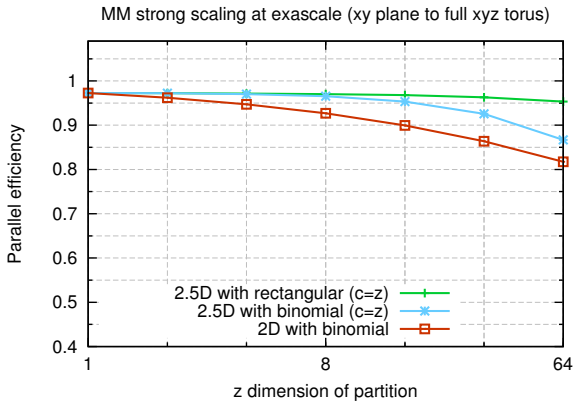


Figure 4.3: Matrix multiplication predicted performance on an exascale architecture. The results show strong scaling from full memory on the first plane ($z = 1$) of the machine to the full machine ($z = 64$).

Figure 4.2(b) shows that 2.5D matrix multiplication outperforms 2D matrix multiplication significantly for small matrices on large partitions. The network latency and bandwidth costs are reduced, allowing small problems to execute much faster (up to 12X faster for the smallest problem size).

4.3.3 Predicted Performance at Exascale

We model performance of matrix multiply with rectangular and binomial collectives at exascale. Our 2D and 2.5D MM algorithms own a single block of each matrix and always communicate in messages that contain the whole block. For the problem we study, this message size should be entirely bandwidth bound for both 1D rectangular and binomial protocols. So we model the

multicast and reduction in the 2.5D algorithm using the maximum bandwidth achieved by these protocols in Figure 2.5. We assume the shifts (sends) in the algorithm achieve the single link peak bandwidth.

Figure 4.3 demonstrates the strong scaling of 2.5D MM with rectangular and binomial collective protocols and 2D MM with a binomial protocol. We scale from a single plane of the 3D torus ($z = 1$) to the entire machine ($z = 64$). We use the largest matrix size that fits in memory on a single plane ($n = 2^{22}$). We calculate the computational time (t_f) based on the peak flop rate. We express parallel efficiency in terms of t_f and the communication time t_c,

$$\textit{efficiency} = \frac{t_f}{t_f + t_c}$$

Evidently, matrix multiplication is dominated by computation at this scale. The 2.5D algorithm and rectangular collectives only make a difference when we are using most of the machine. This result is consistent with the performance of MM on BG/P, where significant speed-ups were achieved only for smaller problem sizes.

4.4 Rectangular Matrix Multiplication

Algorithms for dense factorizations including ones we consider in this thesis primarily employ on non-square (rectangular) matrix multiplications. Most frequently two of the three matrices (the two operands and the result) are rectangular while one is square with the dominant cost coming from operations where the square matrix is larger than the rectangular matrices.

4.4.1 Previous Work on Matrix Multiplication

We first reference the communication costs of multiplication of rectangular matrices on a sequential computer. Here the only communication cost we consider is data movement between the memory and the cache (Q). This cost is dominated either by a term corresponding to reading and multiplying all needed pairs of blocks in cache, or by a term corresponding to reading in all the data of the largest matrix into cache.

Lemma 4.4.1. *The cost of matrix multiplication $C = A \cdot B$ where A is m-by-n and B is n-by-k, on a sequential computer with cache size $\hat{M}$ is*

$$T_{\text{MM}}(m, n, k, \hat{M}) = O\left(\gamma \cdot mnk + \nu \cdot \left[\frac{mnk}{\sqrt{\hat{M}}} + mn + nk + mk\right]\right)$$

Proof. We use the Rec-Mult algorithm with memory bandwidth cost proved in [63, Theorem 1]. □

In the case of parallel computers, we additionally consider interprocessor traffic and synchronization cost necessary to compute rectangular matrix multiplication. Due to the low depth of the

matrix multiplication dependency graph, efficient parallelization with low synchronization cost is possible (unlike in the case of dense matrix factorizations, which have polynomial depth).

Lemma 4.4.2. *The cost of matrix multiplication $C = A \cdot B$ where A is m-by-n and B is n-by-k on a parallel computer with p processors each with local memory $M > (mn + nk + mk)/p$ and cache size $\hat{M}$ is*

$$T_{\text{MM}}(m, n, k, p, \hat{M}) = O\left(\gamma \cdot \frac{mnk}{p} + \beta \cdot \left[\frac{mn + nk + mk}{p} + \frac{mnk}{p \cdot \sqrt{M}} + \left(\frac{mnk}{p}\right)^{2/3}\right] + \nu \cdot \frac{mnk}{p \cdot \sqrt{\hat{M}}} + \alpha \cdot \left[\frac{mnk}{p \cdot M^{3/2}} \log\left(\frac{pm^2}{nk}\right) + \log(p)\right]\right).$$

Proof. We use the CARMA algorithm for parallel matrix multiplication [46]. Throughout this proof we assume (without loss of generality) that $m \leq n \leq k$. We also ignore constants and lower order terms but omit asymptotic notation. Because the computation is load balanced, the computational cost is mnk/p.

We next show that the claimed interprocessor bandwidth cost upper bounds the cost for each of the three cases given in [46, Table I]. If $p < k/n$, then the cost mn is bounded above by mk/p, a term which appears in the expression above. If $k/n \leq p \leq nk/m^2$, then $m^2 \leq nk/p$ and the cost $\sqrt{m^2nk/p}$ is bounded above by nk/p, which also appears above. Finally, if $p > nk/m^2$, the cost is $mnk/(p\sqrt{M}) + (mnk/p)^{2/3}$, which itself appears in the expression above.

The memory bandwidth cost is not considered in [46], so we derive the cost of the CARMA algorithm here. Recall that the total memory bandwidth cost includes the interprocessor bandwidth terms in the expression above. If $p < k/n$, then the single local matrix multiplication on each processor is of dimension $m \times n \times (k/p)$. By Lemma 4.4.1, the memory bandwidth cost is upper bounded by $mnk/(p\sqrt{\hat{M}}) + nk/p$ (note that the largest matrix is the second input), and both terms appear above. If $k/n \leq p \leq nk/m^2$, then the single local matrix multiplication on each processor is of dimension $m \times \sqrt{nk/p} \times \sqrt{nk/p}$. By Lemma 4.4.1, the memory bandwidth cost is again upper bounded by $mnk/(p\sqrt{\hat{M}}) + nk/p$. If $p > nk/m^2$, then there are two possible scenarios depending on the size of the local memory M. If $M \geq (mnk/p)^{2/3}$, then the single local matrix multiplication on each processor is square and of dimension $(mnk/p)^{1/3}$. By Lemma 4.4.1, the memory bandwidth cost is upper bounded by $mnk/(p\sqrt{\hat{M}}) + (mnk/p)^{2/3}$. If $M < (mnk/p)^{2/3}$, then each processor performs $mnk/(pM^{3/2})$ local matrix multiplications, each of square dimension $\sqrt{M}$. By Lemma 4.4.1, the total memory bandwidth cost of these multiplications is $(mnk/[pM^{3/2}]) \cdot (M^{3/2}/\sqrt{\hat{M}} + M) = mnk/(p\sqrt{\hat{M}}) + mnk/(p\sqrt{M})$, which is dominated by the first term.

The claimed latency cost includes the latency costs of each of the three cases analyzed in [46] and is therefore an upper bound. □

4.4.2 Streaming Matrix Multiplication

In the cost analysis of the symmetric eigensolver algorithm, it will be necessary to consider the multiplication of a replicated matrix by a panel, so we first derive Lemma 4.4.3 which quantifies the communication cost of this primitive. In particular, we introduce a streaming matrix multiplication algorithm, where a single large (square or nearly-square) matrix is replicated over a number of sets of processors (on each set the matrix is in a 2D layout) and is multiplied by a smaller rectangular matrix (producing another rectangular matrix of about the same size). The advantage of this primitive over the previous matrix multiplication algorithms we discussed is the fact that it does not require a term corresponding to the data movement required to replicate the large matrix.

This 2.5D algorithm and some of the further ones used in this thesis will be parameterized by δ (same as α in [158]) in contrast to 2.5D algorithms [146], which were parameterized by c. The two parameterizations are closely related $p^\delta = \sqrt{cp}$ and $c = p^{2\delta-1}$. The parameters have corresponding ranges $\delta \in [1/2, 2/3]$ and $c \in [1, p^{1/3}]$. While c is reflective of the number of copies of the matrix that are needed, the exponent δ yields the right factor for recursion with subsets of processors, which will be particularly useful in our successive band reduction algorithm (Algorithm 7.3.1).

Lemma 4.4.3. *Consider a matrix A of dimensions m-by-k where $m, k = O(n)$, that is replicated over $c = p^{2\delta-1}$ groups of $p/c = p^{2(1-\delta)}$ processors for $\delta \in [1/2, 2/3]$ and $c \in [1, p^{1/3}]$. The cost of multiplying A by a matrix B of dimension k-by-b where $b \ll n$ is*

$$T_{\text{repmm}}(n, b, p, \delta) = O\left(\gamma \cdot \frac{n^2 b}{p} + \beta \cdot \frac{nb}{p^\delta} + \alpha \cdot \log(p)\right),$$

when the cache size is $\hat{M} > n^2/p^{2(1-\delta)}$ and the copies of A start inside cache, otherwise the cost is

$$T_{\text{repmm}}(n, b, p, \delta) = O\left(\gamma \cdot \frac{n^2 b}{p} + \beta \cdot \frac{nb}{p^\delta} + \nu \cdot \left[\frac{n^2}{p^{2(1-\delta)}} + \frac{n^2 b}{p\sqrt{\hat{M}}}\right] + \alpha \cdot \log(p)\right).$$

Proof. Assume that B starts in a 1D layout on a group of p^δ processors and that each of $p^{2\delta-1}$ copies of A is distributed on a 2D $p^{1-\delta}$-by-$p^{1-\delta}$ processor grid. We scatter B so that a column of $p^{1-\delta}$ processors on each of the $p^{2\delta-1}$ 2D processor grids owns a k-by-$b/p^{1-\delta}$ vertical slice of B in a 1D layout. This redistribution/scatter costs no more than

$$O\left(\beta \cdot \frac{nb}{p^\delta} + \alpha \cdot \log(p)\right).$$

Now, we replicate each slice B_i across all processor columns in the ith 2D processor grid, multiply locally by A then reduce the result to one row of $p^{1-\delta}$ processors on the ith processor grid, obtaining $C_i = A \cdot B_i$. The costs of the broadcast and reduction are

$$O\left(\beta \cdot \frac{nb}{p^{2(1-\delta)}} + \alpha \cdot \log(p)\right) = O\left(\beta \cdot \frac{nb}{p^\delta} + \alpha \cdot \log(p)\right).$$

The cost of the local matrix multiplication is no more than

$$O\left(\gamma \cdot \frac{n^2 b}{p} + \nu \cdot \left[\frac{n^2}{p^{2(1-\delta)}} + \frac{n^2 b}{p\sqrt{\hat{M}}}\right]\right)$$

by Lemma 4.4.1. However, if the entire matrix A starts in cache, which implies that $\hat{M} > n^2/p^{2(1-\delta)}$, it suffices to read only the entries of B_i from memory into cache and write the entries of C_i out to memory. In this case, the memory bandwidth cost is

$$O\left(\nu \cdot \frac{nb}{p^{2(1-\delta)}}\right),$$

which is dominated by the interprocessor communication term since $\nu \ll \beta$. The result $C = A \cdot B$ can be gathered onto the same processors onto which W started, by first transposing the C_i panel on the 2D processor grid, so that the processor column which owns B_i now owns C_i, then gathering to the p^δ processors which owned B_i. This gather is a direct inverse of the scatter if A done initially and has the same asymptotic communication cost. The transpose has low order cost with respect to broadcasting B_i and reducing C_i. □

4.5 Split-Dimensional Cannon's Algorithm

4.5.1 Algorithm

The following algorithm is designed for a l-ary d-cube bidirectional torus network (a d dimensional network of $p = l^d$ processors). The algorithm requires that the torus network is of even dimension ($d = 2i$ for $i \in \{1, 2, \ldots\}$). Virtualization will be used to extend our approach to odd-dimensional

Matrix Layout on Torus Networks

A matrix is a 2D array of data. To spread this data in a load balanced fashion, we must embed the 2D array in the higher-dimensional torus network. Any l-ary d-cube Π^{dD}, where d is a multiple of 2, can be embedded onto a square 2D grid. Each of two dimensions in this square grid is of length $l^{d/2}$ and is formed by folding a different $d/2$ of the d dimensions. For simplicity we fold the odd $d/2$ dimensions into one of the square grid dimensions and the $d/2$ even dimensions into the other square grid dimension. The algorithms below will assume the initial matrix layout follows this ordering. In actuality, the ordering is irrelevant since a l-ary d-cube network is invariant to dimensional permutations. We define a dimensional embedding for a processor with a d-dimensional index $I^d \in \{0, 1, \ldots, l-1\}^d$, to a two dimensional index (i, j) as

$$G^{dD \to 2D}[I^d] = \left(\sum_{i=0}^{d/2-1} l^i I^d[2i], \sum_{i=0}^{d/2-1} l^i I^d[2i+1]\right).$$

We denote the processor with grid index I^d as $\Pi^{dD}[I^d]$.

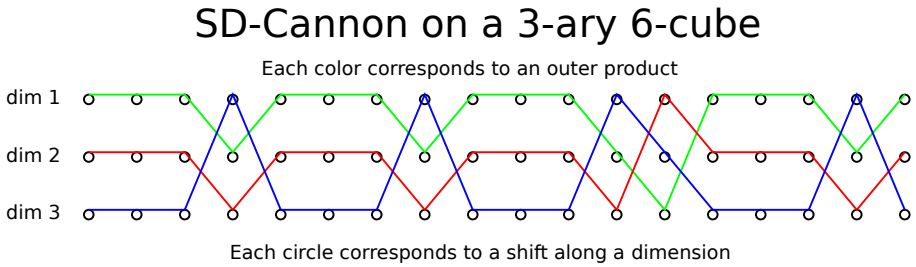


Figure 4.4: Snapshot of network usage in SD-Cannon. A and B use a disjoint set of 3 network dimensions in the same fashion, so it suffices to pay attention to 3.

Split-Dimensional Cannon's Algorithm

We can formulate another version of Cannon's algorithm by using more dimensional shifts. A shift can be performed with a single message sent over a single link from each processor to the next. Since the shifts will be done along dimensions of the l-ary d-cube network, $2d$ links will be available. Algorithm 4.5.1 performs this dimensional shift. Split-dimensional (SD) Cannon's algorithm will use exclusively this shift for communication. In fact, all shifts operate on a static message size. Therefore, communication cost can be calculated by counting shifts. The algorithm achieves complete utilization on any l-ary d-cube network during the shift stage. We specify the algorithm for a bidirectional network as those are much more common. However, the algorithms can be trivially simplified to the unidirectional case.

Algorithm 4.5.1 Shift$<$ dim, dir $>$(l, M, p, Π^{dD}, I^d)

Require: $\Pi^{dD}[I^d]$ owns sub-matrix M.
1: $S^d \leftarrow I^d$
2: **if** dir $= +1$ **then**
3: $\quad S^d[\text{dim}] = (S^d[\text{dim}] + 1) \mod l$
4: **if** dir $= -1$ **then**
5: $\quad S^d[\text{dim}] = (S^d[\text{dim}] - 1) \mod l$
6: Send M to $\Pi^{dD}[S^d]$.
7: % $\Pi^{dD}[I^d]$ sends to $\Pi^{dD}[S^d]$

Algorithm 4.5.2 describes how the stagger step is done inside the SD-Cannon algorithm. A different shift is done on each sub-panel of A and B concurrently. These calls should be done asynchronously and ideally can fully overlap. One interpretation of this stagger algorithm is that sub-panels of each matrix are being staggered recursively along $d/2$ disjoint dimensional orders.

Algorithm 4.5.3 is a recursive routine that loops over each dimension performing shifts on sub-panels of A and B. The order of the shifts is permuted for each sub-panel. Each sub-panel is multiplied via a recursive application of Cannon's algorithm over a given dimensional ordering.

Algorithm 4.5.2 SD-Stagger(A, B, n_b, m_b, k_b, l, p, Π^{dD},I^d)

Require: $\Pi^{dD}[I^d]$ owns $m_b \times k_b$ sub-matrix A and $k_b \times n_b$ sub-matrix B.

Split $A = [A_0, A_1, \ldots, A_d]$ where each A_h is $m_b \times k_b/d$.
Split $B^T = [B_0^T, B_1^T, \ldots, B_d^T]$ where each B_h is $k_b/d \times n_b$.
% At each level, apply index shift
for level $\in [0, d/2 - 1]$ **do**
    % To stagger must shift up to $l - 1$ times
    for $t = 1$ to $l - 1$ **do**
        % Shift the ordering
        for all dh $\in [0, d/2 - 1]$ **do**
            $h \leftarrow (dh + \text{level}) \mod (d/2)$
            if $t \leq I^d[2h + 1]$ **then**
                Shift$< (2 * h), +1 >(l, A_h, p, \Pi^{dD}, I^d)$
            if $t \leq l - I^d[2h + 1]$ **then**
                Shift$< (2 * h), -1 >(l, A_{h+d/2}, p, \Pi^{dD}, I^d)$
            if $t \leq I^d[2h]$ **then**
                Shift$< (2 * h + 1), +1 >(l, B_h, p, \Pi^{dD}, I^d)$
            if $t \leq l - I^d[2h]$ **then**
                Shift$< (2 * h + 1), -1 >(l, B_{h+d/2}, p, \Pi^{dD}, I^d)$

Algorithm 4.5.3 SD-Contract$<$ **level** $>$(A, B, C, n_b, m_b, k_b, l, p, Π^{dD}, I^d)

Require: $\Pi^{dD}[I^d]$ owns $m_b \times k_b$ sub-matrix A and $k_b \times n_b$ sub-matrix B.

Split $A = [A_0, A_1, \ldots, A_d]$ where each A_h is $m_b \times k_b/d$.
Split $B^T = [B_0^T, B_1^T, \ldots, B_d^T]$ where each B_h is $k_b/d \times n_b$.
% Shift and contract l times
for $t = 0$ to $l - 1$ **do**
    if level $= d/2 - 1$ **then**
        $C \leftarrow C + A \cdot B$
    else
        SD-Contract$<$ level $+ 1 >(A, B, C, \frac{n}{\sqrt{p}}, \frac{m}{\sqrt{p}}, \frac{k}{\sqrt{p}}, l, p, \Pi^{dD}, I^d)$
    for all dh $\in [0, d/2 - 1]$ **do**
        % Shift the ordering
        $h \leftarrow (dh + \text{level}) \mod (d/2)$
        Shift$< (2 * h), +1 >(l, A_h, p, \Pi^{dD}, I^d)$
        Shift$< (2 * h + 1), +1 >(l, B_h, p, \Pi^{dD}, I^d)$
        Shift$< (2 * h), -1 >(l, A_{h+d/2}, p, \Pi^{dD}, I^d)$
        Shift$< (2 * h + 1), -1 >(l, B_{h+d/2}, p, \Pi^{dD}, I^d)$

Algorithm 4.5.4 SD-Cannon(A, B, C, n, m, k, l, p, Π^{dD}, $G^{dD\to 2D}$)

Require: $m \times k$ matrix A, $k \times n$ matrix B distributed so that $\Pi^{dD}[I]$ owns $\frac{m}{\sqrt{p}} \times \frac{k}{\sqrt{p}}$ sub-matrix $A[G^{dD\to 2D}[I]]$ and $\frac{k}{\sqrt{p}} \times \frac{n}{\sqrt{p}}$ sub-matrix $B[G^{dD\to 2D}[I]]$

1: *% In parallel with all processors*
2: **for all** $I^d \in \{0, 1, \ldots, l-1\}^d$ **do** $(i, j) \leftarrow G^{dD\to 2D}[I^d]$
3: SD-Stagger($A[i,j]$, $B[i,j]$, $\frac{n}{\sqrt{p}}$, $\frac{m}{\sqrt{p}}$, $\frac{k}{\sqrt{p}}$, l, p, Π^{dD}, I^d)
4: SD-Contract$< 0 >$($A[i,j]$, $B[i,j]$, $C[i,j]$, $\frac{n}{\sqrt{p}}$, $\frac{m}{\sqrt{p}}$, $\frac{k}{\sqrt{p}}$, l, p, Π^{dD}, I^d)

Ensure: square $m \times n$ matrix $C = A \cdot B$ distributed so that $\Pi^{dD}[I]$ owns $\frac{m}{\sqrt{p}} \times \frac{n}{\sqrt{p}}$ block sub-matrix $C[G^{dD\to 2D}[I]]$

Figure 4.4 demonstrates how different network dimensions of a 3-ary 6-cube are used by SD-Cannon. A and B get shifted along 3 of 6 dimensions, so Figure 4.4 records usage along 3 dimensions (corresponding to one of A or B). Each outer product (corresponding to a color) is shifted (each shift corresponds to a circle) along a different dimension at any given step.

Note that the local multiplication call is the same as in Cannon's algorithm. The granularity of the sequential work does not decrease in the SD-Cannon algorithm but only changes its ordering. This is a virtue of splitting into outer-products that accumulate to the same buffer.

The control flow of SD-Cannon is described in Algorithm 4.5.4. The algorithm can be elegantly expressed with one-sided communication since the sends should be asynchronous (puts). Our MPI SD-Cannon code uses one-sided put operations and is compact (a few hundred lines of C).

4.5.2 Analysis

We analyze the communication costs of Cannon's algorithm, SUMMA, and SD-Cannon. We consider bandwidth cost, as the total volume of data sent by each process, and latency cost, as the number of messages sent by each process. As before, we assume a l-ary d-cube bidirectional torus network.

Bandwidth Cost

We can analyze the bandwidth cost of these algorithms by the embedding of the algorithm onto the physical l-ary d-cube network. The bandwidth cost of the algorithm is proportional to the number of shifts along the critical path.

In traditional Cannon's algorithm we shift $2\sqrt{p}$ blocks along the critical path ($\sqrt{p}$ times for stagger and $\sqrt{p}$ times for contraction) of size mk/p and nk/p. Given the ordering of the embedding, we can always find a link which has to communicate $mk/\sqrt{p}$ values and a link that has to

communicate $nk/\sqrt{p}$ values.[1] Therefore the bandwidth cost is

$$W_{\text{Cannon}} = O\left(\frac{\max(m,n)\cdot k}{\sqrt{p}}\right).$$

In the bidirectional, split-dimensional algorithm, all shifts in the communication inner loops (in Algorithm 4.5.2 and Algorithm 4.5.3) can be done simultaneously (so long as the network router can achieve full injection bandwidth). So the communication cost is simply proportional to the number of inner loops, which, for staggering (Algorithm 4.5.2) is $N_T = l \cdot d/2$. For the recursive contraction step (Algorithm 4.5.3), the number of these shift stages is

$$N_S = \sum_{i=1}^{d/2} l^i \leq 2\sqrt{p}.$$

If the network is bidirectional, at each shift stage we send asynchronous messages of sizes $mk/(d \cdot p)$ and $kn/(d \cdot p)$ values. Ignoring the lower-order stagger term in SD-Cannon we have a cost of

$$W_{\text{SD-Cannon}} = O\left(\frac{\max(m,n)\cdot k}{d\cdot\sqrt{p}}\right).$$

So the bandwidth cost of SD-Cannon, $W_{\text{SD-Cannon}}$, is d times lower than that of Cannon's algorithm, W_{Cannon}. In SUMMA, throughout the algorithm A of size mk and B of size kn are multicast along two different directions. An optimal multicast algorithm would utilize d links for the multicasts of A and B respectively. So, the bandwidth cost of SUMMA is

$$W_{\text{SUMMA}} = O\left(\frac{\max(m,n)\cdot k}{d\cdot\sqrt{p}}\right),$$

which is asymptotically the same as the bandwidth cost of SD-Cannon.

Latency Cost

The latency overhead incurred by these algorithms will differ depending on the topology and collective algorithms for SUMMA. The SD-Cannon algorithm sends more messages than Cannon's algorithm, but into different links, so it incurs more sequential and DMA overhead, but no extra network latency overhead. However, both Cannon's algorithm and SD-Cannon will have a lower latency cost than SUMMA on a typical network. In each step of SUMMA, multicasts are done along each dimension of the processor grid. So, on a torus network, a message must travel $l \cdot d/2$ hops at each step, rather than 1 hop as in SD-Cannon. The Blue Gene/P machine provides efficient multicast collectives that work at a fine granularity and incur little latency overhead [57].

[1] This analysis does not consider the wrap around pass. Some messages might have to go multiple hops, but this problem is relieved by interleaving the ordering of the processor grid.

However, on a machine without this type of topology-aware collectives, SD-Cannon would have a strong advantage, as messages would need to travel fewer hops.

If we count latency as the number of hops a message must travel on the network and assume processes can send multiple messages at once, we can derive definite latency costs. For Cannon's algorithm, which does $O(\sqrt{p})$ near neighbor sends, the latency cost is unambiguously

$$S_{\text{Cannon}} = O\left(\sqrt{p}\right).$$

For SD-Cannon, if we assume messages can be sent simultaneously into each dimension, the latency cost is

$$S_{\text{SD-Cannon}} = O\left(\sqrt{p}\right).$$

However, the on-node messaging overhead goes up by a factor of $O(d)$. For SUMMA, there are again $\sqrt{p}$ steps, but at each step a multicast happens among $\sqrt{p}$ processes. On a torus network, the most distant processor would be $l \cdot d/2$ hops away, giving a hop-messaging cost of

$$S_{\text{SUMMA}} = O\left(l \cdot d \cdot \sqrt{p}\right).$$

This latency overhead is higher than Cannon and SD-Cannon, though this cost reflects the number of hops travelled not the number of synchronizations. However, generally it is reasonable to state that a multicast incurs a larger latency overhead than near-neighbor sends, so our qualitative conclusion is valid.

4.5.3 Results

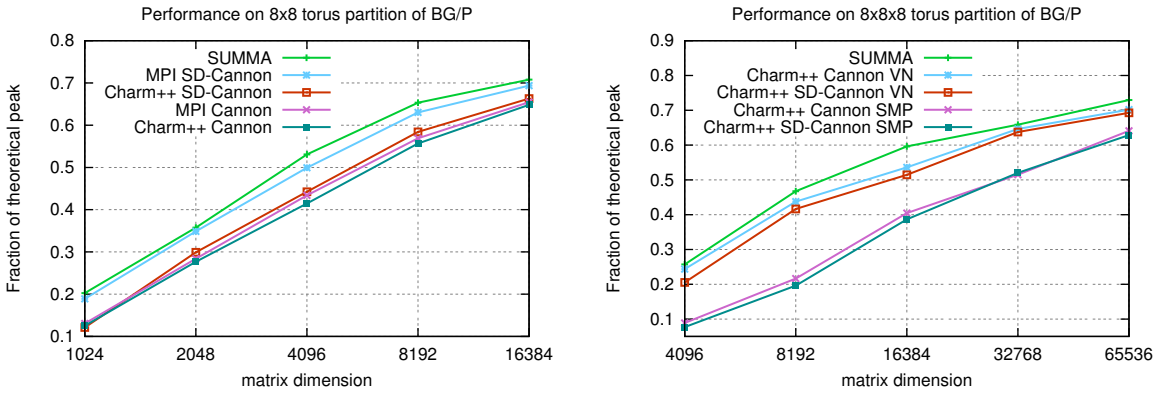
We implemented version of SD-Cannon in MPI [71] and Charm++ [95]. Both versions work on matrices of any shape and size, but we only benchmark square matrices. Both versions assume the virtual decomposition is a k-ary n-cube. In MPI, the process grid is a k-ary n-cube, while in Charm++ we get a k-ary n-cube of chare objects. We use Charm++ to explicitly map the objects onto any unbalanced process grid we define at run time. While we explicitly define the mapping function to fold the chare array onto a smaller processor grid, the Charm++ run-time system manages how the sequential work and messaging get scheduled.

The MPI version uses MPI put operations for communication and barriers for synchronization. The Charm++ version uses the underlying run-time system for messaging between chares, and is dynamically scheduled (no explicit synchronization).

We benchmarked our implementations on a Blue Gene/P (BG/P) [86] machine located at Argonne National Laboratory (Intrepid). We chose BG/P as our target platform because it uses few cores per node (four 850 MHz PowerPC processors) and relies heavily on its interconnect (a bidirectional 3D torus with 375 MB/sec of achievable bandwidth per link).

Since the BG/P network only has three dimensions, the benefit of SD-Cannon is limited to trying to exploit the backwards links and the third dimensional links. The MPI version of our code is limited to even-dimensional tori, so it could only exploit 4 of 6 links. We study relative performance of the MPI version of SD-Cannon on an 8-by-8 torus partition of BG/P. [2]

[2]The partitions allocated by the BG/P scheduler are only toroidal if they have 512 or more nodes. So, we allocated a 512 node partition and worked on the bottom 64 node slice.



(a) Matrix multiplication on a 2D torus partition (b) Matrix multiplication on a 3D torus partition

Figure 4.5: Performance of SD-Cannon on BG/P

Figure 4.5(a) details the performance of MPI versions of SUMMA, Cannon's algorithm, and SD-Cannon on an 8x8 node 2D torus partition. SD-Cannon improves upon Cannon's algorithm as it can utilize the backwards as well as the forwards links simultaneously. The one-dimensional rectangular multicasts used by SUMMA achieve the same effect. We see that the performance of SD-Cannon is higher than Cannon's algorithm (up to 1.5x) and slightly worse than SUMMA. The performance difference between SUMMA and SD-Cannon is due to the extra cost of the initial stagger, which SUMMA does not need. The Charm++ versions were executed with 1 chare per node. In this case, we see that Charm++ has a small overhead and we see a small benefit from bidirectionality.

Figure 4.5(b) shows the performance on a 3D 512 node partition. Since this partition is odd-dimensional, we cannot efficiently map the MPI version of Cannon or SD-Cannon. We execute the Charm++ codes in two modes, one with 1 process per node and 8 chares per process (SMP), and one with 4 processes per node and 2 chares per process (VN). Using multiple processes per node improves the performance of the Charm++ codes, because it is more efficient to perform a separate multiplication on each core, rather than execute each multiplication in sequence across all cores. While the VN-mode version performs almost as well as SUMMA, neither version benefits from multidimensional shifts. Our Charm++ implementations use two-sided communication, while the MPI version uses one-sided. It is likely that a Charm++ implementation with one-sided sends would successfully exploit all of the links.

4.5.4 Conclusions

SD-Cannon is an improvement on top of Cannon's algorithm. While Cannon's algorithm has some nice properties, SUMMA has seen more wide-spread adoption. In this chapter, we demonstrate how SD-Cannon can get closer to the performance of the SUMMA algorithm, and how virtualization can be used to map SD-Cannon and Cannon's algorithm efficiently. On the Blue Gene

hardware it still does not make sense to use SD-Cannon over SUMMA, but SD-Cannon has advantages that could prove to be faster than SUMMA on other hardware. In particular, on networks without optimized collectives or with higher latency cost, the near-neighbor sends performed by Cannon's algorithm and SD-Cannon are preferable to SUMMA's multicasts.

More generally, our work demonstrates how algorithmic design can couple with topology-aware mapping and virtualization. These techniques are already important on modern supercomputers with 3D interconnects as demonstrated by our performance results. As the scale and dimensionality of high performance networks grow, topology-aware mapping and communication-avoidance are becoming pivotal to scalable algorithm design.

Chapter 5

Solving Dense Linear Systems of Equations

We now consider dense linear algebra algorithms with more complicated dependency structures than matrix multiplication, namely triangular solve and Gaussian elimination. We do not consider the use of Strassen-like matrix multiplication algorithms within Gaussian elimination. While the depth (longest path) in the dependency graph of computations for matrix multiplication is logarithmic (due to the sums) in matrix dimension, the longest dependency path in the triangular solve and Gaussian elimination algorithms is proportional to the matrix dimension. Its known that such dependency structure limits the amount of available parallelism and necessitates synchronization. Our lower bounds generalize this tradeoff by quantifying tradeoffs between computation cost, communication cost, and synchronization cost.

In Section 5.1, we derive lower bounds for the triangular dense matrix solve with a single vector, which are the same as previously known diamond DAG tradeoffs [124, 156], but hold for a class of different possible algorithms for the triangular solve. We then by analogous technique derive tradeoffs for Cholesky factorization, which state that for the factorization of any n-by-n symmetric matrix, the computation cost F and the synchronization cost S must be

$$F_{\mathrm{Ch}} \cdot S_{\mathrm{Ch}}^2 = \Omega(n^3).$$

This lower bound means that in order to lower computation cost F (called 'time' in Papadimitriou and Ullman [124]), a larger number of synchronizations S must be done. Our bounds also yield an additional tradeoff between communication cost W and synchronization cost S, which is

$$W_{\mathrm{Ch}} \cdot S_{\mathrm{Ch}} = \Omega(n^2).$$

This lower bound implies that in order to decrease the communication cost of the algorithm W (per-processor cost along most expensive execution path), the number of synchronizations S must go up.

In Section 5.2, we give a triangular solve algorithm that is optimal according to our communication lower bounds. Our algorithm is a wave-front algorithm and is similar to the TRSV algorithm

The lower bounds in this chapter are based on joint work with Erin Carson and Nicholas Knight [145].

given by Heath [78]. The wave-front approach provides a theoretically lower synchronization cost to the TRSV algorithm with respect to approaches which employ collective communication.

We extend the idea of using auxiliary memory as done in 2.5D matrix multiplication in Chapter 4 to Cholesky and LU factorization in Section 5.3. The 2.5D LU algorithm can lower the communication cost by a factor of up to $p^{1/6}$ with respect to the standard 2D approach, but requires more synchronization. In fact, the 2.5D algorithm attains the lower bound tradeoff on interprocessor communication and synchronization cost, for a range of these costs. In Section 5.4 we show how 2.5D LU can be combined with tournament pivoting [70]. This provides an alternative approach to Tiskin's LU with pairwise pivoting [158], which achieves the same asymptotic communication cost to 2.5D LU. Tournament pivoting has been shown to be somewhat more stable than pairwise pivoting in practice [70] although it has worse theoretical stability upper bounds [45, 149] Our approach is also more adaptable to tall and skinny (rectangular matrices) than the panel embedding used by [158].

We detail how the 2.5D approach may be adapted to handle symmetric matrices in Cholesky in Section 5.5. We also give an algorithm of the same cost for the triangular solve with many vectors (TRSM). These algorithms combined allow QR to be done with 2.5D cost via the Cholesky-QR algorithm.

In Section 5.6, we provide implementation results of 2.5D LU factorization on a BlueGene/P supercomputer. We analyze the performance of the algorithm with two different types of collective communication methods, binomial trees and rectangular (torus topology) trees. We also give projected results with the two types of collectives on a hypothetical exascale supercomputer.

The rest of the chapter is organized as follows

- Section 5.1 gives lower bounds on tradeoffs between computation, communication, and synchronization for TRSV and Cholesky,
- Section 5.2 details a triangular solve algorithm that attains the lower bounds,
- Section 5.3 introduces a 2.5D LU algorithm that attains the communication lower bounds,
- Section 5.4 describes an algorithm for 2.5D LU with tournament pivoting,
- Section 5.5 adapts 2.5D LU to perform 2.5D Cholesky and 2.5D TRSM
- Section 5.6 presents measured and projected performance results for 2.5D LU with and without pivoting.

5.1 Lower Bounds for Triangular Solve and Cholesky

We now apply the dependency expansion analysis Section 3.4 to obtain lower bounds on the costs associated with a few specific dense numerical linear algebra algorithms. Our analysis proceeds by obtaining lower bounds on the minimum-cut size of a parent lattice hypergraph corresponding to the computation. By employing parent hypergraphs, we express all possible reduction tree

summation orders. In particular, let $T = (R, E)$ be a tree in a dependency graph which sums a set of vertices $S \subset R$, into vertex $\hat{s} \in R$ via intermediate partial sums $R \setminus (S \cup \{\hat{s}\})$. Since T must be connected, we can define a hyperedge in a parent hypergraph $H = (S \cup \{\hat{s}\}, \{S \cup \{\hat{s}\}\})$, corresponding to this reduction tree, whose vertices contain all summands of H and the output and which has a single hyperedge containing all these vertices. We will then use Theorem 3.3.2 to assert that the smallest vertex separator of T is no smaller than the hyperedge cut of H, the minimum size of which for a single reduction tree is simply one as there is only one hyperedge in H. This lower bound will then apply to any reduction tree (any set of intermediate partial sums), since it considers only the summands and the output.

5.1.1 Lower Bounds for the Triangular Solve

First, we consider a parameterized family of dependency graphs G_{TR} associated with an algorithm for the triangular solve (TRSV) operation. In TRSV, we are interested in computing a vector x of length n, given a dense nonsingular lower-triangular matrix L and a vector y, satisfying $L \cdot x = y$, i.e., $\sum_{j=1}^{i} L_{ij} \cdot x_j = y_i$, for $i \in [1, n]$. A sequential TRSV implementation is given in Algorithm 5.1.1. For analysis, we introduced the intermediate matrix Z (which need not be formed

Algorithm 5.1.1 $[x] \leftarrow \mathrm{TRSV}(L, y, n)$

1: **for** $i = 1$ to n **do**
2: **for** $j = 1$ to $i - 1$ **do**
3: $Z_{ij} = L_{ij} \cdot x_j$
4: $x_i \leftarrow \left(y_i - \sum_{j=1}^{i-1} Z_{ij}\right) / L_{ii}$

explicitly in practice), and corresponding intermediate 'update' vertices $\{Z_{ij} : i \in [1, n], j \in [1, n], j < i\}$. We see that the computation of Z_{ij} for $i = [2, n]$ and some $j < i$ is a dependency of x_i, which is itself a dependency of the computation of Z_{ki} for all $k > i$.

For fixed n, alternative orders exist for the summation on line 4, leading to multiple dependency graphs $G_{\mathrm{TR}} = (V'_{\mathrm{TR}}, E'_{\mathrm{TR}})$. However, the set of vertices, V'_{TR}, defined by any dependency graph for this algorithm, must contain a vertex for each entry of Z and each entry of x (the variability comes from the partial sums computed within the summation). Further, any order of this summation must eventually combine all partial sums; therefore, the vertices corresponding to the computation of each x_i, i.e., Z_{ij} for all $j \in [1, i-1]$, must all be weakly connected by some reduction tree. We will define a $(2, 1)$-lattice hypergraph (with indices in reverse order with respect to the definition in Section 3.3) of breadth n, $H_{\mathrm{TR}} = (V_{\mathrm{TR}}, E_{\mathrm{TR}})$, which is a parent hypergraph for any possible G_{TR}, as we demonstrate below. This hypergraph contains the following vertices and hyperedges,

$$V_{\mathrm{TR}} = \{Z_{ij} : i \in [2, n], j \in [1, i-1]\}$$
$$E_{\mathrm{TR}} = \{e_i : i \in [1, n]\}, \quad \text{where } e_i = \{Z_{ij} : j \in [1, i-1]\} \cup \{Z_{ki} : k \in [i+1, n]\}.$$

We note that the vertices V_{TR} must be present in any G_{TR}, however, the vertices corresponding to x, y, and any intermediate partial summations done in reduction trees are omitted in H_{TR}. To show

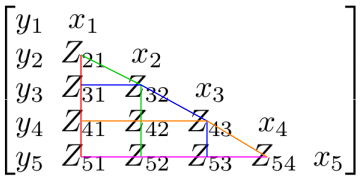


Figure 5.1: Depiction of the hypergraph $H_{\text{TR}}(5)$ along with the inputs and outputs; each line of a different color corresponds to a hyperedge.

that the H_{TR} is a parent hypergraph of any G_{TR}, we need to demonstrate that each hyperedge in H_{TR} corresponds to a unique part (sub-hypergraph) of any G_{TR}. The hyperedges E_{TR} can be enumerated with respect to either vector x or y; the ith hyperedge $e_i \in E_{\text{TR}}$ includes all intermediate values on which x_i depends (Z_{ij} for $j \in [1, i-1]$), or which depend on x_i (Z_{ki} for $k \in [i+1, n]$). Therefore, each hyperedge e_i consists of vertices which are part of a connected component in any G_{TR} that corresponds to a unique set of edges: those in the reduction tree needed to form x_i from each Z_{ij} and those between x_i and each Z_{ki}. This hypergraph is depicted in Figure 5.1.

Lemma 5.1.1. *Any $\frac{1}{q}$-$\frac{1}{x}$-balanced vertex separator of Z (V_{TR}) in any dependency graph G_{TR} which subdivides the $n(n-1)/2$ vertices V_{TR} for any real numbers $4 \leq x \leq q \ll n$ must have size at least*

$$\chi_{q,x}(G_{\text{TR}}) = \Omega\left(n/\sqrt{q}\right).$$

Proof. Any G_{TR} has the parent hypergraph H_{TR}, which we defined above. The maximum vertex degree of the parent hypergraph H_{TR} of G_{TR} is 2, so by application of Theorem 3.3.2, the minimum vertex separator of G_{TR} is at least half the size of the minimum hyperedge cut of H_{TR}. By Theorem 3.3.1, any $\frac{1}{q}$-$\frac{1}{x}$-balanced hyperedge cut of a $(2,1)$-lattice hypergraph of breadth n with $2 \leq x \leq q \ll n$ is of size $\Omega(n/\sqrt{q})$. Therefore, any vertex separator must be of size at least $\chi_{q,x}(G_{\text{TR}}) = \Omega(n/\sqrt{q})$. □

Theorem 5.1.2. *Any parallelization of any dependency graph G_{TR} where some processor computes no more than $\frac{1}{x}$ of the elements in Z and at least $\frac{1}{q}$ elements in Z, (for any $4 \leq x \leq q \ll n$) must incur a communication cost of*

$$W_{\text{TR}} = \Omega\left(n/\sqrt{q}\right).$$

Proof. Consider any dependency graph G_{TR} for Algorithm 5.1.1. Every vertex in G that has an outgoing edge to a vertex computed by a different processor (different color) must be communicated. Since some processor computes no more than $\frac{1}{x}$ of the elements in Z and at least $\frac{1}{q}$ elements in Z, the communicated set can be bounded below by the size of a $\frac{1}{q}$-$\frac{1}{x}$-balanced vertex separator of Z within G_{TR}. By application of Lemma 5.1.1, the size of any such separator is at least $\Omega(n/\sqrt{q})$. □

Theorem 5.1.3. *Any parallelization of any dependency graph G_{TR} where some processor computes no more than $\frac{1}{x}$ of the elements in Z and at least $\frac{1}{q}$ elements in Z for $4 \leq x \leq q \ll n$, incurs the following computation (F), bandwidth (W), and latency (S) costs, for some $b \in [1, n]$,*

$$F_{\mathrm{TR}} = \Omega\left(n \cdot b\right), \qquad W_{\mathrm{TR}} = \Omega\left(n\right), \qquad S_{\mathrm{TR}} = \Omega\left(n/b\right),$$

and furthermore, $F_{\mathrm{TR}} \cdot S_{\mathrm{TR}} = \Omega\left(n^2\right)$.

Proof. Consider any dependency graph $G_{\mathrm{TR}} = (V'_{\mathrm{TR}}, E'_{\mathrm{TR}})$ for Algorithm 5.1.1. We note that the computation of $x_i \subset V'_{\mathrm{TR}}$ for $i \in [1, n]$ requires the computation of Z_{jk} for $j, k \in [1, i]$ with $k < j$. Furthermore, no element Z_{lm} for $l, m \in [i+1, n]$ with $l > m$ may be computed until x_i is computed. Consider any subpath $\mathcal{R} \subset \mathcal{P}$ of the dependency path $\mathcal{P} = \{x_1, \ldots, x_n\}$. We recall that the bubble $\zeta(G_{\mathrm{TR}}, \mathcal{R}) = (V_\zeta, E_\zeta)$ around $\mathcal{R}$ is the set of all computations that depend on an element of $\mathcal{R}$ or influence an element of $\mathcal{R}$. For any $\mathcal{R} = \{x_i, \ldots, x_j\}$, the bubble includes vertices corresponding to a subtriangle of Z, namely, $Z_{kl} \in V_\zeta$ for $k, l \in [i, j]$ with $l < k$. Therefore, $\zeta(G_{\mathrm{TR}}, \mathcal{R})$ is isomorphic to some $G_{\mathrm{TRSV}}(|\mathcal{R}|)$, which implies that $|V_\zeta| = \Theta(|\mathcal{R}|^2)$ and by Lemma 5.1.1, we have a lower bound on its vertex separator size, $\chi_{q,x}(\zeta(G_{\mathrm{TR}}, \mathcal{R})) = \Omega(|\mathcal{R}|/\sqrt{q})$. Since the bubbles for TRSV are 2-dimensional we apply Corollary 3.4.2 with $d = 2$ to obtain the conclusion, for some $b \in [1, n]$. □

5.1.2 Cholesky Factorization

In this section, we show that the Cholesky factorization algorithm has 3-dimensional bubble-growth and dependency graphs which satisfy the path expansion properties necessary for the application of Corollary 3.4.2 with $d = 3$. We consider factorization of a symmetric positive definite matrix via Cholesky factorization. We show that these factorizations of n-by-n matrices form an intermediate 3D tensor Z such that $Z_{ijk} \in Z$ for $i > j > k \in [1, n]$, and Z_{ijk} depends on each Z_{lmn} for $l > m > n \in [1, j-1]$. We note that fast matrix multiplication techniques such as Strassen's algorithm [152], compute a different intermediate and are outside the space of this analysis.

The Cholesky factorization of a symmetric positive definite matrix A, $A = L \cdot L^T$, results in a lower triangular matrix L. A simple sequential algorithm for Cholesky factorization is given in Algorithm 5.1.2. We introduced an intermediate tensor Z, whose elements must be computed

Algorithm 5.1.2 $[L] \leftarrow \mathrm{Cholesky}(A, n)$

```
1: for j ← 1 to n do
2:     L_jj = sqrt(A_jj − Σ_{k=1}^{j−1} L_jk · L_jk)
3:     for i ← j + 1 to n do
4:         for k = 1 to j − 1 do
5:             Z_ijk = L_ik · L_jk
6:         L_ij = (A_ij − Σ_{k=1}^{j−1} Z_ijk)/L_jj
```

during any execution of the Cholesky algorithm, although Z itself need not be stored explicitly

in an actual implementation. We note that the Floyd-Warshall [59, 167] all-pairs shortest-paths graph algorithm has the same dependency structure as Cholesky for undirected graphs (and Gaussian Elimination for directed graphs), so our lower bounds may be easily extended to this case. See Chapter 9 for more details on this extension. However, alternative algorithms, in particular the 'augmented' path-doubling algorithm [157], are capable of solving the all-pairs shortest-paths problem with the same asymptotic communication and computation costs as matrix multiplication. While regular path-doubling exploits the fact that all shortest paths are made up of a series of shortest-paths of up to a given length, the technique in [157] employs the idea that all shortest paths are made of a series of shortest paths of exactly a certain length k and a small shortest path of length less than k, where the length is the number of edges in the path. Tiskin's approach is not immediately extensible to numerical Gaussian elimination due to its exploitation of the structure of shortest paths.

We note that the summations in lines 2 and 6 of Algorithm 5.1.2 can be computed via any summation order (and will be computed in different orders in different parallel algorithms). This implies that the summed vertices are connected in any dependency graph $G_{\text{Ch}} = (V'_{\text{Ch}}, E'_{\text{Ch}})$, but the connectivity structure may be different. Further, we know that Z and L must correspond to vertices in V_{Ch} in any G_{Ch}. We define a $(3, 2)$-lattice hypergraph (with indices in reverse order with respect to the definition in Section 3.3) $H_{\text{Ch}} = (V_{\text{Ch}}, E_{\text{Ch}})$, which is a parent hypergraph of G_{Ch} for the algorithm which allows us to obtain a lower bound for any possible summation order, as

$$\begin{aligned}
V_{\text{Ch}} =& \{Z_{ijk} : i, j, k \in [1, n], i > j > k\}, \\
E_{\text{Ch}} =& \{e_{ij} \quad : i, j \in [1, n] \text{ with } i > j\} \text{ where} \\
e_{ij} =& \{Z_{ijk} : k \in [1, j-1]\} \\
& \cup \{Z_{ikj} : k \in [j+1, i-1]\} \\
& \cup \{Z_{kij} : k \in [i+1, n]\}.
\end{aligned}$$

We note that the vertices V_{Ch} must be present in any G_{Ch}, however, the vertices corresponding to L and any intermediate partial summations done in reduction trees are omitted in H_{Ch}. To show that the H_{Ch} is a parent hypergraph of any G_{Ch}, we need to demonstrate that each hyperedge in H_{Ch} corresponds to a unique part (sub-hypergraph) of any G_{Ch}. The hyperedges E_{Ch} can be enumerated with respect to L. Hyperedge $e_{ij} \in E_{\text{Ch}}$ includes all intermediate values on which L_{ij} depends (Z_{ijk} for $k \in [1, i-1]$), or which depend on L_{ij} (Z_{ikj} for $k \in [j+1, i-1]$ and Z_{kij} for $k \in [i+1, n]$). Therefore, each hyperedge e_{ij} consists of vertices which are part of a connected component in any G_{Ch} that corresponds to a unique set of edges: those in the reduction tree needed to form L_{ij} from each Z_{ijk} and those between L_{ij} and each Z_{ikj} as well as Z_{kij}.

We also define hyperplanes x_i for $i \in [1, n]$, where

$$x_i = e_{i,1} \cup e_{i,2} \cup \cdots \cup e_{i,i-1} \cup e_{i+1,i} \cup e_{i+2,i} \cup \cdots \cup e_{n,i}.$$

Figure 5.2 shows hyperplane x_{12} and hyperedge $e_{12,6}$ on $H_{\text{Ch}}(16)$. These hyperplanes correspond to those in the proof of Theorem 3.3.1.

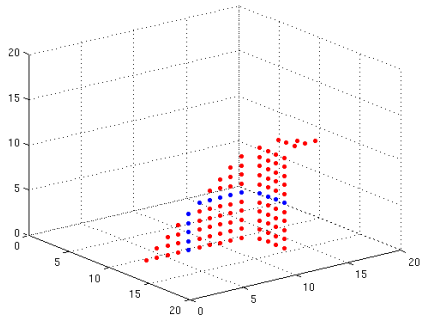


Figure 5.2: A hyperplane (red) and a hyperedge (blue) in $H_{\text{Ch}}(16)$.

Lemma 5.1.4. *Any $\frac{1}{q}$-$\frac{1}{x}$-balanced vertex separator S of Z within dependency graph G_{Ch} for any real numbers $4 \leq x \leq q \ll n$ must have size at least*

$$\chi_q(G_{\text{Ch}}) = \Omega\left(n^2/q^{2/3}\right).$$

Proof. The maximum vertex degree of the parent hypergraph H_{Ch} of G_{Ch} is 3, so by application of Theorem 3.3.2, the minimum size of a $\frac{1}{q}$-$\frac{1}{x}$-balanced vertex separator of G_{Ch} is at least one third that of a $\frac{1}{q}$-$\frac{1}{x}$-balanced hyperedge cut of H_{Ch}. By Theorem 3.3.1 with $m = 3$ and $r = 2$, any $\frac{1}{q}$-$\frac{1}{x}$-balanced hyperedge cut with $4 \leq x \leq q \ll n$ of H_{Ch} is of size $\Omega(n^2/q^{2/3})$. Therefore, any vertex separator of G_{Ch} must be of size at least $\chi_{q,x}(G_{\text{Ch}}) = \Omega(n^2/q^{2/3})$. □

Theorem 5.1.5. *Any parallelization of any dependency graph G_{Ch}, where some processor computes no more than $\frac{1}{x}$ of the elements in Z (V_{Ch}) and at least $\frac{1}{q}$ elements in Z, for any $4 \leq x \leq q \ll n$, must incur a communication of*

$$W_{\text{Ch}} = \Omega\left(n^2/q^{2/3}\right).$$

Proof. For any G_{Ch}, every vertex that has an outgoing edge to a vertex computed by a different processor (different color) must be communicated (is in the communicated set). Since some processor computes no more than $\frac{1}{x}$ of the elements in Z (V_{Ch}) and at least $\frac{1}{q}$ elements in Z, for any $4 \leq x \leq q \ll n$, communicated set can be bounded below by the size of a $\frac{1}{q}$-$\frac{1}{x}$-balanced vertex separator of Z in G_{Ch}. By Lemma 5.1.4, the size of any such separator is $\Omega(n^2/q^{2/3})$. □

Theorem 5.1.6. *Any parallelization of any dependency graph G_{Ch} in which some processor computes no more than $\frac{1}{x}$ of the elements in Z (V_{Ch}) and at least $\frac{1}{q}$ elements in Z, for any $4 \leq x \leq q \ll n$, incurs the following computation (F), bandwidth (W), and latency (S) costs, for some $b \in [1, n]$,*

$$F_{\text{Ch}} = \Omega\left(n \cdot b^2\right), \quad W_{\text{Ch}} = \Omega\left(n \cdot b\right), \quad S_{\text{Ch}} = \Omega\left(n/b\right),$$

and furthermore, $F_{\text{Ch}} \cdot S_{\text{Ch}}^2 = \Omega\left(n^3\right)$, $W_{\text{Ch}} \cdot S_{\text{Ch}} = \Omega\left(n^2\right)$.

Proof. Consider any dependency graph $G_{\text{Ch}} = (V'_{\text{Ch}}, E'_{\text{Ch}})$. We note that the computation of $L_{ii} \in V'_{\text{Ch}}$ for $i \in [1, n]$ requires the computation of $Z_{lmk} \in V_{\text{Ch}} \subset V'_{\text{Ch}}$ for $l, m, k \in [1, i]$ with $l > m > k$. Furthermore, no element $Z_{srt} \in V_{\text{Ch}} \subset V'_{\text{Ch}}$ for $s, r, t \in [i+1, n]$ with $s > r > t$ can be computed until L_{ii} is computed. Consider any subpath $\mathcal{R} \subset \mathcal{P}$ of the dependency path $\mathcal{P} = \{L_{11}, \ldots, L_{nn}\}$. Evidently, if $\mathcal{R} = \{L_{ii}, \ldots, L_{j+1,j+1}\}$, the bubble $\zeta(G_{\text{Ch}}, \mathcal{R}) = (V_\zeta, E_\zeta)$ includes vertices corresponding to a subcube of Z, namely $Z_{klm} \in V_\zeta$ for $k, l, m \in [i, j]$ with $k > l > m$. Therefore, $\zeta(G_{\text{Ch}}, \mathcal{R})$ is isomorphic to some $G_{\text{Ch}}(|\mathcal{R}|)$, which implies that $|V_\zeta| = \Theta(|\mathcal{R}|^3)$ and by Lemma 5.1.4, we have $\chi_{q,x}(\zeta(G_{\text{Ch}}, \mathcal{R})) = \Theta(|\mathcal{R}|^2/q^{2/3})$. Since we have 3-dimensional bubbles with 2-dimensional cross-sections, we apply Corollary 3.4.2 with $d = 3$ to obtain the conclusion, for some $b \in [1, n]$. □

We conjecture that the lower bounds demonstrated for Cholesky in this section, also extend to LU, LDL^T, and QR factorizations, as well as algorithms for computation of the eigenvalue decomposition of a symmetric matrix and the singular value decomposition.

5.2 Parallel Algorithms for the Triangular Solve

The lower bounds presented above for triangular solve, are attained by the communication-efficient blocked schedule suggested in [124]. We give an algorithm specifically for the triangular solve problem, Algorithm 5.2.1, which uses this blocking schedule with blocking factor b to compute the triangular solve. Our algorithm is similar to the wavefront algorithm given by [78].

The parallel Algorithm 5.2.1 can be executed using $p = n/b$ processors. Let processor p_l for $l \in [1, n/b]$ initially own L_{ij}, y_j for $i \in [1, n]$, $j \in [(l-1)b + 1, lb]$. Processor p_l performs parallel loop iteration l at each step of Algorithm 5.2.1. Since it owns the necessary panel of L and vector part x_j, no communication is required outside the vector send/receive calls listed in the code. So at each iteration of the outer loop at least one processor performs $O(b^2)$ work, and $2b$ (each sent vector is of length b) data is sent, requiring 2 messages. Therefore, this algorithm achieves the following costs over the n/b iterations,

$$F_{\text{TR}} = O(nb), \quad W_{\text{TR}} = O(n), \quad S_{\text{TR}} = O(n/b),$$

which attains our communication lower bounds in Theorems 5.1.2 and 5.1.3, for any $b \in [1, n]$. Parallel TRSV algorithms in current numerical libraries such as Elemental [131] and ScaLAPACK [28] employ algorithms that attain our lower bound, up to an $O(\log(p))$ factor on the latency cost, due to their use of collectives for communication rather than the point-to-point communication in our wavefront TRSV algorithm.

5.3 2.5D LU without Pivoting

2D parallelization of LU typically factorizes a vertical and a top panel of the matrix and updates the remainder (the Schur complement). The dominant cost in such a parallel LU algorithm is the

Algorithm 5.2.1 $[x] \leftarrow \text{TRSV}(L, y, n)$

1: $x = y$
2: **for** $k = 1$ to n/b **do**
3: % Each processor p_l executes a unique iteration of the following loop
4: **for** $l = \max(1, 2k - n/b)$ to k **do**
5: **if** $l > 1$ **then** Receive vector $x[(2k - l - 1)b + 1 : (2k - l)b]$ from processor p_{l-1}
6: **for** $i = (2k - l - 1)b + 1$ to $(2k - l)b$ **do**
7: **for** $j = (l - 1)b + 1$ to $\min(i - 1, lb)$ **do**
8: $x_i \leftarrow (x_i - L_{ij} \cdot x_j)$
9: **if** $k = l$ **then**
10: $x_i = x_i / L_{ii}$
11: **if** $l < n/b$ **then** Send vector $x[(2k - l - 1)b + 1 : (2k - l)b]$ to processor p_{l+1}
12: % Each processor p_l executes a unique iteration of the following loop
13: **for** $l = \max(1, 2k + 1 - n/b)$ to k **do**
14: **if** $l > 1$ **then** Receive vector $x[(2k - l)b + 1 : (2k - l + 1)b]$ from processor p_{l-1}
15: **for** $i = (2k - l)b + 1$ to $(2k - l + 1)b$ **do**
16: **for** $j = (l - 1)b + 1$ to lb **do**
17: $x_i \leftarrow (x_i - L_{ij} \cdot x_j)$
18: **if** $l < n/b$ **then** Send vector $x[(2k - l)b + 1 : (2k - l + 1)b]$ to processor p_{l+1}

update to the Schur complement. Our 2.5D algorithm exploits this by accumulating this update over processor layers. However, in order to factorize each panel we must reduce the contributions to the Schur complement. We note that only the panel we need to factorize next on needs to be reduced and the remainder can be further accumulated. Even so, to do the reductions efficiently, a block-cyclic layout is required. This layout allows more processors to participate in the reductions and pushes the bandwidth cost down to the lower bound.

The LU latency lower bound (Theorem 5.1.6) dictates that the block size of the LU algorithm must be $O(n/\sqrt{pc})$ to achieve the 2.5D bandwidth lower bound. Further, with this block size, $\Omega(\sqrt{pc})$ messages must be sent. To achieve this latency the matrix should be laid out in a block-cyclic layout with block-size $O(n/\sqrt{pc})$. Algorithm 5.3 (work-flow diagram in Figure 5.3) is a communication optimal LU factorization algorithm for the entire range of $c \in \{1, 2, \ldots, \lfloor p^{1/3} \rfloor\}$.

With block-size $O(n/\sqrt{pc})$, Algorithm 5.3 has a bandwidth cost of $W_{2.5D}^{\text{LU}} = O\left(\frac{n^2}{\sqrt{cp}}\right)$ words and a latency cost of $S_{2.5D}^{\text{MM}} = O\left(\sqrt{cp}\log(p)\right)$ messages. Therefore, it is asymptotically communication optimal for any choice of c (modulo a $\log(p)$ factor for latency). Further, it is also always asymptotically computationally optimal (the redundant work is a low order cost). These costs are derived in Appendix B in [147].

Algorithm 5.3.1 $[L, U]$ = 2D-LU(A,Λ,n,p)

Require: n-by-n matrix A, spread over a $\sqrt{p}$-by-$\sqrt{p}$ grid Λ so that $\Lambda[i,j]$ owns a square block $A[i,j]$ of dimension $n/\sqrt{p}$

1: **for** $t = 1$ to $t = \sqrt{p}$ **do**
2: Factorize sequentially [$L[t,t]$,$U[t,t]$] = LU($A[t,t]$)
3: Replicate $U[t,t]$ on column $\Lambda[:,t]$
4: Compute $L[t+1:\sqrt{p},t] := A[t+1:\sqrt{p},t]/U[t,t]$
5: Replicate $L[t,t]$ on row $\Lambda[t,:]$
6: Compute $U[t+1:\sqrt{p},t] := A[t+1:\sqrt{p},t]/U[t,t]$
7: Replicate $L[t+1:\sqrt{p},t]$ on columns of $\Lambda[:,:]$
8: Replicate $U[t,t+1:\sqrt{p}]$ on rows of $\Lambda[:,:]$
9: Compute $A[t+1:\sqrt{p},t+1:\sqrt{p}] := A[t+1:\sqrt{p},t+1:\sqrt{p}]$
$-L[t+1:\sqrt{p},t] \cdot U[t,t+1:\sqrt{p}]$

Ensure: triangular n-by-n matrices L, U such that $A = L \cdot U$ distributed so that $L[i,j]$ and $U[i,j]$ reside on $\Lambda[i,j]$ for $i \geq j$ and $j \geq i$, respectively

Algorithm 5.3.2 $[X]$ = 2D-TRSM(B,U,Λ,n,p)

Require: n-by-n matrix B, spread over a square grid Λ. n-by-n upper-triangular matrix U spread over a square grid Λ.

1: $X := B$
2: % *T*he following loop iterations are pipelined
3: **for** $t = 1$ to $t = n$ **do**
4: Replicate $U[t,t]$ on column $\Lambda[:,t]$
5: Compute $X[:,t] := X[:,t]/U[t,t]$
6: Replicate $X[:,t]$ on columns of $\Lambda[:,:]$
7: Replicate $U[t,t+1:n]$ on rows of $\Lambda[:,:]$
8: Compute $X[:,t+1:n] := X[:,t+1:n] - X[:,t] \cdot U[t,t+1:n]$

Ensure: n-by-n matrix X, such that $X \cdot U = B$ and X is spread over Λ in place of B

5.4 2.5D LU with Pivoting

Regular partial pivoting is not latency optimal because it requires $\Omega(n)$ messages if the matrix is in a 2D blocked layout. $\Omega(n)$ messages are required by partial pivoting since a pivot needs to be determined for each matrix column which requires communication unless the entire column is owned by one processor. However, tournament pivoting [70], is a new LU pivoting strategy that can satisfy the general communication lower bound. We will show how partial as well as tournament pivoting can be incorporated into our 2.5D LU algorithm.

Tournament pivoting simultaneously determines b pivots by forming a tree of factorizations as follows,

1. Factorize each $2b$-by-b block $[A[2k-1,1], A[2k,1]]^T = P_k^T L_k U_k$ for $k \in [1, \frac{n}{2b}]$ using partial

Algorithm 5.3.3 $[L, U]$ = 2.5D-LU(A,Π,n,p,c)

Require: n-by-n matrix A distributed so that for each l, m, (n/c)-by-(n/c) block $A[l, m]$ is spread over $\Pi[:, :, 1]$.

Replicate A on each $\Pi[:, :, k]$, for $k \in [1, c]$
for $t = 1$ to c **do**
  % Perform in parallel with some layers:
  for $k = 1$ to $c - t$ **do**
    % Factorize top left block:
    $[L[t, t], U[t, t]]$ = 2D-LU($A[t, t]$, $\Pi[:, :, k]$, n/c, p/c)
    % Update left (L) and top (U) panels of A:
    $[L[t + k, t]^T]$ = 2D-TRSM($U[t, t]^T$, $A[t + k, t]^T$, $\Pi[:, :, k]$, n/c, p/c)
    $[U[t, t + k]]$ = 2D-TRSM($L[t, t]$, $A[t, t + k]$, $\Pi[:, :, k]$, n/c, p/c)
    % All-gather panels among processor layers:
    $\Pi[:, :, k]$ broadcasts $L[t + k, t]$ and $U[t, t + k]$ to $\Pi[:, :, k']$ for all k'
  % Perform outer products on whole Schur complement with each processor layer:
  for $k = 1$ to c **do**
    Partition $[T_1, T_2, \ldots T_k] \leftarrow L[t + 1 : c, t]$
    Partition $[W_1^T, W_2^T, \ldots W_k^T]^T \leftarrow U[t, t + 1 : c]$
    $[S'_k]$ = 2D-MM(T_k, W_k^T, $\Pi[:, :, k]$, n/c, $(c - t) \cdot n/c$, $(c - t) \cdot n/c$)
    % Adjust Schur complement:
    $S[t + 1 : c, t + 1 : c, k] = S[t + 1 : c, t + 1 : c, k] + S'_k$
  % Compute next panels via reductions:
  $A[t + 1 : c, t + 1] \leftarrow A[t + 1 : c, t + 1] - \sum_{k'=1}^{c} S[t + 1 : c, t + 1, k']$
  $A[t + 1, t + 2 : c] \leftarrow A[t + 1, t + 2 : c] - \sum_{k'=1}^{c} S[t + 1, t + 2 : c, k']$

Ensure: triangular n-by-n matrices L, U such that $A = L \cdot U$ and for each l, m, (n/c)-by-(n/c) blocks $L[l, m]$, $U[l, m]$ are spread over $\Pi[:, :, 1]$.

pivoting.

2. Write $B_k = P_k[A[2k - 1, 1], A[2k, 1]]^T$, and $B_k = [B'_k, B''_k]^T$. Each B'_k represents the 'best rows' of each sub-panel of A.

3. Now recursively perform steps 1-3 on $[B'_1, B'_2, ..., B'_{n/(2b)}]^T$ until the number of total best pivot rows is b.

For a more detailed and precise description of the algorithm and stability analysis see [70, 69]. To incorporate pivoting into our LU algorithm, the following modifications are required

1. Previously, we did the side panel Tall-Skinny LU (TSLU) via a redundant top block 2D LU-factorization and 2D TRSMs on lower blocks. To do pivoting, the TSLU factorization needs to be done as a whole rather than in blocks. We can still have each processor layer compute a different '2D TRSM block' but we need to interleave this computation with the top block

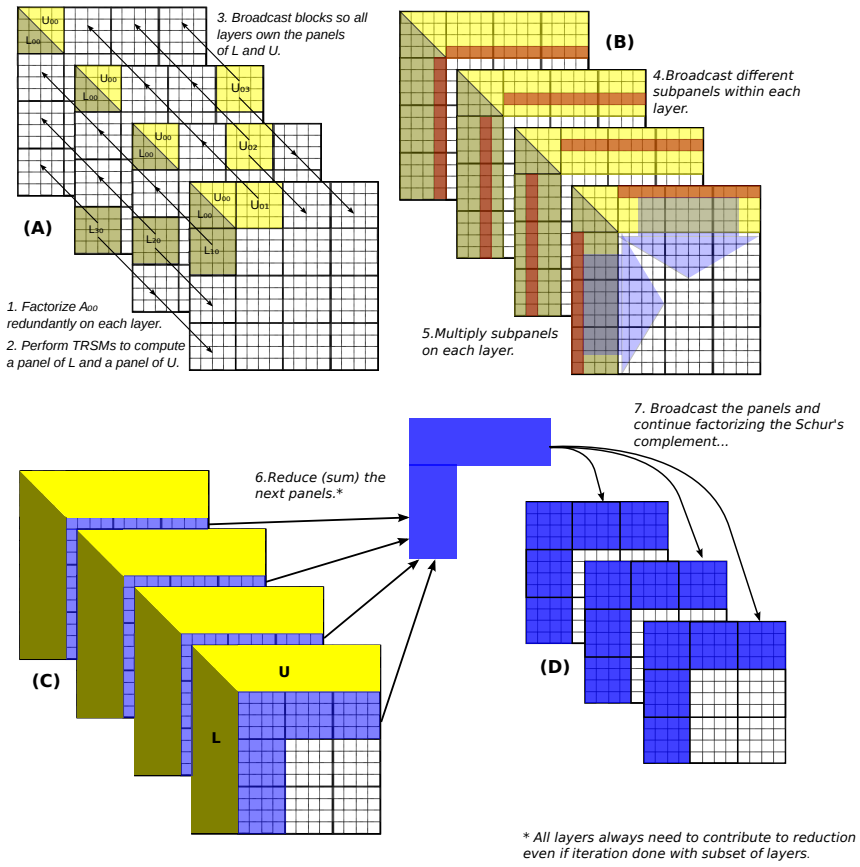


Figure 5.3: 2.5D LU algorithm work-flow

LU factorization and communicate between layers to determine each set of pivots as follows (Algorithm 5.4 gives the full TSLU algorithm),

a) For every small column, we perform pivoting over all layers to determine the best rows. This can be done with partial pivoting on each column or tournament pivoting on blocks of columns.

b) We pivot the rows within the panel on each layer. Interlayer communication is required, since the best rows are spread over the layers (each layer updates a subset of the rows).

c) Each ij processor layer redundantly performs small TRSMs and the Schur complement updates in the top 2D LU block.

d) Each ij processor layer performs TRSMs and updates on its unique subset of rows of the panel.

2. After the TSLU, we need to pivot rows in the rest of the matrix. We do this redundantly on each layer, since each layer will have to contribute to the update of the entire Schur complement.

3. We still reduce the side panel (the one we do TSLU on) at the beginning of each step but we must postpone the reduction of the top panel until pivoting is complete.

Algorithm 5.4.1 $[P, L, U]$ = 2.5D-pivoted-TSLU(A,Π,n,m,p,c)

Require: n-by-m matrix A spread over each square processor layer $\Pi[:, :, k]$ for $k \in [1, c]$, with some block size b.

1: % *Perform in parallel with processor layers:*
2: **for** $k = 1$ to c **do**
3: Let $A_k[:, :] \leftarrow A[k \cdot n/c : (k+1) \cdot n/c, :]$
4: % *Perform with partial or Tournament pivoting*
5: **pipelined**
6: **for** $t = 1$ to m **do**
7: Factorize $A_k[:, t] = P_k \cdot L_k \cdot U[t, t]$ along processor columns
8: Write $R[k] = (P_k^T A_k)[1, t]$
9: Factorize $R = P_R \cdot L_R \cdot U_R$ across processor layers
10: Pivot source row of $(P_R^T R)[1]$ into $A[t, :]$ and $P[:, t]$
11: Compute $A_k[:, t] := A_k[:, t]/A[t, t]$
12: Assign $U[t, t+1 : m] \leftarrow A[t, t+1 : m]$
13: Replicate columns $A_k[:, t]$ on columns of $\Pi[:, :, k]$
14: Replicate $U[t, t+1 : m]$ on rows of $\Pi[:, :, k]$
15: Compute $A_k[:, t+1 : m] := A_k[:, t+1 : m] - A_k[:, t] \cdot U[t, t+1 : m]$
16: Write block of L, $L[k \cdot n/c : (k+1) \cdot n/c, :] \leftarrow A_k[:, :]$

Ensure: n-by-n permutation matrix P and triangular matrices L, U such that $A = P \cdot L \cdot U$, where L and U are stored in place of A.

Algorithm 5.4 shows how each panel of A is pivoted and factorized. This algorithm pivots each column, however, these columns can also be treated as block-columns with tournament pivoting. Figure 5.4 demonstrates the workflow of the new TSLU with tournament pivoting. Algorithm 5.4 details the entire pivoted 2.5D LU algorithm.

Asymptotically, 2.5D LU with tournament pivoting has almost the same communication and computational cost as the original algorithm. Both the flops and bandwidth costs gain an extra asymptotic $\log(p)$ factor (which can be remedied by using a smaller block size and sacrificing some latency). Also, the bandwidth cost derivation requires a probabilistic argument about the locations of the pivot rows, however, the argument should hold up very well in practice. For the full cost derivations of this algorithm see Appendix C in [147].

5.5 2.5D Cholesky-QR

The dependency structure of 2.5D LU is shared by many other dense factorizations and dense numerical linear algebra algorithms. In this section, we formulate algorithms for 2.5D Cholesky

Algorithm 5.4.2 [P, L, U] = 2.5D-pivoted-LU(A,Π,n,p,c)

Require: n-by-n matrix A distributed so that for each l, m, (n/c)-by-(n/c) block $A[l, m]$ is spread over Π[:, :, 1].

Replicate A on each Π[:, :, k], for $k \in [1, c]$
for $t = 1$ to c **do**
  % *Perform in parallel with each layer:*
  for $k = 1$ to c **do**
    % *Factorize top right panel:*
    [P_t, $L[t : c, t]$, $U[t, t]$]
      = 2.5D-pivoted-TSLU($A[t : c, t]$, Π[:, :, $1 : (c - t)$], $n - tn/c$, n/c, p, c)
    Update P with P_t
    % *Pivot remainder of matrix redundantly*
    Swap any rows as required by P_t to $(A, S)[t, :]$
    % *All-reduce the top panel Schur complement*
    $A[t, t + 1 : c] \leftarrow A[t, t + 1 : c] - \sum_{k'=1}^{c} S[t, t + 1 : c, k']$
    [$U[t, t + k]$] = 2D-TRSM($L[t, t]$, $A[t, t + k]$, Π[:, :, k], n/c, p/c)
    Partition $[T_1, T_2, \ldots T_k] \leftarrow L[t + 1 : c, t]$
    Partition $[W_1^T, W_2^T, \ldots W_k^T]^T \leftarrow U[t, t + 1 : c]$
    [S'_k] = 2D-MM(T_k, W_k^T, Π[:, :, k], n/c, $(c - t) \cdot n/c$, $(c - t) \cdot n/c$)
    % *Adjust Schur complement:*
    $S[t + 1 : c, t + 1 : c, k] = S[t + 1 : c, t + 1 : c, k] + S'_k$
  % *Compute next Schur complement panel via reductions:*
  $A[t + 1 : c, t + 1] \leftarrow A[t + 1 : c, t + 1] - \sum_{k'=1}^{c} S[t + 1 : c, t + 1, k']$

Ensure: n-by-n permutation matrix P and and triangular n-by-n matrices L, U such that $A = P \cdot L \cdot U$ and for each l, m, (n/c)-by-(n/c) blocks $L[l, m]$, $U[l, m]$ are spread over Π[:, :, 1].

factorization, 2.5D triangular solve and 2.5D Cholesky-QR factorization. The structure of the algorithms follows that of non-pivoted 2.5D LU. Further, all the communication costs of these algorithms are asymptotically equivalent to the costs of non-pivoted 2.5D LU.

5.5.1 2.5D Cholesky

The Cholesky factorization factorizes a symmetric positive-definite matrix A into matrix product $L \cdot L^T$, where L is a lower-triangular matrix. Cholesky is typically done without pivoting, since given a symmetric-positive definite matrix, the non-pivoted Cholesky algorithm is stable. Therefore, the structure of Cholesky is very similar to non-pivoted LU. The main difference is the symmetry of A, which presents the algorithmic challenge of only storing the unique (lower-triangular) part of A, and avoiding extra computation.

Algorithm 5.5.1 details a 2D algorithm for Cholesky. This algorithm differs from 2D LU (Algorithm 5.3), in that, every iteration, it replicates (broadcasts) panels of L^T rather than U among

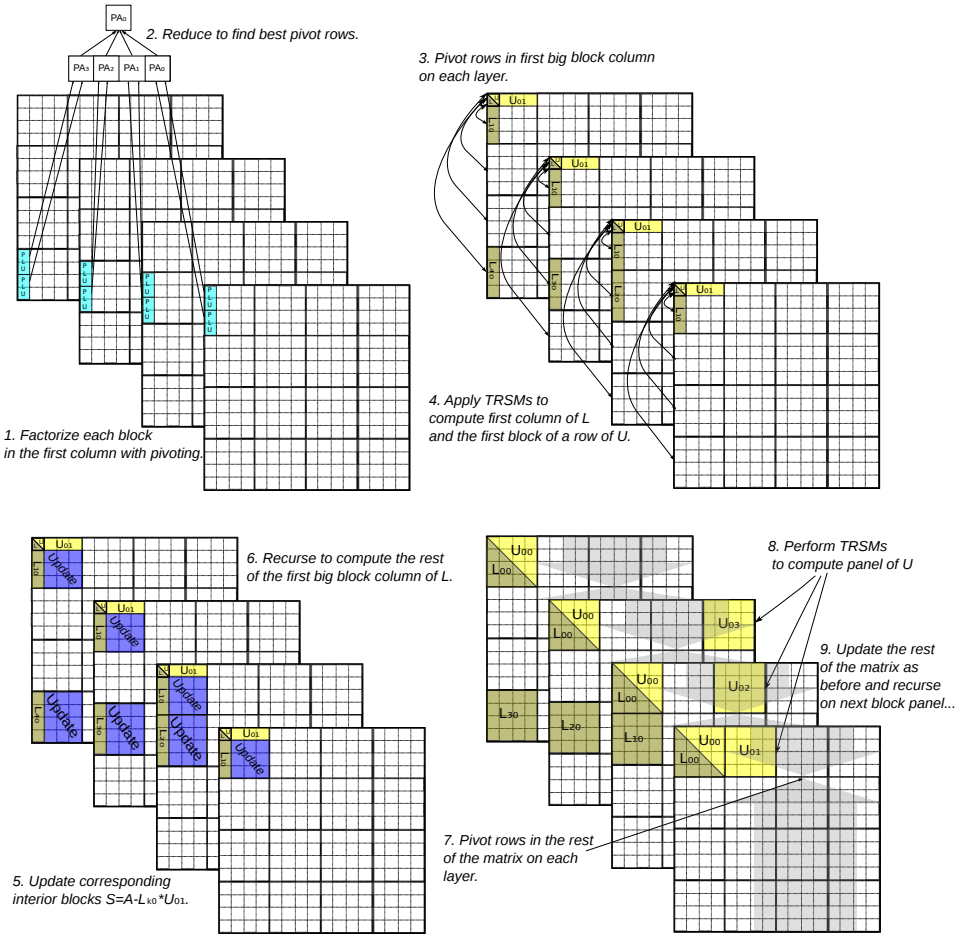


Figure 5.4: 2.5D LU with pivoting panel factorization (step A in Figure 5.3).

rows. However, we do not actually want to store all of L^T. Instead, we notice that the panel L^T we need to perform the update, is the same panel as the panel of L we broadcast among columns. So processors on the diagonal of each processor layer have the panel of L^T they need after the broadcast of the L panel. These diagonal processors should then be used to broadcast the L^T panel within their processor columns.

The above diagonal broadcast technique should also be used in the 2.5D Cholesky algorithm (Algorithm 5.5.1). The structure of 2.5D Cholesky is very similar to non-pivoted 2.5D LU. The upper U panel 2D-TRSM update is no longer necessary. So, this 2.5D Cholesky algorithm performs even less communication that 2.5D LU. Therefore, the same asymptotic costs hold. Since the general lower bound and our latency lower bound also apply to Cholesky, this 2.5D algorithm is communication-optimal.

We note that our 2D and 2.5D Cholesky algorithm have load imbalance due to the symmetry of A. However, this load imbalance should be minimized via the use of a block-cyclic layout for

Algorithm 5.5.1 [L] = 2D-Cholesky(A,Λ,n,p)

Require: n-by-n symmetric matrix A, spread over a square grid Λ

1: % The following loop iterations are pipelined
2: **for** $t = 1$ to $t = n$ **do**
3: Replicate $A[t, t]$ on column $\Lambda[:, t]$
4: Compute $L[t + 1 : n, t] := A[t + 1 : n, t]/A[t, t]$
5: Replicate $L[t + 1 : n, t]$ on columns of $\Lambda[:, :]$
6: Replicate $L^T[t, t + 1 : n]$ on rows of $\Lambda[:, :]$
7: Compute $A[t + 1 : n, t + 1 : n] := A[t + 1 : n, t + 1 : n] - L[t + 1 : n, t] \cdot L^T[t, t + 1 : n]$

Ensure: triangular n-by-n matrices L, such that $A = L \cdot L^T$ and L is spread over Λ in place of A

both 2D and 2.5D Cholesky. Reducing the block-size in this layout would improve the balance of the load, at the cost of increasing latency. The same trade-off also exists in LU factorization, but is more pronounced in Cholesky, due to the symmetry of the matrix.

5.5.2 2.5D Triangular Solve

Triangular solve (TRSM) is used to compute a matrix X, such that $X \cdot U = B$, where U is upper-triangular and B is a dense matrix. This problem has dependencies across the columns of X, but no dependencies across the rows of X. LU and Cholesky have dependencies across both columns and rows. Therefore, efficient parallelization of TRSM is in fact simpler than LU or Cholesky.

Algorithm 5.3 is a 2D algorithm for the triangular solve. Algorithm 5.5.2 gives the corresponding 2.5D version of TRSM. Since there is no dependency across rows, the 2.5D TRSM algorithm can immediately start to compute the left panel of X. Once this update is completed, the rest of the matrix must be updated, and we can move on to computation on the right part of the matrix.

5.5.3 2.5D Cholesky-QR

The QR factorization of a given matrix A can be computed via a combination of matrix multiplication, Cholesky and a triangular solve. This algorithm is called the Cholesky-QR factorization. The algorithm proceeds in 3 steps:

1. Compute symmetric $B = A^T \cdot A$ with matrix multiplication
2. Compute the Cholesky factorization of B, to get $B = R \cdot R^T$
3. Perform a triangular solve to get $Q \cdot R = A$

These 3 steps give us a $A = Q \cdot R$ factorization. We know that this algorithm is bandwidth-optimal since it has the same cost and lower bound as LU factorization [12].

Algorithm 5.2.2 [L] = 2.5D-Cholesky(A,Π,n,p,c)

Require: n-by-n symmetric matrix A distributed so that for each l, m, such that $l \leq m$ (n/c)-by-(n/c) block $A[l, m]$ is spread over Π[**:**, **:**, 1]. Replicate A on each Π[**:**, **:**, k], for $k \in [1, c]$

1: **for** $t = 1$ to c **do**
2: *% Perform in parallel with some layers:*
3: **for** $k = 1$ to $c - t$ **do**
4: *% Factorize top left block:*
5: [$L[t, t]$] = 2D-Cholesky($A[t, t]$, Π[**:**, **:**, k], n/c, p/c)
6: *% Update left (L) panel of A:*
7: [$L[t + k, t]^T$] = 2D-TRSM($L[t, t]$, $A[t + k, t]^T$, Π[**:**, **:**, k], n/c, p/c)
8: *% All-gather panels among processor layers:*
9: Π[**:**, **:**, k] broadcasts $L[t + k, t]$ to Π[**:**, **:**, k'] for all k'
10: **for** $k = 1$ to c **do**
11: Partition $[W_1, W_2, \ldots W_c] \leftarrow L[t + 1 : c, t]$
12: *% Perform a distributed transpose on each panel W_k*
13: $\Pi[i, j, k]$ sends its part of W_k to $\Pi[j, i, k]$.
14: *% Perform outer products on the triangular Schur complement:*
15: [S'_k] = 2D-MM(W_k, W_k^T, Π[**:**, **:**, k], n/c, $(c - t) \cdot n/c$, $(c - t) \cdot n/c$)
16: *% Adjust triangular Schur complement:*
17: $S[t + 1 : c, t + 1 : c, k] = S[t + 1 : c, t + 1 : c, k] + S'_k$
18: *% Compute next panel via reduction:*
19: $A[t + 1 : c, t + 1] \leftarrow A[t + 1 : c, t + 1] - \sum_{k'=1}^{c} S[t + 1 : c, t + 1, k']$

Ensure: triangular n-by-n matrices L, U such that $A = L \cdot U$ and for each l, m, (n/c)-by-(n/c) blocks $L[l, m]$, are spread over Π[**:**, **:**, 1].

5.6 2.5D LU Performance Results

We use the same supercomputer and configurations for 2.5D LU as we did for 2.5D MM, which we described in Section 4.3.1.

5.6.1 Performance of 2.5D LU without Pivoting

We implemented a version of 2.5D LU without pivoting. While this algorithm is not stable for general dense matrices, it provides a good upper-bound on the performance of 2.5D LU with pivoting. The performance of non-pivoted 2.5D LU is also indicative of how well a 2.5D Cholesky and 2.5D TRSM implementation might perform.

Our 2.5D LU implementation has a structure closer to that of Algorithm 5.4 rather than Algorithm 5.3. Processor layers perform all updates at once rather than 2D TRSMs on sub-blocks. This implementation made heavy use of subset broadcasts (multicasts). All communication is done in the form of broadcasts or reductions along axis of the 3D virtual topology. This design allowed

Algorithm 5.5.3 $[X]$ = 2.5D-TRSM(B,U,Π,n,p,c)

Require: n-by-n matrix B distributed so that for each l, m, (n/c)-by-(n/c) block $B[l, m]$ is spread over $\Pi[:, :, 1]$. n-by-n upper-triangular matrix U distributed so that for each l, m, such that $m \geq l$, (n/c)-by-(n/c) block $U[l, m]$ is spread over $\Pi[:, :, 1]$.

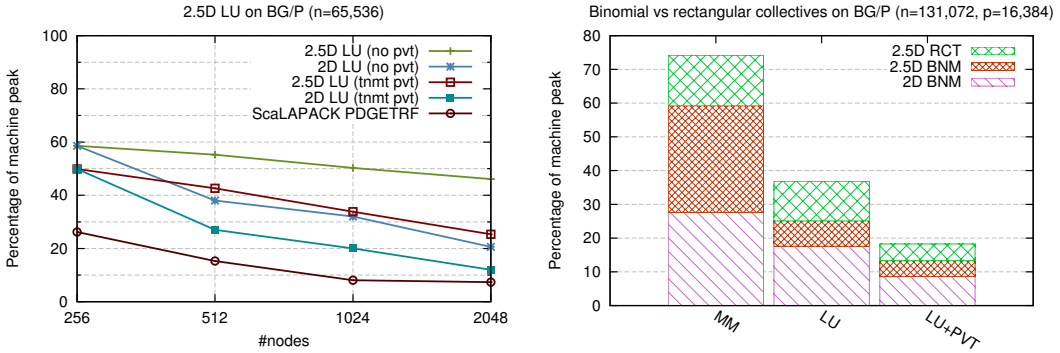
Replicate B and U on each $\Pi[:, :, k]$, for $k \in [1, c]$
for $t = 1$ to c **do**
  % Perform in parallel with some layers:
  for $k = 1$ to $c - t$ **do**
    % Solve a panel of X:
    $[X[t + k, t]]$ = 2D-TRSM($B[t + k, t]$, $U[t, t]$ $\Pi[:, :, k]$, n/c, p/c)
    % All-gather panels among processor layers:
    $\Pi[:, :, k]$ broadcasts $X[t + k, t]$ to $\Pi[:, :, k']$ for all k'
  % Perform outer products on whole Schur complement with each processor layer:
  for $k = 1$ to c **do**
    Partition $[T_1, T_2, \ldots T_c] \leftarrow X[:, t]$
    Partition $[W_1^T, W_2^T, \ldots W_c^T]^T \leftarrow U[t, t + 1 : c]$
    $[S'_k]$ = 2D-MM(T_k, W_k^T, $\Pi[:, :, k]$, n/c, n, $(c - t) \cdot n/c$)
    % Adjust Schur complement:
    $S[:, t + 1 : c, k] = S[:, t + 1 : c, k] + S'_k$
  % Compute next panels via reductions:
  $B[:, t + 1] \leftarrow B[:, t + 1] - \sum_{k'=1}^{c} S[:, t + 1, k']$

Ensure: n-by-n matrix X, such that $X \cdot U = B$ and for each l, m, (n/c)-by-(n/c) blocks $X[l, m]$ are spread over $\Pi[:, :, 1]$.

our code to utilize efficient line broadcasts on the BG/P supercomputer.

Figure 5.5(a) shows that 2.5D LU achieves more efficient strong scaling than 2D LU. 2D LU maps well to the 2D processor grid on 256 nodes. However, the efficiency of 2D LU suffers when we use more nodes, since the network partition becomes 3D. On 3D partitions, the broadcasts within 2D LU are done via topology-oblivious binomial trees and suffer from contention. For this problem configuration, 2.5D LU achieves a two-fold speed-up over the 2D algorithm on 2048 nodes.

Figure 5.5(b) demonstrates that 2.5D LU is also efficient and beneficial at a larger scale. Figure 5.5(b) also demonstrates the effect of topology-aware collectives on all the algorithms. When using topology-aware rectangular (RCT) collectives, 2.5D algorithms gain a significant amount of efficiency. 2D algorithms do not map to the BG/P architecture so they can only use binomial (BNM) collectives. However, we see that 2.5D algorithms with binomial collectives still outperform 2D algorithms by a significant factor.



(a) LU strong scaling

(b) Performance on 16,384 nodes

Figure 5.5: Performance of 2.5D LU on BG/P

5.6.2 Performance of 2.5D LU with Tournament Pivoting

2.5D LU performs pivoting in two stages. First, pivoting is performed only in the big-block panel. Then the rest of the matrix is pivoted according to a larger, accumulated pivot matrix. We found it most efficient to perform the sub-panel pivoting via a broadcast and a reduction, which minimize latency. For the rest of the matrix, we performed scatter and gather operations to pivot, which minimize bandwidth. We found that this optimization can also be used to improve the performance of 2D LU and used it accordingly.

Figure 5.5(a) shows that 2.5D LU with tournament pivoting strongly scales with higher efficiency than its 2D counter-part. It also outperforms the ScaLAPACK PDGETRF implementation. Though, we note that ScaLAPACK uses partial pivoting rather than tournament pivoting and therefore computes a different answer. Our peak efficiency is much lower than the peak efficiency achieved by LINPACK on this machine, but LINPACK is heavily optimized and targeted for very large matrices than those we tested.

Figure 5.5(b) details the efficiency of pivoted 2.5D LU with binomial and rectangular collectives. Again, we observe that part of the improvement is due to the algorithmic decrease in communication cost, and part is due to the topology-aware mapping (collectives).

The absolute efficiency achieved by our 2.5D LU with tournament pivoting algorithm is better than ScaLAPACK and can be improved even further. Our implementation does not exploit overlap between communication and computation and does not use prioritized scheduling. We observed that, especially at larger scales, processors spent most of their time idle (waiting to synchronize). Communication time, on the other hand, was heavily reduced in our implementation and was no longer the major bottleneck. While overlap can be combined with 2.5D algorithms, we noted that for small matrices, computation was a small fraction of the total time, so overlap could not have been used to completely hide communication. Further, we achieved speed-ups of over 2X for all algorithms, which cannot be done with just overlap.

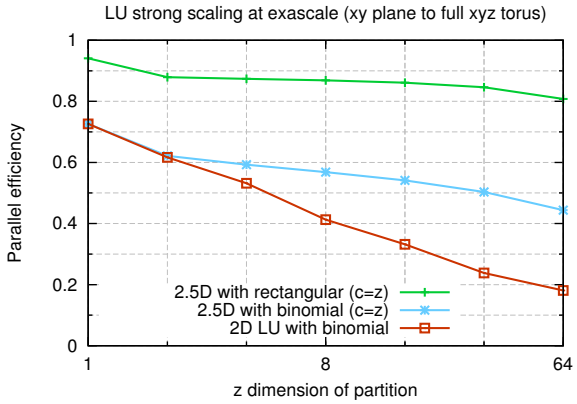


Figure 5.6: LU factorization predicted performance on an exascale architecture. The results show strong scaling from full memory on the first plane ($z = 1$) of the machine to the full machine ($z = 64$).

5.6.3 Predicted Performance at Exascale

LU factorization requires more careful modeling, since the block-cyclic layout makes the message sizes variable throughout the algorithm. Further, the 2.5D LU algorithm performs collectives on messages of different sizes and to different partitions depending on the stage of the algorithm. The LU communication costs are itemized in Appendix B of [147]. We scale these communication costs based on the message size and partition size they operate on using Eq. 2.2.4 (rectangular collectives) and Eq. 2.2.7 (binomial collectives).

Our LU exascale study uses the same problem size and measures the efficiency in the same way as the MM study (previous section). Figure 5.6 details the scaling from a plane to the full machine for a matrix size of $n = 2^{22}$. We model only LU without pivoting. We see that using the 2.5D LU algorithm increases performance significantly even without rectangular collectives. 2.5D LU with rectangular collectives achieves very good parallel scalability with comparison to the 2D algorithm. When using all available nodes ($z = 64$), 2.5D LU with rectangular collectives reduces communication time by 95% and achieves a speed-up of 4.5x over 2D LU.

Chapter 6

QR Factorization

The standard algorithm for QR decomposition, which is implemented in LAPACK [5], ScaLAPACK [28], and Elemental [131] is known as Householder-QR (given below as Algorithm 6.1.1). For tall and skinny matrices, the algorithm works column-by-column, computing a Householder vector and applying the corresponding transformation for each column in the matrix. When the matrix is distributed across a parallel machine, this requires one parallel reduction per column. The TSQR algorithm (given below as Algorithm 6.1.2), on the other hand, performs only one reduction during the entire factorization of a tall and skinny matrix. Therefore, TSQR requires asymptotically less inter-processor synchronization than Householder-QR on parallel machines (TSQR also achieves asymptotically higher cache reuse on sequential machines).

Computing the QR decomposition of a tall and skinny matrix is an important kernel in many contexts, including standalone least squares problems, eigenvalue and singular value computations, and Krylov subspace and other iterative methods. In addition, the tall and skinny factorization is a standard building block in the computation of the QR decomposition of general (not necessarily tall and skinny) matrices. In particular, most algorithms work by factoring a tall and skinny panel of the matrix, applying the orthogonal factor to the trailing matrix, and then continuing on to the next panel. Although Householder-QR is bottlenecked by communication in the panel factorization, it can apply the orthogonal factor as an aggregated Householder transformation efficiently, using matrix multiplication [139].

The Communication-Avoiding QR (CAQR) [47] algorithm uses TSQR to factor each panel of a general matrix. One difficulty faced by CAQR is that TSQR computes an orthogonal factor that is implicitly represented in a different format than that of Householder-QR. While Householder-QR represents the orthogonal factor as a set of Householder vectors (one per column), TSQR computes a tree of smaller sets of Householder vectors (though with the same total number of nonzero parameters). In CAQR, this difference in representation implies that the trailing matrix update is done using the implicit tree representation rather than matrix multiplication as possible with Householder-QR. From a software engineering perspective, this means writing and tuning more

This chapter is based on joint work with Grey Ballard, Matthias Jacquelin, Laura Grigori, and Hong Diep Nguyen [8] as well as Nicholas Knight.

complicated code. Furthermore, from a performance perspective, the trailing matrix update within CAQR is less communication efficient than the update within Householder-QR by a logarithmic factor in the number of processors.

Building on a method introduced by Yamamoto [169], we show that the standard Householder vector representation may be recovered from the implicit TSQR representation for roughly the same cost as the TSQR itself. The key idea is that the Householder vectors that represent an orthonormal matrix can be computed via LU decomposition (without pivoting) of the orthonormal matrix subtracted from a diagonal sign matrix. We prove that this reconstruction is as numerically stable as Householder-QR (independent of the matrix condition number), and validate this proof with experimental results.

This reconstruction method allows us to get the best of the TSQR algorithm (avoiding synchronization) as well as the best of the Householder-QR algorithm (efficient trailing matrix updates via matrix multiplication). By obtaining Householder vectors from the TSQR representation, we can logically decouple the block size of the trailing matrix updates from the number of columns in each TSQR and use two levels of aggregation of Householder vectors. This abstraction makes it possible to optimize panel factorization and the trailing matrix updates independently. Our resulting aggregated parallel implementation outperforms ScaLAPACK, Elemental, and a binary-tree CAQR implementation on the Hopper Cray XE6 platform at NERSC by factors of up to 1.4X. While we do not experimentally study sequential performance, we expect our algorithm will also be beneficial in this setting, due to the cache efficiency gained by using TSQR.

Two other contributions of the chapter include improvements to the Householder QR and CAQR algorithms for square matrices. We employ the two-level aggregation technique within Householder QR to alleviate a memory bandwidth bottleneck (see Section 6.2.3), and we use a more efficient trailing matrix update within CAQR that improves both the computation and communication costs of that algorithm (see Section 6.2.4). Both optimizations lead to significant performance improvement (up to 1.4X for Householder QR due to aggregation and up to 4X with respect to binary tree CAQR due to scatter-apply algorithm) for the two algorithms.

Lastly, we give an algorithm for 2.5D QR factorization, which attains the same asymptotic cost as the 2.5D LU factorization algorithm in Chapter 5. The algorithm is left-looking unlike the 2.5D LU algorithm, due to the fact that the QR update cannot be accumulated in the same away as a Schur complement update in LU. We use this 2.5D QR as well as the 2D QR with reconstruction algorithms as building blocks in the next chapter on the symmetric eigenvalue problem.

The rest of the chapter is organized as follows,

- Section 6.1 identifies and reviews previous work needed for our parallel algorithms,
- Section 6.2 gives the 2D QR algorithm with Householder reconstruction and shows how the update may be aggregated,
- Section 6.3 benchmarks the scalability of our new aggregated 2D QR algorithm, comparing to existing implementations,

- Section 6.4 presents a 2.5D algorithm that does a factor of up to $p^{1/6}$ less communication than the 2D version.

6.1 Previous Work

We distinguish between two types of QR factorization algorithms. We call an algorithm that distributes entire rows of the matrix to processors a 1D algorithm. Such algorithms are often used for tall and skinny matrices. Algorithms that distribute the matrix across a 2D grid of $p_r \times p_c$ processors are known as 2D algorithms. Many right-looking 2D algorithms for QR decomposition of nearly square matrices divide the matrix into column panels and work panel-by-panel, factoring the panel with a 1D algorithm and then updating the trailing matrix. We consider two such existing algorithms in this section: 2D-Householder-QR (using Householder-QR) and CAQR (using TSQR).

6.1.1 Householder-QR

We first present Householder-QR in Algorithm 6.1.1, following [67] so that each Householder vector has a unit diagonal entry. We use LAPACK [5] notation for the scalar quantities. However, we depart from the LAPACK code in that there is no check for a zero norm of a subcolumn. We present Algorithm 6.1.1 in Matlab-style notation as a sequential algorithm. The algorithm works column-by-column, computing a Householder vector and then updating the trailing matrix to the right. The Householder vectors are stored in an $m \times b$ lower triangular matrix Y. Note that we do not include τ as part of the output because it can be recomputed from Y.

Algorithm 6.1.1 $[Y, R] = \text{Householder-QR}(A)$

Require: A is $m \times b$

1: **for** $i = 1$ to b **do**
2: *% Compute the Householder vector*
3: $\alpha = A(i,i)$, $\beta = \|A(i{:}m, i)\|_2$
4: **if** $\alpha > 0$ **then**
5: $\beta = -\beta$
6: $A(i,i) = \beta$, $\tau(i) = \frac{\beta - \alpha}{\beta}$
7: $A(i{+}1{:}m, i) = \frac{1}{\alpha - \beta} \cdot A(i{+}1{:}m, i)$
8: *% Apply the Householder transformation to the trailing matrix*
9: $z = \tau(i) \cdot [A(i, i{+}1{:}b) + A(i{+}1{:}m, i)^T \cdot A(i{+}1{:}m, i{+}1{:}b)]$
10: $A(i{+}1{:}m, i{+}1{:}b) = A(i{+}1{:}m, i{+}1{:}b) - A(i{+}1{:}m, i) \cdot z$

Ensure: $A = \left(\prod_{i=1}^{n} (I - \tau_i y_i y_i^T)\right) R$

Ensure: R overwrites the upper triangle and Y (the Householder vectors) has implicit unit diagonal and overwrites the strict lower triangle of A; τ is an array of length b with $\tau_i = 2/(y_i^T y_i)$

While the algorithm works for general m and n, it is most commonly used when $m \gg n$, such as a panel factorization within a square QR decomposition. In LAPACK terms, this algorithm

corresponds to `geqr2` and is used as a subroutine in `geqrf`. In this case, we also compute an upper triangular matrix T so that

$$Q = \prod_{i=1}^{n}(I - \tau_i y_i y_i^T) = I - YTY^T,$$

which allows the application of Q^T to the trailing matrix to be done efficiently using matrix multiplication. Computing T is done in LAPACK with `larft` but can also be computed from Y^TY by solving the equation $Y^TY = T^{-1} + T^{-T}$ for T^{-1} (since Y^TY is symmetric and T^{-1} is triangular, the off-diagonal entries are equivalent and the diagonal entries differ by a factor of 2) [132].

1D Algorithm

We will make use of Householder-QR as a sequential algorithm, but there are parallelizations of the algorithm in libraries such as ScaLAPACK [28] and Elemental [131] against which we will compare our new approach. Assuming a 1D distribution across p processors, the parallelization of Householder-QR (Algorithm 6.1.1) requires communication at lines 3 and 9, both of which can be performed using an all-reduction. Because these steps occur for each column in the matrix, the total latency cost of the algorithm is $2b \log p$. This synchronization cost is a potential parallel scaling bottleneck, since it grows with the number of columns of the matrix and does not decrease with the number of processors. The algorithm also performs $2mb^2/p$ flops and communicates $(b^2/2) \log p$ words.

2D Algorithm

In the context of a 2D algorithm, in order to perform an update with the computed Householder vectors, we must also compute the T matrix from Y in parallel. The leading order cost of computing T^{-1} from Y^TY is mb^2/p flops plus the cost of reducing a symmetric $b \times b$ matrix, $\alpha \cdot \log p + \beta \cdot b^2/2$; note that the communication costs are lower order terms compared to computing Y. We present the costs of parallel Householder-QR in the first row of Table 6.2.1, combining the costs of Algorithm 6.1.1 with those of computing T.

We refer to the 2D algorithm that uses Householder-QR as the panel factorization as 2D-Householder-QR. In ScaLAPACK terms, this algorithm corresponds to `pxgeqrf`. The overall cost of 2D-Householder-QR, which includes panel factorizations and trailing matrix updates, is given to leading order by

$$\gamma \cdot \left(\frac{6mnb - 3n^2b}{2p_r} + \frac{n^2b}{2p_c} + \frac{2mn^2 - 2n^3/3}{p} \right) +$$
$$\beta \cdot \left(nb \log p_r + \frac{2mn - n^2}{p_r} + \frac{n^2}{p_c} \right) +$$
$$\alpha \cdot \left(2n \log p_r + \frac{2n}{b} \log p_c \right).$$

If we pick $p_r = p_c = \sqrt{p}$ (assuming $m \approx n$) and $b = n/(\sqrt{p}\log p)$ then we obtain the leading order costs

$$\gamma \cdot (2mn^2 - 2n^3/3)/p + \beta \cdot (mn + n^2)/\sqrt{p} + \alpha \cdot n \log p.$$

Note that these costs match those of [47, 28], with exceptions coming from the use of more efficient collectives. The choice of b is made to preserve the leading constants of the parallel computational cost. We present the costs of 2D-Householder-QR in the first row of Table 6.2.2.

6.1.2 Communication-Avoiding QR

In this section we present parallel Tall-Skinny QR (TSQR) [47, Algorithm 1] and Communication-Avoiding QR (CAQR) [47, Algorithm 2], which are algorithms for computing a QR decomposition that are more communication efficient than Householder-QR, particularly for tall and skinny matrices.

1D Algorithm (TSQR)

We present a simplified version of TSQR in Algorithm 6.1.2: we assume the number of processors is a power of two and use a binary reduction tree (TSQR can be performed with any tree). Note also that we present a reduction algorithm rather than an all-reduction (*i.e.*, the final R resides on only one processor at the end of the algorithm). TSQR assumes the tall-skinny matrix A is distributed in block row layout so that each processor owns a $(m/p) \times n$ submatrix. After each processor computes a local QR factorization of its submatrix (line 1), the algorithm works by reducing the p remaining $n \times n$ triangles to one final upper triangular $R = Q^T A$ (lines 2–8). The Q that triangularizes A is stored implicitly as a tree of sets of Householder vectors, given by $\{Y_{i,k}\}$. In particular, $\{Y_{i,k}\}$ is the set of Householder vectors computed by processor i at the kth level of the tree. The ith leaf of tree, $Y_{i,0}$ is the set of Householder vectors which processor i computes by doing a local QR on its part of the initial matrix A.

In the case of a binary tree, every internal node of the tree consists of a QR factorization of two stacked $b \times b$ triangles (line 6). This sparsity structure can be exploited, saving a constant factor of computation compared to a QR factorization of a dense $2b \times b$ matrix. In fact, as of version 3.4, LAPACK has subroutines for exploiting this and similar sparsity structures (`tpqrt`). Furthermore, the Householder vectors generated during the QR factorization of stacked triangles have similar sparsity; the structure of the $Y_{i,k}$ for $k > 0$ is an identity matrix stacked on top of a triangle.

The costs and analysis of TSQR are given in [47, 48]:

$$\gamma \cdot \left(\frac{2mb^2}{p} + \frac{2b^3}{3}\log p\right) + \beta \cdot \left(\frac{b^2}{2}\log p\right) + \alpha \cdot \log p.$$

We tabulate these costs in the second row of Table 6.2.1. We note that the TSQR inner tree factorizations require an extra computational cost $O(b^3 \log p)$ and a bandwidth cost of $O(b^2 \log p)$. Also note that in the context of a 2D algorithm, using TSQR as the panel factorization implies that there is no $b \times b$ T matrix to compute; the update of the trailing matrix is performed differently.

Algorithm 6.1.2 $[\{Y_{i,k}\}, R] = \text{TSQR}(A)$

Require: Number of processors is p and i is the processor index
Require: A is $m \times b$ matrix distributed in block row layout; A_i is processor i's block
$[Y_{i,0}, \bar{R}_i] =$ Householder-QR(A_i)
for $k = 1$ to $\lceil \log p \rceil$ **do**
  if $i \equiv 0 \mod 2^k$ and $i + 2^{k-1} < p$ **then**
    $j = i + 2^{k-1}$
    Receive $\bar{R}_j$ from processor j
    $[Y_{i,k}, \bar{R}_i] =$ Householder-QR $\left(\begin{bmatrix} \bar{R}_i \\ \bar{R}_j \end{bmatrix} \right)$
  else if $i \equiv 2^{k-1} \mod 2^k$ **then**
    Send $\bar{R}_i$ to processor $i - 2^{k-1}$
if $i = 0$ **then**
  $R = \bar{R}_0$
Ensure: $A = QR$ with Q implicitly represented by $\{Y_{i,k}\}$
Ensure: R is stored by processor 0 and $Y_{i,k}$ is stored by processor i

2D Algorithm (CAQR)

The 2D algorithm that uses TSQR for panel factorizations is known as CAQR. In order to update the trailing matrix within CAQR, the implicit orthogonal factor computed from TSQR needs to be applied as a tree in the same order as it was computed. See [47, Algorithm 2] for a description of this process, or see [7, Algorithm 4] for pseudocode that matches the binary tree in Algorithm 6.1.2. We refer to this application of implicit Q^T as Apply-TSQR-Q^T. The algorithm has the same tree dependency flow structure as TSQR but requires a bidirectional exchange between paired nodes in the tree. We note that in internal nodes of the tree it is possible to exploit the additional sparsity structure of $Y_{i,k}$ (an identity matrix stacked on top of a triangular matrix), which our implementation does via the use of the LAPACK v3.4+ routine `tpmqrt`.

Further, since A is $m \times n$ and intermediate values of rows of A are communicated, the trailing matrix update costs more than TSQR when $n > b$. In the context of CAQR on a square matrix, Apply-TSQR-Q^T is performed on a trailing matrix with $n \approx m$ columns. The extra work in the application of the inner leaves of the tree is proportional to $O(n^2 b \log(p)/\sqrt{p})$ and bandwidth cost proportional to $O(n^2 \log(p)/\sqrt{p})$. Since the cost of Apply-TSQR-Q^{T}is almost leading order in CAQR, it is desirable in practice to optimize the update routine. However, the tree dependency structure complicates this manual developer or compiler optimization task.

The overall cost of CAQR is given to leading order by

$$\gamma \cdot \left(\frac{6mnb - 3n^2 b}{2p_r} + \left(\frac{4nb^2}{3} + \frac{3n^2 b}{2p_c} \right) \log p_r + \frac{6mn^2 - 2n^3}{3p} \right) +$$
$$\beta \cdot \left(\left(\frac{nb}{2} + \frac{n^2}{p_c} \right) \log p_r + \frac{2mn - n^2}{p_r} \right) + \alpha \cdot \left(\frac{3n}{b} \log p_r + \frac{4n}{b} \log p_c \right).$$

See [47] for a discussion of these costs and [48] for detailed analysis. Note that the bandwidth cost is slightly lower here due to the use of more efficient broadcasts. If we pick $p_r = p_c = \sqrt{p}$ (assuming $m \approx n$) and $b = \frac{n}{\sqrt{p}\log^2 p}$ then we obtain the leading order costs

$$\gamma \cdot \left(\frac{2mn^2 - 2n^3/3}{p}\right) + \beta \cdot \left(\frac{2mn + n^2 \log p}{\sqrt{p}}\right) + \alpha \cdot \left(\frac{7}{2}\sqrt{p}\log^3 p\right).$$

Again, we choose b to preserve the leading constants of the computational cost. Note that b needs to be chosen smaller here than in Section 6.1.1 due to the costs associated with Apply-TSQR-Q^T.

It is possible to reduce the costs of Apply-TSQR-Q^T further using ideas from efficient recursive doubling/halving collectives; see Section 6.2.4 for more details. Another important practical optimization for CAQR is pipelining the trailing matrix updates [52], though we do not consider this idea here as it cannot be applied with the Householder reconstruction approach.

Constructing Explicit Q from TSQR

Algorithm 6.1.3 $[B] =$ Apply-TSQR-Q$^T(\{Y_{i,k}\}, A)$

Require: Number of processors is p and i is the processor index
Require: A is $m \times n$ matrix distributed in block row layout; A_i is processor i's block
Require: $\{Y_{i,k}\}$ is the implicit tree TSQR representation of b Householder vectors of length m.

$B_i =$ Apply-Householder-$Q^T(Y_{i,0}, A_i)$
Let $\bar{B}_i$ be the first b rows of B_i
for $k = 1$ to $\lceil \log p \rceil$ **do**
  if $i \equiv 0 \mod 2^k$ and $i + 2^{k-1} < p$ **then**
    $j = i + 2^{k-1}$
    Receive $\bar{B}_j$ from processor j
    $\begin{bmatrix} \bar{B}_i \\ \bar{B}_j \end{bmatrix} =$ Apply-Householder-$Q^T\left(Y_{i,k}, \begin{bmatrix} \bar{B}_i \\ \bar{B}_j \end{bmatrix}\right)$
    Send $\bar{B}_j$ back to processor j
  else if $i \equiv 2^{k-1} \mod 2^k$ **then**
    Send $\bar{B}_i$ to processor $i - 2^{k-1}$
    Receive updated rows $\bar{B}_i$ from processor $i - 2^{k-1}$
    Set the first b rows of B_i to $\bar{B}_i$

Ensure: $B = Q^T A$ with processor i owning block B_i, where Q is the orthogonal matrix implicitly represented by $\{Y_{i,k}\}$

In many use cases of QR decomposition, an explicit orthogonal factor is not necessary; rather, we need only the ability to apply the matrix (or its transpose) to another matrix, as done in the previous section. For our purposes (see Section 6.2), we will need to form the explicit $m \times b$ orthonormal matrix from the implicit tree representation.[1] Though it is not necessary within CAQR,

[1] In LAPACK terms, constructing (*i.e.*, generating) the orthogonal factor when it is stored as a set of Householder vectors is done with `orgqr`.

we describe it here because it is a known algorithm (see [82, Figure 4]) and the structure of the algorithm is very similar to TSQR.

Algorithm 6.1.4 presents the method for constructing the $m \times b$ matrix Q by applying the (implicit) square orthogonal factor to the first b columns of the $m \times m$ identity matrix. Note that while we present Algorithm 6.1.4 assuming a binary tree, any tree shape is possible, as long as the implicit Q is computed using the same tree shape as TSQR. While the nodes of the tree are computed from leaves to root, they will be applied in reverse order from root to leaves. Note that in order to minimize the computational cost, the sparsity of the identity matrix at the root node and the sparsity structure of $\{Y_{i,k}\}$ at the inner tree nodes is exploited.

Since the communicated matrices $\bar{Q}_j$ are triangular just as $\bar{R}_j$ was triangular in the TSQR algorithm, Construct-TSQR-Q incurs the exact same computational and communication costs as TSQR. So, we can reconstruct the unique part of the Q matrix from the implicit form given by TSQR for the same cost as the TSQR itself.

Algorithm 6.1.4 $Q =$ Construct-TSQR-$Q(\{Y_{i,k}\})$

Require: Number of processors is p and i is the processor index
Require: $\{Y_{i,k}\}$ is computed by Algorithm 6.1.2 so that $Y_{i,k}$ is stored by processor i
if $i = 0$ **then**
  $\bar{Q}_0 = I_b$
for $k = \lceil \log p \rceil$ down to 1 **do**
  if $i \equiv 0 \mod 2^k$ and $i + 2^{k-1} < p$ **then**
    $j = i + 2^{k-1}$
    $\begin{bmatrix} \bar{Q}_i \\ \bar{Q}_j \end{bmatrix} =$ Apply-Householder-Q $\left(Y_{i,k}, \begin{bmatrix} \bar{Q}_i \\ 0 \end{bmatrix} \right)$
    Send $\bar{Q}_j$ to processor j
  else if $i \equiv 2^{k-1} \mod 2^k$ **then**
    Receive $\bar{Q}_i$ from processor $i - 2^{k-1}$
$Q_i =$ Apply-Q-to-Triangle $\left(Y_{i,0}, \begin{bmatrix} \bar{Q}_i \\ 0 \end{bmatrix} \right)$
Ensure: Q is orthonormal $m \times b$ matrix distributed in block row layout; Q_i is processor i's block

6.1.3 Yamamoto's Basis-Kernel Representation

The main goal of this work is to combine Householder-QR with CAQR; Yamamoto [169] proposes a scheme to achieve this. As described in Section 6.1.1, 2D-Householder-QR suffers from a communication bottleneck in the panel factorization. TSQR alleviates that bottleneck but requires a more complicated (and slightly less efficient) trailing matrix update. Motivated in part to improve the performance and programmability of a hybrid CPU/GPU implementation, Yamamoto suggests computing a representation of the orthogonal factor that triangularizes the panel that mimics the representation in Householder-QR.

As described by Sun and Bischof [153], there are many so-called "basis-kernel" representations of an orthogonal matrix. For example, the Householder-QR algorithm computes a lower triangular

matrix Y such that $A = (I - YTY_1^T)R$, so that

$$Q = I - YTY^T = I - \begin{bmatrix} Y_1 \\ Y_2 \end{bmatrix} T \begin{bmatrix} Y_1^T & Y_2^T \end{bmatrix}. \tag{6.1.1}$$

Here, Y is called the "basis" and T is called the "kernel" in this representation of the square orthogonal factor Q. However, there are many such basis-kernel representations if we do not restrict Y and T to be lower and upper triangular matrices, respectively.

Yamamoto [169] chooses a basis-kernel representation that is easy to compute. For an $m \times b$ matrix A, let $A = \begin{bmatrix} Q_1 \\ Q_2 \end{bmatrix} R$ where Q_1 and R are $b \times b$. Then define the basis-kernel representation

$$Q = I - \tilde{Y}\tilde{T}\tilde{Y}^T = I - \begin{bmatrix} Q_1 - I \\ Q_2 \end{bmatrix} \begin{bmatrix} I - Q_1 \end{bmatrix}^{-T} \begin{bmatrix} (Q_1 - I)^T & Q_2^T \end{bmatrix}, \tag{6.1.2}$$

where $I - Q_1$ is assumed to be nonsingular. It can be easily verified that $Q^T Q = I$ and $Q^T A = \begin{bmatrix} R \\ 0 \end{bmatrix}$; in fact, this is the representation suggested and validated by [27, Theorem 3]. Note that both the basis and kernel matrices $\tilde{Y}$ and $\tilde{T}$ are dense.

The main advantage of basis-kernel representations is that they can be used to apply the orthogonal factor (or its transpose) very efficiently using matrix multiplication. In particular, the computational complexity of applying Q^T using any basis-kernel is the same to leading order, assuming Y has the same dimensions as A and $m \gg b$. Thus, it is not necessary to reconstruct the Householder vectors; from a computational perspective, finding any basis-kernel representation of the orthogonal factor computed by TSQR will do. Note also that in order to apply Q^T with the representation in Equation equation 6.1.2, we need to apply the inverse of $I - Q_1$, so we need to perform an LU decomposition of the $b \times b$ matrix and then apply the inverses of the triangular factors using triangular solves.

The assumption that $I - Q_1$ is nonsingular can be dropped by replacing I with a diagonal sign matrix S chosen so that $S - Q_1$ is nonsingular [168]; in this case the representation becomes

$$QS = I - \tilde{Y}\tilde{T}\tilde{Y}^T = I - \begin{bmatrix} Q_1 - S \\ Q_2 \end{bmatrix} S \begin{bmatrix} S - Q_1 \end{bmatrix}^{-T} \begin{bmatrix} (Q_1 - S)^T & Q_2^T \end{bmatrix}. \tag{6.1.3}$$

Yamamoto's approach is very closely related to TSQR-HR (Algorithm 6.2.2), presented in Section 6.2. We compare the methods in Section 6.2.1.

6.2 New 2D QR Algorithms

We first present our main contribution, a parallel algorithm that performs TSQR and then reconstructs the Householder vector representation from the TSQR representation of the orthogonal factor. We then show that this reconstruction algorithm may be used as a building block for more efficient 2D QR algorithms. In particular, the algorithm is able to combine two existing approaches

	Flops	Words	Messages
Householder-QR	$\frac{3mb^2}{p} - \frac{2b^3}{3p}$	$\frac{b^2}{2}\log p$	$2b\log p$
TSQR	$\frac{2mb^2}{p} + \frac{2b^3}{3}\log p$	$\frac{b^2}{2}\log p$	$\log p$
TSQR-HR	$\frac{5mb^2}{p} + \frac{4b^3}{3}\log p$	$b^2\log p$	$4\log p$

Table 6.2.1: Costs of QR factorization of tall-skinny $m \times b$ matrix distributed over p processors in 1D fashion. We assume these algorithms are used as panel factorizations in the context of a 2D algorithm applied to an $m \times n$ matrix. Thus, costs of Householder-QR and TSQR-HR include the costs of computing T.

for 2D QR factorizations, leveraging the efficiency of TSQR in panel factorizations and the efficiency of Householder-QR in trailing matrix updates. While Householder reconstruction adds some extra cost to the panel factorization, we show that its use in the 2D algorithm reduces overall communication compared to both 2D-Householder-QR and CAQR.

6.2.1 TSQR with Householder Reconstruction

The basic steps of our 1D algorithm include performing TSQR, constructing the explicit tall-skinny Q factor, and then computing the Householder vectors corresponding to Q. The key idea of our reconstruction algorithm is that performing Householder-QR on an orthonormal matrix Q is the same as performing an LU decomposition on $Q - S$, where S is a diagonal sign matrix corresponding to the sign choices made inside the Householder-QR algorithm. Informally, ignoring signs, if $Q = I - YTY_1^T$ with Y a matrix of Householder vectors, then $Y \cdot (-TY_1^T)$ is an LU decomposition of $Q - I$ since Y is unit lower triangular and TY_1^T is upper triangular.

In this section we present Modified-LU as Algorithm 6.2.1, which can be applied to any orthonormal matrix (not necessarily one obtained from TSQR). Ignoring lines 1, 3, and 4, it is exactly LU decomposition without pivoting. Note that with the choice of S, no pivoting is required since the effective diagonal entry will be at least 1 in absolute value and all other entries in the column are bounded by 1 (the matrix is orthonormal).[2]

Parallelizing this algorithm is straightforward. Since $m \geq b$ and no pivoting is required, the Modified-LU algorithm can be applied on one processor to the top $b \times b$ block. The rest of the lower triangular factor is updated with a triangular solve involving the upper triangular factor. After the upper triangular factor has been broadcast to all processors, the triangular solve is performed in parallel. Thus, the cost of Algorithm 6.2.1 is given by

$$\gamma \cdot \left(\frac{mb^2}{p} + \frac{2b^3}{3}\right) + \beta \cdot b^2 + \alpha \cdot 2\log p.$$

Given the algorithms of the previous sections, we now present the full approach for computing the QR decomposition of a tall-skinny matrix using TSQR and Householder reconstruction. That

[2]We use the convention $\mathrm{sgn}(0) = 1$.

Algorithm 6.2.1 $[L, U, S] = \text{Modified-LU}(Q)$

Require: Q is $m \times b$ orthonormal matrix

1: $S = 0$
2: **for** $i = 1$ to b **do**
3: $\quad S(i,i) = -\operatorname{sgn}(Q(i,i))$
4: $\quad Q(i,i) = Q(i,i) - S(i,i)$
5: $\quad$ *% Scale ith column of L by diagonal element*
6: $\quad Q(i{+}1{:}m, i) = \frac{1}{Q(i,i)} \cdot Q(i{+}1{:}m, i)$
7: $\quad$ *% Perform Schur complement update*
8: $\quad Q(i{+}1{:}m, i{+}1{:}b) = Q(i{+}1{:}m, i{+}1{:}b) - Q(i{+}1{:}m, i) \cdot Q(i, i{+}1{:}b)$

Ensure: U overwrites the upper triangle and L has implicit unit diagonal and overwrites the strict lower triangle of Q; S is diagonal so that $Q - S = LU$

is, in this section we present an algorithm such that the format of the output of the algorithm is identical to that of Householder-QR. However, we argue that the communication costs of this approach are much less than those of performing Householder-QR.

The method, given as Algorithm 6.2.2, is to perform TSQR (line 1), construct the tall-skinny Q factor explicitly (line 2), and then compute the Householder vectors that represent that orthogonal factor using Modified-LU (line 3). The R factor is computed in line 1 and the Householder vectors (the columns of Y) are computed in line 3. An added benefit of the approach is that the triangular T matrix, which allows for block application of the Householder vectors, can be computed very cheaply. That is, a triangular solve involving the upper triangular factor from Modified-LU computes the T so that $A = (I - YTY_1^T)R$. To compute T directly from Y (as is necessary if Householder-QR is used) requires $O(nb^2)$ flops; here the triangular solve involves $O(b^3)$ flops. Our approach for computing T is given in line 4, and line 5 ensures sign agreement between the columns of the (implicitly stored) orthogonal factor and rows of R.

Algorithm 6.2.2 $[Y, T, R] = \text{TSQR-HR}(A)$

Require: A is $m \times b$ matrix distributed in block row layout

1: $[\{Y_{i,k}\}, \tilde{R}] = \text{TSQR}(A)$
2: $Q = \text{Construct-TSQR-}Q(\{Y_{i,k}\})$
3: $[Y, U, S] = \text{Modified-LU}(Q)$
4: $T = -USY_1^{-T}$
5: $R = S\tilde{R}$

Ensure: $A = (I - YTY_1^T)R$, where Y is $m \times b$ and unit lower triangular, Y_1 is top $b \times b$ block of Y, and T and R are $b \times b$ and upper triangular

On p processors, Algorithm 6.2.2 incurs the following costs (ignoring lower order terms):

1. Compute $[\{Y_{i,k}\}, R'] = \text{TSQR}(A)$

   The computational costs of this step come from lines 1 and 6 in Algorithm 6.1.2. Line 1 corresponds to a QR factorization of a $(m/p) \times b$ matrix, with a flop count of $2(m/p)b^2 - 2b^3/3$

(each processor performs this step simultaneously). Line 6 corresponds to a QR factorization of a $b \times b$ triangle stacked on top of a $b \times b$ triangle. Exploiting the sparsity structure, the flop count is $2b^3/3$; this occurs at every internal node of the binary tree, so the total cost in parallel is $(2b^3/3) \log p$.

The communication costs of Algorithm 6.1.2 occur in lines 5 and 8. Since every $R_{i,k}$ is a $b \times b$ upper triangular matrix, the cost of a single message is $\alpha + \beta \cdot (b^2/2)$. This occurs at every internal node in the tree, so the total communication cost in parallel is a factor $\log p$ larger.

Thus, the cost of this step is

$$\gamma \cdot \left(\frac{2mb^2}{p} + \frac{2b^3}{3} \log p \right) + \beta \cdot \left(\frac{b^2}{2} \log p \right) + \alpha \cdot \log p.$$

2. $Q =$ Construct-TSQR-$Q(\{Y_{i,k}\})$

The computational costs of this step come from lines 6 and 10 in Algorithm 6.1.4. Note that for $k > 0$, $Y_{i,k}$ is a $2b \times b$ matrix: the identity matrix stacked on top of an upper triangular matrix. Furthermore, $Q_{i,k}$ is an upper triangular matrix. Exploiting the structure of $Y_{i,k}$ and $Q_{i,k}$, the cost of line 6 is $2b^3/3$, which occurs at every internal node of the tree. Each $Y_{i,0}$ is a $(m/p) \times b$ lower triangular matrix of Householder vectors, so the cost of applying them to an upper triangular matrix in line 10 is $2(m/p)b^2 - 2b^3/3$. Note that the computational cost of these two lines is the same as those of the previous step in Algorithm 6.1.2.

The communication pattern of Algorithm 6.1.4 is identical to Algorithm 6.1.2, so the communication cost is also the same as that of the previous step.

Thus, the cost of constructing Q is

$$\gamma \cdot \left(\frac{2mb^2}{p} + \frac{2b^3}{3} \log p \right) + \beta \cdot \left(\frac{b^2}{2} \log p \right) + \alpha \cdot \log p.$$

3. $[Y, U, S] =$ Modified-LU(Q)

Ignoring lines 3–4 in Algorithm 6.2.1, Modified-LU is the same as LU without pivoting. In parallel, the algorithm consists of a $b \times b$ (modified) LU factorization of the top block followed by parallel triangular solves to compute the rest of the lower triangular factor. The flop count of the $b \times b$ LU factorization is $2b^3/3$, and the cost of each processor's triangular solve is $(m/p)b^2$. The communication cost of parallel Modified-LU is only that of a broadcast of the upper triangular $b \times b$ U factor (for which we use a bidirectional-exchange algorithm): $\beta \cdot (b^2) + \alpha \cdot (2 \log p)$.

Thus, the cost of this step is

$$\gamma \cdot \left(\frac{mb^2}{p} + \frac{2b^3}{3} \right) + \beta \cdot b^2 + \alpha \cdot 2 \log p$$

4. $T = -USY_1^{-T}$ and $R = SR'$

   The last two steps consist of computing T and scaling the final R appropriately. Since S is a sign matrix, computing US and SR' requires no floating point operations and can be done locally on one processor. Thus, the only computational cost is performing a $b \times b$ triangular solve involving Y_1^T. If we ignore the triangular structure of the output, the flop count of this operation is b^3. However, since this operation occurs on the same processor that computes the top $m/p \times b$ block of Y and U, it can be overlapped with the previous step (Modified-LU). After the top processor performs the $b \times b$ LU factorization and broadcasts the U factor, it computes only $m/p - b$ rows of Y (all other processors update m/p rows). Thus, performing an extra $b \times b$ triangular solve on the top processor adds no computational cost to the critical path of the algorithm.

Thus, TSQR-HR(A) where A is m-by-b incurs the following costs (ignoring lower order terms):

$$\gamma \cdot \left(\frac{5mb^2}{p} + \frac{4b^3}{3} \log p \right) + \beta \cdot \left(b^2 \log p \right) + \alpha \cdot (4 \log p) . \tag{6.2.1}$$

Note that the LU factorization required in Yamamoto's approach (see Section 6.1.3) is equivalent to performing Modified-LU($-Q_1$). In Algorithm 6.2.2, the Modified-LU algorithm is applied to an $m \times b$ matrix rather than to only the top $b \times b$ block; since no pivoting is required, the only difference is the update of the bottom $m - b$ rows with a triangular solve. Thus it is not hard to see that, ignoring signs, the Householder basis-kernel representation in Equation equation 6.1.1 can be obtained from the representation given in Equation equation 6.1.2 with two triangular solves: if the LU factorization gives $I - Q_1 = LU$, then $Y = \tilde{Y}U^{-1}$ and $T = UL^{-T}$. Indeed, performing these two operations and handling the signs correctly gives Algorithm 6.2.2.

While Yamamoto's approach avoids performing the triangular solve on Q_2, it still involves performing both TSQR and Construct-TSQR-Q. Avoiding the triangular solve saves 20% of the arithmetic of the panel factorization with Householder reconstruction, though we found in our performance experiments that the triangular solve accounts for only about 10% of the running time (mostly due to the broadcast of the triangular factor).

The main advantages of TSQR-HR over Yamamoto's algorithm are that the storage of the basis-kernel representation is more compact (since Y is unit lower triangular and T is upper triangular) and that this basis-kernel representation is backward-compatible with (Sca)LAPACK and other libraries using the compact WY representation [139], offering greater performance portability.

6.2.2 CAQR-HR

We refer to the 2D algorithm that uses TSQR-HR for panel factorizations as CAQR-HR. Because Householder-QR and TSQR-HR generate the same representation as output of the panel factorization, the trailing matrix update can be performed in exactly the same way. Thus, the only difference between 2D-Householder-QR and CAQR-HR, presented in Algorithm 6.2.3, is the subroutine call for the panel factorization (line 3).

Algorithm 6.2.3 $[Y, T, R] = \text{CAQR-HR}(A)$

Require: A is $m \times n$ and distributed block-cyclically on $p = p_r \cdot p_c$ processors with block size b, so that each $b \times b$ block A_{ij} is owned by processor $\Pi(i, j) = (i \bmod p_r) + p_r \cdot (j \bmod p_c)$

1: **for** $i = 0$ to $n/b - 1$ **do**
2: *% Compute TSQR and reconstruct Householder representation using column of p_r processors*
3: $\left[Y_{i:m/b-1,i}, T_i, R_{ii}\right] = \text{Hh-Recon-TSQR}(A_{i:m/b-1,i})$
4: *% Update trailing matrix using all p processors*
5: $\Pi(i, i)$ broadcasts T_i to all other processors
6: **for** $r \in [i, m/b - 1], c \in [i + 1, n/b - 1]$ **do in parallel**
7: $\Pi(r, i)$ broadcasts Y_{ri} across proc. row $\Pi(r, \mathbf{:})$
8: $\Pi(r, c)$ computes $\tilde{W}_{rc} = Y_{ri}^T \cdot A_{rc}$
9: Allreduce $W_c = \sum_r \tilde{W}_{rc}$ along proc. column $\Pi(\mathbf{:}, c)$
10: $\Pi(r, c)$ computes $A_{rc} = A_{rc} - Y_{ri} \cdot T_i^T \cdot W_c$
11: Set $R_{ic} = A_{ic}$

Ensure: $A = \left(\prod_{i=1}^{n} (I - Y_{:,i} T_i Y_{:,i}^T)\right) R$

	Flops	**Words**	**Messages**
2D-Householder-QR	$\frac{2mn^2 - 2n^3/3}{p}$	$\frac{2mn + n^2/2}{\sqrt{p}}$	$n \log p$
CAQR	$\frac{2mn^2 - 2n^3/3}{p}$	$\frac{2mn + n^2 \log p}{\sqrt{p}}$	$\frac{7}{2}\sqrt{p} \log^3 p$
CAQR-HR	$\frac{2mn^2 - 2n^3/3}{p}$	$\frac{2mn + n^2/2}{\sqrt{p}}$	$6\sqrt{p} \log^2 p$
Scatter-Apply CAQR	$\frac{2mn^2 - 2n^3/3}{p}$	$\frac{2mn + n^2/2}{\sqrt{p}}$	$7\sqrt{p} \log^2 p$

Table 6.2.2: Costs of QR factorization of $m \times n$ matrix distributed over p processors in 2D fashion. Here we assume a square processor grid ($p_r = p_c$). We also choose block sizes for each algorithm independently to ensure the leading order terms for flops are identical.

Algorithm 6.2.3 with block size b, and matrix m-by-n matrix A, with $m \geq n$ and $m, n \equiv 0 \pmod b$ distributed on a 2D p_r-by-p_c processor grid incurs the following costs over all n/b iterations.,

1. Compute $\left[Y_{i:m/b-1,i}, T_i, R_{ii}\right] = \text{Hh-Recon-TSQR}\left(A_{i:m/b-1,i}\right)$

   Equation equation 6.2.1 gives the cost of a single panel TSQR factorization with Householder reconstruction. We can sum over all iterations to obtain the cost of this step in the 2D QR algorithm (line 3 in Algorithm 6.2.3),

$$\sum_{i=0}^{n/b-1} \left(\gamma \cdot \left(\frac{5(m - ib)b^2}{p_r} + \frac{4b^3}{3} \log p_r\right) + \beta \cdot \left(b^2 \log p_r\right) + \alpha \cdot (4 \log p_r)\right) =$$
$$\gamma \cdot \left(\frac{5mnb}{p_r} - \frac{5n^2 b}{2p_r} + \frac{4nb^2}{3} \log p_r\right) + \beta \cdot (nb \log p_r) + \alpha \cdot \left(\frac{4n \log p_r}{b}\right)$$

2. $\Pi(i,i)$ broadcasts T_i to all other processors

   The matrix T_i is b-by-b and triangular, so we use a bidirectional exchange broadcast. Since there are a total of n/b iterations, the total communication cost for this step of Algorithm 6.2.3 is

$$\beta \cdot (nb) + \alpha \cdot \left(\frac{2n}{b} \log p \right)$$

3. $\Pi(r,i)$ broadcasts Y_{ri} to all processors in its row, $\Pi(r,:)$

   At this step, each processor which took part in the TSQR and Householder reconstruction sends its local chunk of the panel of Householder vectors to the rest of the processors. At iteration i of Algorithm 6.2.3, each processor owns at most $\lceil \frac{m/b-i}{p_r} \rceil$ blocks Y_{ri}. Since all the broadcasts happen along processor rows, we assume they can be done simultaneously on the network. The communication along the critical path is then given by

$$\sum_{i=0}^{n/b-1} \left(\beta \cdot \left(\frac{2(m/b-i) \cdot b^2}{p_r} \right) + \alpha \cdot 2 \log p_c \right) = \beta \cdot \left(\frac{2mn - n^2}{p_r} \right) + \alpha \cdot \left(\frac{2n}{b} \log p_c \right)$$

4. $\Pi(r,c)$ computes $\tilde{W}_{rc} = Y_{ri}^T \cdot A_{rc}$

   Each block $\tilde{W}_{rc}$ is computed on processor $\Pi(r,c)$, using A_{rc}, which it owns, and Y_{ri}, which it just received from processor $\Pi(r,i)$. At iteration i, processor j may own up to $\lceil \frac{m/b-i}{p_r} \rceil \cdot \lceil \frac{n/b-i}{p_c} \rceil$ blocks $\tilde{W}_{rc}$, $\Pi(r,c) = j$. Each, block-by-block multiply incurs $2b^3$ flops, therefore, this step incurs a total computational cost of

$$\gamma \cdot \left(\sum_{i=0}^{n/b-1} \frac{2(m/b-i)(n/b-i)b^3}{p} \right) = \gamma \cdot \left(\frac{mn^2 - n^3/3}{p} \right)$$

5. Allreduce $W_c = \sum_r \tilde{W}_{rc}$ so that processor column $\Pi(:,c)$ owns W_c

   At iteration i of Algorithm 6.2.3, each processor j may own up to $\lceil \frac{m/b-i}{p_r} \rceil \cdot \lceil \frac{n/b-i}{p_c} \rceil$ blocks W_{rc}. The local part of the reduction can be done during the computation of W_{rc} on line 8 of Algorithm 6.2.3. Therefore, no process should contribute more than $\lceil \frac{n/b-i}{p_c} \rceil$ b-by-b blocks W_{rc} to the reduction. Using a bidirectional exchange all-reduction algorithm and summing over the iterations, we can obtain the cost of this step throughout the entire execution of the 2D algorithm:

$$\sum_{i=0}^{n/b-1} \left(\beta \cdot \left(\frac{2(n/b-i) \cdot b^2}{p_c} \right) + \alpha \cdot 2 \log p_r \right) = \beta \cdot \left(\frac{n^2}{p_c} \right) + \alpha \cdot \left(\frac{2n}{b} \log p_r \right)$$

6. $\Pi(r, c)$ computes $A_{rc} = A_{rc} - Y_{ri} \cdot T_i \cdot W_c$

   Since in our case, $m \geq n$, it is faster to first multiply T_i by W_c rather than Y_{ri} by T_i. Any processor j may update up to $\lceil \frac{m/b-i}{p_r} \rceil \cdot \lceil \frac{n/b-i}{p_c} \rceil$ blocks of A_{rc} using $\lceil \frac{m/b-i}{p_r} \rceil$ blocks of Y_{ri} and $\lceil \frac{n/b-i}{p_c} \rceil$ blocks of W_c. When multiplying T_i by W_c, we can exploit the triangular structure of T, to lower the flop count by a factor of two. Summed over all iterations, these two multiplications incur a computational cost of

$$\gamma \cdot \left(\sum_{i=0}^{n/b-1} \frac{2(m/b-i)(n/b-i)b^3}{p} + \frac{(n/b-i)b^3}{p_c} \right) = \gamma \cdot \left(\frac{mn^2 - n^3/3}{p} + \frac{n^2 b}{2p_c} \right)$$

The overall costs of CAQR-HR are given to leading order by

$$\gamma \cdot \left(\frac{10mnb - 5n^2 b}{2p_r} + \frac{4nb^2}{3} \log p_r + \frac{n^2 b}{2p_c} + \frac{2mn^2 - 2n^3/3}{p} \right) +$$
$$\beta \cdot \left(nb \log p_r + \frac{2mn - n^2}{p_r} + \frac{n^2}{p_c} \right) + \alpha \cdot \left(\frac{8n}{b} \log p_r + \frac{4n}{b} \log p_c \right).$$

If we pick $p_r = p_c = \sqrt{p}$ (assuming $m \approx n$) and $b = \frac{n}{\sqrt{p} \log p}$ then we obtain the leading order costs

$$\gamma \cdot \left(\frac{2mn^2 - 2n^3/3}{p} \right) + \beta \cdot \left(\frac{2mn + n^2/2}{\sqrt{p}} \right) + \alpha \cdot \left(6\sqrt{p} \log^2 p \right),$$

shown in the third row of Table 6.2.2.

Comparing the leading order costs of CAQR-HR with the existing approaches, we note again the $O(n \log p)$ latency cost incurred by the 2D-Householder-QR algorithm. CAQR and CAQR-HR eliminate this synchronization bottleneck and reduce the latency cost to be independent of the number of columns of the matrix. Further, both the bandwidth and latency costs of CAQR-HR are factors of $O(\log p)$ lower than CAQR (when $m \approx n$). As previously discussed, CAQR includes an extra leading order bandwidth cost term ($\beta \cdot n^2 \log p / \sqrt{p}$), as well as a computational cost term ($\gamma \cdot (n^2 b/p_c) \log p_r$) that requires the choice of a smaller block size and leads to an increase in the latency cost.

6.2.3 Two-Level Aggregation

The Householder-QR algorithm attains an efficient trailing matrix update by aggregating Householder vectors into panels (the compact-WY representation). Further, it is straightforward to combine aggregated sets (panels) of Householder vectors into a larger aggregated form. While it is possible to aggregate any basis-kernel representation in this way [153, Corollary 2.8], the Householder form allows for maintaining trapezoidal structure of the basis and triangular structure of

the kernel (note that Yamamoto's representation would yield a block-trapezoidal basis and block-triangular kernel). We will refer to the aggregation of sets of Householder vectors as two-level aggregation.

Given the Householder vector reconstruction technique, two-level aggregation makes it possible to decouple the block sizes used for the trailing matrix update from the width of each TSQR. Adding the second blocking parameter to achieve two-level aggregation is a simple algorithmic optimization to our 2D algorithm, and does not change the leading order interprocessor communication costs. This two-level algorithm is given in full in the technical report [7]; it calls Algorithm 6.2.3 recursively on large panels of the matrix and then performs an aggregated update on the trailing matrix. While this algorithm does not lower the interprocessor communication, it lowers the local memory-bandwidth cost associated with reading the trailing matrix from memory. This two-level aggregation is analogous to the two-level blocking technique employed in [146] for LU factorization, albeit only on a 2D grid of processors. We also implemented a 2D Householder algorithm with two-level aggregation, which employs ScaLAPACK to factor each thin panel. We refer to this algorithm as Two-Level 2D Householder. We note that ScaLAPACK could be easily modified to use this algorithm with the addition of a second algorithmic blocking factor. Both of our two-level algorithms obtained a significant performance improvement over their single-level aggregated counterparts.

6.2.4 Scatter-Apply CAQR

We also found an alternative method for improving the CAQR trailing matrix update that does not reconstruct the Householder form. A major drawback with performing the update via a binary tree algorithm is heavy load imbalance. This problem may be resolved by exploiting the fact that each column of the trailing matrix may be updated independently and subdividing the columns among more processors to balance out the work. This can be done with ideal load balance using a butterfly communication network instead of a binary tree.

Doing the CAQR trailing matrix update via a butterfly network requires storing the implicit representation of the Householder vectors redundantly. We compute the Householder vectors redundantly by doing the TSQR via a butterfly network as done in [81]. Algorithm 6.2.4 shows how the trailing matrix update can be efficiently computed using a butterfly communication network, which effectively performs recursive halving on the columns of the trailing matrix, then recombines the computed updates via an inverse butterfly network (recursive doubling). We call this algorithm Scatter-Apply TSQR-Q^T to emphasize that its structure is analogous to performing a broadcast via a scatter-allgather algorithm, which generalizes recursive halving and doubling and lowers the asymptotic bandwidth cost of a large message broadcast over a simple binary tree broadcast by a factor of $O(\log p)$. Within the context of a 2D QR implementation, Algorithm 6.2.4 would be used for each processor column.

Algorithm 6.2.4 reduces the bandwidth and computational costs of the trailing matrix update by a factor of $O(\log p)$, since these costs are now dominated by the first level of the butterfly. The leading order costs of a CAQR algorithm which uses Scatter-Apply TSQR-Q^T for the trailing

matrix update are

$$\gamma \cdot \left(\frac{2mn^2 - 2n^3/3}{p} \right) + \beta \cdot \left(\frac{2mn + 2n^2}{\sqrt{p}} \right) + \alpha \cdot \left(7\sqrt{p} \log^2 p \right).$$

We extended Algorithm 6.2.4 to a non-power-of-two number of processes via an additional level of the butterfly, which cuts to the nearest power-of-two, though there are alternatives which could be cheaper. An interesting remaining question is whether pipelined CAQR with flat trees, such as the algorithms presented in [52] can yield the same improvement in costs as Algorithm 6.2.4.

Algorithm 6.2.4 $[B] =$ Scatter-Apply TSQR-Q$^T(\{Y_{i,k}\}, A)$

Require: No. of processors p is a power of 2 and i is the processor index

Require: A is $m \times n$ matrix distributed in block row layout; A_i is processor i's block; and $\{Y_{i,k}\}$ is the implicit representation of b Householder vectors computed via a butterfly TSQR

$B_i =$ Apply-Householder-$Q^T(Y_{i,0}, A_i)$
Let $\bar{B}_i$ be the first b rows of B_i
for $k = 1$ to $\log p$ **do**
  $j = 2^k \lfloor \frac{i}{2^k} \rfloor + (i + 2^{k-1} \bmod 2^k)$
  Let $\bar{B}_i = [\bar{B}_{i1}, \bar{B}_{i2}]$ where each block is b-by-$n/2^k$
  if $i < j$ **then**
    Swap $\bar{B}_{i2}$ with $\bar{B}_{j1}$ from processor j
    $\begin{bmatrix} \bar{B}_i \\ \bar{B}^k_{j1} \end{bmatrix} =$ Apply-Householder-$Q^T \left(Y_{i,k}, \begin{bmatrix} \bar{B}_{i1} \\ \bar{B}_{j1} \end{bmatrix} \right)$
  else
    Swap $\bar{B}_{j2}$ with $\bar{B}_{i1}$ from processor j
    $\begin{bmatrix} \bar{B}_i \\ \bar{B}^k_{i2} \end{bmatrix} =$ Apply-Householder-$Q^T \left(Y_{i,k}, \begin{bmatrix} \bar{B}_{i2} \\ \bar{B}_{j2} \end{bmatrix} \right)$
for $k = \log p$ down to 1 **do**
  $j = 2^k \lfloor \frac{i}{2^k} \rfloor + (i + 2^{k-1} \bmod 2^k)$
  if $i < j$ **then**
    Swap $\bar{B}^k_{j1}$ with $\bar{B}_j$ from processor j
    $\bar{B}_i = [\bar{B}_i, \bar{B}_j]$
  else
    Swap $\bar{B}_i$ with $\bar{B}^k_{i1}$ from processor j
    $\bar{B}_i = [\bar{B}^k_{i1}, \bar{B}^k_{i2}]$
Set the first b rows of B_i to $\bar{B}_i$

Ensure: $B = Q^T A$ where Q is the orthogonal matrix implicitly represented by $\{Y_{i,k}\}$

6.3 Performance

Having established the stability of our algorithm, we now analyze its experimental performance. We demonstrate that for tall and skinny matrices TSQR-HR achieves better parallel scalability than

library implementations (ScaLAPACK and Elemental) of Householder-QR. Further, we show that for square matrices Two-Level CAQR-HR outperforms our implementation of CAQR, and library implementations of 2D-Householder-QR.

6.3.1 Architecture

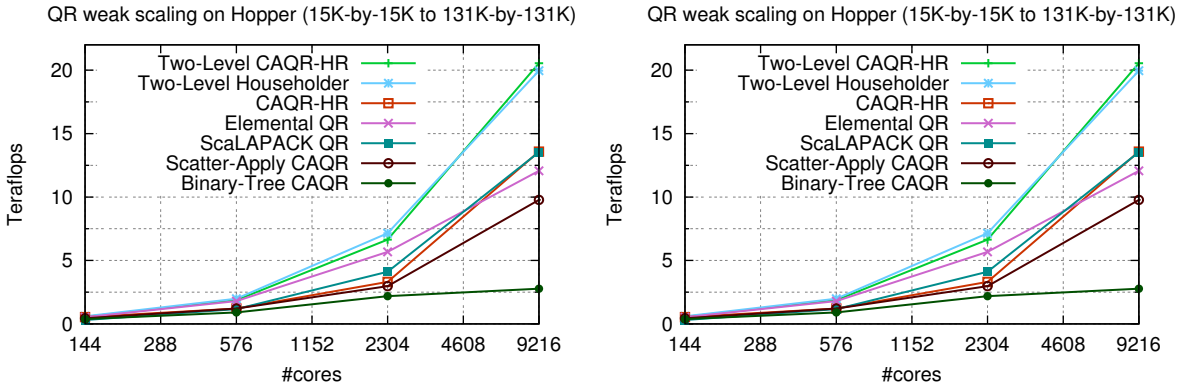
The experimental platform is "Hopper," which is a Cray XE6 supercomputer, built from dual-socket 12-core "Magny-Cours" Opteron compute nodes. We used the Cray LibSci BLAS routines. This machine is located at the NERSC supercomputing facility. Each node can be viewed as a four-chip compute configuration due to NUMA domains. Each of these four chips have six super-scalar, out-of-order cores running at 2.1 GHz with private 64 KB L1 and 512 KB L2 caches. Nodes are connected through Cray's "Gemini" network, which has a 3D torus topology. Each Gemini chip, which is shared by two Hopper nodes, is capable of 9.8 GB/s bandwidth.

6.3.2 Parallel Scalability

In this section, we give performance results based on our C++/MPI/LAPACK implementations of TSQR, TSQR-HR, Two-Level CAQR-HR, CAQR, Scatter-Apply CAQR, and Two-Level 2D Householder, as well as two library implementations of 1D Householder-QR and 2D-Householder-QR, Elemental (version 0.80) and ScaLAPACK (native LibSci installation on Hopper, October 2013). Our implementations aim to do minimal communication and arithmetic, and do not employ low-level tuning or overlap between communication and computation. All the benchmarks use one MPI process per core, despite the fact that is favorable on Hopper to use one process per socket and six threads per process. This decision was made because we observed that some of the many LAPACK routines used throughout our codes (`geqrf`, `ormqr`, `tpqrt`, `tmpqrt`, etc.) were not threaded.

First, we study the performance of QR factorization of tall-skinny matrices using a 1D processor grid. Figure 6.1(a) gives the strong scaling performance for a matrix of size $122{,}880$-by-32. We also tested a range of reasonable panel sizes that are not detailed here and observed similar performance trends. We observe from Figure 6.1(a) that TSQR-HR takes roughly twice the execution time of TSQR, which is in line with our theoretical cost analysis. Figure 6.1(a) also gives the time to solution of Elemental and ScaLAPACK, which both use the Householder-QR algorithm, albeit with different matrix blocking and collectives. We see that TSQR obtains a performance benefit over Householder-QR due to the lower synchronization cost and TSQR-HR preserves the scaling behavior and remains competitive with Householder-QR.

We collected these results by taking the best observed time over a few runs including ones where a subset of the nodes in the scheduled partition was used. We note that ScaLAPACK performance was highly variable and benefited significantly from using only a fraction of the partition. For instance, on 768 nodes the best ScaLAPACK observed performance was 3.4 ms for this problem size when using half of a 1536 node partition, but over 7 ms when using a 768 node partition. This variability could be justified by the hypothesis that using a subset of a partition on Hopper yields better locality on the network, which alleviates the latency bottleneck of the Householder-



(a) Tall-skinny QR performance on Cray XE6

(b) Square QR performance on Cray XE6

QR algorithm. This claim is supported by the fact that the performance variability of algorithms employing TSQR was smaller and much less benefit was yielded from these algorithms being executed on a subset of a partition.

Second, we study the parallel scaling of QR factorization on square matrices. In Figure 6.1(b), we compare our implementation of CAQR with a binary tree update (no pipelining or other optimizations), Scatter-Apply CAQR, CAQR-HR, and Two-Level CAQR-HR, with Elemental and ScaLAPACK, which use 2D-Householder-QR, as well as Two-Level 2D Householder. We tuned the block sizes of all the codes (the Two-Level CAQR-HR required tuning two block sizes), though fewer data points were collected for larger scale runs, due to timing and allocation constraints.

Comparing the performance of Two-Level CAQR-HR and CAQR-HR in Figure 6.1(b), we observe that significant benefit is obtained from aggregating the trailing matrix update. Similarly, we note that two-level aggregation of the 2D Householder algorithm yields a similar performance improvement (compare ScaLAPACK with Two-Level 2D Householder). On the other hand, the binary-tree CAQR performance is relatively poor due to the overhead of the implicit tree trailing update. This overhead is significantly alleviated by the Scatter-Apply TSQR-Q^T algorithm for the trailing matrix update, though the Scatter-Apply CAQR is still slower than algorithms which perform the trailing matrix update using the Householder form.

Overall, these results suggest that latency cost is not a significant overhead on this platform, though as explained in the performance analysis of the 1D algorithms, heavy latency cost contributes to performance variability. Further, other architectures such as cloud and grid environments typically have higher latency payloads. We also expect that the relative cost of messaging latency will grow in future architectures and larger scales of parallelism as the topological distance between computing elements grows. Lastly, we note that for Elemental, ScaLAPACK, and all of our QR implementations, it was often better to utilize a rectangular processor grid with more rows than columns. Having more processes in each column of the processor grid accelerates the computation of each tall-skinny panel.

6.4 2.5D QR Factorization

Having found a method to perform TSQR and then reconstruct the Householder vector representation paves the way for the design of a synchronization and interprocessor communication efficient 2.5D QR algorithm. In designing this algorithm, we have to consider potential memory-bandwidth overheads. So, we start by restating the communication costs of sequential QR and TSQR with memory-bandwidth factors that are dependent on the cache size $\hat{M}$.

We will use both parallel and sequential versions of the Communication-Avoiding QR (CAQR) algorithm, which is based on an improved algorithm for computing QR factorizations of tall and skinny matrices known as Tall-Skinny QR (TSQR). The TSQR and CAQR algorithms are analyzed in terms of communication cost in [47], although the idea of doing a QR reduction tree with Givens rotations goes back to [68], a blocked flat tree approach (optimal sequentially) was presented in [72], and a parallel block reduction tree approach was given earlier in [42] (more relevant references given in [47]). Lower bounds for the communication costs of QR factorization are proven under some assumptions in [12].

6.4.1 Sequential QR Factorization

We first consider sequential QR factorization and its memory bandwidth cost for a given cache size. The cost of this sequential algorithm is analogous to that of sequential matrix multiplication, except in this case only one matrix needs to be read from memory to cache.

Lemma 6.4.1. *Given $m \times n$ dense matrix A with $m \geq n$, the cost of a sequential QR factorization of A is*

$$T_{\text{QR}}(m, n, \hat{M}) = O\left(\gamma \cdot mn^2 + \nu \cdot \left[\frac{mn^2}{\sqrt{\hat{M}}} + mn\right]\right).$$

Proof. We use the sequential CAQR algorithm (and its analysis) given by Demmel et al. [47]. □

6.4.2 Parallel QR Factorization for Tall-and-Skinny Matrices

An important use-case of parallel QR is the factorization of tall-and-skinny matrices. Tall-and-skinny QR factorizations are needed both in applications which employ orthogonalization of subspaces, as well as within dense linear algebra algorithms such as the symmetric eigensolvers and square-matrix QR algorithms, in both of which QR is done on matrix subpanels. We now consider the memory-bandwidth cost involved in the TSQR algorithm (given in Section 6.1).

Lemma 6.4.2. *Given a $m \times n$ matrix A with $m/p \geq n$, on any architecture 1 processor and with $M > \frac{mn}{p} + n^2$ memory and cache size $\hat{M}$ a QR factorization of A may be performed with the cost*

$$T_{\text{QR}}(m,n,p,\hat{M}) = O\left(\gamma \cdot \left[\frac{mn^2}{p} + n^3\log(p)\right] + \beta \cdot n^2\log(p) \right.$$
$$\left. + \nu \cdot \left[\frac{mn^2/p + n^3\log(p)}{\sqrt{\hat{M}}} + \frac{mn}{p}\right] + \alpha \cdot \log(p)\right).$$

Proof. We use the parallel TSQR algorithm presented in [47]. The analysis for all but the memory bandwidth cost is provided in [47].

In order to determine the memory bandwidth cost, we assume use of the sequential CAQR algorithm of for all local QR decompositions and appeal to Lemma 6.4.1. Along the critical path of the algorithm, the local computations consist of one QR decomposition of an $(m/p) \times n$ matrix and $\log(p)$ QR decompositions of $(2n) \times n$ matrices (we ignore the sparsity structure of these matrices here). Thus, the total memory bandwidth cost is $(m/p)n^2/\sqrt{\hat{M}} + (m/p)n + (n^3/\sqrt{\hat{M}} + n^2)\log(p)$, which simplifies to the expression above. □

6.4.3 Parallel QR Factorization for Square Matrices

Having established the memory-bandwidth cost of an algorithm for factorizing tall-and-skinny matrices, we now consider another special case, namely when the matrices are square. The following algorithm, due to Alexander Tiskin [158] achieves an optimal communication cost according to the lower bounds in [12], and is also optimal (modulo a $\log(p)$ factor) in synchronization assuming that the tradeoffs proven for Cholesky in [145] also apply to QR.

Lemma 6.4.3. *Let $\frac{1}{2} \leq \delta \leq \frac{2}{3}$. Given a $n \times n$ matrix A on any architecture p processors and with $M > \frac{n^2}{p^{2(1-\delta)}}$ memory and cache size $\hat{M}$ a QR factorization of A may be performed with the cost*

$$T_{\text{QR}}(n,p,\hat{M},\delta) = O\left(\gamma \cdot n^3/p + \beta \cdot \frac{n^2}{p^\delta} + \nu \cdot \frac{n^3}{p\sqrt{\hat{M}}} + \alpha \cdot p^\delta \log(p)\right).$$

Proof. Tiskin's pseudo-panel QR algorithm [158] should achieve this cost for square matrices. The memory-bandwidth complexity was not analyzed in Tiskin's paper, however, since at each level of recursion the algorithm performs a trailing matrix update where the trailing matrix is not significantly larger than the matrices with which it is updated, there is no reason the memory bandwidth complexity should be higher than the interprocessor communication cost, aside from a factor corresponding to reading the inputs to the computation into cache, in particular the overall memory bandwidth cost $\hat{W}$ should be the following function of the bandwidth cost W and cache size $\hat{M}$,

$$\hat{W} = O\left(W + \frac{n^3}{p\sqrt{\hat{M}}}\right).$$

Tiskin's algorithm was formulated under a BSP model, which counts global synchronizations (BSP timesteps), but each BSP timestep may incur a higher synchronization cost in our model, if a processor receives or sends many messages. We argue that the synchronization cost of Tiskin's algorithm incurs an extra factor of $\Theta(\log(p))$ under our model. In factorizing the panel during the base case, each processor performs two QR factorizations and updates on a small local subset of the matrix, so near-neighbor synchronizations should suffice. However, when performing the trailing matrix update at the base case, two n-by-n banded matrices with bandwidth $k = n/p^{\delta}$ are multiplied by partitioning the nk^2 multiplications into cubic blocks of size nk^2/p. The dimension of each such block is then $\Theta((nk^2/p)^{1/3})$, and since the number of non-trivial multiplications is $\Theta(k)$ in each of three directions from any point in the 3D graph of multiplications, the number of processors which must receive the same inputs or contribute to the same outputs (whose cubic blocks overlap in one of the directions) yields the group of processors which must synchronize at each step of recursion,

$$\hat{p} = \Theta\left(k/[nk^2/p]^{1/3}\right) = \Theta\left(\left[\frac{pk}{n}\right]^{1/3}\right) = \Theta(p^{1-\delta}).$$

Each group of $\hat{p}$ processors can synchronize with $O(\log(\hat{p})) = O(\log(p))$ synchronizations, and the base case is reached $O(p^{\delta})$ times, so the latency cost is $\Theta(p^{\delta}\log(p))$. □

6.4.4 Parallel QR Factorization for Rectangular Matrices

We now give a 2.5D QR algorithm, which achieves the desired costs for rectangular matrices Its cost generalize those of the TSQR algorithm and Tiskin's QR algorithm for square matrices [158]. This algorithm uses Householder transformations rather than Givens rotations as done in [158], and does not require embedding in a slanted matrix panel as needed in [158].

Lemma 6.4.4. *Let $\frac{1}{2} < \delta \leq \frac{2}{3}$. Given a $m \times n$ matrix A (with $1 \leq m/n \leq p/\log(p)$) on any architecture with p processors, $M > \left(\frac{n(m/n)^{1-\delta}}{p^{1-\delta}}\right)^2$ memory, and cache size $\hat{M}$ a QR factorization of A may be performed with the cost*

$$T_{\mathrm{QR}}(m,n,p,\hat{M},\delta) = O\left(\gamma \cdot mn^2/p + \beta \cdot \frac{m^{\delta}n^{2-\delta}}{p^{\delta}} + \nu \cdot \frac{mn^2}{p\sqrt{\hat{M}}} + \alpha \cdot (np/m)^{\delta}\log(p)\right).$$

Proof. This cost may be achieved by an algorithm similar to the one given by Elmroth and Gustavson [55] that calls a recursive QR on a subset of the matrix columns with a subset of the processors and performs the trailing matrix update using 2.5D matrix multiplication. We present this approach in Algorithm 6.4.1 providing some of the details in the below cost analysis.

We quantify the costs of 2.5D-QR below. During the computations of the ith for loop iteration,

- the size of Y is m-by-in/k
- the size of T is in/k-by-in/k

Algorithm 6.4.1 $[Y, T, R] \leftarrow$ 2.5D-QR(A, Π, k, ζ)

Require: Let $k > 1$ and $\zeta \in [\frac{1}{2}, 1)$. Let Π be a set of p processors, define $\hat{\Pi} \subset \Pi$ be a group of p/k^{ζ} processors. Let A be a m-by-n matrix, and A_i for $i \in \{0, \ldots k-1\}$ be a m-by-n/k panel of A.

1: **if** $m/p > n$ **then**
2: Perform recursive tall-skinny matrix QR algorithm on A.
3: Let $Y = \{\}$ and $T = \{\}$
4: **for** $i = 0$ to $k - 1$ **do**
5: Compute $A_i = (I - YTY^T)^T A_i$
6: $[Y_i, T_i, R_i] \leftarrow$ 2.5D-QR$(A_i, \hat{\Pi}, k)$
7: Compute $\bar{T}_i = T(Y^T Y_i)T_i$
8: Let $T = \begin{bmatrix} T & \bar{T}_i \\ 0 & T_i \end{bmatrix}$ and $Y = \begin{bmatrix} Y & 0 \\ \vdots & Y_i \end{bmatrix}$

- the size of Y_i is m-by-n/k
- the size of T_i is n/k-by-n/k
- the size of $\bar{T}_i$ is in/k-by-n/k

We calculate the costs of each recursive level of 2.5D-QR using these matrix sizes,

- in the base-case, the cost of TSQR is given by Lemma 6.4.2

$$O\left(\gamma \cdot mn^2/p + \beta \cdot n^2 \log(p) + \nu \cdot (n^3 \log(p) + mn^2/p)/\sqrt{\hat{M}} + \alpha \cdot \log(p)\right).$$

- line 5 requires three multiplications which should be done right-to-left for a total cost of (using Theorem 4.4.2)

$$O\left(\gamma \cdot \frac{imn^2}{k^2 p} + \beta \cdot \left(\frac{imn}{kp} + \frac{imn^2}{k^2 p \cdot \sqrt{M}} + \left(\frac{imn^2}{k^2 p}\right)^{2/3}\right) + \nu \cdot \left(\frac{imn^2}{k^2 p \cdot \sqrt{\hat{M}}}\right) + \alpha \cdot \left(\frac{imn^2}{k^2 p \cdot M^{3/2}}\right)\right),$$

we obtain the cost of this line over all iterations by summing over the k iterations

$$O\left(\gamma \cdot \frac{mn^2}{p} + \beta \cdot \left(\frac{kmn}{p} + \frac{mn^2}{p \cdot \sqrt{M}} + k^{1/3}\left(\frac{mn^2}{p}\right)^{2/3}\right) + \nu \cdot \left(\frac{mn^2}{p \cdot \sqrt{\hat{M}}}\right) + \alpha \cdot \left(\frac{mn^2}{p \cdot M^{3/2}}\right)\right).$$

If we keep Y replicated throughout the k iterations we do not incur the factor of $k^{1/3}$ on the third interprocessor bandwidth term, instead achieving the cost of doing a single n-by-m times m-by-n parallel matrix multiplication. However, if the matrix does not fit in cache $\hat{M} < mn/p^{2(1-\delta)}$, we still incur a memory bandwidth cost with this extra factor, yielding an overall cost of

$$O\left(\gamma \cdot \frac{mn^2}{p} + \beta \cdot \left(\frac{kmn}{p} + \frac{mn^2}{p \cdot \sqrt{M}} + \left(\frac{mn^2}{p}\right)^{2/3}\right) + \nu \cdot \left(\frac{mn^2}{p \cdot \sqrt{\hat{M}}} + k^{1/3}\left(\frac{mn^2}{p}\right)^{2/3}\right) + \alpha \cdot \left(\frac{mn^2}{p \cdot M^{3/2}}\right)\right),$$

if the whole computation fits in cache, namely $\hat{M} > mn/p^{2(1-\delta)}$ we can keep Y in cache throughout multiple iterations, yielding the cost

$$O\left(\gamma \cdot \frac{mn^2}{p} + \beta \cdot \left(\frac{kmn}{p} + \frac{mn^2}{p \cdot \sqrt{M}} + \left(\frac{mn^2}{p}\right)^{2/3}\right) + \nu \cdot \left(\frac{mn^2}{p \cdot \sqrt{\hat{M}}}\right) + \alpha \cdot \left(\frac{mn^2}{p \cdot M^{3/2}}\right)\right).$$

Keeping Y replicated in this fashion is valuable if we want to pick $k = p^{\zeta - 1/2}$ rather than a constant.

- line 7 requires three matrix multiplication of the same or smaller cost as the rightmost multiplication of line 5, so the asymptotic cost is the same.

Therefore, the complete cost recurrence for the algorithm is

$$\begin{aligned} T_{\text{2.5D-QR}}(p,m,n,k,\zeta) =& k \cdot T_{\text{2.5D-QR}}(p/k^{\zeta}, m, n/k, k) \\ &+ O\left(\gamma \cdot \frac{mn^2}{p} + \beta \cdot \left[\frac{kmn}{p} + \frac{mn^2}{p \cdot \sqrt{M}} + \left(\frac{mn^2}{p}\right)^{2/3}\right]\right. \\ &\left. + \nu \cdot \left[\frac{mn^2}{p \cdot \sqrt{\hat{M}}} + k^{1/3}\left(\frac{mn^2}{p}\right)^{2/3}\right] + \alpha \cdot \left(\frac{mn^2}{p \cdot M^{3/2}}\right)\right) \end{aligned}$$

with the base case

$$\begin{aligned} T_{\text{2.5D-QR}}(p,m,m/p,k,\zeta) = O\Big(& \gamma \cdot m^3/p^3 + \beta \cdot m^2 \log(p)/p^2 \\ &+ \nu \cdot m^3 \log(p)/(p^3 \cdot \sqrt{\hat{M}}) + \alpha \cdot \log(p)\Big) \end{aligned}$$

If $M > mn/\left(\frac{n}{p^{1-\delta}q^{\delta-1}}\right)^2$, we now select $\zeta = (1-\delta)/\delta$, which implies that $\delta = 1/(1+\zeta)$. In order to avoid a $\log(p)$ factor on the bandwidth cost which arises when $\delta = \frac{1}{2}$ and $\zeta = 1$ as the cost $\beta\frac{mn^2}{p\cdot\sqrt{M}}$ stays the same at each recursive level, we restrict $\delta \in (1/2, 2/3]$, which means $\zeta < 1$ and implies that the cost of the aforementioned term decreases geometrically with each recursive level. Further, we pick k to be a constant, e.g. 4. For a m-by-n with $q_0 = m/n$ matrix starting with p_0 processors the recursion continues until reaching the base case problems of configuration

$$T_{\text{2.5D}-\text{QR}}(p_0^\delta q_0^{1-\delta}, m, m/(p_0^\delta q_0^{1-\delta}), k, \zeta)$$

There are $\log((p/q)^\delta)$ levels and $O((p/q)^\delta)$ subproblems on the last level. The term $\beta\cdot k^{1/3}\left(\frac{mn^2}{p}\right)^{2/3}$ is amplified at each recursive level by a factor of

$$k\cdot\left(\frac{1/k^2}{1/k^\zeta}\right)^{2/3} = k^{1+2/3(\zeta-2)} = k^{(1/3)(2\zeta-1)} = k^{(1/3)(2(1-\delta)/\delta-1)} = k^{2/3\delta-1}.$$

Therefore, at the last level the term is amplified to

$$\beta\cdot k^{\log((p/q)^\delta)(2/3\delta-1)}\cdot\left(\frac{mn^2}{p}\right)^{2/3} = \beta\cdot(p/q)^{2/3-\delta}\cdot\left(\frac{mn^2}{p}\right)^{2/3} = \beta\cdot\frac{mn}{p^\delta q^{1-\delta}}.$$

At the base case, we note that the base case term $\beta\cdot m^2\log(p)/p^2$ is greater than $\frac{mn}{p^\delta q^{1-\delta}}$ by a factor of $\log(p)$ when $q = p$ (in which case there would be only one level of recursion). However, we have restricted $q \le p/\log(p)$, and we now show that this is sufficient for the base-case term to become low order. We have that the recursive term is amplified to at least

$$\begin{aligned}\frac{mn}{p^\delta q^{1-\delta}} &\ge \frac{mn}{p^\delta(p/\log(p))^{1-\delta}} = \frac{mn}{p}\log(p)^{1-\delta}\\ &= \frac{m^2}{pq}\log(p)^{1-\delta} \ge \frac{m^2}{p^2}\log(p)^{2-\delta},\end{aligned}$$

since $\log(p)^{2-\delta} > \log(p)$ the recursive term always dominates the base-case term, so we can omit the $\log(p)$ factor from the interprocessor bandwidth cost. Therefore, given $M > \left(\frac{n}{p^{1-\delta}q^{\delta-1}}\right)^2$ for $\delta \in (1/2, 2/3]$ the overall cost is

$$\begin{aligned}T_{\text{2.5D}-\text{QR}}(m,n,p,\hat{M},\delta) &= O\left(\gamma\cdot mn^2/p + \beta\cdot\frac{mn}{p^\delta q^{1-\delta}} + \nu\cdot\frac{mn^2}{p\cdot\sqrt{\hat{M}}} + \alpha\cdot(p/q)^\delta\log(p)\right),\\ &= O\left(\gamma\cdot mn^2/p + \beta\cdot\frac{m^\delta n^{2-\delta}}{p^\delta} + \nu\cdot\frac{mn^2}{p\cdot\sqrt{\hat{M}}} + \alpha\cdot\left(\frac{np}{m}\right)^\delta\log(p)\right),\end{aligned}$$

alternatively, we can rewrite this for $mn/p < M < (n/p^{1-\delta}q^{\delta-1})^2$ as,

$$T_{\text{2.5D}-\text{QR}}(m,n,p,\hat{M},\delta) = O\left(\gamma \cdot mn^2/p + \beta \cdot \left(\frac{mn^2}{p \cdot \sqrt{M}}\right) + \nu \cdot \frac{mn^2}{p \cdot \sqrt{\hat{M}}} + \alpha \cdot \frac{p\sqrt{M}}{m}\log(p)\right).$$

□

Chapter 7

Computing the Eigenvalues of a Symmetric Matrix

We study algorithms for computation of the eigenvalue decomposition of a real symmetric matrix $A = QDQ^T$ where Q is an orthogonal matrix and D is a diagonal matrix of eigenvalues. The most efficient general approaches for solving this problem reduce A via orthogonal transformations to a tridiagonal matrix with the same eigenvalues, then employ a tridiagonal eigensolver algorithm to compute the eigenvalues of the tridiagonal matrix. The eigenvectors may be constructed by applying the orthogonal transformations backwards. We focus on computing the eigenvalues of the symmetric matrix and do not give algorithms for performing the orthogonal transformations needed to compute the eigenvectors. However, we note when our new algorithms require more work to perform the back-transformations needed to compute the eigenvectors.

A key motivation for distributed-memory parallelization of symmetric eigenvalue algorithms is their utility for electronic structure calculations. The most common numerical schemes employed in these methods are Hartree Fock (HF) [76, 60], which is also known as the Self-Consistent Field (SCF) iterative procedure. At each iteration of SCF, it is necessary to compute the eigendecomposition of dense symmetric matrix. Not only do HF and post-HF methods (e.g. Møller-Plesset perturbation theory [118] and the coupled-cluster method [166]) employ symmetric eigensolvers, but the commonly-used cheaper alternative, Density Functional Theory (DFT) [100, 83] also usually requires a symmetric eigensolve at each iteration, which is leading order in cost (although a number of alternative DFT adaptations which avoid solving the full eigenproblem exist [61]). Efficient parallelization of these methods requires fast parallel calculation of the symmetric eigenvalue decomposition of matrices that are often not very large, so that strong scaling (solving problems faster with more processors) is a key demand.

The computation of the eigenvalues of the tridiagonal matrix can be done in $O(n^2)$ work for a matrix with dimension n via QR or bisection [127] or the MRRR algorithm if eigenvectors are also desired [51]. The cost of the eigenvalue computation is therefore typically dominated by

This chapter is based on joint work with Grey Ballard and Nicholas Knight.

the reduction from dense to tridiagonal form. It has been further noted that in order to do the full to tridiagonal reduction with good memory-bandwidth efficiency (in either the sequential or parallel case), the matrix should first be reduced to banded form [6, 12]. Once in banded form, the matrix can be reduced efficiently to tridiagonal form via a technique known as successive band reduction [13]. We review this recent work in Section 7.1.

Our work aims to extend algorithms for the symmetric eigensolver to achieve the same communication costs as 2.5D LU and 2.5D QR. We give an algorithm that lowers the interprocessor bandwidth cost in this manner by a factor of up to $p^{1/6}$ in Section 7.2. The algorithm reduces the band width of the symmetric matrix (we will refer to the width of the band in a banded matrix as "band width" to disambiguate from communication bandwidth cost) to a small band in a single step, which requires minimal interprocessor communication asymptotically. The resulting band width of the banded matrix is small enough for the banded to tridiagonal step to be done locally on one processor or via a parallel successive band reduction technique such as those given by [6, 13]. However, when the locally-owned chunk of the full dense matrix does not fit into cache, the algorithm presented in Section 7.2 incurs a higher memory-bandwidth cost than standard (previously known) approaches.

The 2.5D successive band reduction algorithm in Section 7.3 alleviates the memory-bandwidth overhead of the algorithm in Section 7.2. The 2.5D successive band reduction algorithm lowers the band size of the matrix from n (full) successively by a parameterized factor. To achieve the full 3D $p^{1/6}$ reduction in both the interprocessor communication and memory-bandwidth communication costs with a small cache size, a factor of $\log(n)$ band reduction stages are required by the 2.5D successive band reduction algorithm. However, the algorithm can take fewer steps to achieve a smaller communication cost reduction (which may be optimal due to memory constraints), as well as when the cache size is enough to store a matrix with a large-enough band width. The 2.5D successive band reduction algorithm is in effect a generalization of both the successive band reduction approaches [6, 13] that does less communication as well as a generalization of the 2.5D one-step full-to-banded reduction algorithm (Section 7.2) that is more cache-efficient.

The rest of this section is organized as follows

- Section 7.1 reviews relevant previous work on algorithms for the symmetric eigenvalue problem,
- Section 7.2 gives a 2.5D algorithm for reduction from a fully dense matrix to a banded matrix in optimal interprocessor bandwidth cost,
- Section 7.3 presents a 2.5D successive band reduction algorithm, which reduces a matrix to banded form with optimal interprocessor communication and memory-bandwidth costs, but requires more computation to obtain the eigenvectors.

7.1 Previous Work

Algorithms for blocked computation of the eigenvalue decomposition of a symmetric matrix via a tridiagonal matrix were studied by [53, 54, 91]. These algorithms reduce the symmetric matrix to tridiagonal (or banded) form via a series of n/b blocked Householder transformations of b Householder vectors,

$$A = Q_1 \cdots Q_k D Q_k^T \cdots Q_1^T,$$

where each $Q_i = (I - Y_1 T_i Y_i^T)$. A key property employed by these algorithms is that each two-sided trailing matrix update of blocked Householder transformations may be done as a rank-$2b$ symmetric update on the symmetric matrix $\bar{A}$,

$$\begin{aligned}
Q^T \bar{A} Q =& (I - Y T^T Y^T) \bar{A} (I - Y T Y^T) \\
=& \bar{A} - Y T^T Y^T \bar{A} - \bar{A} Y T Y^T + Y T^T Y^T \bar{A} Y T Y^T \\
=& \bar{A} - Y T^T (Y^T \bar{A}) - (\bar{A} Y) T Y^T + \frac{1}{2} Y T^T (Y^T \bar{A} Y T Y^T) + \frac{1}{2} (Y T^T Y^T \bar{A} Y) T Y^T \\
=& \bar{A} + Y T^T \left(\frac{1}{2} Y^T \bar{A} Y T Y^T - Y^T \bar{A} \right) + \left(\frac{1}{2} Y T^T Y^T \bar{A} Y - \bar{A} Y \right) T Y^T \\
=& \bar{A} + U V^T + V U^T,
\end{aligned}$$

where $U \equiv Y T^T$ and $V \equiv \frac{1}{2} Y T^T Y^T \bar{A} Y - \bar{A} Y = \frac{1}{2} U Y^T W - W$ where $W = AY$. This form of the update is cheaper to compute than the explicit two-sided update and is easy to aggregate by appending additional vectors (to aggregate the Householder form itself requires computing a larger T matrix). Since the trailing matrix update does not have to be applied immediately, but only to the columns which are factorized, this two-sided updated can also be aggregated and used in a left-looking algorithm. For instance, if $\bar{A}$ is an updated version of A meaning $\bar{A} = A + \bar{U}\bar{V}^T + \bar{V}\bar{U}^T$, we could delay the computation of the update to A (not form $\bar{A}$) and instead employ $A + \bar{U}\bar{V}^T + \bar{V}\bar{U}^T$ in explicit form in place of $\bar{A}$ during the eigensolver computation. So, when computing V_i, which requires multiplication of a tall-and-skinny matrix by $\bar{A}$, we could instead multiply by $A + \bar{U}\bar{V}^T + \bar{V}\bar{U}^T$ where $\bar{U}$ and $\bar{V}$ are the aggregated transformations corresponding to the concatenation of all U_j and V_j for $j \in [1, i-1]$. Early approaches used such left-looking aggregation with $b = 1$, reducing the symmetric matrix directly to tridiagonal form [53].

However, there are disadvantages to reducing the symmetric matrix directly to tridiagonal form, since it requires that a vector be multiplied by the trailing matrix for each computation of V_i of which there are $n - 1$. This requires $O(n)$ synchronizations and also considerable vertical data movement (memory-bandwidth cost) if the trailing matrix does not fit into cache as then it needs to be read in for each matrix column (update). These disadvantages motivated approaches where the matrix is not reduced directly to tridiagonal form, but rather to banded form, which allows for $b > 1$ Householder vectors to be computed via QR at each step without needing to touch the trailing matrix from within the QR. After such a reduction to banded form, it is then necessary to reduce the banded matrix to tridiagonal form, which is often cheaper because the trailing matrix is banded and therefore smaller in memory footprint than the trailing matrix in the full-to-banded

reduction step. This multi-stage reduction approach was introduced by Bischof et al. [26, 25] with the aim of achieving BLAS 3 reuse. ELPA [6] implements this approach in the parallel setting with the primary motivation of reducing memory-bandwidth cost. ELPA also employs the parallel banded-to-tridiagonal algorithm introduced by Lang [106]. Lang's algorithms reduces the banded matrix to tridiagonal form directly, but it is also possible to perform more stages of reduction, reducing the matrix to more intermediate bandwidths. Performing more stages of successive band reduction can improve the synchronization cost of the overall approach, from $O(n)$ as needed by Lang's approach, to $O(\sqrt{p})$ as shown by [13].

Our work leverages the ideas of update aggregation, left-looking algorithms, and multi-stage band reduction in order to achieve a 2.5D symmetric eigenvalue computation algorithm with communication costs not asymptotically greater (modulo $O(\log(p))$ factors) than those of 2.5D QR (Section 6.4.4).

7.2 Direct Symmetric-to-Banded Reduction

Algorithm 7.2.1 (2.5D Symmetric Eigensolver) reduces a symmetric n-by-n matrix A directly to band width b using replication of data and aggregation aiming to achieve an interprocessor communication cost of

$$W = O(n^2/\sqrt{cp}) = O(n^2/p^\delta),$$

when the amount of available memory on each processor is

$$M = \Theta(cn^2/p) = O(n^2/p^{2(1-\delta)}).$$

As in Section 6.4, we will employ the parameter $\delta \in [1/2, 2/3]$ rather than $c \in [1, p^{1/3}]$, as it is more convenient in the forthcoming analysis. The algorithm replicates the matrix A and aggregates as well as replicates the updates $U^{(0)}$ and $V^{(0)}$ over $c = p^{2\delta-1}$ groups of $p/c = p^{2(1-\delta)}$ processors each. The algorithm is left looking, and so updates the next matrix panel (line 6) immediately prior to performing the QR of the panel. Each group of $p^{2(1-\delta)}$ processors in Algorithm 7.2.1 should be arranged in a $p^{1-\delta}$-by-$p^{1-\delta}$ processor grid, but we avoid discussing the details of how the multiplications are scheduled until the cost analysis. Throughout Algorithm 7.2.1 we also use the convention that for any $m \geq n \geq 1$, m-by-n matrix X_1 is lower triangular and $(m-n)$-by-k matrix X_2 is lower triangular, we write their concatenation in short form as

$$[X_1, X_2] \equiv \begin{bmatrix} X_1 & 0 \\ \vdots & X_2 \end{bmatrix}.$$

Further given trapezoidal $X = [X_0, X_1]$ where X_0 is m-by-l and dense and X_1 is as before m-by-n and lower triangular, we similarly abbreviate

$$[X, X_2] \equiv \begin{bmatrix} X & 0 \\ \vdots & X_2 \end{bmatrix}.$$

Algorithm 7.2.1 $[U, V, B] \leftarrow$ 2.5D-SE$(A, U^{(0)}, V^{(0)}, \Pi, b)$

Require: Let Π be a set of p processors, define $\hat{\Pi} \subset \Pi$ be a group of p^δ processors. Let A be a n-by-n symmetric matrix. Let $U^{(0)}$ and $V^{(0)}$ be n-by-m matrices that are trapezoidal (zero in top right upper b-by-b triangle). Let A, $U^{(0)}$, and $V^{(0)}$ be replicated over $p^{2\delta-1}$ subsets of $p^{2(1-\delta)}$ processors.

if $n \leq b$ **then**
  Compute $B = A - U^{(0)}V^{(0)^T} + V^{(0)}U^{(0)^T}$
  Return $[\{\}, \{\}, B]$
Subdivide $A = \begin{bmatrix} A_{11} & A_{21}^T \\ A_{21} & A_{22} \end{bmatrix}$ where A_{11} is b-by-b
Subdivide $U^{(0)} = \begin{bmatrix} U_1^{(0)} \\ U_2^{(0)} \end{bmatrix}$ and $V^{(0)} = \begin{bmatrix} V_1^{(0)} \\ V_2^{(0)} \end{bmatrix}$ where $U_1^{(0)}$ and $V_1^{(0)}$ are b-by-m
Compute $\begin{bmatrix} \bar{A}_{11} \\ \bar{A}_{21} \end{bmatrix} = \begin{bmatrix} A_{11} \\ A_{21} \end{bmatrix} + U^{(0)}V_1^{(0)^T} + V^{(0)}U_1^{(0)^T}$
% Compute QR using TSQR algorithm (see Lemma 6.4.2)
$[Y, T, R] \leftarrow$ TSQR$(\bar{A}_{21}, \hat{\Pi})$
Compute $W = (A_{22} + U^{(0)}V^{(0)^T} + V^{(0)}U^{(0)^T})Y$
Compute $U_1 = YT^T$
Compute $V_1 = \frac{1}{2}U_1Y^TW - W$
Replicate U_1 and V_1 over $p^{2\delta-1}$ subsets of $p^{2(1-\delta)}$ processors
% Recursively reduce the trailing matrix to banded form
$[U_2, V_2, B_2] =$ 2.5D-SE$(A_{22}, [U_2^{(0)}, U_1], [V_2^{(0)}, V_1], \Pi, b)$
$B = \begin{bmatrix} \bar{A}_{11} & R^T & 0 \\ R & B_2 & \cdots \\ 0 & \vdots & \ddots \end{bmatrix}, \quad U = [U_1, U_2], \quad V = [V_1, V_2]$

Ensure: B is a banded n-by-n matrix with band width b, U and V are lower-triangular $(n-b)$-by-$(n-b)$ matrices, and $B = A + [U^{(0)}, U] \cdot [V^{(0)}, V]^T + [V^{(0)}, V] \cdot [U^{(0)}, U]^T$

Algorithm 7.2.1 computes the update correctly since

$$V = \frac{1}{2}UY^TW - W = \frac{1}{2}YT^TY^TAY - AY$$

is the aggregated update derived in Section 7.1. The correctness follows in general by the ensured condition on the result of the tail recursion, which performs the update and factorization of the trailing matrix. In the base case, the matrix dimension is less than or equal to the desired matrix band width, which means it suffices to perform the aggregated update and return the result, which would appear in the lower right subblock of the full banded matrix. We state and prove the costs of Algorithm 7.2.1 (for the full symmetric eigenvalue problem, with the band width of B picked in the proof) in Theorem 7.2.1.

Theorem 7.2.1. *On a parallel homogeneous machine with p processors, each with local memory $M > 3n^2/p^{2(1-\delta)}$, for $\delta \in [1/2, 2/3]$, and any cache size $1 \ll \hat{M} \leq M$, Algorithm 7.2.1 can be used to compute the eigenvalues of any n-by-n matrix A with time complexity*

$$T_{\text{2.5D-SE}}(n,p,\delta,\hat{M}) = O\left(\gamma \cdot \frac{n^3}{p} + \beta \cdot \frac{n^2}{p^\delta} + \nu \cdot \left[\frac{n^2}{p^{2-3\delta}}\log(p) + \frac{n^3}{p\sqrt{\hat{M}}}\right] + \alpha \cdot p^\delta \log^2(p)\right).$$

Further, when $\hat{M} > 3n^2/p^{2(1-\delta)}$ (all intermediates fit into cache), the algorithm can be executed in time

$$T_{\text{2.5D-SE}}(n,p,\delta,\hat{M}) = O\left(\gamma \cdot n^3/p + \beta \cdot \frac{n^2}{p^\delta} + \nu \cdot \frac{n^3}{p\sqrt{\hat{M}}} + \alpha \cdot p^\delta \log^2(p)\right).$$

Proof. We consider the costs within each recursive step of Algorithm 7.2.1. We note that the dimensions of A, $U^{(0)}$ and $V^{(0)}$ will always be less than the dimension of the original matrix n and analyze the cost of each recursive step with n being the dimension of the original matrix (rather than the smaller dimension of the trailing matrix).

- The update to the trailing matrix update done on line 6 of Algorithm 7.2.1 requires two multiplications of an $O(n)$-by-$O(n)$ matrix by a $O(n)$-by-b matrix. Since the matrices $U^{(0)}$ and $V^{(0)}$ are replicated over $p^{2\delta-1}$ groups of $p^{2(1-\delta)}$ processors, by application of Lemma 4.4.3 the cost of this step is, if $U^{(0)}$ and $V^{(0)}$ start in cache,

$$O\left(\gamma \cdot \frac{n^2 b}{p} + \beta \cdot \frac{nb}{p^\delta} + \alpha \cdot \log(p)\right),$$

  and in general (for any cache size),

$$O\left(\gamma \cdot \frac{n^2 b}{p} + \beta \cdot \frac{nb}{p^\delta} + \nu \cdot \left[\frac{n^2}{p^{2(1-\delta)}} + \frac{n^2 b}{p\sqrt{\hat{M}}}\right] + \alpha \cdot \log(p)\right).$$

- The cost of a $O(n)$-by-b TSQR done on line 8 using p^δ processors is given by Lemma 6.4.2

$$O\left(\gamma \cdot \left[nb^2/p^\delta + b^3\log(p)\right] + \beta \cdot b^2\log(p) + \nu \cdot \left[\frac{b^3\log(p) + nb^2/p^\delta}{\sqrt{\hat{M}}} + \frac{nb}{p}\right] + \alpha \cdot \log(p)\right).$$

- Computing line 9, $W = (A_{22} + U^{(0)}{V^{(0)}}^T + V^{(0)}{U^{(0)}}^T)Y$, right-to-left requires 5 matrix multiplications, namely

  1. $K_1 = {U^{(0)}}^T Y$ where ${U^{(0)}}^T$ is $O(n)$-by-$O(n)$ and Y is $O(n)$-by-b,
  2. $V^{(0)}K_1$ where $V^{(0)}$ is $O(n)$-by-$O(n)$ and K_1 is $O(n)$-by-b,

3. $K_2 = {V^{(0)}}^T Y$ where ${V^{(0)}}^T$ is $O(n)$-by-$O(n)$ and Y is $O(n)$-by-b,
4. $U^{(0)} K_1$ where $U^{(0)}$ is $O(n)$-by-$O(n)$ and K_1 is $O(n)$-by-b,
5. $K_3 = A_{22} Y$ where A_{22} is $O(n)$-by-$O(n)$ and Y is $O(n)$-by-b,

where the result is then $W = K_1 + K_2 + K_3$. The additions are low order with respect to the multiplications and each matrix multiplication is between an $O(n)$-by-$O(n)$ matrix and an $O(n)$-by-$O(b)$ matrix. These can be done most efficiently by keeping A, $U^{(0)}$, and $V^{(0)}$ in place and replicated (and if possible, in cache) and moving only Y and the intermediate. Since the three matrices A, $U^{(0)}$, and $V^{(0)}$ all start replicated, we can apply Lemma 4.4.3 to obtain that when $\hat{M} > 3n^2/p^{2(1-\delta)}$ (and all three replicated matrices fit into cache), the overall cost of line 9 is

$$O\left(\gamma \cdot \frac{n^2 b}{p} + \beta \cdot \frac{nb}{p^\delta} + \alpha \cdot \log(p)\right),$$

and in the second scenario, when the cache is small ($\hat{M} < 3n^2/p^{2(1-\delta)}$), the matrices A, $U^{(0)}$ and $V^{(0)}$ need to be moved in and out of cache for each matrix multiplication, leading to the cost

$$O\left(\gamma \cdot \frac{n^2 b}{p} + \beta \cdot \frac{nb}{p^\delta} + \nu \cdot \left[\frac{n^2}{p^{2(1-\delta)}} + \frac{n^2 b}{p\sqrt{\hat{M}}}\right] + \alpha \cdot \log(p)\right).$$

- Forming U and V is strictly less expensive than forming W, since these computations involve only $O(n)$-by-b matrices. So the costs of these tasks contribute no additional cost asymptotically.

- Replicating U and V over $p^{2\delta-1}$ subsets of $p^{2(1-\delta)}$ processors costs

$$O\left(\beta \cdot \left[nb/p^{2(1-\delta)} + \alpha \cdot \log(p)\right]\right).$$

The cost over all n/b recursive levels when all replicated matrices fit into cache (when $\hat{M} > 3n^2/p^{2(1-\delta)}$) is

$$O\left(\gamma \cdot \frac{n^3}{p} + \beta \cdot \left[nb\log(p) + \frac{n^2}{p^\delta}\right] + \alpha \cdot (n/b)\log(p)\right).$$

We want to maximize b without raising the communication bandwidth cost in order to lower latency cost. We can pick $b = n/(p^\delta \log(p))$ and obtain the desired costs,

$$O\left(\gamma \cdot \frac{n^3}{p} + \beta \cdot \frac{n^2}{p^\delta} + \alpha \cdot p^\delta \log^2(p)\right).$$

In the second scenario (when $\hat{M} < 3n^2/p^{2(1-\delta}$), the cost of the algorithm has a higher memory-bandwidth factor

$$O\left(\gamma \cdot \frac{n^3}{p} + \beta \cdot \left[nb\log(p) + \frac{n^2}{p^\delta}\right] + \nu \cdot \left[\frac{n^3/b}{p^{2(1-\delta)}} + \frac{n^3}{p\sqrt{\hat{M}}}\right] + \alpha \cdot (n/b)\log(p)\right).$$

Again picking $b = n/(p^\delta \log(p))$ we obtain the cost

$$O\left(\gamma \cdot \frac{n^3}{p} + \beta \cdot \frac{n^2}{p^\delta} + \nu \cdot \left[\frac{n^2}{p^{2-3\delta}}\log(p) + \frac{n^3}{p\sqrt{\hat{M}}}\right] + \alpha \cdot p^\delta \log^2(p)\right).$$

Computing the eigenvalues of the resulting banded matrix can be done with low-order cost by collecting the matrix on one processor and reducing to tridiagonal form (then computing the eigenvalues of the tridiagonal matrix) sequentially. The reduction from band width $b = n/(p^\delta \log(p))$ to tridiagonal can be done with low-order cost, although in practice using a parallel successive band reduction algorithm [13] with p^δ processors is likely preferrable and also attains the desired costs. □

When the matrix does not fit in cache, Algorithm 7.2.1 has a tradeoff between interprocessor communication cost and memory bandwidth cost. For the the first term in the memory bandwidth cost of the second scenario, $\frac{n^2}{p^{2-3\delta}}\log(p)$, to be $O(\nu \cdot n^2/p^\delta)$, we need $\delta = 1/2$, which is only true in the 2D case. So, if the cache is small, as we increase delta, the interprocessor bandwidth cost decreases while the memory-bandwidth cost increases. We note that in the 3D case, when $\delta = 2/3$, the memory-bandwidth cost term $\frac{n^3/b}{p^{2(1-\delta)}}\log(p)$ is $O(\nu \cdot n^2/p^\delta)$ only when the intermediate band width is $b = \Omega(n)$. This motivates the use of of a successive band reduction approach that reduces the band width by a constant factor at each level, which we present in the next section.

7.3 Successive Symmetric Band Reduction

Algorithm 7.3.1 (2.5D Successive Band Reduction) describes the QR factorizations and applications necessary to reduce a symmetric banded matrix A from band width b to band width b_{term} via bulge chasing. The algorithm uses multiple stages of band reduction, reducing the band width from b to b/k at each stage. At each stage, the algorithm eliminates nk/b trapezoidal panels each of which gets chased $O(n/b)$ times down the band. The number of stages is dependent on the parameter k, which we later pick to be a constant. As we conjecture at the end of the proof, it ought to be possible to extend the algorithm to efficiently handle k being proportional to $p^{1-\zeta}$, where $\zeta = (1-\delta)/\delta$, so $\zeta \in [\frac{1}{2}, 1]$, which means that in the 3D case $\zeta = 1$ and k would still have to be a constant, but in the 2D case we could pick $k = \sqrt{p}$. Reducing the number of band reduction stages, which shrinks as k grows with $\log(b)/\log(k)$, is particularly important if we also desire to compute the eigenvectors, since the cost of the back-transformations scales linearly with the number of band-reduction stages (each one costs $O(n^3)$ computational work). We leave this extension and the consideration of eigenvector construction for future work and focus on attaining the desired communication costs for computation of only the eigenvalues of the symmetric matrix. We also restrict $\delta > 1/2$ ($\zeta < 1$) in order to efficiently employ the 2.5D QR algorithm from Section 6.4.4. In the case of $\delta = 1/2$, the regular 2D algorithm used by ELPA and ScaLAPACK attains the desired communication costs and when combined with a synchronization-efficient successive band reduction technique also attains the desired latency cost [13].

Algorithm 7.3.1 $[B] \leftarrow$ 2.5D-SBR$(A, \Pi, k, \zeta, b_{\text{term}})$

Require: Let $k \geq 2$ and $\zeta \in [\frac{1}{2}, 1)$. Let A be a banded symmetric matrix of dimension n with band width $b \leq n$. Assume $n \bmod b \equiv 0$ and $b \bmod k \equiv 0$. Let Π be a set of $\bar{p}$ processors, define $\hat{\Pi}_j \subset \Pi$ be the jth group of $\hat{p} \equiv \bar{p} \cdot (b/n)$ processors for $j \in [1, n/b]$.

Set $B = A$
if $b \leq b_{\text{term}}$ **then** Return B.
Initialize $\hat{U} = \emptyset$ and $\hat{V} = \emptyset$
Let $B[(j-1)b + 1 : jb\ ,\ (j-1)b + 1 : jb]$ be replicated in $\hat{\Pi}_j$ over $(b\bar{p}/n)^{2\delta-1}$ subsets of $(b\bar{p}/n)^{2(1-\delta)}$ processors.
% Iterate over panels of B
for $i = 1 : kn/b$ **do**
    % $\hat{\Pi}_j$ applies jth chase of bulge i as soon as $\hat{\Pi}_{j-1}$ executes the $(j-1)$th chase
    for $j = 1 : \lfloor (n - ib/k - 1)/b \rfloor$ **do**
        % Define offsets for bulge block
        Let $o_{\text{blg}} = (j + i/k)b$
        if $j = 1$ **then** $o_{\text{qr}} = b/k$
        else $o_{\text{qr}} = b$
        % Define index ranges needed for bulge chase
        $n_{\text{row}} = \min(n - o_{\text{blg}}, 2b - b/k)$
        $n_{\text{col}} = \min(n - o_{\text{blg}} + o_{\text{qr}} - b/k, 3b - b/k)$
        $I_{\text{qr.rs}} = \{o_{\text{blg}} + 1 : o_{\text{blg}} + n_{\text{row}}\}$
        $I_{\text{qr.cs}} = \{o_{\text{blg}} - o_{\text{qr}} + 1 : o_{\text{blg}} - o_{\text{qr}} + b/k\}$
        $I_{\text{up.rs}} = \{o_{\text{qr}} - b/k + 1 : o_{\text{qr}} - b/k + n_{\text{row}}\}$
        $I_{\text{up.cs}} = \{o_{\text{blg}} - o_{\text{qr}} + b/k + 1 : o_{\text{blg}} - o_{\text{qr}} + b/k + n_{\text{col}}\}$
        % Perform a rectangular 2.5D QR (Algorithm 6.4.1)
        $[Y, T, R] \leftarrow$ 2.5D-QR$(B[I_{\text{qr.rs}}, I_{\text{qr.cs}}])$
        $B[I_{\text{qr.rs}}, I_{\text{qr.cs}}] = \begin{bmatrix} R \\ 0 \end{bmatrix}$, $B[I_{\text{qr.cs}}, I_{\text{qr.rs}}] = \begin{bmatrix} R \\ 0 \end{bmatrix}^T$
        Compute $U = YT^T$
        Compute $W = -B[I_{\text{up.cs}}, I_{\text{qr.rs}}]Y$, set $V = -W$
        Compute $V[I_{\text{up.rs}}, :] = V[I_{\text{up.rs}}, :] + \frac{1}{2}UY^TW[I_{\text{up.rs}}, :]$
        Compute $B[I_{\text{qr.rs}}, I_{\text{up.cs}}] = B[I_{\text{qr.rs}}, I_{\text{up.cs}}] + UV^T$
        Compute $B[I_{\text{up.cs}}, I_{\text{qr.rs}}] = B[I_{\text{up.cs}}, I_{\text{qr.rs}}] + VU^T$
% Perform tail-recursion call to reduce matrix from band width b/k to b_{term} using a fraction of the processors
Let $\bar{\Pi} \subset \Pi$ be a group of $\bar{p}/k^{\zeta}$ processors
$[B] \leftarrow$ 2.5D-SBR$(B, \bar{\Pi}, k, \zeta, b_{\text{term}})$

Ensure: B is a banded matrix with band width less than b_{term} and the same eigenvalues as A

Theorem 7.3.1. *On a parallel homogeneous machine with p processors, each with local memory $M > 3n^2/p^{2(1-\delta)}$, for $\delta \in (1/2, 2/3]$, and any cache size $1 \ll \hat{M} \leq M$, Algorithm 7.3.1 can be used to compute the eigenvalues of any $n \times n$ symmetric matrix A with time complexity*

$$T_{\text{2.5D-SBR}}(n, p, \delta, \hat{M}) = O\left(\gamma \cdot n^3/p + \beta \cdot \frac{n^2}{p^\delta}\log(p) + \nu \cdot \frac{n^3}{p\sqrt{\hat{M}}} + \alpha \cdot p^\delta \log^2(p)\right).$$

Proof. Since by assumption $M > n^2/p^{2(1-\delta)}$ and each recursive step reduces the number of processors by a factor of k^ζ and the matrix band width by a factor of k, at every recursive level l it will always be the case that $M > b^2/\hat{p}^{2(1-\delta)}$ since

$$\begin{aligned}\frac{b^2}{\hat{p}^{2(1-\delta)}} =& \frac{b^2}{(\bar{p}b/n)^{2(1-\delta)}} = \frac{(n/k^l)^2}{[(p/k^{l\zeta})(n/k^l)/n]^{2(1-\delta)}} \\ =& \frac{n^2}{[k^{l/(1-\delta)}k^{-l\delta/(1-\delta)}k^{-l}p]^{2(1-\delta)}} = \frac{n^2}{p^{2(1-\delta)}}.\end{aligned}$$

The cost of each inner loop iteration (loop on line 8 can be derived from the costs of the matrix multiplications and QR done inside it. Each iteration of the inner loop ((i, j) iteration starting on line 8 in Algorithm 7.3.1) incurs the following costs (in the following analysis we ignore the fact that the matrices may be smaller in the last iteration of the loop over j)

- The multiplication needed to form $U = YT^T$ on line 23 is of a $(2b - b/k)$-by-b/k matrix Y and a b/k-by-b/k matrix T. The cost of this step is strictly less than that of the multiplication on line 24 where Y is multiplied by a $(3b - b/k)$-by-$(2b - b/k)$ matrix.

- The multiplication needed to form $W = B[I_{\text{up.cs}}, I_{\text{qr.rs}}]Y$ on line 24 is of a $(3b - b/k)$-by-$(b - b/k)$ matrix $B[I_{\text{up.cs}}, I_{\text{qr.rs}}]$ and a $(b - b/k)$-by-b/k matrix Y. By Lemma 4.4.2 the cost of this multiplication is

  $$O\left(\gamma \cdot \frac{b^3}{k\hat{p}} + \beta \cdot \left[\frac{b^2}{\hat{p}} + \frac{b^3}{k\hat{p}\sqrt{M}} + \frac{b^2}{k^{2/3}\hat{p}^{2/3}}\right] + \nu \cdot \frac{b^3}{k\hat{p}\sqrt{\hat{M}}} + \alpha \cdot \left[\frac{b^3}{k\hat{p} \cdot M^{3/2}} + \log(p)\right]\right).$$

  Since $M > b^2/\hat{p}^{2(1-\delta)}$, this cost may be upper-bounded as

  $$O\left(\gamma \cdot \frac{b^3}{k\hat{p}} + \beta \cdot \frac{b^2}{\hat{p}^\delta} + \nu \cdot \frac{b^3}{k\hat{p}\sqrt{\hat{M}}} + \alpha \cdot \left[\frac{\hat{p}^{2-3\delta}}{k} + \log(\hat{p})\right]\right).$$

- The multiplications needed to compute $\frac{1}{2}UY^TW[I_{\text{up.rs}}, :]$ (line 25) can be done right to left by first forming $Z = Y^TW[I_{\text{up.rs}}, :]$ then UZ. Forming Z requires a multiplication of a b/k-by-$(b - b/k)$ matrix Y^T and a $(b - b/k)$-by-b/k matrix $W[I_{\text{up.rs}}, :]$ into a b/k-by-b/k matrix Z. The cost of this multiplication is strictly cheaper than the one needed to compute W. The second multiplication needed is $U \cdot Z$ and is again low-order since U is b-by-b/k and Z is b/k-by-b/k.

- The trailing matrix updates on line 26 and on line 27 require two multiplications UV^T and VU^T whose operands are $(3b-b/k)$-by-b/k and b/k-by-$(b-b/k)$ matrices into a $(3b-b/k)$-by-$(b-b/k)$ matrix (or transposes of these). Lemma 4.4.2 again applies here and yields an asymptotically equivalent cost to the computation of W.

- The cost of the 2.5D-QR of the $(b-b/k)$-by-b/k matrix $B[I_{\mathrm{qr.rs}}, I_{\mathrm{qr.cs}}]$ performed on line 21 with $\hat{p}$ processors is by Lemma 6.4.4 for $\delta \in (1/2, 2/3]$,

$$O\left(\gamma \cdot \frac{b^3}{k^2\hat{p}} + \beta \cdot \frac{b^2}{k^{2-\delta}\hat{p}^\delta} + \nu \cdot \frac{b^3}{k^2\hat{p} \cdot \sqrt{\hat{M}}} + \alpha \cdot \hat{p}^\delta \log(\hat{p})\right).$$

The overall cost for each iteration of Algorithm 7.3.1 is the sum of the two different costs above, which with low order terms discarded is

$$O\left(\gamma \cdot \frac{b^3}{k\hat{p}} + \beta \cdot \frac{b^2}{\hat{p}^\delta} + \nu \cdot \frac{b^3}{k\hat{p}\sqrt{\hat{M}}} + \alpha \cdot \hat{p}^\delta \log(\hat{p})\right).$$

For a given outer loop (line 6) iteration i, each j loop iteration (line 8) is done concurrently by a different processor group. Therefore, along the critical path $O(kn/b)$ inner loop iterations are executed. Combining this with the tail recursion of 2.5D-SBR with a matrix B of band width b/k and using $\bar{p}/k^\zeta$ processors (line 30), the total cost of Algorithm 7.3.1 is

$$\begin{aligned} T_{\mathrm{2.5D-SBR}}\ (n, \bar{p}, \delta, \hat{M}, b, k) &= T_{\mathrm{2.5D-SBR}}(n, \bar{p}/k^\zeta, \delta, \hat{M}, b/k, k) \\ &+ O\left(\gamma \cdot \frac{nb^2}{\hat{p}} + \beta \cdot \frac{knb}{\hat{p}^\delta} + \nu \cdot \frac{nb^2}{\hat{p}\sqrt{\hat{M}}} + \alpha \cdot \frac{kn\hat{p}^\delta}{b} \log(\hat{p})\right), \end{aligned}$$

plugging in $\hat{p} = \bar{p}b/n$ we obtain

$$\begin{aligned} T_{\mathrm{2.5D-SBR}}\ (n, \bar{p}, \delta, \hat{M}, b, k) &= T_{\mathrm{2.5D-SBR}}(n, \bar{p}/k^\zeta, \delta, \hat{M}, b/k, k) \\ &+ O\left(\gamma \cdot \frac{n^2 b}{\bar{p}} + \beta \cdot \frac{k^{1-\delta} n^{1+\delta} b^{1-\delta}}{\bar{p}^\delta} + \nu \cdot \frac{n^2 b}{\bar{p}\sqrt{\hat{M}}} + \alpha \cdot \frac{kn^{1-\delta}\bar{p}^\delta}{b^{1-\delta}} \log(\bar{p})\right), \end{aligned}$$

after l levels of recursion starting from a n-by-n matrix with p processors and $b = n/k^l$, each subproblem has the cost

$$\begin{aligned} T_{\mathrm{2.5D-SBR}}(\quad n, p/k^{l\zeta}, \delta, \hat{M}, n/k^l, k) &= T_{\mathrm{2.5D-SBR}}(n, p/k^{(l+1)\zeta}, \delta, \hat{M}, n/k^{l+1}, k) \\ &+ O\left(\gamma \cdot \frac{n^3}{k^{l(1-\zeta)}p} + \beta \cdot \frac{n^2 k^{l(\delta-1)+l\zeta\delta+(1-\delta)}}{p^\delta} + \nu \cdot \frac{n^3}{k^{l(1-\zeta)}p\sqrt{\hat{M}}} + \alpha \cdot \frac{k^{1-\delta}p^\delta}{k^{l(\delta-1)+l\zeta\delta}} \log(p)\right). \end{aligned}$$

Firstly, since $\delta > 1/2$ and so $\zeta = (1-\delta)/\delta < 1$, the floating point cost decreases geometrically at each level, and therefore is dominated by the top level. Now $\zeta = (1-\delta)/\delta$ implies that

$\delta(1+\zeta) = 1$ and further that $k^{l(\delta-1)+l\zeta\delta} = 1$. Therefore, the bandwidth and latency cost at every one of $O(\log(p))$ recursive levels stay the same and pick up a $\log(p)$ factor due to the number of recursive levels,

$$T_{\text{2.5D-SBR}}(n,p,\delta,\hat{M},n,k) = O\left(\gamma\cdot\frac{n^3}{p} + \beta\cdot\frac{k^{1-\delta}n^2}{p^\delta}\log(p) + \nu\cdot\frac{n^3}{p\sqrt{\hat{M}}} + \alpha\cdot k^{1-\delta}p^\delta\log^2(p)\right).$$

We can now attain the desired costs by picking k to be a constant (e.g. 2),

$$T_{\text{2.5D-SBR}}(n,p,\delta,\hat{M},n,2) = O\left(\gamma\cdot\frac{n^3}{p} + \beta\cdot\frac{n^2}{p^\delta}\log(p) + \nu\cdot\frac{n^3}{p\sqrt{\hat{M}}} + \alpha\cdot p^\delta\log^2(p)\right).$$

The cost given in the statement theorem has two fewer parameters,

$$T_{\text{2.5D-SBR}}(n,p,\delta,\hat{M}) = T_{\text{2.5D-SBR}}(n,p,\delta,\hat{M},n,2).$$

□

We conjecture that it is possible to extend the algorithm in a way that achieves good asymptotic communication costs when k is not a constant when $\delta < 2/3$. One approach to achieving this would be to integrate the left-looking aggregation idea used in Algorithm 7.2.1 into Algorithm 7.3.1. The updates to each bulge should be aggregated up to the band width b of the matrix A as $U^{(0)}$ and $V^{(0)}$ in Algorithm 7.2.1 and applied only thereafter (or once a panel must be factorized). This optimization would reduce the communication costs associated with the inner loop multiplications that involve b-by-b matrices, as these matrices could be kept replicated between different inner loop iterations. When $\delta = 1/2$, it should then be possible to pick $k = \sqrt{p}$ and reduce Algorithm 7.3.1 into Algorithm 7.2.1 with $b = n/\sqrt{p}$. However, the analysis of this algorithm would be somewhat more complicated since the bulge chased by processor group j on the $i+1$th iteration of the outer loop is shifted by b/k from the bulge on the ith iteration, so a portion of the replicated matrix within each processor group would need to be shifted between processor groups.

Chapter 8

Sparse Iterative Methods

We now consider sparse numerical linear algebra computations, which are typically dominated by repeated sparse matrix-vector multiplications. Krylov subspace methods are a large class of iterative methods which, given a sparse matrix A and a vector $x^{(0)}$, employ such repeated sparse matrix-vector products to construct a an s-step Krylov (subspace) basis, $\{x^{(0)}, Ax^{(0)}, \ldots, A^s x^{(0)}\}$. This construction is interleaved with operations which use the Krylov basis to iteratively solve eigenvalue-type problems or systems of equations. In this chapter, we do not consider more general bases such as polynomial in A given by three-term recurrences While these methods achieve good asymptotic scalability with respect to problem size by preserving sparsity, the amount of data reuse which these algorithms can exploit is limited (especially when only a single vector is being multiplied repeatedly) with respect to the dense linear algebra algorithms. One technique that remedies this lack of data-reuse is performing blocking across sparse matrix-vector products rather than doing each one in sequence [90, 109, 49, 154]. While this technique reduces synchronization and memory-bandwidth cost [116], it usually requires extra interprocessor communication costs [49]. In this chapter, we introduce the dependency graphs of Krylov basis computations and apply our lower bound techniques to derive tradeoffs between synchronization cost and the costs of computation and interprocessor communication.

In particular, we employ the path-expansion-based tradeoff lower bounds derived in Chapter 3 to demonstrate that an s-step Krylov basis computation with a $(2m+1)^d$ point stencil must incur the following computation F, communication W, and synchronization S costs:

$$F_{\mathrm{Kr}} \cdot S_{\mathrm{Kr}}^d = \Omega\left(m^{2d} \cdot s^{d+1}\right), \quad W_{\mathrm{Kr}} \cdot S_{\mathrm{Kr}}^{d-1} = \Omega\left(m^d \cdot s^d\right).$$

These lower bounds are not dependent on the number of processors or the global problem size, but only on the path expansion properties of the dependency graph. They signify that in order to reduce synchronization cost below s, a certain amount of work and communication needs to be done along the critical path. The communication tradeoff occurs only for meshes of dimension two or higher ($d \geq 2$). We rigorously derive these bounds in Section 8.2.

This chapter is based on joint work with Erin Carson and Nicholas Knight [145].

These lower bounds are attained in the parallel (strong scaling) limit by known algorithms, which we recall in Section 8.3. These algorithms can reduce synchronization cost, but at the overhead of extra interprocessor communication cost. An additional advantage of these known algorithms in the case when only $A^s x^{(0)}$ is needed rather than the entire Krylov basis (as is often the case in multigrid smoothers and preconditioners) is that they yield an improvement in memory-bandwidth efficiency when the mesh entries assigned to each processor does not fit into cache. We demonstrate in Section 8.4 that memory-bandwidth efficiency may be improved in an alternative manner that does not reduce synchronization cost, and does not incur an associated overhead in interprocessor communication cost. However, the new algorithm incurs extra interprocessor communication cost when the cache size is small. The algorithm has promising potential for the scenario when the portion of the data owned by each processor is too large to fit completely in cache, but only by some constant factor. In this case, the new algorithm can achieve asymptotically better cache reuse at the cost of a constant factor overhead in interprocessor communication cost. This scenario is common for current hybrid supercomputer architectures, which use accelerators with a smaller local memory than that of the host CPU processor. If the mesh (vector x) does not fit into the local memory of the accelerator, it may be sensible to use the technique in Section 8.4 to lower the amount of data transfered between the accelerator and the CPU (i.e. over the PCI connection).

The rest of this chapter is organized as follows

- Section 8.1 formally defines the Krylov subspace computations we consider and constructs their dependency graphs,
- Section 8.2 derives lower bounds on tradeoffs between computation, communication, and synchronization cost in Krylov subspace methods
- Section 8.3 reviews known algorithms which avoid synchronization and attain the stated lower bounds,
- Section 8.4 presents a new algorithm that aims to achieve low interprocessor communication efficiency as well as low memory-bandwidth cost.

8.1 Definition and Dependency Graphs of Krylov Basis Computations

We consider the s-step Krylov basis computation

$$x^{(l)} = A \cdot x^{(l-1)},$$

for $l \in [1, s]$, $x^{(0)}$ given as input, and A is a sparse matrix corresponding to a $(2m+1)^d$-point stencil (with $m \geq 1$), i.e., a d-dimensional n-by-$\cdots$-by-n mesh T, where each entry in A represents an interaction between vertices $v_{i_1,\ldots,i_d}, w_{j_1,\ldots,j_d} \in T$, such that for $k \in [1, d]$, $i_k, j_k \in [1, n]$, $|j_k - i_k| \leq$

m. Thus, matrix A and vectors $x^{(l)}$, $l \in [0, s]$, have dimension n^d, and A has $\Theta(m^d)$ nonzeros per row/column. Our interprocessor communication and synchronization bounds will apply both when the entries of A are implicit and when they are explicit, but the memory bandwidth analysis for the algorithms will assume that A is implicit. We also assume that $s \leq n/m$ (the lower bound could also be applied to subsets of the computation with $s' = \lfloor n/m \rfloor$). We note that the dependency structure of this computation is analogous to direct force evaluations in particle simulations and the Bellman-Ford shortest-paths algorithm, which may be expressed as sparse matrix-vector multiplication, but on a different algebraic semiring.

We let $G_{\text{Kr}} = (V_{\text{Kr}}, E_{\text{Kr}})$ be the dependency graph of the s-step Krylov basis computation defined above. We index the vertices $V_{\text{Kr}} \ni v_{i_1,\ldots,i_d,l}$ with $d+1$ coordinates, each corresponding to the computation of an intermediate vector element $x_k^{(l)}$ for $l \in [1, s]$ and $k = \sum_{j=1}^{d}(i_j - 1)n^{j-1} + 1$, supposing the lexicographical ordering of $[1, n]^d$. For each edge $(v_{i_1,\ldots,i_d}, w_{j_1,\ldots,j_d})$ in T and each $l \in [1, s]$, there is an edge $(v_{i_1,\ldots,i_d,l-1}, w_{j_1,\ldots,j_d,l})$ in E_{Kr}. By representing the computation of each $x_k^{(l)}$ as a single vertex, we have precluded the parallelization of the m^d scalar multiplications needed to compute each such vertex (instead we simply have edges corresponding to these dependencies). Accordingly, the computation costs we give subsequently will be scaled by m^d for each vertex computed.

8.2 Communication Lower Bounds for Krylov Basis Computation

We now extract a subset of G_{Kr}, and lower bound the execution cost of a load-balanced computation of this subset. Consider the following dependency path $\mathcal{P}$,

$$\mathcal{P} = \{v_{1,\ldots,1,1}, \ldots, v_{1,\ldots,1,s}\}.$$

The bubble $\zeta(G_{\text{Kr}}, \mathcal{P}) = \hat{G}_{\text{Kr}} = (\hat{V}_{\text{Kr}}, \hat{E}_{\text{Kr}})$ includes

$$\begin{aligned}\hat{V}_{\text{Kr}} = \big\{ & v_{i_1,\ldots,i_d,i_{d+1}} : 1 \leq i_{d+1} \leq s, \\ & \text{and } \forall j \in [1, d],\ i_j \leq \max(1, m \cdot \min(i_{d+1}, s - i_{d+1}))\big\}.\end{aligned}$$

as well as all edges between these vertices $\hat{E}_{\text{Kr}} = (\hat{V}_{\text{Kr}} \times \hat{V}_{\text{Kr}}) \cap E_{\text{Kr}}$. In the following theorem we lower bound the communication and synchronization costs required to compute this subset of the computation. The lower bound is independent of n (independent of the global problem size), since we are only considering a subset. This lower bound is effectively a limit on the available parallelism.

Theorem 8.2.1. *Any execution of an s-step Krylov basis computation for a $(2m+1)^d$-point stencil for $m \geq 1$ on a d-dimensional mesh with $d \ll s$, where some processor computes at most $\frac{1}{x}$ of $\hat{V}_{\text{Kr}}$ and at least $\frac{1}{q}$ of $\hat{V}_{\text{Kr}}$, for some $4 \leq x \leq q \ll s$, requires the following computational, bandwidth, and latency costs for some $b \in [1, s]$,*

$$F_{\text{Kr}} = \Omega\left(m^{2d} \cdot b^d \cdot s\right), W_{\text{Kr}} = \Omega\left(m^d \cdot b^{d-1} \cdot s\right), S_{\text{Kr}} = \Omega\left(s/b\right).$$

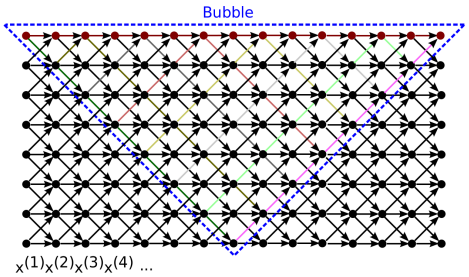


Figure 8.1: Depiction of the bubble (blue parallelogram) along a dependency path (top red path) within a Krylov basis computation on a 1-dimensional mesh. Each vertex corresponds to a block as defined in the proof of Theorem 8.2.1. Edges within the bubble are colored according to the hypergraph edge to which they correspond in the constructed hypergraph, H'.

and furthermore,

$$F_{\mathrm{Kr}} \cdot S_{\mathrm{Kr}}^{d} = \Omega\left(m^{2d} \cdot s^{d+1}\right), \quad W_{\mathrm{Kr}} \cdot S_{\mathrm{Kr}}^{d-1} = \Omega\left(m^{d} \cdot s^{d}\right).$$

Proof. In the following analysis, we will discard factors of d, as we assume $d \ll s$ (e.g., d is a constant), which is reasonable for most problems of interest ($d \in [2, 4]$), but some assumptions in this analysis may need to be revisited if lower bounds for Krylov computations on high-dimensional meshes are desired.

Consider the dependency path $\mathcal{P}$, which defines $\hat{G}_{\mathrm{Kr}}$, and any subpath $\mathcal{R}$ of $\mathcal{P}$, where $|\mathcal{R}| = r \geq 3$,

$$\begin{aligned}\mathcal{P} &= \{v_{1,\ldots,1,1}, \ldots, v_{1,\ldots,1,s}\} \\ \mathcal{P} \supset \mathcal{R} &= \{v_{1,\ldots,1,h+1}, \ldots, v_{1,\ldots,1,h+r}\}.\end{aligned}$$

The bubble $\zeta(\hat{G}_{\mathrm{Kr}}, \mathcal{R}) = (V_\zeta, E_\zeta)$ includes

$$\begin{aligned}V_\zeta = \big\{&v_{i_1,\ldots,i_d,i_{d+1}} : h+1 \leq i_{d+1} \leq h+r, \\ &\text{and } \forall j \in [1,d],\ i_j \leq \max(1, m \cdot \min(i_{d+1} - h - 1, h + r - i_{d+1}))\big\}.\end{aligned}$$

For each $(u_1, \ldots, u_{d+1}) \in [0, r-3]^{d+1}$ (starting at 0 to simplify later transformations, and ending at $r-3$ as we exclude the two end-points of the path, since they are single vertices rather than blocks), we define the following block of the vertices inside G_{Kr}

$$\begin{aligned}B_{u_1,\ldots,u_{d+1}} = \big\{&v_{i_1,\ldots,i_{d+1}} \in V_{\mathrm{Kr}} : \\ &\forall j \in [1,d],\ i_j \in \{\lceil m/2 \rceil u_j + 1, \ldots, \lceil m/2 \rceil (u_j + 1)\} \\ &i_{d+1} = u_{d+1} + h + 2\big\}.\end{aligned}$$

Thus, each block should contain $\lceil m/2 \rceil^d$ vertices on the same level, u_{d+1}. We note that because the breadth of each block is $\lceil m/2 \rceil$ and the interaction distance (stencil radius) is m, every vertex in $B_{u_1,\ldots,u_{d+1}}$ depends on every vertex in $B_{y_1,\ldots,y_d,u_{d+1}-1}$ such that $\forall i \in [1,d]$, $|y_i - u_i| \leq 1$.

We now construct a DAG $G'_{\rm Kr} = (V'_{\rm Kr}, E'_{\rm Kr})$ corresponding to the connectivity of (a subset of) the blocks within the given bubble $\zeta(G_{\rm Kr}, \mathcal{R})$, enumerating them on a lattice graph of dimension $d+1$ and breadth $g = \lfloor (r-2)/(d+1) \rfloor$. For each $(\nu_1, \ldots, \nu_{d+1}) \in [0, g-1]^{d+1}$ with $\nu_1 \leq \nu_2 \leq \ldots \nu_{d+1}$, we have a vertex in the lattice $w_{\nu_1,\ldots,\nu_{d+1}} \in V'_{\rm Kr}$, which corresponds to block $B_{u_1,\ldots,u_{d+1}} \subset V_{\rm Kr}$, where for $i \in [1,d]$, $u_i = \nu_{i+1} - \nu_i$ and $u_{d+1} = \sum_{j=1}^{d+1} \nu_j$. We first show that this mapping of lattice vertices to blocks is unique, by considering any pair of two different vertices in the lattice: $w_{\nu_1,\ldots,\nu_{d+1}}, w_{\nu'_1,\ldots,\nu'_{d+1}} \in V'_{\rm Kr}$. If they map to the same block $B_{u_1,\ldots,u_{d+1}}$, they must have the same u_{d+1}, which implies that $\sum_{j=1}^{d+1} \nu_j = \sum_{j=1}^{d+1} \nu'_j$. Consider the highest index j in which they differ, $\max\{j : \nu_j - \nu'_j \neq 0\}$, and assume without loss of generality that $\nu_j > \nu'_j$. Now if $j = 1$, this is the only differing index, so the sum of the indices cannot be the same. Otherwise, since $u_{j-1} = \nu_j - \nu_{j-1} = \nu'_j - \nu'_{j-1}$, we must also have $\nu_{j-1} > \nu'_{j-1}$, which further implies that for any $k \leq j$, $\nu_k > \nu'_k$, which implies that $\sum_{j=1}^{d+1} \nu_j > \sum_{j=1}^{d+1} \nu'_j$, and is a contradiction. Therefore, the mapping of lattice vertices to blocks is unique.

We now argue that the blocks corresponding to the lattice vertices of $G'_{\rm Kr}$ have all their vertices inside the bubble $\zeta(G_{\rm Kr}, \mathcal{R})$. The first lattice vertex $w_{0,\ldots,0} = B_{0,\ldots,0}$, since for $i \in [1,d]$, $u_i = \nu_{i+1} - \nu_i = 0$ and $u_{d+1} = \sum_{j=1}^{d+1} \nu_j = 0$. So the first lattice vertex (block $B_{0,\ldots,0}$) contains the following vertices of $V_{\rm Kr}$,

$$B_{0,\ldots,0} = \left\{ v_{i_1,\ldots,i_{d+1}} \in V_{\rm Kr} : \forall j \in [1,d],\ i_j \in \{1, \ldots, \lceil m/2 \rceil\},\ i_{d+1} = h+2 \right\},$$

all of these vertices are dependent on the first vertex in the path, $v_{1,1,\ldots,1,h+1}$.

Further, the last lattice vertex $w_{g,\ldots,g}$ corresponds to $B_{0,\ldots,0,u_{d+1}}$ with $u_{d+1} \in [r-2-d, r-2]$, since for $i \in [1,d]$, $u_i = \nu_{i+1} - \nu_i = 0$ and

$$u_{d+1} = \sum_{j=1}^{d+1} \nu_j = (d+1)g = (d+1)\lfloor (r-2)/(d+1) \rfloor \in [r-2-d, r-2].$$

The set of vertices of $V_{\rm Kr}$ in this last in this last block are

$$B_{0,\ldots,0,u_{d+1}} = \left\{ v_{i_1,\ldots,i_{d+1}} \in V_{\rm Kr} : \forall j \in [1,d],\ i_j \in \{1, \ldots, \lceil m/2 \rceil\},\ i_{d+1} = u_{d+1} + h + 2 \right\}.$$

Since $u_{d+1} + h + 1 \leq h + r - 1$, all of these vertices appear at least one level (value of index u_{d+1}) before the last vertex on the path $\mathcal{R}$, which is $v_{1,\ldots,1,h+r}$. Further, since this last block contains values at most $m/2$ away from the origin of the mesh ($\{i_1, \ldots, i_d\} = \{1, \ldots, 1\}$), the last vertex on the path $\mathcal{R}$, $v_{1,\ldots,1,h+r}$ depends on all vertices in the last lattice vertex (block).

Each vertex $w_{\nu_1,\ldots,\nu_{d+1}}$ is connected to vertices $w_{\nu_1,\ldots,\nu_j+1,\ldots,\nu_{d+1}}$ for $j \in [1, d+1]$ (so long as either $j = d+1$ and $\nu_{d+1} \leq g$ or $j \leq d$ and $\nu_{j+1} > \nu_j$) by edges in $E'_{\rm Kr}$, since if $w_{\nu_1,\ldots,\nu_{d+1}}$ corresponds to $B_{u_1,\ldots,u_{d+1}}$ then $w_{\nu_1,\ldots,\nu_j+1,\ldots,\nu_{d+1}}$ corresponds to $B_{u_1,\ldots,u_{j-1}+1,u_j-1,\ldots,u_d,u_{d+1}+1}$, which is dependent on the former block. Therefore, for any lattice vertex $w_{\nu_1,\ldots,\nu_{d+1}}$, there is a dependency path in the lattice from the first lattice vertex $w_{0,\ldots,0}$ to $w_{\nu_1,\ldots,\nu_{d+1}}$ and a dependency path from $w_{\nu_1,\ldots,\nu_{d+1}}$ to the last lattice vertex $w_{g,\ldots,g}$. Since the first lattice vertex is dependent on the origin of the path $\mathcal{R}$ and the last lattice vertex is a dependency of the last vertex on this path, this implies

that all lattice vertices (blocks) are part of the bubble $\zeta(G_{\mathrm{Kr}}, \mathcal{R})$. We give a depiction of the lattice of blocks in the bubble with $d = 1$ in Figure 8.1.

We transform G'_{Kr} into a $(d+1, d)$-lattice hypergraph $H' = (V_H, E_H)$ of breadth g, so that for $4 \leq x \leq q \ll s$, a $\frac{1}{q}$-$\frac{1}{x}$-balanced vertex separator of G'_{Kr} is proportional to a $\frac{1}{q}$-$\frac{1}{x}$-balanced hyperedge cut of H'. We define $V_H = \{w_{i_1,\ldots,i_{d+1}} \in V'_{\mathrm{Kr}} : i_1 < i_2 < \cdots < i_{d+1}\}$ (note that V'_{Kr} included diagonals, but V_H does not), and the hyperedges E_H correspond to a unique member of a disjoint partition of the edges in G'_{Kr}. In particular, we define sets of hyperedges $e_{i_1,\ldots,i_d} \in E_H$ for $i_1, \ldots, i_d \in [1, g]$ with $i_1 < \cdots < i_d$, to contain all $w_{j_1,\ldots,j_{d+1}}$ which satisfy $j_1 < \cdots < j_{d+1}$ and $\{i_1, \ldots, i_d\} \subset \{j_1, \ldots, j_{d+1}\}$. Each of these hyperedges, $e_{i_1,\ldots,i_d}$ corresponds to vertices along an edge-disjoint path within G'_{Kr},

$$\begin{aligned} y_{i_1,\ldots,i_d} \subset & \bigcup_{k=1}^{i_1-1} (w_{k,i_1,\ldots,i_d}, w_{k+1,i_1,\ldots,i_d}) \\ & \cup \bigcup_{k=i_1}^{i_2-1} (w_{i_1,k,i_2\ldots,i_d}, w_{i_1,k+1,i_2,\ldots,i_d}) \cup \cdots \\ & \cup \bigcup_{k=i_d}^{g-1} (w_{i_1,\ldots,i_d,k}, w_{i_1,\ldots,i_d,k+1}), \end{aligned}$$

where each pair of vertices in the union corresponds to a unique edge in G'_{Kr}. Because these hyperedges correspond to disjoint subsets of E'_{Kr}, any $\frac{1}{q}$-$\frac{1}{x}$-balanced vertex separator of G'_{Kr} can be transformed into a $\frac{1}{q}$-$\frac{1}{x}$-balanced hyperedge cut of size no greater than $d+1$ times than the size of some $\frac{1}{q}$-$\frac{1}{x}$-balanced vertex separator by application of Theorem 3.3.2, since the degree of any vertex in H' is at most $d+1$ and since H' is a parent hypergraph of G'_{Kr}.

Since the hypergraph H' is a $(d+1, d)$-lattice hypergraph of breadth $g = \lfloor (|\mathcal{R}| - 2)/(d+1) \rfloor$, by Theorem 3.3.1, its $\frac{1}{q}$-$\frac{1}{x}$-balanced hyperedge cut for $4 \leq x \leq q \ll s$ has size $\epsilon_q(H') = \Omega(g^d/q^{d/(d+1)})$. Furthermore, since the $\frac{1}{q}$-$\frac{1}{x}$-balanced vertex separator of $\zeta(G'_{\mathrm{Kr}}, \mathcal{R})$ is at least $\frac{1}{d+1}\epsilon_q(H')$,

$$\chi_{q,x}(\zeta(G'_{\mathrm{Kr}}, \mathcal{R})) = \Omega\left(\frac{g^d}{(d+1)q^{d/(d+1)}}\right) = \Omega(|\mathcal{R}|^d),$$

where the last bound follows since we have $d, x, q \ll s$.

This lower bound on edge cut size in the block graph G'_{Kr} allows us to obtain a lower bound on the size of any $\frac{1}{q}$-$\frac{1}{x}$-balanced vertex separator of $\hat{G}_{\mathrm{Kr}}$ that is larger by a factor of $\Omega(m^d)$. Consider any $\frac{1}{q}$-$\frac{1}{x}$-balanced vertex separator of $\hat{G}_{\mathrm{Kr}}$ that separates the vertices into three disjoint subsets, the separator Q and the parts V_1 and V_2. If two vertices $u, v \in \hat{V}_{\mathrm{Kr}}$ are in two different parts (are of different color), $u \in V_1$ and $v \in V_2$, and are in the same block, and all vertices in the d adjacent blocks must have all their vertices entirely in Q, since all vertices in adjacent blocks in G'_{Kr} are adjacent to u and v in $\hat{G}_{\mathrm{Kr}}$. Therefore, the number of blocks which contain vertices of different colors is less than $|Q|/m^d$ and therefore small with respect to $|V'_{\mathrm{Kr}}|/q$. Therefore, Q should yield

$\Omega(|V'_{\mathrm{Kr}}|/q)$ blocks which contain vertices that are either in the separator or in V_1, and similarly for V_2. Now, since two blocks $B_1 \subset (V_1 \cup Q)$ and $B_2 \subset (V_2 \cup Q)$ which contain non-separator vertices of different color may not be adjacent, there must exist a separator block $B_3 \subset Q$ for any path on the lattice between B_1 and B_2. Therefore, Q also induces a separator of G'_{Kr} of size no larger than $|Q| \cdot \lceil m/2 \rceil^d$. So, we obtain the following lower bound on the size of a $\frac{1}{q}$-$\frac{1}{x}$-balanced vertex separator of $\hat{G}_{\mathrm{Kr}}$: $\chi_{q,x}(\zeta(\hat{G}_{\mathrm{Kr}}, \mathcal{R})) = \Omega(m^d \cdot \chi_{q,x}(\zeta(G'_{\mathrm{Kr}}, \mathcal{R}))) = \Omega(m^d |\mathcal{R}|^d)$.

Now, the bubble size is $|V_\zeta| = \Omega(m^d |\mathcal{R}|^{d+1})$ and the total length of our main dependency path in $\hat{G}_{\mathrm{Kr}}$ is $|\mathcal{P}| = s$. By Definition 4.1, $\hat{G}_{\mathrm{Kr}}$ is a (ϵ, σ)-path-expander with $\epsilon(b) = m^d b^d$ and $\sigma(b) = m^d b^{d+1}$. Therefore, by application of Theorem 3.4.1 with $k = 3$, we obtain the following lower bounds (scaling by m^d the computation cost for each vertex), for some integer $b \in [3, s]$,

$$F_{\mathrm{Kr}} = \Omega\left(m^{2d} \cdot b^d \cdot s\right), W_{\mathrm{Kr}} = \Omega\left(m^d \cdot b^{d-1} \cdot s\right), S_{\mathrm{Kr}} = \Omega\left(s/b\right).$$

□

8.3 Previous Work on Krylov Basis Algorithms

Having established lower bounds for Krylov basis computations (defined in Section 8.1) in Section 8.2, we now check their attainability. We first recall the costs of a known technique which attains these bounds. In the case that the matrix A is implicitly represented, we propose an alternate idea which improves memory-bandwidth efficiency while keeping interprocessor-bandwidth communication cost low, albeit at the price of increasing network latency cost. We limit our analysis to the case when only the last vector, $x^{(s)}$ is needed, although potentially the algorithms discussed could be extended to the case where a Gram matrix, $[x^{(0)}, \ldots, x^{(s)}]^T \cdot [x^{(0)}, \ldots, x^{(s)}]$ is needed via recomputation of the Krylov vectors [33, 116].

A parallel communication-avoiding algorithm for computing a Krylov basis, termed 'PA1', is presented in [49]. The idea of doing this type of blocking (or tiling) for stencil methods is much older; see, e.g., [90, 109]. Computing an s-step Krylov basis with a $(2m+1)^d$-point stencil with block size $b \in [1, s]$ can be accomplished by $\lceil s/b \rceil$ invocations of PA1 with basis size parameter b. For each invocation of PA1, every processor imports a volume of the mesh of size $(n/p^{1/d} + bm)^d$ and computes b sparse-matrix vector-multiplications on these mesh points without further communication, updating a smaller set at each level so as to keep all the dependencies local. Over s/b invocations of PA1, the computation cost, which includes m^d updates to the locally stored and imported vertices, is given by

$$F_{\mathrm{PA1}} = O\left(s \cdot m^d \cdot (n/p^{1/d} + b \cdot m)^d\right).$$

The communication cost of each PA1 invocation is proportional to the import volume, $(n/p^{1/d} + bm)^d$, minus the part of the mesh that is initially owned locally, which is of size n^d/p, so

$$W_{\mathrm{PA1}} = O\left((s/b) \cdot \left((n/p^{1/d} + b \cdot m)^d - n^d/p\right)\right).$$

Since at every invocation only one round of communication is done, the synchronization cost is

$$S_{\text{PA1}} = O(s/b),$$

so long as $n/p^{1/d} \geq bm$, which means the ghost (import) zones do not extend past the mesh points owned by the nearest-neighbor processors in the processor grid (although the algorithm can be extended to import data from processors further away with a proportionally higher synchronization cost). When the problem assigned to each processor is large enough, i.e., $n/p^{1/d} \gg bm$, the costs for PA1 are

$$F_{\text{PA1}} = O\left(m^d \cdot s \cdot n^d/p\right), \quad W_{\text{PA1}} = O\left(m \cdot s \cdot n^{d-1}/p^{(d-1)/d}\right), \quad S_{\text{PA1}} = O\left(s/b\right).$$

However, in the strong scaling limit, $n/p^{1/d} = \Theta(bm)$, the costs become dominated by

$$F_{\text{PA1}} = O\left(m^{2d} \cdot b^d \cdot s\right), \quad W_{\text{PA1}} = O\left(m^d \cdot b^{d-1} \cdot s\right), \quad S_{\text{PA1}} = O\left(s/b\right),$$

limiting the scalability of the algorithm. These costs of PA1 asymptotically match the lower bounds and lower bound tradeoffs we proved in Theorem 8.2.1, demonstrating that the algorithm is optimal in this scaling limit. We also remark that, strictly speaking, Theorem 8.2.1 does not apply to PA1 because PA1 performs redundant computation, which changes the dependency graph (albeit it is only a constant factor more work when $n/p^{1/d} = \Omega(bm)$). However, we claim that PA1 can be extended to avoid redundant computation altogether, avoiding the overlap between blocks using a similar strategy to [154], with at most constant factor increases in bandwidth and latency, thus showing that the lower bounds and tradeoffs can be attained without recomputation.

In addition to (interprocessor) latency savings, another advantage of executing PA1 with $b > 1$ is improved memory-bandwidth efficiency. For instance, assuming the matrix A is implicit and so only the vector entries (mesh points rather than edges) need to be stored in cache, when $b = 1$ (this standard case is referred to as 'PA0' in [49]), and when the local mesh chunk size $n^d/p > \hat{M}$ (as well as $n^{1/d}/p \gg bm$),

$$\hat{W}_{\text{PA0}} = O(s \cdot n^d/p)$$

data is moved between main memory and cache throughout execution. This memory bandwidth cost corresponds to $O(m^d)$ cache reuse per local mesh point and can be much greater than the interprocessor communication cost when m is small relative to $n/p^{1/d}$. However, when PA1 is executed with $b > 1$, an extra factor of b in cache reuse may be achieved, by computing up to bm^d updates to each mesh point loaded into cache, yielding a memory-bandwidth cost of

$$\hat{W}_{\text{PA1}} = O\left(\frac{s \cdot n^d}{b \cdot p}\right),$$

for $b \leq \hat{M}^{1/d}$. This may be accomplished via a second level of blocking, similar to PA1, except with blocks sized to fit in cache, to minimize the number of transfers between cache and memory. This two-level approach has been studied in [116] in a shared-memory context; see also [154] for a related blocking approach. Analysis in [49] for the cases $d \in [1, 3]$ show that cache-blocking can

reduce memory bandwidth by $O(\hat{M}^{1/d})$, attaining the lower bounds in [90], when n is sufficiently large.

Noting that the memory-bandwidth cost of this computation can exceed the interprocessor-bandwidth cost, we see that PA1 with larger b has lower synchronization cost $S_{\text{PA1}} = O(s/b)$, but higher interprocessor-bandwidth cost $W_{\text{PA1}} = O(msn^{d-1}/p^{(d-1)/d})$ (when $d \geq 2$), and also features a lower memory-bandwidth cost $\hat{W}_{\text{PA1}} = O(sn^d/(pb))$. It is then natural to ask whether there is a tradeoff (in a lower bound sense) between interprocessor and memory-bandwidth costs in the computation, or if a different schedule exists, which achieves a low memory-bandwidth cost without incurring the extra interprocessor-bandwidth cost from which PA1 suffers (when $d \geq 2$). If such a schedule with low or optimal interprocessor communication cost and memory bandwidth cost existed, our lower bounds necessitate that it would have $\Omega(s)$ synchronizations. Indeed, such a schedule exists and we introduce it in the following section.

8.4 A Communication-Efficient Schedule for Krylov Basis Computation

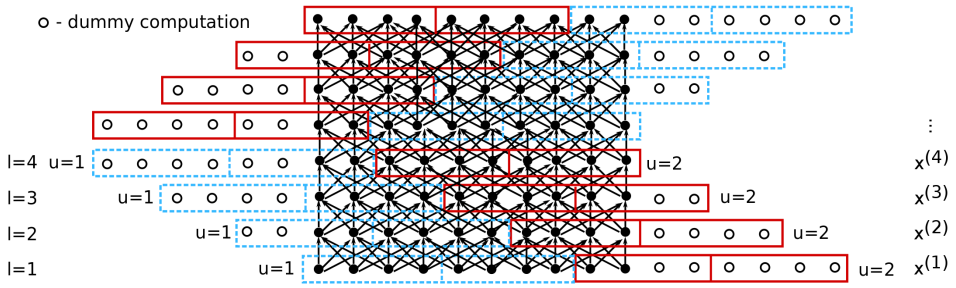


Figure 8.2: Depiction of Algorithm 8.4.1 with $b = 4$, $r = 4$, and $p = 2$ to perform a Krylov basis computation with $n = 10$, $m = 2$, $d = 1$, and $s = 8$. All the blue (dashed outline) blocks for each of two levels (the first is labeled for $l = 1$ to $l = 4$) are computed prior to all the red (solid outline) blocks, with the two subblocks composing each blue and red block being computed in parallel by the two processors.

Algorithm 8.4.1 is a schedule for $G_{\text{Kr}} = (V_{\text{Kr}}, E_{\text{Kr}})$, which computes the vertices V_{Kr} for the computation of $x^{(s)}$ from the input $x^{(0)}$ while satisfying the dependencies E_{Kr}. The algorithm assumes the entries of A are implicit and so do not need to be stored in memory or cache. Algorithm 8.4.1 also has three execution-related parameters:

- b, which is analogous to the parameter b for PA1 in the previous section, as it is the blocking parameter on the $(d+1)$th dimension (non-mesh dimension) of the iteration space which ranges from 1 to s;
- r, which defines the side length of the mesh chunk each processor works on at a given time;

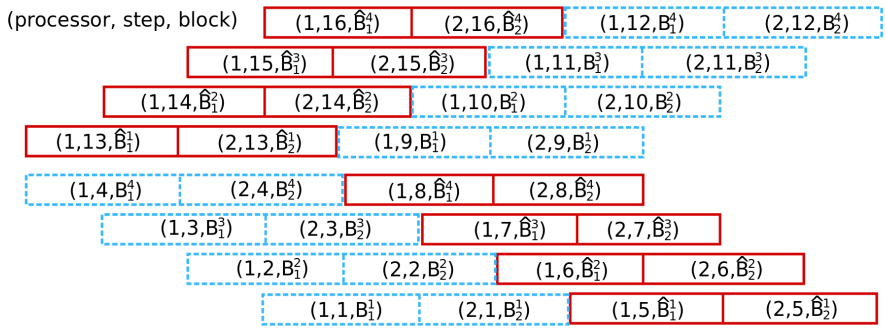


Figure 8.3: Work assignment and order of Algorithm 8.4.1 labelled according to (executing processor number, execution step number, block name corresponding to Algorithm 8.4.1) for execution in Figure 8.2.

- p, which is the number of processors as before.

Like PA1 with $b > 1$, Algorithm 8.4.1 defines $(d+1)$-dimensional blocks that enable improved cache reuse when a second level of blocking is imposed. The key difference is that in Algorithm 8.4.1 the blocking is done on the global problem, rather than the local chunk of the work. Each of these global blocks is executed in parallel, with synchronizations done between each d-dimensional slice of the block. Improved memory-bandwidth cost is achieved by Algorithm 8.4.1 by ensuring that each processor executes subslices which, along with their dependencies, are small enough to be stored in cache between synchronizations.

Figure 8.2 demonstrates the schedule assignment of Algorithm 8.4.1 for computing a Krylov basis of length $s = 8$ for a 1D mesh problem with $n = 10$ mesh points and stencil width $m = 2$, using $b = 4$ and $r = 4$. The figure also depicts the dummy computations introduced to simplify Algorithm 8.4.1, the number of which is proportional to $sn^{d-1}mb$ and thus negligible relative to the overall work, $O(sn^d)$, when $n \gg mb$. The figure also shows why Algorithm 8.4.1 contains two alternating loops from $l = 1$ to b, as this allows the blocks computed by each processor at the last iteration of the first loop over l to satisfy the dependencies of the block needed by the same processor for the first iteration of the second loop over l, and so forth in an alternating manner. Figure 8.2 demonstrates at what step the two processors compute each block.

Algorithm 8.4.1 satisfies the dependencies E_{Kr}, since the pieces of the work executed in parallel within the same block in the first loop nest (for-loop on line 7 of Algorithm 8.4.1), $B^{u_1,\ldots,u_d,l}$, correspond to chunks of a single matrix-vector multiplication with no interdependencies for each l. Further, each $B^{u_1,\ldots,u_d,l}$ depends only on $B^{s_1,\ldots,s_d,l-1}$ for each $(s_j \in \{u_j - 1, u_j\})$, due to the fact that the blocks are shifted back by $(l-1)\cdot m$ in all dimensions at inner loop iteration l (and each vertex depends only on vertices at most m away from its index in the block at the previous level) and the fact that every such block $B^{s_1,\ldots,s_d,l-1}$ was computed previously due to the lexicographical order on the iteration index tuples $(u_1, \ldots, u_d)$. The second loop nest (for-loop on line 18 of Algorithm 8.4.1) reverses the order of the first loop nest, with blocks shifted forward by m with

each increment of l, and starting at the opposite end of the mesh, so that each $B^{u_1,\ldots,u_d,l}$ depends only on the vertices from $B^{s_1,\ldots,s_d,l-1}$ for all $(s_j \in \{u_j, u_j+1\})$. The point of this reversal is to line up the data each processor owns when computing the block for $l = b$ with the data each processor needs when computing the block for $l = 1$ for the next loop over l.

In particular, we argue that every block $B^{u_1,\ldots,u_d,0}_{q_1,\ldots,q_d}$ and $\hat{B}^{\hat{u}_1,\ldots,\hat{u}_d,0}_{q_1,\ldots,q_d}$ accessed by processor $p_{q_1,\ldots,q_d}$ resides in the local memory of the processor. For the first loop nest (line 7) with $w = 1$, this block is the input block defined on line 5 of Algorithm 8.4.1. For each first inner loop iteration $(l = 1)$ of the second loop nest (line 18), the global block defined in the loop iteration,

$$\hat{B}^{\hat{u}_1,\ldots,\hat{u}_d,0} = \big\{v_{i_1,\ldots,i_d,(w-1)2b+b} \in V_{\mathrm{Kr}} : i_j \in [b\cdot(z-\hat{u}_j) - (b-1)\cdot m + 1, \\ b\cdot(z-\hat{u}_j+1) - (b-1)\cdot m\big\},$$

is the same block as the one computed in the first loop nest at the last inner loop iteration $(l = b)$ with $(u_1, \ldots, u_d) = (z - \hat{u}_1 + 1, \ldots, z - \hat{u}_d + 1)$, namely

$$B^{u_1,\ldots,u_d,b} = \big\{v_{i_1,\ldots,i_d,(w-1)2b+b} \in V_{\mathrm{Kr}} : i_j \in [b\cdot(u_j-1) - (b-1)\cdot m + 1 \\ b\cdot u_j - (b-1)\cdot m]\big\},$$

since $u_i = z - \hat{u}_i + 1$ for all $i \in [1, d]$. In both cases, the blocks are subdivided among processors in the exact same fashion, so we have demonstrated that all blocks $B^{u_1,\ldots,u_d,l}_{q_1,\ldots,q_d}$ and $\hat{B}^{\hat{u}_1,\ldots,\hat{u}_d,l}_{q_1,\ldots,q_d}$ for $l \in [0, b-1]$ are in memory of processor $q_1, \ldots, q_d$ each time they are requested as a dependency.

Now $B^{u_1,\ldots,u_d,l}_{q_1,\ldots,q_d}$ depends on all r^d entries of $B^{\hat{u}_1,\ldots,\hat{u}_d,l-1}_{q_1,\ldots,q_d}$ for all (q_j), and similarly for $\hat{B}$, since each block is shifted by m (the interaction range). Further, there are a total of at most $(2m + r)^d$ entries that each block depends on. Therefore, at most $(2m + r)^d - r^d$ entries need to be communicated from remote processors at each step and these processors are the nearest neighbors of the recipient, namely $p_{s_1,\ldots s_d}$ for each $(s_j \in \{q_j - 1, q_j\})$, for the first loop, over $(u_1, \ldots, u_d)$, and for each $(s_j \in \{q_j, q_j+1\})$, for the second loop, over $(\hat{u}_1, \ldots, \hat{u}_d)$. So the interprocessor bandwidth cost for each inner loop given a particular iteration $(u_1, \ldots, u_d)$ or $(\hat{u}_1, \ldots, \hat{u}_d)$ of Algorithm 8.4.1 is (assuming $r > m$),

$$W^u_{\mathrm{ObKs}} \leq \sum_{l=1}^{b} \left[(2m + r)^d - r^d\right] = O(b \cdot m \cdot r^{d-1}).$$

Over $s/(2b)$ (assumed an integer) iterations of the outer loop and

$$z^d \approx \frac{(n + m\cdot(b-1))^d}{r^d \cdot p} = O\left(\frac{n^d}{r^d \cdot p}\right)$$

iterations of loops over $(u_1, \ldots, u_d)$ and $(\hat{u}_1, \ldots, \hat{u}_d)$, assuming that $m \cdot b \ll n$ (b will be picked

to be no more than $\hat{M}^{1/d}$), the total interprocessor bandwidth communication cost is then

$$\begin{aligned} W_{\text{ObKs}}(n, m, s, d, b, r, p) &= O\left(\frac{s}{2b} \cdot \frac{n^d}{r^d \cdot p} \cdot W^u_{\text{ObKs}}\right) \\ &= O\left(\frac{s}{2b} \cdot \frac{n^d}{r^d \cdot p} \cdot b \cdot m \cdot r^{d-1}\right) \\ &= O\left(\frac{s \cdot n^d \cdot m}{r \cdot p}\right). \end{aligned}$$

If we pick $r = n/p^{1/d}$, this cost matches the cost of PA0 (equivalently PA1 with $b = 1$); however, we will pick r so as to allow the local mesh chunks to fit into cache, $r = \Theta(\hat{M}^{1/d})$, which will result in a greater interprocessor bandwidth cost. On the other hand, the quantity b does not appear in this interprocessor bandwidth cost, unlike that of PA1.

To achieve a good memory-bandwidth cost, the parameter r should be picked so that the slices $B^{u_1,\ldots,u_d,l}_{q_1,\ldots,q_d}$ (of size r^d) computed by each process, as well as their dependencies (of size at most $(2m + r)^d$), fit into the cache of size $\hat{M}$, i.e., $\hat{M} \geq (2m + r)^d$. Assuming that $m \ll \hat{M}^{1/d}$, this implies that we can pick $r = \Theta(\hat{M}^{1/d})$. If this is done, the memory-bandwidth cost of Algorithm 8.4.1 is not much greater than the interprocessor bandwidth cost for large enough b, since the dependencies from the previous block on a given processor may be kept in cache prior to execution of the next block. However, for each loop iteration over $(u_1, \ldots, u_d)$ and $(\hat{u}_1, \ldots, \hat{u}_d)$, the blocks $B^{u_1,\ldots,u_d,0}_{q_1,\ldots,q_d}$ and $\hat{B}^{\hat{u}_1,\ldots,\hat{u}_d,0}_{q_1,\ldots,q_d}$ must be read from main memory to cache; the memory-bandwidth cost for any any one such iteration is

$$\hat{W}^u_{\text{ObKs}} \leq r^d + W^u_{\text{ObKs}}.$$

The total memory bandwidth cost of the algorithm over $z^d \approx \frac{(n+m\cdot(b-1))^d}{r^d \cdot p}$ iterations is, therefore,

$$\begin{aligned} \hat{W}_{\text{ObKs}}(n, m, s, d, b, r, p) &= O\left(\frac{s}{2b} \cdot \frac{n^d}{r^d \cdot p} \cdot \hat{W}^u_{\text{ObKs}}\right) + W_{\text{ObKs}} \\ &= O\left(\frac{s \cdot n^d}{b \cdot p}\right) + W_{\text{ObKs}} \\ &= O\left(\frac{s \cdot n^d}{b \cdot p} + \frac{s \cdot n^d \cdot m}{r \cdot p}\right) \\ &= O\left(\frac{s \cdot n^d}{b \cdot p} + \frac{s \cdot n^d \cdot m}{\hat{M}^{1/d} \cdot p}\right). \end{aligned}$$

Therefore, if we pick $b = \Omega(\hat{M}^{1/d}/m)$, the memory bandwidth cost will be asymptotically the same as the interprocessor bandwidth cost. Since, as stated above, the interprocessor bandwidth cost is optimal when $\hat{M}^{1/d} = \Omega(n/p^{1/d})$, the memory bandwidth will be optimal too. Assuming that $\hat{M} \leq (n + 2m)^d$, in which case the local mesh chunk and its dependencies do not fit into

cache, the memory bandwidth cost of the new algorithm, $\hat{W}_{\text{ObKs}}$, will be a factor of $\Theta(1/b)$ less than the memory-bandwidth cost of doing each matrix-vector multiplication in sequence, using $\hat{W}_{\text{PA0}}$. Further, the new algorithm avoids the increased interprocessor-bandwidth cost of PA1 with $b > 1$ (when $d \geq 2$). We can potentially select b up to $\hat{M}^{1/d}$ and achieve a factor of up to $\Theta(\frac{1}{\hat{M}^{1/d}})$ reduction in memory bandwidth cost. We see that this algorithm has a sweet spot in terms of performance when $\hat{M} < n^d/p$ but not $\hat{M} \ll n^d/p$, so that the local mesh chunk does not fit into cache (so the memory-bandwidth efficiency of PA0 deteriorates), but simultaneously the cache size is big enough so that the blocking done by Algorithm 8.4.1 does not significantly increase the interprocessor-communication cost (and also the memory-bandwidth cost). The total communication volume of the new algorithm is proportional to the surface area of all the mesh blocks multiplied by $s \cdot m$ and therefore grows when the blocks, whose size cannot exceed the cache size, are small.

The synchronization cost of this algorithm is

$$S_{\text{ObKs}} = O(s \cdot n^d/(p \cdot \hat{M})),$$

since it requires a synchronization for every loop iteration from $l = 1$ to b (over $s/(2b)$ invocations), and because the number of outer loop iterations is $O(n^d/(p \cdot \hat{M}))$. This synchronization cost is larger than the s/b synchronizations needed by PA1, especially when the cache size ($\hat{M}$) is small with respect to n^d/p.

Algorithm 8.4.1 Outer-blocked-Krylov-schedule($V_{\mathrm{Kr}}, x^{(0)}, n, m, d, s, b, r, p$)

Require: Restrict V_{Kr} to the set of computations needed to compute $x^{(s)}$ from $x^{(0)}$; assume $p^{1/d}$ and $s/2b$ are integers and $m \leq \frac{1}{2} r \cdot p^{1/d}$; assume elements of the input mesh $x^{(0)}$ may be referenced as $x_{i_1,\ldots,i_d}$.

Let $b = r \cdot p^{1/d}$ and $z = \lceil \frac{n+m\cdot(b-1)}{b} \rceil$.
Augment V_{Kr} with dummy computations $v_{i_1,\ldots,i_d,l}$ for all $(i_1,\ldots,i_d) \notin [1,n]^d$, the result of which is zero.
Arrange processors into d-dimensional grid as $p_{q_1,\ldots,q_d}$ for each $q_j \in [1, p^{1/d}]$.
Let $B^{u_1,\ldots,u_d,0} = \left\{ x_{i_1,\ldots,i_d} \in x^{(0)} : i_j \in [1, b \cdot u_j] \right\}$ for each $(u_1,\ldots,u_d) \in [1,z]^d$.
Let sub-block $B^{u_1,\ldots,u_d,0}_{q_1,\ldots,q_d} = \left\{ x_{i_1,\ldots,i_d} \in B^{u_1,\ldots,u_d,0} : i_j \in [(q_j-1)\cdot r + 1, q_j \cdot r] \right\}$ start in the memory of processor $p_{q_1,\ldots,q_d}$.
for $w = 1$ to $s/2b$ **do**
  for $(u_1,\ldots,u_d) = (1,\ldots,1)$ to $(z,\ldots,z)$ **do** in lexicographical ordering
    for $l = 1$ to b **do**
      Let $B^{u_1,\ldots,u_d,l} = \left\{ v_{i_1,\ldots,i_d,(w-1)2b+l} \in V_{\mathrm{Kr}} : \right.$
        $\left. i_j \in [b\cdot(u_j-1) - (l-1)\cdot m + 1, b\cdot u_j - (l-1)\cdot m] \right\}$.

      Define sub-blocks $B^{u_1,\ldots,u_d,l}_{q_1,\ldots,q_d} = \left\{ v_{i_1,\ldots,i_d,l} \in B^{u_1,\ldots,u_d,l} : \right.$
        $\left. i_j \in [(q_j-1)\cdot r + 1, q_j \cdot r] \right\}$ for each $(q_j \in [1, p^{1/d}])$.
      Compute $B^{u_1,\ldots,u_d,l}_{q_1,\ldots,q_d}$ with processor $p_{q_1,\ldots q_d}$,
        fetching dependencies from $B^{u_1,\ldots,u_d,l-1}_{s_1,\ldots,s_d}$ for each $(s_j \in \{q_j - 1, q_j\})$.
    % Store sub-block from last loop iteration for use in next loop
    Store sub-block $B^{u_1,\ldots,u_d,b}_{q_1,\ldots,q_d}$ in memory as $\hat{B}^{\hat{u}_1,\ldots,\hat{u}_d,0}_{q_1,\ldots,q_d}$ with $u_i = z - \hat{u}_i + 1$.
  for $(\hat{u}_1,\ldots,\hat{u}_d) = (1,\ldots,1)$ to $(z,\ldots,z)$ **do** in lexicographical ordering
    for $l = 1$ to b **do**
      Let $\hat{B}^{\hat{u}_1,\ldots,\hat{u}_d,l} = \left\{ v_{i_1,\ldots,i_d,(w-1)2b+b+l} \in V_{\mathrm{Kr}} : \right.$
        $\left. i_j \in [b\cdot(z-\hat{u}_j) + (l-b+1)\cdot m + 1, b\cdot(z-\hat{u}_j+1) + (l-b+1)\cdot m] \right\}$.

      Define sub-blocks $\hat{B}^{\hat{u}_1,\ldots,\hat{u}_d,l}_{q_1,\ldots,q_d} = \left\{ v_{i_1,\ldots,i_d,l} \in \hat{B}^{\hat{u}_1,\ldots,\hat{u}_d,l} : \right.$
        $\left. i_j \in [(q_j-1)\cdot r + 1, q_j \cdot r] \right\}$ for each $(q_j \in [1, p^{1/d}])$.
      Compute $\hat{B}^{\hat{u}_1,\ldots,\hat{u}_d,l}_{q_1,\ldots,q_d}$ with processor $p_{q_1,\ldots q_d}$
        fetching dependencies from $\hat{B}^{\hat{u}_1,\ldots,\hat{u}_d,l-1}_{s_1,\ldots,s_d}$ for each $(s_j \in \{q_j, q_j+1\})$.
    % Store sub-block from last loop iteration for use in next loop or the output $x^{(s)}$
    Store sub-block $\hat{B}^{\hat{u}_1,\ldots,\hat{u}_d,b}_{q_1,\ldots,q_d}$ in memory as $B^{u_1,\ldots,u_d,0}_{q_1,\ldots,q_d}$ with $u_i = z - \hat{u}_i + 1$.

Ensure: $x^{(s)}$ has been computed and saved into memory via computation of all of V_{Kr}.

Chapter 9

Finding the Shortest Paths in Graphs

The idea of 2.5D algorithms, which previous chapters detailed for dense linear algebra, may also be extended to algorithms for finding all-pairs shortest paths in a graph. The all-pairs shortest paths (APSP) is a fundamental graph problem with many applications in urban planning and simulation [107], datacenter network design [41], metric nearness problem [29], and traffic routing. APSP is also used as a subroutine in other graph algorithms, such as Ullman and Yannakakis's breadth-first search algorithm [161], which is suitable for high diameter graphs.

Given a directed graph $G = (V, E)$ with n vertices $V = \{v_1, v_2, ..., v_n\}$ and m edges $E = \{e_1, e_2, ..., e_m\}$, the distance version of the algorithm computes the length of the shortest path from v_i to v_j for all (v_i, v_j) pairs. The full version also returns the actual paths in the form of a predecessor matrix. Henceforth, we will call the distance-only version "all-pairs shortest distances" (APSD) to avoid confusion.

The classical dynamic programming algorithm for APSP is due to Floyd [59] and Warshall [167]. Serial blocked versions of the Floyd-Warshall algorithm have been formulated [125] to increase data locality. The algorithm can also be recast into semiring algebra over vectors and matrices. This vectorized algorithm (attributed to Kleene) is rich in matrix multiplications over the (min,+) semiring [117]. Several theoretical improvements have been made, resulting in subcubic algorithms for the APSD problem [34]. However, in practice, these algorithms are typically not competitive with simpler cubic algorithms.

Variants of the Floyd-Warshall algorithm are most suitable for dense graphs. Johnson's algorithm [92], which is based on repeated application of Dijkstra's single-source shortest path algorithm (SSSP), is theoretically faster than the Floyd-Warshall variants on sufficiently sparse graphs. However, the data dependency structure of this algorithm (and Dijkstra's algorithm in general) make scalable parallelization difficult (computing Dijkstra for different sources from each processor would require replicating the graph). SSSP algorithms based on Δ-stepping [115] scale better in practice but their performance is input dependent and scales with $O(m + d \cdot L \cdot \log n)$, where d is the maximum vertex degree and L is the maximum shortest path weight from the source. Con-

This chapter is based on joint work with Aydin Buluc [144].

sequently, it is likely that a Floyd-Warshall based approach would be competitive even for sparse graphs, as realized on graphical processing units [30].

Given the $\Theta(n^2)$ output of the algorithm, large instances can not be solved on a single node due to memory limitations. Further, a distributed memory approach is favorable over a sequential out-of-core method, because of the high computational complexity of the problem, which motivates using parallel computing to lower the time to solution. In this chapter, we are concerned with obtaining high performance in a practical implementation by reducing communication cost and increasing data locality through optimized matrix multiplication over semirings.

The major contribution of this chapter is the implementation of a 2.5D algorithm for the all-pairs shortest path problem. This implementation leverages a divide-and-conquer formulation of the Floyd-Warshall algorithm (Section 9.3). Our communication-efficient parallelization of this algorithm is described in Section 9.4. The parallel algorithm we give is similar to the algorithm given by Tiskin [157]. We give the first implementation of this method, and show that with respect to a standard (2D) parallelization of APSP, the 2.5D algorithm can attain 2X speed-ups for large problems and up to 6.2X speed-ups for small graphs on a Cray XE6 supercomputer (Section 9.5).

The rest of the chapter is organized as follows

- Section 9.1 reviews previous work on parallel algorithms for the all-pairs shortest paths problem,
- Section 9.2 derives communication lower bounds for the Floyd-Warshall algorithm based on the techniques presented in Chapter 3,
- Section 9.3 gives the divide-and-conquer version of the Floyd-Warshall algorithm,
- Section 9.4 describes our parallelization of the divide-and-conquer algorithm and analyzes the communication costs,
- Section 9.5 presents the performance results of our implementation on a distributed-memory supercomputer,
- Section 9.6 discusses the performance of alternative approaches to the all-pairs shortest path graph problem,
- Section 9.7 overviews conclusions of our analysis,
- Section 9.8 gives pseudocodes for some of the algorithms.

9.1 Previous Work

Jenq and Sahni [89] were the first to give a 2D distributed memory algorithm for the APSP problem, based on the original Floyd-Warshall schedule. Since the algorithm does not employ blocking, it has to perform n global synchronizations, resulting in a latency lower bound of $\Omega(n)$. This

SUMMA-like algorithm [1, 163] is improved further by Kumar and Singh [104] by using pipelining to avoid global synchronizations. Although they reduced the synchronization costs, both of these algorithms have low data reuse: each processor performs n unblocked rank-1 updates on its local submatrix in sequence. Obtaining high-performance in practice requires increasing temporal locality and is achieved by the blocked divide-and-conquer algorithms we consider in this work.

The main idea behind the divide-and-conquer (DC) algorithm is based on a proof by Aho et al. [3] that shows that costs of semiring matrix multiplication and APSP are asymptotically equivalent in the random access machine (RAM) model of computation. Actual algorithms based on this proof are given by various researchers, with minor differences. Our decision to use the DC algorithm as our starting point is inspired by its demonstrated better cache reuse on CPUs [125], and its impressive performance attained on the many-core graphical processor units [30].

Previously known communication bounds [11, 90, 88] for 'classical' (triple-nested loop) matrix multiplication also apply to our algorithm, because Aho et al.'s proof shows how to get the semiring matrix product for free, given an algorithm to compute the APSP. These lower bounds, however, are not necessarily tight because the converse of their proof (to compute APSP given matrix multiplication) relies on the cost of matrix multiplication being $\Omega(n^2)$, which is true for its RAM complexity but not true for its bandwidth and latency costs. In Section 9.2, we show that a tighter bound exists for latency, one similar to the latency lower bound of Cholesky decomposition (Chapter 5).

Seidel [140] showed a way to use fast matrix multiplication algorithms, such as Strassen's algorithm, for the solution of the APSP problem by embedding the $(\min, +)$ semiring into a ring. However, his method only works for undirected and unweighted graphs. We cannot, therefore, utilize the recently discovered communication-optimal Strassen based algorithms [11, 10] directly for the general problem.

Habbal et al. [73] gave a parallel APSP algorithm for the Connection Machine CM-2 that proceeds in three stages. Given a decomposition of the graph, the first step constructs SSSP trees from all the 'cutset' (separator) vertices, the second step runs the classical Floyd-Warshall algorithm for each partition independently, and the last step combines these results using 'minisummation' operations that is essentially semiring matrix multiplication. The algorithm's performance depends on the size of separators for balanced partitions. Without good sublinear (say, $O(\sqrt{n})$) separators, the algorithm degenerates into Johnson's algorithm. Almost all graphs, including those from social networks, lack good separators [110]. Note that the number of partitions are independent (and generally much less) from the number of active processors. The algorithm sends $\Theta(n)$ messages and moves $\Theta(n^2)$ words for the 5-point stencil (2-D grid).

A recent distributed algorithm by Holzer and Wattenhofer [85] runs in $O(n)$ communication rounds. Their concept of communication rounds is similar to our latency concept with the distinction that in each communication round, every node can send a message of size at most $O(\log(n))$ to each one of its neighbors. Our cost model clearly differentiates between bandwidth and latency costs without putting a restriction on message sizes. Their algorithm performs breadth-first search from every vertex with carefully chosen starting times. The distributed computing model used in their work, however, is incompatible with ours.

Brickell et al. [29] came up with a linear programming formulation for the APSP problem, by

exploiting its equivalence to the decrease-only version of the metric nearness problem (DOMN). Their algorithm runs in $O(n^3)$ time using a Fibonacci heap, and the dual problem can be used to obtain the actual paths. Unfortunately, heaps are inherently sequential data structures that limit parallelism. Since the equivalence between APSP and DOMN goes both ways, our algorithm provides a highly parallel solution to the DOMN problem as well.

A considerable amount of effort has been devoted into precomputing transit nodes that are later used for as shortcuts when calculating shortest paths. The PHAST algorithm [44], which is based on contraction hierarchies [65], exploits this idea to significantly improve SSSP performance on road graphs with non-negative edge weights. The impressive performance achieved on the SSSP problem makes APSP calculation on large road networks feasible by repeatedly applying the PHAST algorithm. These algorithms based on precomputed transit nodes, however, do not dominate the classical algorithms such as Dijkstra and Δ-stepping for general types of inputs. Precomputation yields an unacceptable number of shortcuts for social networks, making the method inapplicable for networks that do not have good separators. This is analogous to the fill that occurs during sparse Gaussian elimination [136], because both algorithms rely on some sort of vertex elimination.

Due to their similar triple nested structure and data access patterns, APSP, matrix multiplication, and LU decomposition problems are sometimes classified together. The Gaussian elimination paradigm of Chowdhury and Ramachandran [36] provides a cache-oblivious framework for these problems, similar to Toledo's recursive blocked LU factorization [159]. Our APSP work is orthogonal to that of Chowdhury and Ramachandran in the sense we provide distributed memory algorithms that minimize internode communication (both latency and bandwidth), while their method focuses on cache-obliviousness and multithreaded (shared memory) implementation.

A communication-avoiding parallelization of the recursive all-pairs shortest-paths algorithm was given by Tiskin under the BSP theoretical model [157]. Our algorithm is similar, though we pay closer attention to data layout, lower-bound the communication, and study the performance of a high-performance implementation.

Our main motivating work will be 2.5D formulations of matrix multiplication and LU factorization for dense linear algebra (Chapter 5). These algorithms are an adaptation and generalization of 3D matrix multiplication [43, 1, 2, 23, 93]. The main idea is to store redundant intermediate data, in order to reduce communication bandwidth. Bandwidth is reduced by a factor of $\sqrt{c}$ at the cost of a memory usage overhead of a factor of c. The technique is particularly useful for the strong scaling regime, where one can solve problems faster by storing more intermediates spread over more processors.

9.2 Lower Bounds

Algorithm 9.2.1 is the Floyd-Warshall algorithm for symmetric graphs. We note that its structure is quite similar to the Cholesky algorithm (Algorithm 5.1.2 in Chapter 5). In fact, we note that each P_{ij} in Algorithm 9.2.1 depends on Z_{ijk} for $k \in [1, i-1]$ and is a dependency of Z_{ikj} for $k \in [j+1, i-1]$ as well as Z_{kij} for $k \in [i+1, n]$. This dependency structure is the same as

that of Algorithm 5.1.2. While in the Cholesky algorithm the summations could be computed in any order, in the Floyd-Warshall algorithm the minimum computed on line 9.2.1, can be computed in any order. Each reduction tree for computing such a min corresponds to a reduction tree for computing a sum in the Cholesky algorithm. Therefore, any dependency graph G_{FW} for computing Algorithm 9.2.1 is isomorphic to some G_{Ch} the dependency graph for computing Cholesky as defined in Chapter 5.

Algorithm 9.2.1 $[P] \leftarrow$ Floyd-Warshall(A)

Require: $A \in \mathbb{S}^{n\times n}$ is a symmetric n-by-n adjacency matrix of undirected graph G.

1: Initialize $P = A$
2: **for** $j \leftarrow 1$ to n **do**
3: **for** $i \leftarrow j+1$ to n **do**
4: **for** $k = 1$ to $j-1$ **do**
5: $Z_{ijk} = P_{ik} + P_{jk}$
6: $P_{ij} = \min\left(P_{ij}, \min_{k\in[1,j-1]} Z_{ijk}\right)$

Ensure: $P \in \mathbb{S}^{n\times n}$ is an n-by-n path distance matrix derived from adjacency matrix of graph G.

The following communication lower bounds then follow as direct extensions from the Cholesky case.

Theorem 9.2.1. *Any parallelization of Algorithm 9.2.1 where some processor computes no more than $\frac{1}{x}$ of the elements in Z and at least $\frac{1}{q}$ elements in Z, for any $4 \leq x \leq q \ll n$, must incur a communication of*

$$W_{\mathrm{FW}} = \Omega\left(n^2/q^{2/3}\right).$$

Proof. This theorem holds since every parallelization of Algorithm 9.2.1 corresponds to a dependency graph that is isomorphic to some G_{Ch}, and by application of Theorem 5.1.5 the lower bound on communication holds. □

Theorem 9.2.2. *Any parallelization of Algorithm 9.2.1 in which some processor computes no more than $\frac{1}{x}$ of the elements in Z and at least $\frac{1}{q}$ elements in Z, for any $4 \leq x \leq q \ll n$, incurs the following computation (F), bandwidth (W), and latency (S) costs, for some $b \in [1, n]$,*

$$F_{\mathrm{FW}} = \Omega\left(n \cdot b^2\right), \qquad W_{\mathrm{FW}} = \Omega\left(n \cdot b\right), \qquad S_{\mathrm{FW}} = \Omega\left(n/b\right),$$

and furthermore, $F_{\mathrm{FW}} \cdot S_{\mathrm{FW}}^2 = \Omega\left(n^3\right), \quad W_{\mathrm{FW}} \cdot S_{\mathrm{FW}} = \Omega\left(n^2\right).$

Proof. This theorem holds since every parallelization of Algorithm 9.2.1 corresponds to a dependency graph that is isomorphic to some G_{Ch}, and by application of Theorem 5.1.6 the lower bounds on tradeoffs hold. □

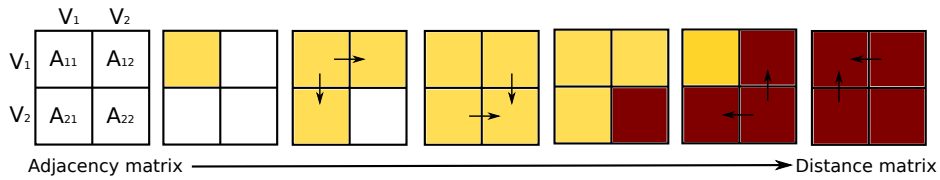


Figure 9.1: DC-APSP algorithm, with initial adjacency distance denoted in white, partially complete path distances in yellow, and final path distances in red

9.3 Divide-and-Conquer APSP

The all-pairs shortest-paths problem corresponds to finding the matrix closure on the tropical $(\min, +)$ semiring [117]. A semiring is denoted by $(\mathbb{S}, \oplus, \otimes, 0, 1)$, where $\oplus$ and $\otimes$ are binary operations defined on the set $\mathbb{S}$ with identity elements 0 and 1, respectively [58]. In the case of the *tropical semiring*, $\oplus$ is min, $\otimes$ is $+$, the additive identity is $+\infty$, and the multiplicative identity is 0. Compared to the classical matrix multiplication over the ring of real numbers, in our semiring-matrix-matrix multiplication (also called the distance product [171]), each multiply operation is replaced with an addition (to calculate the length of a larger path from smaller paths or edges) and each add operation is replaced with a minimum operation (to get the minimum in the presence of multiple paths).

Algorithm 9.3.1 gives the high-level structure of the divide-and-conquer all-pairs-shortest-path algorithm (DC-APSP). The workflow of the DC-APSP algorithm is also pictured in Figure 9.1. The correctness of this algorithm has been proved by many researchers [3, 30, 125] using various methods. Edge weights can be arbitrary, including negative numbers, but we assume that the graph is free of negative cycles. The tropical semiring does not have additive inverses, hence fast matrix multiplication algorithms like those by Strassen [152] and Coppersmith-Winograd [38] are not applicable for this problem.

For simplicity, we formulate our algorithms and give results only for adjacency matrices of power-of-two dimension. Extending the algorithms and analysis to general adjacency matrices is straightforward.

Each semiring-matrix-matrix multiplication performs $O(n^3)$ additions and $O(n^2)$ minimum (min) operations. If we count each addition and min operation as $O(1)$ flops, the total computation cost of DC-APSP, F, is given by a recurrence

$$F(n) = 2 \cdot F(n/2) + O(n^3) = O(n^3).$$

Thus the number of operations is the same as that required for matrix multiplication.

9.4 Parallelization of DC-APSP

In this section, we introduce techniques for parallelization of the divide-and-conquer all-pairs-shortest-path algorithm (DC-APSP). Our first approach uses a 2D block-cyclic parallelization. We

Algorithm 9.3.1 A=DC-APSP(A, n)

Require: $A \in \mathbb{S}^{n \times n}$ is a graph adjacency matrix of a n-node graph G

if $n = 1$ **then**
    return.
Partition $A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}$, where all A_{ij} are $n/2$-by-$n/2$
% *P*artition the vertices $V = (V_1, V_2)$
$A_{11} \leftarrow$ DC-APSP$(A_{11}, n/2)$
% *F*ind all-pairs shortest paths between vertices in V_1
$A_{12} \leftarrow A_{11} \cdot A_{12}$
% *P*ropagate paths from V_1 to V_2
$A_{21} = A_{21} \cdot A_{11}$
% *P*ropagate paths from V_2 to V_1
$A_{22} = \min(A_{22}, A_{21} \cdot A_{12})$
% *U*pdate paths to V_2 via paths from V_2 to V_1 and back to V_2
$A_{22} \leftarrow$ DC-APSP$(A_{22}, n/2)$
% *F*ind all-pairs shortest paths between vertices in V_2
$A_{21} = A_{22} \cdot A_{21}$
% *F*ind shortest paths from V_2 to V_1
$A_{12} = A_{12} \cdot A_{22}$
% *F*ind shortest paths from V_1 to V_2
$A_{11} = \min(A_{11}, A_{12} \cdot A_{21})$
% *F*ind all-pairs shortest paths for vertices in V_1

Ensure: $A \in \mathbb{S}^{n \times n}$ is the APSP distance matrix of G

demonstrate that a careful choice of block-size can minimize both latency and bandwidth costs simultaneously. Our second approach utilizes a 2.5D decomposition [143, 146]. Our cost analysis shows that the 2.5D algorithm reduces the bandwidth cost and improves strong scalability.

9.4.1 2D Divide-and-Conquer APSP

We start by deriving a parallel DC-APSP algorithm that operates on a square 2D processor grid and consider cyclic and blocked variants.

2D Semiring-Matrix-Matrix-Multiply

Algorithm 9.4.1 describes an algorithm for performing Semiring-Matrix-Matrix-Multiply (SMMM) on a 2D processor grid denoted by Λ. Since the data dependency structure of SMMM is identical to traditional matrix multiply, we employ the popular SUMMA algorithm [163]. The algorithm is formulated in terms of distributed rank-1 updates. These updates are associative and commutative so they can be pipelined or blocked. To achieve optimal communication performance, the matrices should be laid out in a blocked fashion, and each row and column of processors should broadcast

Algorithm 9.4.1 $C \leftarrow$ 2D-SMMM$(A, B, C, \Lambda[1 : \sqrt{p}, 1 : \sqrt{p}], n, p)$

Require: process $\Lambda[i, j]$ owns $A_{ij}, B_{ij}, C_{ij} \in \mathbb{S}^{\frac{n}{\sqrt{p}} \times \frac{n}{\sqrt{p}}}$

1: % *e*xecute loop iterations in parallel
2: **for** $i, j \leftarrow 1$ to $\sqrt{p}$ **do**
3: **for** $k \leftarrow 1$ to $\sqrt{p}$ **do**
4: Broadcast A_{ik} to processor columns $\Lambda[i, :]$
5: Broadcast B_{kj} to processor rows $\Lambda[:, j]$
6: $C_{ij} \leftarrow \min(C_{ij}, A_{ik} \cdot B_{kj})$

Ensure: process $\Lambda[i, j]$ owns $C_{ij} \in \mathbb{S}^{\frac{n}{\sqrt{p}} \times \frac{n}{\sqrt{p}}}$

its block-row and block-column in turn. Given p processors, each processor would then receive $O(\sqrt{p})$ messages of size $O(n^2/p)$, giving a bandwidth cost of $O(n^2/\sqrt{p})$. We note that any different classical distributed matrix multiplication algorithm (e.g. Cannon's algorithm [32]) can be used here in place of SUMMA.

2D Blocked Divide-and-Conquer APSP

Algorithm 9.8.1 (psuedocode given in the Appendix) displays a parallel 2D blocked version of the DC-APSP algorithm. In this algorithm, each SMMM is computed on the quadrant of the processor grid on which the result belongs. The operands, A and B, must be sent to the processor grid quadrant on which C is distributed. At each recursive step, the algorithm recurses into one quadrant of the processor grid. Similar to SMMM, this is also an owner computes algorithm in the sense that the processor that owns the submatrix to be updated does the computation itself after receiving required inputs from other processors.

This blocked algorithm has a clear flaw, in that at most a quarter of the processors are active at any point in the algorithm. We will alleviate this load-imbalance by introducing a block-cyclic version of the algorithm.

2D Block-Cyclic Divide-and-Conquer APSP

Algorithm 9.8.2 (given in the Appendix) details the full 2D block-cyclic DC-APSP algorithm. This block-cyclic algorithm operates by performing *cyclic-steps* until a given block-size, then proceeding with *blocked-steps* by calling the blocked algorithm as a subroutine. At each cyclic-step, each processor operates on sub-blocks of its local block, while at each blocked-step a sub-grid of processors operate on their full matrix blocks. In other words, a cyclic-step reduces the local working sets, while a blocked-step reduces the number of active processors. These two steps are demonstrated in sequence in Figure 9.2 with 16 processors.

We note that no redistribution of data is required to use a block-cyclic layout. Traditionally, (e.g. in ScaLAPACK [28]) using a block-cyclic layout requires that each processor own a block-cyclic portion of the starting matrix. However, the APSP problem is invariant to permutation (permuting the numbering of the node labels does not change the answer). We exploit permutation

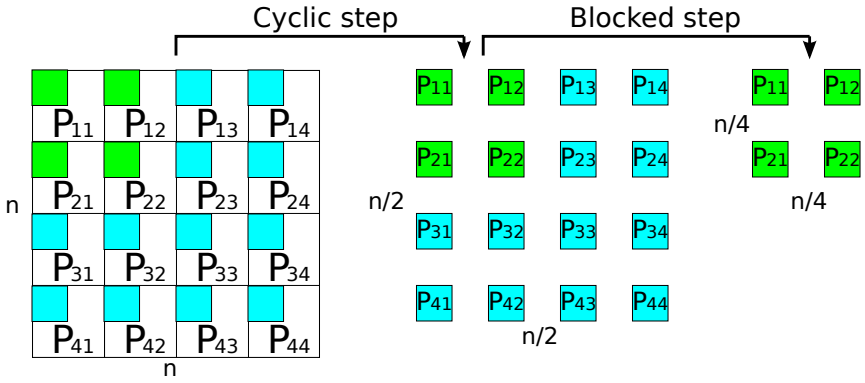


Figure 9.2: Our block-cyclic 2D APSP algorithm performs cyclic-steps until a given block-size, then performs blocked-steps as shown in this diagram.

invariance by assigning each process the same sub-block of the adjacency and distance matrices, no matter how many blocked or cyclic steps are taken.

As derived in Appendix A in [144], if the block size is picked as $b = O(n/\log(p))$ (execute $O(\log\log(p))$ cyclic recursive steps), the bandwidth and latency costs are

$$W_{\text{bc-2D}}(n,p) = O(n^2/\sqrt{p}),$$
$$S_{\text{bc-2D}}(p) = O(\sqrt{p}\log^2(p)).$$

These costs are optimal (modulo the polylog latency term) when the memory size is $M = O(n^2/p)$. The costs are measured along the critical path of the algorithm, showing that both the computation and communication are load balanced throughout execution.

9.4.2 2.5D DC-APSP

In order to construct a communication-optimal DC-APSP algorithm, we utilize 2.5D-SMMM. Transforming 2D SUMMA (Algorithm 9.4.1) to a 2.5D algorithm can be done by performing a different subset of updates on each one of c processor layers. Algorithm 9.8.5 (given in the Appendix) details 2.5D SUMMA, modified to perform SMMM. The three dimensional processor grids used in 2.5D algorithms are denoted by Π.

Given a replication factor $c \in [1, p^{1/3}]$, each $\sqrt{p/c}$-by-$\sqrt{p/c}$ processor layer performs n/c outer products. Since each length n outer product vector is subdivided into $\sqrt{p/c}$ chunks, the bandwidth cost is $O(n^2/\sqrt{cp})$ words. These outer products can be blocked into bundles of up to $n/\sqrt{p/c}$ to lower the latency cost to $O(\sqrt{p/c^3})$ messages.

Algorithm 9.8.4 (pseudo-code given in the Appendix) displays the blocked version of the 2.5D DC-APSP algorithm. The blocked algorithm executes multiplies and recurses on octants of the processor grid (rather than quadrants in the 2D version). The algorithm recurses until $c = 1$, which must occur while $p \geq 1$, since $c \leq p^{1/3}$. The algorithm then calls the 2D block-cyclic algorithm on the remaining 2D sub-partition.

The 2.5D blocked algorithm suffers from load-imbalance. In fact, the top half of the processor grid does no work. We can fix this by constructing a block-cyclic version of the algorithm, which performs cyclic steps with the entire 3D processor grid, until the block-size is small enough to switch to the blocked version. The 2.5D block-cyclic algorithm looks exactly like Algorithm 9.8.2, except each call to 2D SMMM is replaced with 2.5D SMMM. This algorithm is given in full in [144].

As derived in Appendix B in [144], if the 2.5D block size is picked as $b_1 = O(n/c)$ (execute $O(\log(c))$ 2.5D cyclic recursive steps), the bandwidth and latency costs are

$$W_{\text{bc-2.5D}}(n,p) = O(n^2/\sqrt{cp}),$$
$$S_{\text{bc-2.5D}}(p) = O(\sqrt{cp}\log^2(p)).$$

These costs are optimal for any memory size (modulo the polylog latency term).

9.5 Experiments

In this section, we show that the distributed APSP algorithms do not just lower the theoretical communication cost, but actually improve performance on large supercomputers. We implement the 2D and 2.5D variants of DC-APSP recursively, as described in the previous section. For fairness, both variants have the same amount of optimizations applied and use the same kernel. We were not able to find any publicly available distributed memory implementations of APSP for comparison.

9.5.1 Implementation

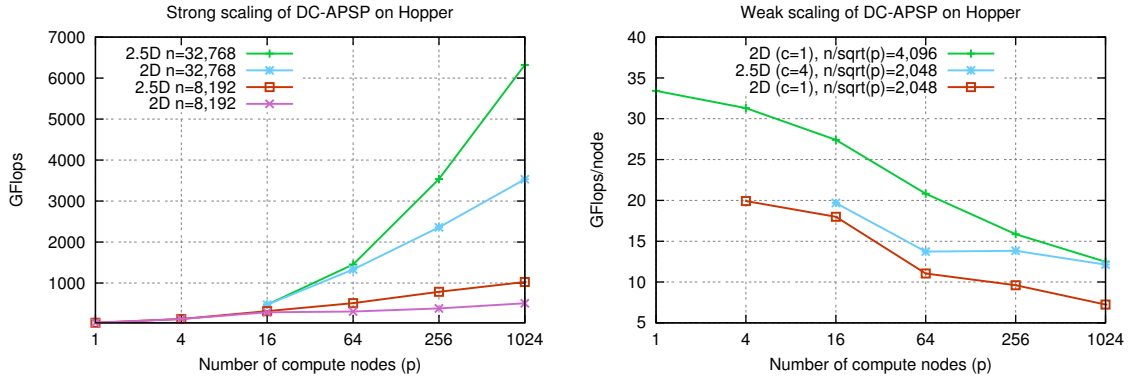
The dominant sequential computational work of the DC-APSP algorithm is the Semiring-Matrix-Matrix-Multiplies (SMMM) called at every step of recursion. Our implementation of SMMM uses two-level cache-blocking, register blocking, explicit SIMD intrinsics, and loop unrolling. We implement threading by assigning L1-cache blocks of C to different threads.

Our 2.5D DC-APSP implementation generalizes the following algorithms: 2D cyclic, 2D blocked, 2D block-cyclic, 2.5D blocked, 2.5D cyclic, and 2.5D block-cyclic. Block sizes b_1 and b_2 control how many 2.5D and 2D cyclic and blocked steps are taken. These block-sizes are set at run-time and require no modification to the algorithm input or distribution.

We compiled our codes with the GNU C/C++ compilers (v4.6) with the -O3 flag. We use Cray's MPI implementation, which is based on MPICH2. We run 4 MPI processes per node, and use 6-way intra-node threading with the GNU OpenMP library. The input is an adjacency matrix with entries representing edge-weights in double-precision floating-point numbers.

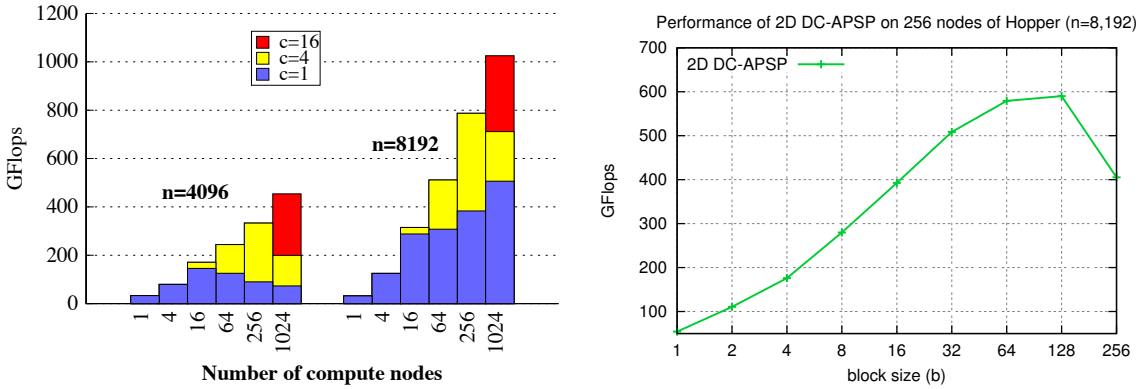
9.5.2 Performance

Our experimental platform is 'Hopper', which is a Cray XE6 supercomputer, the architecture of which we described in Section 6.3.



(a) DC-APSP strong scaling (b) DC-APSP weak scaling

Figure 9.3: Scaling of 2D and 2.5D block-cyclic DC-APSP on Hopper (Cray XE6)



(a) 2.5D DC-APSP small matrix performance (b) Performance of DC-APSP with respect to block size

Figure 9.4: Performance of DC-APSP on Small Matrices

Our threaded Semiring-Matrix-Matrix-Multiply achieves up to 13.6 GF on 6-cores of Hopper (we count each min and plus as a flop), which is roughly 25% of theoretical floating-point peak. This is a fairly good fraction in the absence of an equivalent fused multiply-add operation for our semiring. Our implementation of DC-APSP uses this subroutine to perform APSP at 17% of peak computational performance on 1 node (24 cores, 4 processes, 6 threads per process).

Figure 9.3(a) demonstrates the strong scaling performance of 2D and 2.5D APSP. Strong scaling performance is collected by keeping the adjacency matrix size constant and computing APSP with more processors. The 2.5D performance is given as the best performing variant for any replication factor c (in almost all cases, $c = 4$). Strong scaling a problem to a higher core-count lowers the memory usage per processor, allowing increased replication (increased c). Performing 2.5D style replication improves efficiency significantly, especially at large scale. On 24,576 cores of

Hopper, the 2.5D algorithm improves on the performance of the 2D APSP algorithm by a factor of 1.8x for $n = 8,192$ and 2.0x for $n = 32,768$.

Figure 9.3(b) shows the weak scaling performance of the 2D and 2.5D DC-APSP algorithms. To collect weak scaling data, we keep the problem size per processor $(n/\sqrt{p})$ constant and grow the number of processors. Since the memory usage per processor does not decrease with the number of processors during weak scaling, the replication factor cannot increase. No data-point is given for the 2.5D algorithm with $c = 4$ on four nodes, since this run involved 16 processors, and we need $c \leq p^{1/3}$ We compare data with $n/\sqrt{p} = 2048, 4096$ for 2D ($c = 1$) and with $n/\sqrt{p} = 2048$ for 2.5D ($c = 4$). The 2.5D DC-APSP algorithm performs almost as well as the 2D algorithm with a larger problem size and significantly better than the 2D algorithm with the same problem size.

The overall weak-scaling efficiency is good all the way up to the 24,576 cores (1024 nodes), where the code achieves an impressive aggregate performance over 12 Teraflops (Figure 9.3(b)). At this scale, our 2.5D implementation solves the all-pairs shortest-paths problem for 65,536 vertices in roughly 2 minutes. With respect to 1-node performance, strong scaling allows us to solve a problem with 8,192 vertices over 30x faster on 1024 compute nodes (Figure 9.4(a)). Weak scaling gives us a performance rate up to 380x higher on 1024 compute nodes than on one node.

Figure 9.4(a) shows the performance of 2.5D DC-APSP on small matrices. The bars are stacked so the $c = 4$ case shows the added performance over the $c = 1$ case, while the $c = 16$ case shows the added performance over the $c = 4$ case. A replication factor of $c = 16$ results in a speed-up of 6.2x for the smallest matrix size $n = 4,096$. Overall, we see that 2.5D algorithm hits the scalability limit much later than the 2D counterpart. Tuning over the block sizes (Figure 9.4(b)), we also see the benefit of the block-cyclic layout for the 2D algorithm. The best performance over all block sizes is significantly higher than either the cyclic ($b = 1$) or blocked ($b = n/\sqrt{p}$) performance. We found that the use of cyclicity in the 2.5D algorithm supplanted the need for cyclic steps in the nested call to the 2D algorithm. The 2.5D blocked algorithm can call the 2D blocked algorithm directly without a noticeable performance loss.

9.6 Discussion of Alternatives

We solved the APSP problem using a distributed memory algorithm that minimizes communication and maximizes temporal locality reuse through BLAS-3 subroutines. There are at least two other alternatives to solving this problem. One alternative is to use a sparse APSP algorithm and the other one is to leverage an accelerator architecture such as GPU.

If the graph is big enough so that it requires distribution to multiple processors, the performance of sparse APSP algorithms become heavily dependent on the structure of the graph; and rather poor in general. For the case that the graph is small enough so that it can be fully replicated along different processors, one can parallelize Johnson's algorithm in an embarrassingly parallel way. We experimented with this case, where each core runs many to all shortest paths. Specifically, we wanted to know how sparse the graph needs to get in order to make this fully replicated approach a strong alternative. The break-even points for density depend both on the graph size (the number of vertices) and the number of cores. For example, using 384 cores, solving the APSP problem

on a 16,384 vertex, 5% dense graph, is slightly faster using our approach (18.6 vs. 22.6 seconds) than using the replicated Johnson's algorithm. Keeping the number of vertices intact and further densifying the graph favors our algorithm while sparsifying it favors Johnson's algorithm. Larger cores counts also favor Johnson's algorithm; but its major disadvantage is its inability to run any larger problems due to graph replication.

On the architecture front, we benchmarked a highly optimized CUDA implementation [30] on a single Fermi (NVIDIA X2090) GPU. This GPU implementation also runs the dense recursive algorithm described in this chapter. On a graph with 8,192 vertices, our distributed memory CPU based implementation running on 4 nodes achieved 80% of the performance of the Fermi (which takes 9.9 seconds to solve APSP on this graph). This result shows the suitability of the GPU architecture to the APSP problem, and provides us a great avenue to explore as future work. As more supercomputers become equipped with GPU accelerators, we plan to reimplement our 2.5D algorithm in a way that it can take advantage of the GPUs as coprocessors on each node. The effect of communication avoidance will become more pronounced as local compute phases get faster due to GPU acceleration.

9.7 Conclusions

The divide-and-conquer APSP algorithm is well suited for parallelization in a distributed memory environment. The algorithm resembles well-studied linear algebra algorithms (e.g. matrix multiply, LU factorization). We exploit this resemblance to transfer implementation and optimization techniques from the linear algebra domain to the graph-theoretic APSP problem. In particular, we use a block-cyclic layout to load-balance the computation and data movement, while simultaneously minimizing message latency overhead. Further, we formulate a 2.5D DC-APSP algorithm, which lowers the bandwidth cost and improves parallel scalability. Our implementations of these algorithms achieve good scalability at very high concurrency and confirm the practicality of our analysis. Our algorithm provides a highly parallel solution to the decrease-only version of the metric nearness problem problem as well, which is equivalent to APSP.

Our techniques for avoiding communication allow for a scalable implementation of the divide-and-conquer APSP algorithm. The benefit of such optimizations grows with machine size and level of concurrency. The performance of our implementation can be further improved upon by exploiting locality via topology-aware mapping. The current Hopper job scheduler does not allocate contiguous partitions but other supercomputers (e.g. IBM BlueGene) allocate toroidal partitions, well-suited for mapping of 2D and 2.5D algorithms [143].

9.8 Appendix

Detailed Pseudocodes

Algorithm 9.8.1 $A \leftarrow$ BLOCKED-DC-APSP$(A, \Lambda[1 : \sqrt{p}, 1 : \sqrt{p}], n, p)$

Require: process $\Lambda[i, j]$ owns a block of the adjacency matrix, $A_{ij} \in \mathbb{S}^{\frac{n}{\sqrt{p}} \times \frac{n}{\sqrt{p}}}$

1: **if** $p = 1$ **then**
2: $A =$ DC-APSP(A, n)
3: % *P*artition the vertices $V = (V_1, V_2)$ by partitioning the processor grid
Partition $\Lambda = \begin{bmatrix} \Lambda_{11} & \Lambda_{12} \\ \Lambda_{21} & \Lambda_{22} \end{bmatrix}$, where all Λ_{ij} are $\sqrt{p}/2$-by-$\sqrt{p}/2$
4: % *e*xecute following loop iterations in parallel
5: **for** $i, j \leftarrow 1$ to $\sqrt{p}/2$ **do**
6: % *F*ind all-pairs shortest paths between vertices in V_1
7: $A_{ij} \leftarrow$ BLOCKED-DC-APSP$(A_{ij}, \Lambda_{11}, n/2, p/4)$
8: % *P*ropagate paths from V_1 to V_2
9: $\Lambda_{11}[i, j]$ sends A_{ij} to $\Lambda_{12}[i, j]$.
10: $A_{i,j+\sqrt{p}/2} \leftarrow$ 2D-SMMM$(A_{ij}, A_{i,j+\sqrt{p}/2}, A_{i,j+\sqrt{p}/2}, \Lambda_{12}, n/2, p/4)$
11: % *P*ropagate paths from V_2 to V_1
12: $\Lambda_{11}[i, j]$ sends A_{ij} to $\Lambda_{21}[i, j]$.
13: $A_{i+\sqrt{p}/2,j} \leftarrow$ 2D-SMMM$(A_{i+\sqrt{p}/2,j}, A_{ij}, A_{i+\sqrt{p}/2,j}, \Lambda_{21}, n/2, p/4)$
14: % *U*pdate paths to V_2 via paths from V_2 to V_1 and back to V_2
15: $\Lambda_{12}[i, j]$ sends $A_{i,j+\sqrt{p}/2}$ to $\Lambda_{22}[i, j]$.
16: $\Lambda_{21}[i, j]$ sends $A_{i+\sqrt{p}/2,j}$ to $\Lambda_{22}[i, j]$.
17: $A_{i+\sqrt{p}/2,j+\sqrt{p}/2} \leftarrow$ 2D-SMMM$(A_{i+\sqrt{p}/2,j}, A_{i,j+\sqrt{p}/2}, A_{i+\sqrt{p}/2,j+\sqrt{p}/2}, \Lambda_{22}, n/2, p/4)$
18: % *F*ind all-pairs shortest paths between vertices in V_2
19: $A_{i+\sqrt{p}/2,j+\sqrt{p}/2} \leftarrow$ BLOCKED-DC-APSP$(A_{i+\sqrt{p}/2,j+\sqrt{p}/2}, \Lambda_{22}, n/2, p/4)$
20: % *F*ind shortest paths paths from V_2 to V_1
21: $\Lambda_{22}[i, j]$ sends $A_{i+\sqrt{p}/2,j+\sqrt{p}/2}$ to $\Lambda_{21}[i, j]$.
22: $A_{i+\sqrt{p}/2,j} \leftarrow$ 2D-SMMM$(A_{i+\sqrt{p}/2,j+\sqrt{p}/2}, A_{i+\sqrt{p}/2,j}, A_{i+\sqrt{p}/2,j}, \Lambda_{21}, n/2, p/4)$
23: % *F*ind shortest paths paths from V_1 to V_2
24: $\Lambda_{22}[i, j]$ sends $A_{i+\sqrt{p}/2,j+\sqrt{p}/2}$ to $\Lambda_{12}[i, j]$.
25: $A_{i,j+\sqrt{p}/2} \leftarrow$ 2D-SMMM$(A_{i,j+\sqrt{p}/2}, A_{i+\sqrt{p}/2,j+\sqrt{p}/2}, A_{i,j+\sqrt{p}/2}, \Lambda_{12}, n/2, p/4)$
26: % *F*ind all-pairs shortest paths for vertices in V_1
27: $\Lambda_{12}[i, j]$ sends $A_{i,j+\sqrt{p}/2}$ to $\Lambda_{11}[i, j]$.
28: $\Lambda_{21}[i, j]$ sends $A_{i+\sqrt{p}/2,j}$ to $\Lambda_{11}[i, j]$.
29: $A_{ij} \leftarrow$ 2D-SMMM$(A_{i,j+\sqrt{p}/2}, A_{i+\sqrt{p}/2,j}, A_{ij}, \Lambda_{22}, n/2, p/4)$

Ensure: process $\Lambda[i, j]$ owns a block of the APSP distance matrix, $A_{ij} \in \mathbb{S}^{\frac{n}{\sqrt{p}} \times \frac{n}{\sqrt{p}}}$

Algorithm 9.8.2 $A \leftarrow$ BLOCK-CYCLIC-DC-APSP$(A, \Lambda[1 : \sqrt{p}, 1 : \sqrt{p}], n, p, b)$

Require: process $\Lambda[i, j]$ owns a block of the adjacency matrix, $A_{ij} \in \mathbb{S}^{\frac{n}{\sqrt{p}} \times \frac{n}{\sqrt{p}}}$

1: **if** $n \leq b$ **then**
2: $A \leftarrow$ BLOCKED-DC-APSP(A, Λ, n, p)
3: % *S*witch to blocked algorithm once the matrix is small
4: % *e*xecute following loop iterations in parallel
5: **for** $i, j \leftarrow 1$ to $\sqrt{p}$ **do**
6: $A^l \leftarrow A_{ij}$
7: % A^l *denotes the local matrix owned by* $\Lambda[i, j]$
8: Partition $A^l = \begin{bmatrix} A^l_{11} & A^l_{12} \\ A^l_{21} & A^l_{22} \end{bmatrix}$, where all A^l_{kl} are $n/2$-by-$n/2$
9: % *P*artition the vertices $V = (V_1, V_2)$
10: $A^l_{11} \leftarrow$ BLOCK-CYLIC-DC-APSP$(A^l_{11}, \Lambda, n/2, p, b)$
11: % *F*ind all-pairs shortest paths between vertices in V_1
12: $A^l_{12} \leftarrow$ 2D-SMMM$(A^l_{11}, A^l_{12}, A^l_{12}, \Lambda, n/2, p)$
13: % *P*ropagate paths from V_1 to V_2
14: $A^l_{21} \leftarrow$ 2D-SMMM$(A^l_{21}, A^l_{11}, A^l_{21}, \Lambda, n/2, p)$
15: % *P*ropagate paths from V_2 to V_1
16: $A^l_{22} \leftarrow$ 2D-SMMM$(A^l_{21}, A^l_{12}, A^l_{22}, \Lambda, n/2, p)$
17: % *U*pdate paths among vertices in V_2 which go through V_1
18: $A^l_{22} \leftarrow$ BLOCK-CYLIC-DC-APSP$(A^l_{22}, \Lambda, n/2, p, b)$
19: % *F*ind all-pairs shortest paths between vertices in V_2
20: $A^l_{21} \leftarrow$ 2D-SMMM$(A^l_{22}, A^l_{21}, A^l_{21}, \Lambda, n/2, p)$
21: % *F*ind shortest paths from V_2 to V_1
22: $A^l_{12} \leftarrow$ 2D-SMMM$(A^l_{12}, A^l_{22}, A^l_{12}, \Lambda, n/2, p)$
23: % *F*ind shortest paths from V_1 to V_2
24: $A^l_{11} \leftarrow$ 2D-SMMM$(A^l_{12}, A^l_{21}, A^l_{11}, \Lambda, n/2, p)$
25: % *F*ind all-pairs shortest paths for vertices in V_1

Ensure: process $\Lambda[i, j]$ owns a block of the APSP distance matrix, $A_{ij} \in \mathbb{S}^{\frac{n}{\sqrt{p}} \times \frac{n}{\sqrt{p}}}$

Algorithm 9.8.3 $C \leftarrow$ 2.5D-SMMM$(A, B, C, \Pi[1 : \sqrt{p/c}, 1 : \sqrt{p/c}, 1 : c], n, p, c)$

Require: process $\Pi[i, j, 1]$ owns $A_{ij}, B_{ij}, C_{ij} \in \mathbb{S}^{\frac{n}{\sqrt{p/c}} \times \frac{n}{\sqrt{p/c}}}$

1: % *e*xecute following loop iterations in parallel
2: **for** $m \leftarrow 1$ to c **do**
3: % *e*xecute following loop iterations in parallel
4: **for** $i, j \leftarrow 1$ to $\sqrt{p}$ **do**
5: Π_{ij1} sends A_{ij} to process $\Pi_{i,j,j/c}$
6: Π_{ij1} sends B_{ij} to process $\Pi_{i,j,i/c}$
7: **if** $m = 1$ **then**
8: $C_{ijm} = C_{ij}$
9: **else**
10: $C_{ijm}[:, :] = \infty$
11: **for** $k \leftarrow 1$ to $\sqrt{p/c^3}$ **do**
12: Broadcast $A_{i,m\sqrt{p/c^3}+k}$ to processor columns $\Pi[i, :, m]$
13: Broadcast $B_{m\sqrt{p/c^3}+k,j}$ to processor rows $\Pi[:, j, m]$
14: $C_{ijm} \leftarrow \min(C_{ij}, A_{i,m\sqrt{p/c^3}+k} \cdot B_{m\sqrt{p/c^3}+k,j})$
15: Reduce to first processor layer, $C_{ij} = \sum_{m=1}^{c} C_{ijm}$

Ensure: process $\Pi[i, j, 1]$ owns $C_{ij} \in \mathbb{S}^{\frac{n}{\sqrt{p/c}} \times \frac{n}{\sqrt{p/c}}}$

Algorithm 9.8.4 $A \leftarrow$ 2.5D-BLOCKED-DC-APSP$(A, \Pi[1 : \sqrt{p/c}, 1 : \sqrt{p/c}, 1 : c], n, p, c, b)$

Require: process $\Pi[i, j, 1]$ owns a block of the adjacency matrix, $A_{ij} \in \mathbb{S}^{\frac{n}{\sqrt{p/c}} \times \frac{n}{\sqrt{p/c}}}$

if $c = 1$ **then**
  $A =$ BLOCK-CYCLIC-DC-APSP(A, n, p, c, b)
% *P*artition the vertices $V = (V_1, V_2)$ by partitioning the processor grid
Partition Π into 8 cubic block Π_{abc}, for $a, b, c \in \{1, 2\}$, where all Π_{abc} are $\sqrt{p/c}/2$-by-$\sqrt{p/c}/2$-by-$c/2$
% *e*xecute following loop iterations in parallel
for $k \leftarrow 1$ to $c/2$ **do**
  % *e*xecute following loop iterations in parallel
  for $i, j \leftarrow 1$ to $\sqrt{p}/2$ **do**
    % *F*ind all-pairs shortest paths between vertices in V_1
    $A_{ij} \leftarrow$ 2.5D-BLOCKED-DC-APSP$(A_{ij}, \Pi_{111}, n/2, p/8)$
    % *P*ropagate paths from V_1 to V_2
    $\Pi_{111}[i, j]$ sends A_{ij} to $\Pi_{121}[i, j]$.
    $A_{i,j+\sqrt{p/c}/2} \leftarrow$ 2.5D-SMMM$(A_{ij}, A_{i,j+\sqrt{p/c}/2}, A_{i,j+\sqrt{p/c}/2}, \Pi_{121}, n/2, p/8, c/2)$
    % *P*ropagate paths from V_2 to V_1
    $\Pi_{111}[i, j]$ sends A_{ij} to $\Pi_{211}[i, j]$.
    $A_{i+\sqrt{p}/2,j} \leftarrow$ 2.5D-SMMM$(A_{i+\sqrt{p}/2,j}, A_{ij}, A_{i+\sqrt{p}/2,j}, \Pi_{211}, n/2, p/8, c/2)$
    % *U*pdate paths to V_2 via paths from V_2 to V_1 and back to V_2
    $\Pi_{121}[i, j]$ sends $A_{i,j+\sqrt{p}/2}$ to $\Pi_{221}[i, j]$.
    $\Pi_{211}[i, j]$ sends $A_{i+\sqrt{p}/2,j}$ to $\Pi_{221}[i, j]$.
    $A_{i+\sqrt{p}/2,j+\sqrt{p}/2} \leftarrow$ 2.5D-SMMM$(A_{i+\sqrt{p}/2,j}, A_{i,j+\sqrt{p}/2}, A_{i+\sqrt{p}/2,j+\sqrt{p}/2}, \Pi_{221}, n/2, p/8, c/2)$
    % *F*ind all-pairs shortest paths between vertices in V_2
    $A_{i+\sqrt{p}/2,j+\sqrt{p}/2} \leftarrow$ 2.5D-BLOCKED-DC-APSP$(A_{i+\sqrt{p}/2,j+\sqrt{p}/2}, \Pi_{221}, n/2, p/8, c/2)$
    % *F*ind shortest paths paths from V_2 to V_1
    $\Pi_{221}[i, j]$ sends $A_{i+\sqrt{p}/2,j+\sqrt{p}/2}$ to $\Pi_{211}[i, j]$.
    $A_{i+\sqrt{p}/2,j} \leftarrow$ 2.5D-SMMM$(A_{i+\sqrt{p}/2,j+\sqrt{p}/2}, A_{i+\sqrt{p}/2,j}, A_{i+\sqrt{p}/2,j}, \Pi_{211}, n/2, p/8, c/2)$
    % *F*ind all-pairs shortest paths for vertices in V_1
    $\Pi_{121}[i, j]$ sends $A_{i,j+\sqrt{p}/2}$ to $\Pi_{111}[i, j]$.
    $\Pi_{211}[i, j]$ sends $A_{i+\sqrt{p}/2,j}$ to $\Pi_{111}[i, j]$.
    $A_{ij} \leftarrow$ 2.5D-SMMM$(A_{i,j+\sqrt{p}/2}, A_{i+\sqrt{p}/2,j}, A_{ij}, \Pi_{111}, n/2, p/8, c/2)$

Ensure: process $\Pi[i, j, 1]$ owns a block of the APSP distance matrix, $A_{ij} \in \mathbb{S}^{\frac{n}{\sqrt{p/c}} \times \frac{n}{\sqrt{p/c}}}$

Algorithm 9.8.5 [C] = 2.5D-SMMM(A, B, C, Π, n, p, c)

Require: n-by-n matrices A, B, C, spread over $\sqrt{p/c}$-by-$\sqrt{p/c}$ processor grid $\Pi[:, :, 1]$.

```
% Do with each processor in parallel
for all i, j ∈ [1, √(p/c)], k ∈ [1, c] do
    Replicate A[i, j], B[i, j] on all layers Π[i, j, :]
    if k > 1 then Initialize C[:, :, k] = ∞
    % perform loop iterations in a pipelined fashion
    for t = (k − 1) · n/c to t = k · n/c do Replicate A[:, t] on columns of Λ[:, :]
        Replicate B[t, :] on rows of Λ[:, :]
        % Perform Semiring-Matrix-Matrix-Multiply
        C[:, :, k] := min(C[:, :], A[:, t] + B[t, :])
    C[:, :, 1] := min(C[:, :, :])
    % min-reduce C across layers
```

Ensure: n-by-n matrix $C = \min(C, A \cdot B)$, spread over $\sqrt{p/c}$-by-$\sqrt{p/c}$ processor grid $\Pi[:, :, 1]$.

Chapter 10

Distributed-Memory Tensor Contractions

In this chapter, we will consider numerical applications in quantum chemistry. Among the most common methods of quantum chemistry are many-body (QMB) methods, which attempt to explicitly solve the Schrödinger equation using a variety of models. The explicit treatment of electrons in molecules leads to a steep computational cost, which is nonetheless often of polynomial complexity, but with the benefit of systematic improvement achieved through appropriate elaborations of the models. The coupled-cluster (CC) family of methods [165, 17, 39] is currently the most popular QMB method in chemistry due to its high accuracy, polynomial time and space complexity, and systematic improvability. Coupled-cluster excels in the description of molecular systems due to its ability to accurately describe electron correlation – the dynamic effect of each electron on the others. In simpler (and hence cheaper) methods, electron correlation is either neglected in favor of an averaged interaction (as in self-consistent field theory [135, 130]) or approximated by an assumed functional form as in DFT [128], while correlation is treated explicitly in CC methods for pairs of electrons (CCSD), triples (CCSDT), and so on in a systematically improvable way. Additionally CC is rigorously size-extensive [17] and easily extensible to excited states [150], derivatives [138, 151], and properties [119].

This chapter focuses on the fundamental kernels of coupled-cluster calculations – tensor contractions – and documents a completely new set of parallel algorithms implemented as a distributed-memory tensor contraction library, the Cyclops Tensor Framework (CTF) [1]. CTF has enabled coupled-cluster with excitations of up to three electrons (CCSDT) to scale on state-of-the-art architectures while achieving a high degree of efficiency in computation, communication and storage. On a Cray XC30 supercomputer, CTF outperforms NWChem [31], the most commonly used distributed-memory coupled-cluster software, solving problems many times faster by virtue of better strong scaling, as well as achieving higher performance for large problems (weak scaling). We also demonstrate that the framework can maintain a high-fraction of the peak performance on thousands of nodes of both a Cray XC30 and a BlueGene/Q architecture.

This chapter is based on joint work with Devin Matthews, Jeff Hammond, and John F. Stanton [148].

[1] Cyclops Tensor Framework is publicly available under a BSD license: `https://github.com/solomonik/ctf`

Any tensor contraction can be performed via a series of index reorderings and matrix multiplications. Parallel matrix multiplication is a well-studied problem and existing algorithms are capable of achieving high efficiency for the problem on a variety of scales/architectures, as we demonstrate in Chapter 4. In the field of dense linear algebra, optimized numerical libraries have achieved success and popularity by exploiting the efficiency of primitives such as matrix multiplication to provide fast routines for matrix operations specified via a high-level interface. CTF raises the level of abstraction to provide contraction routines which employ library-optimized matrix multiplication calls to maintain efficiency and portability. In addition to contractions, CTF provides optimized distributed data transposes which can reorder tensor indices and extract subtensors. This capability allows CTF to dynamically make data-decomposition decisions, which maintain load-balance and communication efficiency throughout the execution of any given tensor contraction. Other libraries, such as NWChem, use dynamic load-balancing to deal with the irregular structure of symmetric tensor contractions. The high-dimensional blocking used in CTF permits the capability to exploit tensor symmetry to lower computational and/or memory costs of contractions whenever possible.

We demonstrate the generality of this framework in the Aquarius quantum chemistry program [2] via a concise interface that closely corresponds to Einstein notation, capable of performing arbitrary dimensional contractions on symmetric tensors. This interface is a domain specific language well-suited to theoretical chemists. To demonstrate correctness and performance we have implemented coupled-cluster methods with single, double, and triple excitations (CCSDT) using this infrastructure.

The rest of the chapter is organized as follows

- Section 10.1 reviews previous work on coupled-cluster frameworks and tensor contractions,
- Section 10.2 presents the tensor blocking and redistribution algorithms used inside CTF,
- Section 10.3 explains how CTF performs tensor contractions and arranges the data via folding and mapping techniques,
- Section 10.4 gives performance results for CCSD and CCSDT codes implemented on top of CTF,
- Section 10.5 discusses future directions for work on distributed-memory tensor frameworks.

10.1 Previous work

In this section, we provide an overview of existing applications and known algorithms for distributed memory CC and tensor contractions. We also discuss parallel numerical linear algebra algorithms which will serve as motivation and building blocks for the design of Cyclops Tensor Framework.

[2] A pre-alpha version of Aquarius is publicly available under the New BSD license: `https://github.com/devinamatthews/aquarius`

10.1.1 NWChem and TCE

NWChem [31] is a computational chemistry software package developed for massively parallel systems. NWChem includes an implementation of CC that uses tensor contractions, which are of interest in our analysis. We will detail the parallelization scheme used inside NWChem and use it as a basis of comparison for the Cyclops Tensor Framework design.

NWChem uses the Tensor Contraction Engine (TCE) [79, 19, 64], to automatically generate sequences of tensor contractions based on a diagrammatic representation of CC schemes. TCE attempts to form the most efficient sequence of contractions while minimizing memory usage of intermediates (computed tensors that are neither inputs nor outputs). We note that TCE or a similar framework can function with any distributed library which actually executes the contractions. Thus, TCE can be combined with Cyclops Tensor Framework since they are largely orthogonal components. However, the tuning decisions done by such a contraction-generation layer should be coupled with performance and memory usage models of the underlying contraction framework.

To parallelize and execute each individual contraction, NWChem employs the Global Arrays (GA) framework [120]. Global Arrays is a partitioned global-address space model (PGAS) [170] that allows processors to access data (via explicit function calls, e.g., Put, Get and Accumulate) that may be laid out physically on a different processor. Data movement within GA is performed via one-sided communication, thereby avoiding synchronization among communicating nodes, while accessing distributed data on-demand. NWChem performs different block tensor sub-contractions on all processors using GA as the underlying communication layer to satisfy dependencies and obtain the correct blocks. NWChem uses dynamic load balancing among the processors because the work associated with block-sparse representation of symmetric tensors within GA is not intrinsically balanced. Further, since GA does not explicitly manage contractions and data redistribution, the communication pattern resulting from one-sided accesses is often irregular. The dynamic load-balancer attempts to alleviate this problem, but assigns work without regard to locality or network topology. Cyclops Tensor Framework eliminates the scalability bottlenecks of load imbalance and irregular communication, by using a regular decomposition which employs a structured communication pattern especially well-suited for torus network architectures.

10.1.2 ACES III and SIAL

The ACES III package uses the SIAL framework [112, 50] for distributed memory tensor contractions in coupled-cluster theory. Like the NWChem TCE, SIAL uses tiling to extract parallelism from each tensor contraction. However, SIAL has a different runtime approach that does not require one-sided communication, but rather uses intermittent polling (between tile contractions) to respond to communication requests, which allows SIAL to be implemented using MPI two-sided communication.

10.1.3 MRCC

MRCC [96] is unique in its ability to perform arbitrary-order calculations for a variety of CC and related methods. Parallelism is enabled to a limited extent by either using a multi-threaded BLAS library or by parallel MPI features of the program. However, the scaling performance is severely limited due to highly unordered access of the data and excessive inter-node communication. MRCC is currently the only tenable solution for performing any type of CC calculation which takes into account quadruple and higher excitations. MRCC uses a string-based approach to tensor contractions which originated in the development of Full CI codes [99, 122]. In this method, the tensors are stored using a fully-packed representation, but must be partially unpacked in order for tensor contractions to be performed. The indices of the tensors are then represented by index "strings" that are pre-generated and then looped over to form the final product. The innermost loop contains a small matrix-vector multiply operation (the dimensions of this operation are necessarily small, and become relatively smaller with increasing level of excitation as this loop involves only a small number of the total indices). The performance of MRCC is hindered by its reliance on the matrix-vector multiply operation, which is memory-bandwidth bound. Other libraries, including NWChem and our approach, achieve better cache locality by leveraging matrix multiplication.

10.1.4 QChem

The QChem [141] quantum chemistry package employs libtensor [56], a general tensor contraction library for shared-memory architectures. The libtensor framework exploits spatial and permutational symmetry of tensors, and performs tensor blocking to achieve parallelization. However, libtensor does not yet provide support for distributed memory tensor contractions and redistributions, as done by CTF. The libtensor module in QChem also contains a tensor contraction Domain Specific Language (DSL) somewhat similar to our own. One of the main differences between the two DSLs is that in libtensor, the index symmetry of the output tensor is implicitly described by the symmetry of the inputs and any explicit antisymmetrization operations to be performed. In our DSL (the tensor contraction interface to CTF and Aquarius), it is the other way around in that the antisymmetrization operators are implicitly specified by the symmetry of the inputs and output. We feel that the latter approach gives a more convenient and compact representation of the desired operation, as well as a more robust one in that antisymmetrization operators must be specified for each contraction, while the output tensor structure must be specified only once. In addition, the use of repeated and common indices in the tensor index labels and operator overloading in our approach is more general and flexible than providing functions for each type of common tensor operation as in libtensor.

10.1.5 Other Tensor Contraction Libraries

There are other tensor contraction libraries, which are not part of standard chemistry packages previously mentioned in this thesis. One concurrently-developed distributed memory tensor library effort is given by Rajbahandri et al [134], and shares many similarities with our work. This library

employs similar matrix multiplication primitives (SUMMA and 2.5D algorithms) for distributed tensor contractions and mapping of data onto torus networks. A few of the important differences between our work and this framework are the overdecomposition and redistribution mechanisms which we provide. Recent studies have also demonstrated the efficacy of scheduling many different contractions simultaneously within coupled-cluster [105], an approach that is particularly useful for higher order coupled-cluster methods. Our work focuses on parallel algorithms for the execution of a single tensor contraction, leaving it for future work to integrate this single-contraction parallelization with a second layer of multi-contraction parallelism.

There have also been efforts for efficient implementation of tensor contractions for coupled-cluster which do not focus on distributed memory parallelism. Hanrath and Engels-Putzka [74] give a sequential framework which performs tensor transpositions to efficiently exploit threaded matrix multiplication primitives within tensor contractions. Parkhill and Head-Gordon [126] as well as Kats and Manby [97] give sequential implementations of sparse tensor contraction libraries, targeted at coupled-cluster methods which exploit spatial locality to evaluate a sparse set of interactions. Support for parallel sparse tensor contractions is not within the scope of this thesis, but is being pursued as an important direction of future work since many of the parallel algorithmic ideas discussed in this thesis extend to sparse tensors.

10.2 Algorithms for Tensor Blocking and Redistribution

CTF decomposes tensors into blocks which are cyclic sub-portions of the global tensor and assigns them to processors in a regular fashion. The partitioning is contraction-specific and tensor data is transferred between different distributions when necessary. Further, it is necessary to allow the user to modify, enter, and read the tensor data in a general fashion. In this section, we explain how the cyclic blocking scheme works and give data redistribution algorithms which are designed to shuffle the data around efficiently.

10.2.1 Cyclic Tensor Blocking

A blocked distribution implies each processor owns a contiguous piece of the original tensor. In a cyclic distribution, a cyclic phase defines the periodicity of the set of indices whose elements are owned by a single processor. For example, if a vector is distributed cyclically among 4 processors, each processor owns every fourth element of the vector. For a tensor of dimension d, we can define a set of cyclic phases $(p_1, \ldots, p_d)$, such that processor $P_{i_1,\ldots,i_d}$ owns all tensor elements whose index $(j_1, \ldots, j_d)$ satisfies

$$j_k \equiv i_k \bmod p_k$$

for all $k \in \{1, \ldots, d\}$ and where p_k gives the length of the processor grid in the $k-th$ dimension. A block-cyclic distribution generalizes blocked and cyclic distributions, by distributing contiguous blocks of any size b cyclically among processors. Cyclic decompositions are commonly used in parallel numerical linear algebra algorithms and frameworks such as ScaLAPACK (block-cyclic) [28] and Elemental (cyclic) [131]. Our method extends this decomposition to tensors.

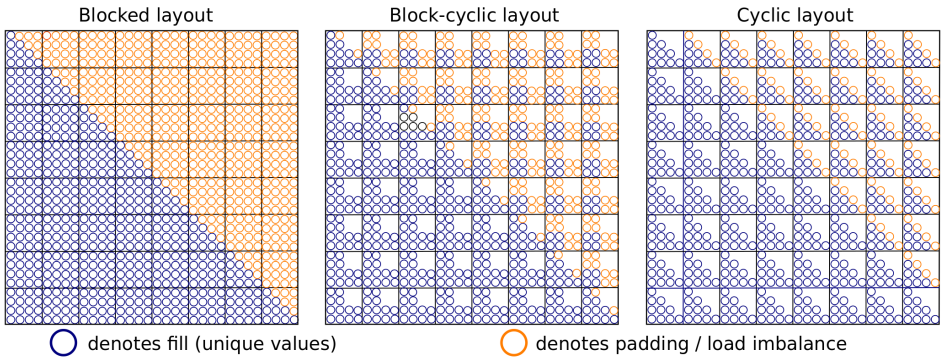


Figure 10.1: The load-imbalance incurred or padding necessary for blocked, block-cyclic, and cyclic layouts.

Cyclops Tensor Framework employs a cyclic distribution in order to preserve packed symmetric full structure in sub-tensors, minimize padding, and generate a completely regular decomposition, susceptible to classical linear algebra optimizations. Each processor owns a cyclic sub-tensor, where the choice of cyclic phases in each dimension has the same phase for all symmetric indices. By maintaining the same cyclic phase in each dimension, the algorithm ensures that each of the sub-tensors owned by any processor has the same fill structure as the whole tensor. For instance, CTF might decompose the integral tensor V which represents two-electron interactions via two basis functions and is therefore anti-symmetric in two index pairs, into 36 blocks $\hat{V}_{w_1w_2w_3w_4}$, where $w_1, w_2 \in \{0,1\}$ and $w_3, w_4 \in \{0,1,2\}$ so that

$$V_{ij}^{ab} \in \hat{V}_{w_1w_2w_3w_4} : V_{ij}^{ab} \in V, a \equiv w_1 \bmod 2,\ b \equiv w_2 \bmod 2, a < b$$
$$i \equiv w_3 \bmod 3,\ j \equiv w_4 \bmod 3, i < j.$$

Each sub-tensor $\hat{V}_{w_1w_2w_3w_4}$ has the same structure as the unique part of V (V_{ij}^{ab} for $a < b, i < j$), though some blocks have extra elements (e.g. $\hat{V}_{0101}$ is larger than $\hat{V}_{1101}$ since $V_{ij}^{aa} = 0$, which implies that the latter block does not contain entries that are on the diagonal of the first two indices). However, it is important to clarify that while structure is preserved, cyclicity does not preserve symmetry. If we drop the conditions $a < b$ and $i < j$ from the definition of the $\hat{V}_{w_1w_2w_3w_4}$ block, only the blocks $\hat{V}_{w_1w_1w_3w_3}$ will have the same symmetry as V (the same would be true if the blocks were selected contiguously rather than cyclically).

The cyclic blocks differ in fill, but padding along the diagonals of symmetric indices allows all the blocks to be defined in the same shape. Figure 10.1 demonstrates the padding required to store a lower-triangular matrix in a 4-by-4 cyclic layout. For comparison, we also show the amount of padding one would need to preserve identical block structure in block-cyclic and contiguously-blocked layouts. The cyclic layout is the only one able to preserve the fill structure and does not require a significant amount of padding. Preserving a regular layout, regardless of symmetry, also necessitates padding to make the tensor edge lengths divisible by the processor grid edge lengths. In a cyclic layout, the amount of padding required due to symmetry and divisibility grows with

the number of processors, presenting a potential limit to strong scaling (e.g. if the number of processors is equal to the number of tensor elements, cyclic and blocked layouts are the same and no structure is preserved). However, the relative padding overhead decreases as we increase the tensor size, so we expect the cyclic layout to at least maintain good weak scalability.

10.2.2 Overdecomposition

Our cyclic decomposition allows us to preserve symmetric fill structure so long as we satisfy the requirement that all symmetric tensor indices are decomposed with the same cyclic phase. This requirement presents a significant challenge to the naive method of assigning a single block to each processor, since for a n-dimensional symmetric index group, preserving symmetric fill structure would require the number of processors to have an integer nth root to have the same cyclic phase in each dimension. This restriction could be overcome by using a subset of the processors to do the computation, but this would sacrifice efficiency and generality. Our solution is to add a layer of abstraction between the decomposition and the machine by assigning multiple sub-tensors to each processor.

Cyclops Tensor Framework has the capability to overdecompose the tensor into a number of blocks that is a multiple of the number of processors. We can represent a d-dimensional torus processor grid as a d-tuple $(p_1, \dots p_d)$, where $p_d \in \{1, \dots\}$. In CTF, each n-dimensional tensor has n mappings $(m_1, \dots, m_n)$, where each mapping m_k consists of a processor grid dimension of an integer overdecomposition factor $v_k \in \{1, 2, \dots\}$ and a processor grid dimension index $q_k \in \{0, \dots, d\}$, so that $m_k = (v_k, q_k)$. We use the zeroth processor grid index to denote a completely virtual grid mapping, by letting $p_0 = 1$. This means that processor $P_{i_1,\dots,i_d}$ owns all tensor elements whose index $(j_1, \dots, j_n)$ satisfies

$$j_k \equiv i_k \bmod (v_k \cdot q_k)$$

for all $k \in \{1, \dots, n\}$. This overdecomposition capability allows tensors of order n to be mapped onto processor grids of order $d \leq n$, since any mapping $m_k = (v_k, 0)$ does not correspond to an actual processor grid dimension but only a local blocking. We also support mappings of tensors onto processor grids of dimension $d > n$ either by replicating the tensor along some of the processor grid dimensions, or by folding multiple processor grid dimensions into a single larger dimension.

CTF attempts to use the least amount of overdecomposition, since larger blocks typically achieve higher matrix multiplication efficiency. The overdecomposition factor is then set to be equal to the least common multiple (lcm) of the physical dimensions to which indices in the symmetric group are mapped. For instance if a tensor has a 3-index symmetry with the first two indices mapped along processor grid dimensions p_1 and p_2, we will have the three indices mapped as

$$\begin{aligned} m_1 &= (\text{lcm}(p_1, p_2)/p_1, 1) \\ m_2 &= (\text{lcm}(p_1, p_2)/p_2, 2) \\ m_3 &= (\text{lcm}(p_1, p_2), 0). \end{aligned}$$

We note that the overall phase of these mappings is the same, so the symmetric fill structure of the index group is preserved within each sub-tensor block. Further, we see that overdecomposition allows a tensor to be distributed among any number of processors (in this case 6), whereas a naive decomposition would have required a processor count of $\{1, 8, 27, \ldots\}$, and would have very limited capability for topology-aware mapping.

We do not use a dynamically scheduled overdecomposition approach such as that of the Charm++ runtime system [95]. Instead, our overdecomposition is set so that the dimensions of the cyclic decomposition are a multiple of the physical torus dimensions (by the factors v as above) and generate a regular mapping. For dense tensors, our approach maintains perfect load-balance and achieves high communication and task granularity by managing each overdecomposed sub-grid of blocks explicitly within each processor. However, we are exploring the utility of dynamic scheduling in allowing CTF to efficiently support sparse tensors.

10.2.3 Redistribution of Data

As we have seen with overdecomposition, the symmetries of a tensor place requirements on the cyclic decomposition of the tensor. Further parallel decomposition requirements arise when the tensor participates in a summation or contraction with other tensors, in order to properly match up the index mappings among the tensors. Therefore, it is often necessary to change the mapping of the tensor data between each contraction. In general, we want to efficiently transform the data between any two given index to processor grid mappings.

Further, redistribution of data is necessary if the framework user wants to read or write parts of the tensor. We enable a user to read/write to any set of global indices from any processor in a bulk synchronous sparse read/write call. Additionally, CTF allows the user to extract any sub-block from a tensor and define it as a new CTF tensor. CTF has three redistribution kernels of varying generality and efficiency, all of which are used in the execution of coupled-cluster calculations. The functionality of each kernel is summarized as follows (the kernels are listed in order of decreasing generality and increasing execution speed)

- sparse redistribution – write or read a sparse set of data to or from a dense mapped CTF tensor
- dense redistribution – shuffles the data of a dense tensor from one CTF mapping to any other CTF mapping
- block redistribution – shuffling the assignment of the tensor blocks to processors without changing the ordering of data within tensor blocks

Of these three kernels, the second is the most complex, the most optimized, and is used the most intensively during CCSD/CCSDT calculations. The third kernel serves as a faster version of dense redistribution. The sparse redistribution kernel currently serves for input and output of tensor data, but in the future could also be used internally to execute sparse tensor contractions. We detail the architecture of each kernel below.

Sparse Redistribution

There are multiple reasons for CTF to support sparse data redistribution, including data input and output, verification of the dense redistribution kernel, and support for sparse tensor storage. Our algorithm for reading and writing sparse data operates on a set of key-value pairs to read/write. The key is the global data index in the tensor. For a tensor of dimension d and edge lengths $(l_1, \ldots, l_d)$, the value at the position described by the tuple $(i_1, \ldots, i_d),\ 0 \leq i_k < l_k\ :\ \forall k \in [1, d]$ is given by

$$i_{global} = \sum_{k=1}^{d} \left(i_k \prod_{m=1}^{k-1} l_m \right).$$

The sparse algorithm is as follows:

1. sort the keys by global index
2. place the local key/value pairs into bins based on the indices' block in the final distribution
3. collect the bins for each block into bins for each destination processor
4. exchange keys among processors via all-to-all communication
5. combine the received keys for each block and sort them by key (the data is in global order within a block, but not across multiple blocks)
6. iterate over the local data of the target tensor, computing the key of each piece of data and performing a read or write if the next sparse key is the same
7. if the operation is a read, send the requested values back to the processors which requested them

This sparse read/write algorithm can be employed by a kernel which goes between two mappings of a tensor by iterating over local data and turning it into sparse format (key-value pairs), then calling a sparse write on a zero tensor in the new mapping. Further, by extracting only the values present in the sparse block of a tensor and changing the offsets of the keys, we allow general sub-block to sub-block redistribution.

This redistribution kernel is very general and fairly straightforward to implement and thread, however, it clearly does redundant work and communication for a redistribution between two mappings due to the formation, sorting, and communication of the keys. If the amount of data stored locally on each processor is n, this algorithm requires $O(n \log n)$ local work and memory traffic for the sort. The next kernel we detail is specialized to perform this task in a more efficient manner.

Dense Redistribution

Between contractions, it is often the case that a CTF tensor must migrate from one mapping to another. These mappings can be defined on processor grids of different dimension and can have

different overall cyclic phases along each dimension. This change implies that the padding of the data also changes, and it becomes necessary to move only the non-padded data. So, we must iterate over the local data, ignoring the padding, and send each piece of data to its new destination processor, then recover the data on the destination processor and write it back to the new local blocks in the proper order.

Our dense redistribution kernel utilizes the global ordering of the tensor to avoid forming or communicating keys with the index of data values. The kernel places the values into the send buffers in global order, performs the all-to-all then retrieves them from the received buffers using the knowledge that the data is in global order and thereby computing which processor sent each value. The same code with swapped parameters is used to compute the destination processor from the sender side, as well as to compute the sender processor from the destination side. The kernel is threaded by partitioning the tensor data among threads according to global order. We ensure a load-balanced decomposition of work between threads, by partitioning the global tensor in equal chunks according to its symmetric-packed layout, and having each thread work on the local part of this global partition, which is now balanced because the layout is cyclic. Another important optimization to the kernel, which significantly lowered the integer-arithmetic cost, was precomputing the destination blocks along each dimension of the tensor. For instance, to redistribute an n-by-n matrix A from a p_r-by-p_c grid to another distribution, we precompute two destination vectors v and w of size n/p_r and n/p_c, respectively, on each processor, which allow us to quickly determine the destination of local matrix element A_{ij} via look ups to v_i and w_j.

The code necessary to implement this kernel is complex because it requires iterating over local tensor blocks data in global order, which requires striding over blocks, making the index arithmetic complex and the access pattern non-local. However, overall the algorithm performs much less integer arithmetic than the sparse redistribution kernel, performing $O(n)$ work and requiring $O(n)$ memory reads (although with potentially with less locality than sorting), if the local data size is n. Further, the all-to-all communication required for this kernel does not need to communicate keys along with the data like in the sparse redistribution kernel. In practice, we found the execution time to be roughly 10X faster than the sparse kernel. Nevertheless, these dense redistributions consume a noticeable fraction of the execution time of CTF during most CC calculations we have tested.

Block Redistribution

Often, it is necessary to perform a redistribution that interchanges the mapping assignments of tensor dimensions. For instance, if we would like to symmetrize a tensor, we need to add the tensor to itself with two indices transposed. We also perform a similar addition to 'desymmetrize' a tensor by defining a nonsymmetric tensor object and adding the original tensor object, which is in packed storage to the unpacked nonsymmetric tensor object (although the data is still symmetric, but now redundantly stored). In a distributed memory layout, this requires a distribution if either of the two indices are mapped onto a processor grid dimension. If the tensor was in a mapping where the two indices had the same cyclic phase (it had to if the indices were symmetric), it suffices to redistribute the blocks of the tensor. With this use-case in mind, we wrote a specialized kernel that goes from any pair of mappings with the same overall cyclic phases and therefore the same blocks.

We implemented this type of block redistribution by placing asynchronous receive calls for each block on each processor, sending all the blocks from each processor, and waiting for all the exchanges to finish. Since blocks are stored contiguously they are already serialized, so this kernel requires no local computational work and has a low memory-bandwidth cost. However, this kernel still incurs the network communication cost of an all-to-all data exchange. This block redistribution kernel is used whenever possible in place of the dense redistribution kernel.

10.3 Algorithms for Tensor Contraction

In the previous section we've established the mechanics of how CTF manages tensor data redistribution among mappings. This section will focus on detailing the algorithms CTF uses to select mappings for contractions and to perform the tensor contraction computation. We start by discussing what happens to tensor symmetries during a contraction and how symmetry may be exploited. Then we detail how the intra-node contraction is performed and how the inter-node communication is staged. Lastly, we detail how full contraction mappings are automatically selected based on performance models of redistribution and distributed contraction.

10.3.1 Tensor Folding

Any nonsymmetric tensor contraction of $A \in \mathbb{R}^{I_{a_1} \times \ldots \times I_{a_l}}$ and $B \in \mathbb{R}^{I_{b_1} \times \ldots \times I_{b_m}}$ into $C \in \mathbb{R}^{I_{c_1} \times \ldots \times I_{c_n}}$ is an assignment of indices to each dimension of the three tensors, such that every index is assigned to two dimensions in different tensors. The total number of indices involved in the contraction is therefore $t = (l+m+n)/2$. We can define the assignment of these indices to the three tensors, by three index sets, $\{i_1, \ldots, i_l\}, \{j_1, \ldots, j_m\}, \{k_1, \ldots, k_n\} \subset \{1, \ldots, t\}$ which correspond to three projections of some index (tuple) $r = \{r_1, \ldots, r_t\}$:

$$p_A(r) = \{r_{i_1}, \ldots, r_{i_l}\}, \quad p_B(r) = \{r_{j_1}, \ldots, r_{j_m}\}, \quad p_C(r) = \{r_{k_1}, \ldots, r_{k_n}\}.$$

We can define the full index space via these projections as $\{I_1, \ldots, I_t\}$, where

$$p_A(\{I_1, \ldots, I_t\}) = \{I_{a_1}, \ldots, I_{a_l}\} \qquad p_B(\{I_1, \ldots, I_t\}) = \{I_{b_1}, \ldots, I_{b_m}\}$$
$$p_C(\{I_1, \ldots, I_t\}) = \{I_{c_1}, \ldots, I_{c_n}\}.$$

These projection mappings (indices) also completely define the element-wise contraction over this complete index space $\{I_1, \ldots, I_t\}$,

$$\forall r \in \{1, \ldots, I_1\} \times \ldots \times \{1, \ldots, I_t\} \; : \; C_{p_C(r)} = C_{p_C(r)} + A_{p_A(r)} \cdot B_{p_B(r)}.$$

For example, matrix multiplication ($C = C + A \cdot B$) may be defined by mappings $p_A(r_1, r_2, r_3) = (r_1, r_3)$, $p_B(r_1, r_2, r_3) = (r_3, r_2)$ and $p_C(r_1, r_2, r_3) = (r_1, r_2)$. These projections are provided to the framework via the domain specific language in Section 3.1. The code is in fact somewhat more general than the definition of contractions we analyze here, as it allows the same index to appear in all three tensors (weigh operation) as well as only one of three tensors (reduction).

Nonsymmetric tensor contractions reduce to matrix multiplication via index folding. Index folding corresponds to transforming sets of indices into larger compound indices, and may necessitate transposition of indices. We define a folding of index set $s = \{i_1, \ldots, i_n\}$ into a single compound index q as a one-to-one map

$$f : \{1, \ldots, I_1\} \times \ldots \times \{1, \ldots, I_n\} \rightarrow \left\{1, \ldots, \prod_{i=1}^{n} I_i\right\},$$

for instance with $s = \{i, j, k\}$, we have $f(s) = i + I_1 \cdot (j-1) + I_1 \cdot I_2 \cdot (k-1)$. Given a two-way partition function $(\pi_1(s), \pi_2(s))$, and two index set foldings f and g, we define a matrix folding $m(s) = (f(\pi_1(s)), g(\pi_2(s)))$ as a folding of a tensor's indices into two disjoint compound indices. Therefore, $\bar{A} = m(A)$ with elements $\bar{A}_{m(s)} = A_s$.

Any contraction can be folded into matrix multiplication in the following manner, by defining mappings f_{AB}, f_{BC}, and f_{AC} and matrix foldings:

$$\begin{aligned} m_A(i_A) &= (f_{AC}(i_{AC}), f_{AB}(i_{AB})), \\ m_B(i_B) &= (f_{AB}(i_{AB}), f_{BC}(i_{BC})), \\ m_C(i_C) &= (f_{AC}(i_{AC}), f_{BC}(i_{BC})), \end{aligned}$$

where $i_X = p_X(i_E)$ for all $X \in \{A, B, C\}$ and $i_{XY} = p_X(i_E) \cap p_Y(i_E)$ for all $X, Y \in \{A, B, C\}$. Now the contraction may be computed as a matrix multiplication of $\bar{A} = m_A(A)$ with $\bar{B} = m_B(B)$ into $\bar{C} = m_C(C)$

$$\bar{C}_{ij} = \bar{C}_{ij} + \sum_k \bar{A}_{ik} \cdot \bar{B}_{kj}$$

Tensors can also have symmetry, we denote antisymmetric (skew-symmetric) index groups within a fully symmetric order n tensor T as

$$T_{[i_1,\ldots,i_j,\ldots,i_k,\ldots,i_n]} = -T_{[i_1,\ldots,i_k,\ldots,i_j,\ldots,i_n]}$$

for any $j, k \in [1, n]$ (the value is zero when $j = k$ and is not stored). We also employ this notation for partially-antisymmetric tensors that have multiple index groups in which the indices are symmetric, e.g. for a partially-antisymmetric order $n + m$ tensor W,

$$\begin{aligned} &W_{[i_1,\ldots,i_n],[j_1,\ldots j_m]} = -W_{[i_n,i_2,\ldots,i_{n-1},i_1],[j_1,\ldots,j_m]} \\ &= -W_{[i_1,\ldots,i_n],[j_m,j_2,\ldots,j_{m-1},j_1]} = W_{[i_n,i_2,\ldots,i_{n-1},i_1],[j_m,j_2,\ldots,j_{m-1},j_1]}, \end{aligned}$$

where we can also permute any other pairs of indices (rather than the first with the last) within the two antisymmetric index groups with the same effect on the sign of the value. For the purpose of this analysis, we will only treat antisymmetric tensors; for symmetric tensors the non-zero diagonals require more special consideration. We denote a packed (folded) antisymmetric compound

index as an onto map from a packed set of indices to a interval of size binomial in the tensor edge length

$$\hat{f}(i_1, \ldots, i_n) : \{1, \ldots, I\}^n \to \left\{1, \ldots, \binom{I}{n}\right\}.$$

So given a simple contraction of antisymmetric tensors, such as

$$C_{[i_1 \ldots, i_{k-s}], [i_{k-s+1} \ldots, i_m]} = \sum_{j_1 \ldots, j_s} A_{[i_1 \ldots, i_{k-s}], [j_1 \ldots, j_s]} \cdot B_{[j_1 \ldots, j_s], [i_{k-s+1} \ldots, i_m]},$$

we can compute it in packed antisymmetric layout via matrix foldings:

$$\begin{aligned} m_C(\{i_1, \ldots, i_m\}) &= (\hat{f}(i_1, \ldots, i_{k-s}), \hat{g}(i_{k-s+1}, \ldots, i_m)), \\ m_A(\{i_1, \ldots, i_{k-s}, j_1, \ldots, j_s\}) &= (\hat{f}(i_1, \ldots, i_{k-s}), \hat{h}(j_1, \ldots, j_s)), \\ m_B(\{j_1, \ldots, j_s, i_{k-s+1}, i_m\}) &= (\hat{h}(j_1, \ldots, j_s), \hat{g}(i_{k-s+1}, \ldots, i_m)). \end{aligned}$$

So that, for $\bar{A} = m_A(A)$, we can rewrite the unfolded contraction and apply the foldings $\bar{B} = m_B(B)$ and $\bar{C} = m_C(C)$,

$$\begin{aligned} \forall \{i_1 \ldots, i_{k-s}\} \in \{1, \ldots, I\}^{k-s}, \{i_{k-s+1} \ldots, i_m\} \in \{1, \ldots, I\}^{m-k+s}, \\ \{j_1 \ldots, j_s\} \in \{1, \ldots, I\}^s \ : \\ \bar{C}_{m_C(\{i_1, \ldots, i_m\})} = \bar{C}_{m_C(\{i_1, \ldots, i_m\})} + \bar{A}_{m_A(\{i_1, \ldots, i_{k-s}, j_1, \ldots, j_s\})} \cdot \bar{B}_{m_B(\{j_1, \ldots, j_s, i_{k-s+1}, i_m\})} \end{aligned}$$

which yields the following folded form of the contraction

$$\forall i \in \left\{1, \ldots, \binom{I}{k-s}\right\}, j \in \left\{1, \ldots, \binom{I}{m-k+s}\right\} \ : \ \bar{C}_{ij} = s! \sum_{k=1}^{\binom{I}{s}} \bar{A}_{ik} \cdot \bar{B}_{kj}.$$

The above contraction is an example where all symmetries or antisymmetries are *preserved*. Any preserved symmetries must be symmetries of each tensor within the whole contraction term. We can consider a tensor Z corresponding to the uncontracted set of scalar multiplications whose entries are 3-tuples associated with scalar multiplications and the entry of the output to which the multiplicaiton is accumulated. For instance for a contraction expressed as

$$C_{i_1, \ldots, i_s, i_{s+t+1}, \ldots i_{s+t+v}} = \sum_{i_{s+1}, \ldots i_{s+v}} A_{i_1, \ldots, i_{s+v}} \cdot B_{i_{s+1}, \ldots, i_{s+t+v}}$$

the entries of Z are defined as

$$Z_{i_1, \ldots, i_{s+t+v}} = (A_{i_1, \ldots, i_{s+v}}, B_{i_{s+1}, \ldots, i_{s+t+v}}, C_{i_1, \ldots, i_s, i_{s+t+1}, \ldots i_{s+t+v}}).$$

If Z is symmetric (or antisymmetric) in a pair of indices, we say this symmetry (or antisymmetry) is preserved in the contraction. *Broken* symmetries are symmetries which exist in one of A, B,

or C, but not in Z (symmetries can exist in C and not in Z if the contraction is symmetrized or antisymmetrized). For example, we can consider the contraction

$$C_{[ij]kl} = \sum_{pq} A_{[ij][pq]} \cdot B_{pk[ql]},$$

the tensor Z would in this case be

$$Z_{[ij]klpq} = (A_{[ij][pq]}, B_{pk[ql]}, C_{[ij]kl}).$$

The symmetry $[ij]$ is preserved but the symmetries $[pq]$ and $[ql]$ are broken, since the three values in the 3-tuple $Z_{[ij]klpq}$ are unchanged (or change sign if antisymmetric) when i is permuted with j, but the value of B changes when p is permuted with q, and the value of A as well C change when q is permuted with l. For each preserved symmetry in a contraction we can achieve a reduction in floating point operations via folding. For broken symmetries we currently only exploit preservation of storage, although Chapter 11 introduces a method for lowering the computation cost in the case of broken symmetries via computing an alternative set of intermediate quantities. Within the framework the broken symmetries are unpacked and the contraction computed as

$$\bar{C}_{\hat{f}(i,j),k,l} = \sum_{p,q} \bar{A}_{\hat{f}(i,j),p,q} \cdot B_{p,k,q,l}$$

or the broken symmetries can remain folded, in which case multiple permutations are required,

$$\begin{aligned} \bar{C}_{\hat{f}(i,j),k,l} &= \sum_{p<q} \left[\bar{A}_{\hat{f}(i,j),\hat{g}(p,q)} \cdot \bar{B}_{p,k,\hat{h}(q,l)} - \bar{A}_{\hat{f}(i,j),\hat{g}(p,q)} \cdot \bar{B}_{p,k,\hat{h}(l,q)} \right] \\ &- \sum_{q<p} \left[\bar{A}_{\hat{f}(i,j),\hat{g}(q,p)} \cdot \bar{B}_{p,k,\hat{h}(q,l)} - \bar{A}_{\hat{f}(i,j),\hat{g}(q,p)} \cdot \bar{B}_{p,k,\hat{h}(l,q)} \right] \end{aligned}$$

Our framework makes dynamic decisions to unpack broken symmetries in tensors or to perform the packed contraction permutations, based on the amount of memory available. Unpacking each pair of indices is done via a tensor summation which requires either a local transpose of data of a global transpose with inter-processor communication, which can usually be done via a block-wise redistribution which was detailed in Section 10.2.3. It may be possible to efficiently reduce or avoid the desymmetrization and symmetrization calls necessary to deal with broken symmetries and we are currently exploring possible new algorithms that can achieve this. However, currently unpacking leads to highest efficiency due to the ability to fold the unpacked indices into the matrix multiplication call.

10.3.2 On-Node Contraction of Tensors

To prepare a folded form for the on-node contractions, we perform non-symmetric transposes of each tensor block. In particular, we want to separate out all indices which take part in a broken

symmetric group within the contraction and linearize the rest of the indices. If a symmetry group is not broken, we can simply fold the symmetric indices into one bigger dimension linearizing the packed layout. We perform an ordering transposition on the local tensor data to make dimensions which can be folded, as fastest increasing and the broken symmetric dimensions as slowest increasing within the tensors. To do a sequential contraction, we can then iterate over the broken symmetric indices (or unpack the symmetry) and call matrix multiplication over the linearized indices. For instance, the contraction from above,

$$C_{[ij]kl} = \sum_{pq} A_{[ij][pq]} \cdot B_{pk[ql]}$$

would be done as a single matrix multiplication for each block, if the $[pq]$ and $[ql]$ symmetries are unpacked. However, if all the broken symmetries are kept folded, the non-symmetric transpose would make fastest increasing the folded index corresponding to $\hat{f}(i,j)$, as well as the k index, so that the sequential kernel could iterate over p, q, l and call a vector outer-product operation over $\hat{f}(i,j), k$ for each p, q, l. The underlying call is generally a matrix multiplication, though in this case it reduces to an outer-product of two vectors, and in other cases could reduce to a vector or scalar operation. No matter what the mapping of the tensor is, the non-symmetric transpose required to fold the tensor blocks can be done locally within each block and requires no network communication. Therefore, while the non-symmetric permutations present an overhead to the sequential performance, their cost decreased comparatively as we perform strong or weak scaling.

10.3.3 Distributed Contraction of Tensors

In CTF, tensors are distributed in cyclic layouts over a virtual torus topology. The distributed algorithm for tensor contractions used by CTF is a combination of replication, as done in 3D and 2.5D algorithms [146] and a nested SUMMA [1, 163] algorithm (detailed in Chapter 4). If the dimensions of two tensors with the same contraction index are mapped onto different torus dimensions, a SUMMA algorithm is done on the plane defined by the two torus dimensions. For each pair of indices mapped in this way, a nested level of SUMMA is done. The indices must be mapped with the same cyclic phase so that the blocks match, adding an extra mapping restriction which necessitates overdecomposition.

We employ three versions of SUMMA, each one communicates a different pair of the the three tensors and keeps one tensor in place. The operand tensors are broadcast before each subcontraction, while the output tensor is reduced after each subcontraction. So, in one version of SUMMA (the outer-product version) two broadcasts are done (one of each operand tensor block) and in the other two of the three versions of SUMMA one reduction is done (instead of one of the broadcasts) to accumulate partial sums to blocks of the output. If the processor dimensions onto which the pair of indices corresponding to a SUMMA call are mapped are of different lengths, the indices must be overdecomposed to the same cyclic phase, and our SUMMA implementation does as many communication steps as the cyclic phase of the indices.

Blocking the computation on 3D grids (replicating over one dimension) allows for algorithms which achieve better communication efficiency in the strong scaling limit. Strong scaling efficiency

is necessary for higher order CC methods which require many different small contractions to be computed rapidly. When a small contraction is computed on a large number of processors there is typically much more memory is available than the amount necessary to store the tensor operands and output. The additional memory can be exploited via replication of tensor data and reduction of overall communication volume. We always replicate the smallest one of the three tensors involved in the contraction to minimize the amount of memory and communication overhead of replication.

10.3.4 Topology-Aware Network Mapping

Each contraction can place unique restrictions on the mapping of the tensors. In particular, our decomposition needs all symmetric tensor dimensions to be mapped with the same cyclic phase. Further, we must satisfy special considerations for each contraction, that can be defined in terms of indices (we will call them paired tensor dimensions) which are shared by a pair of tensors in the contraction. These considerations are

1. dimensions which are paired must be mapped with the same phase
2. for the paired tensor dimensions which are mapped to different dimensions of the processor grid (are mismatched)
    a) the mappings of two pairs of mismatched dimensions cannot share dimensions of the processor grid
    b) the subspace formed by the mappings of the mismatched paired dimensions must span all input data

The physical network topology is a d-dimensional toroidal partition specified by the d-tuple $(p_1, \ldots, p_d)$. CTF considers all foldings of physical topology that preserve the global ordering of the processors (e.g. p_2 may be folded with p_3 but not only with p_4). If the physical network topology is not specified as a torus, CTF factorizes the number of processes up to 8 factors and treats the physical topology as an 8-dimensional processor grid and attempts to map to all lower-dimensional foldings of it. When there are more tensor indices than processor grid dimensions, additional grid dimensions of length 1 are added.

For any given topology, CTF attempts to map all three possible pairs of tensors so that their indices are mismatched on the processor grid and they are communicated in the contraction kernel. The mapping of two tensors automatically defines the mapping of the third, though the overdecomposition factor must be adjusted to satisfy all symmetries and matching indices among tensors. CTF also considers partial replication of tensors among the processor grid dimensions. Figure 10.2 shows an example of an overdecomposed mapping with a properly matched cyclic phase along all tensor dimensions.

The search through mappings is done entirely in parallel among processors, then the best mapping is selected across all processors. The mapping logic is done without reading or moving any of the tensor data and is generally composed of integer logic that executes in an insignificant amount of time with respect to the contraction. We construct a 'ghost' mapping for each valid topology

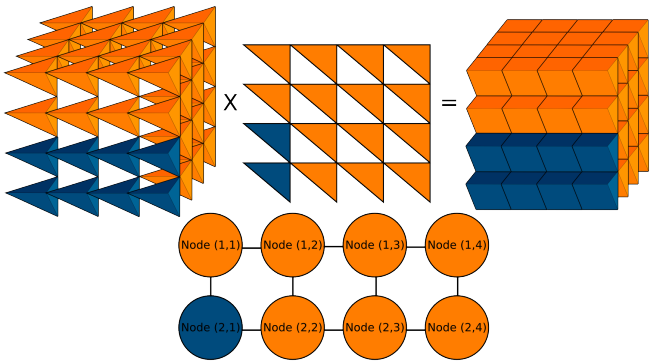


Figure 10.2: Overdecomposition as used in CTF to perform contractions. This diagram demonstrates a mapping for a contraction of the form $c_{[kl]i} = \sum_j a_{[jkl]} \cdot b_{[ij]}$. In this case, we have a 4-by-2 processor grid, and a 4-by-4-by-4 set of blocks.

and each ordering of tensors. The distributed contraction algorithm is constructed on each ghost mapping, and its communication and memory overheads are evaluated. If the ghost mapping is suboptimal it is thrown out without ever dictating data movement. Once a mapping is decided upon, the tensors are redistributed. Currently, our performance cost models predominantly consider communication, with controls on the amount of overdecomposition and memory usage. We are seeking to extend this cost model and to add infrastructure for training the parameters of our performance models.

10.4 Application Performance

The implementation of CTF employs MPI [71] for interprocessor communication, BLAS for matrix multiplication and summation, as well as OpenMP for threading. All other functionalities in CTF were developed from scratch and have no other dependencies. We used vendor provided optimized on-node parallel BLAS implementations (IBM ESSL and Cray LibSci) on all architectures for benchmarking. While the interface is C++, much of the internals of CTF are in C-style with C++ library support. Outside of special network topology considerations on BlueGene/Q, CTF does not employ any optimizations which are specific to an architecture or an instruction set. Performance profiling is done by hand and with TAU [142].

10.4.1 Architectures

Cyclops Tensor Framework targets massively parallel architectures and is designed to take advantage of network topologies and communication infrastructure that scale to millions of nodes. Parallel scalability on commodity clusters should benefit significantly from the load balanced characteristics of the workload, while high-end supercomputers will additionally benefit from reduced inter-processor communication which typically becomes a bottleneck only at very high degrees of

parallelism. We collected performance results on two state-of-the-art supercomputer architectures, IBM Blue Gene/Q and Cray XC30. We also tested sequential and multi-threaded performance on a Xeon desktop.

The sequential and non-parallel multi-threaded performance of CTF is compared to NWChem and MRCC. The platform is a commodity dual-socket quad-core Xeon E5620 system. On this machine, we used the sequential and threaded routines of the Intel Math Kernel Library. This platform, as well as the problem sizes tested reflect a typical situation for workloads on a workstation or small cluster, which is where the sequential performance of these codes is most important. Three problem sizes are timed, spanning a variety of ratios of the number of virtual orbitals to occupied orbitals.

The second experimental platform is 'Edison', a Cray XC30 supercomputer with two 12-core Intel "Ivy Bridge" processors at 2.4GHz per node (19.2 Gflops per core and 460.8 Gflops per node). Edison has a Cray Aries high-speed interconnect with Dragonfly topology ($0.25\mu s$ to $3.7\mu s$ MPI latency, 8 GB/sec MPI bandwidth). Each node has 64 GB of DDR3 1600 MHz memory (four 8 GB DIMMs per socket) and two shared 30 MB L3 caches (one per Ivy Bridge). Each core has its own L1 and L2 caches, of size 64 KB and 256 KB.

The final platform we consider is the IBM Blue Gene/Q (BG/Q) architecture. We use the installations at Argonne and Lawrence Livermore National Laboratories. On both installations, IBM ESSL was used for BLAS routines. BG/Q has a number of novel features, including a 5D torus interconnect and 16-core SMP processor with 4-way hardware multi-threading, transactional memory and L2-mediated atomic operations [75], all of which serve to enable high performance of the widely portable MPI/OpenMP programming model. The BG/Q cores run at 1.6 GHz and the QPX vector unit supports 4-way fused multiply-add for a single-node theoretical peak of 204.8 GF/s. The BG/Q torus interconnect provides 2 GB/s of theoretical peak bandwidth per link in each direction, with simultaneous communication along all 10 links achieving 35.4 GB/s for 1 MB messages [35].

10.4.2 Results

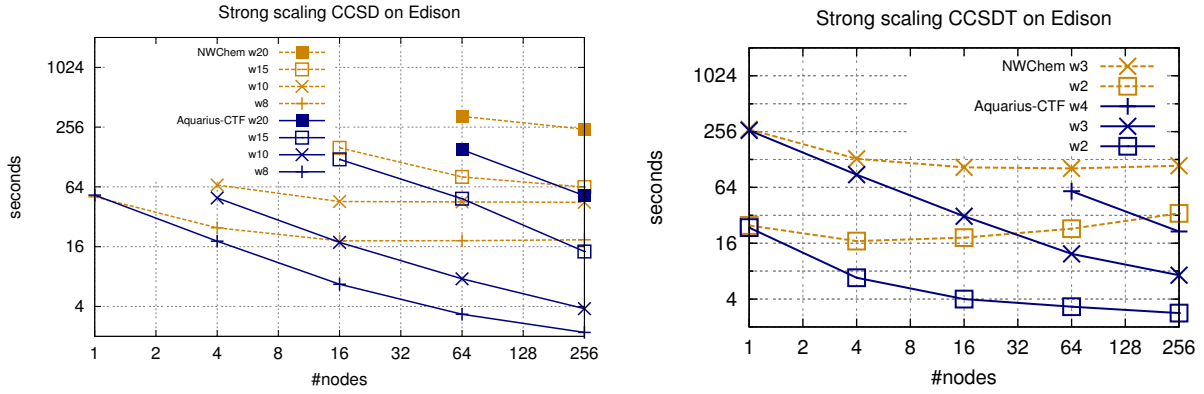
We present the performance of a CCSD and CCSDT implementation contained within Aquarius and executed using Cyclops Tensor Framework. For each contraction, written in one line of Aquarius code, CTF finds a topology-aware mapping of the tensors to the computer network and performs the necessary set of contractions on the packed structured tensors.

Sequential CCSD Performance

The results of the sequential and multi-threaded comparison are summarized in Table 10.4.1. The time per CCSD iteration is lowest for NWChem in all cases, and similarly highest for MRCC. The excessive iteration times for MRCC when the $\frac{n_v}{n_o}$ ratio becomes small reflect the fact that MRCC is largely memory-bound, as contractions are performed only with matrix-vector products. The multi-threaded speedup of CTF is significantly better than NWChem, most likely due to the lack of multi-threading of tensor transposition and other non-contraction operations in NWChem.

Table 10.4.1: Sequential and non-parallel multi-threaded performance comparison of CTF, NWChem, and MRCC. Entries are average time for one CCSD iteration, for the given number of virtual (n_v) and occupied (n_o) orbitals.

		$n_v = 110$ $n_o = 5$	$n_v = 94$ $n_o = 11$	$n_v = 71$ $n_o = 23$
NWChem	1 thread	6.80 sec	16.8 sec	49.1 sec
CTF	1 thread	23.6 sec	32.5 sec	59.8 sec
MRCC	1 thread	31.0 sec	66.2 sec	224. sec
NWChem	8 threads	5.21 sec	8.60 sec	18.1 sec
CTF	8 threads	9.12 sec	9.37 sec	18.5 sec
MRCC	8 threads	67.3 sec	64.3 sec	86.6 sec



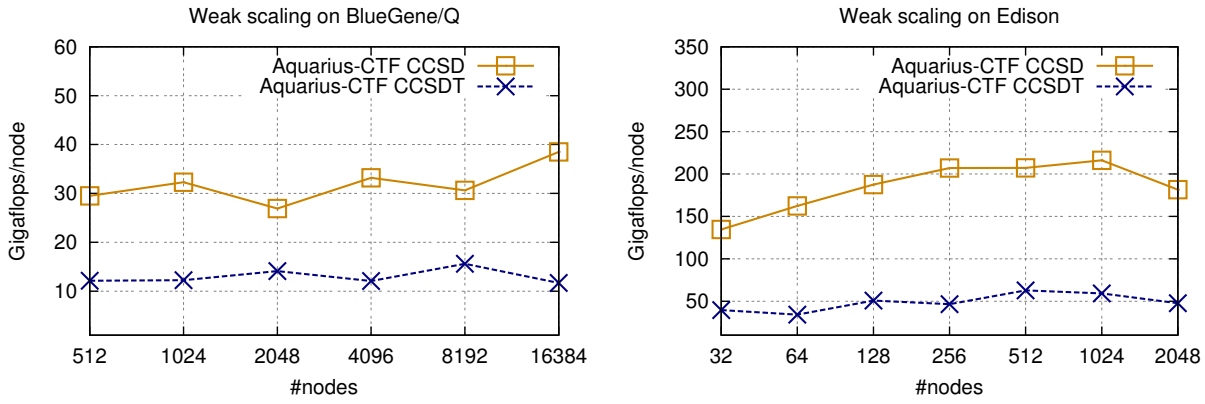
(a) CCSD (b) CCSDT

Figure 10.3: CCSD and CCSDT strong scaling of water clusters with cc-pVDZ basis set on Edison (Cray XC30).

Strong Scaling

On the Cray XC30 machine, we compared the performance of our CCSD implementation with that of NWChem. We benchmarked the two codes for a series of water systems – wn, where n is the number of water monomers in the system. Water clusters give a flexible benchmark while also representing an interesting chemical system for their role in describing bulk water properties. In Figure 10.3(a), we compare the scaling of CCSD using CTF with the scaling of NWChem. Our version of NWChem 6.3 used MPI-3 and was executed using one MPI process per core. The tile size was 40 for CCSD and 20 for CCSDT. The CCSD performance achieved by CTF becomes significantly higher than NWChem when the number of processors used for the calculation is increased. CTF can both solve problems in shorter time than NWChem (strong scale) and solve larger problems faster (weak scale). Figure 10.3(b) gives the CCSDT strong scaling comparison. CCSDT for system sizes above 3 molecules did not work successfully with our NWChem build,

but a large performance advantage for CTF is evident for the smaller systems.



(a) IBM BlueGene/Q (Mira)

(b) Cray XC30 (Edison)

Figure 10.4: CCSD and CCSDT weak scaling on water clusters with cc-pVDZ basis set.

Weak Scaling

The parallel weak scaling efficiency of our CCSD and CCSDT implementation on Blue Gene/Q is displayed in Figure 10.4(a) and on Cray XC30 is displayed in Figure 10.4(b). This weak scaling data was collected by doing the largest CCSD and CCSDT run that would fit in memory on each node count and normalizing the efficiency by the number of floating point operations performed by CTF. Going from 512 to 16384 nodes (256K cores), the efficiency actually often increases, since larger problems can be solved, which increases the ratio of computation over communication. For CCSD, CTF maintains 30 GF/node on BG/Q, which is about one sixth of peak. On Edison, CCSD surpasses 200 GF/node, which is over 50% of peak. The application was run with 4 MPI processes per node and 16 threads per process on BG/Q and with 4 MPI processes per node and 6 threads per process on Edison.

Table 10.4.2 lists profiling data for a CCSD iteration of CTF on 1024 nodes (16K cores) of BG/Q and 256 nodes (6K cores) of Edison on 25 water molecules with a basis set size of 600 virtual orbitals. This problem took 228 seconds on 1024 nodes of BG/Q and 167 seconds on 256 nodes of Edison. The table reports the percentage of execution time of a CCSD iteration spent in the main kernels in CTF. The table also lists the architectural bounds for each kernel, demonstrating the components of the hardware being utilized by each computation. The time inside matrix multiplication reflects the time spent working on tensor contractions sequentially on each processor, while broadcasts represent the time spent communicating data for replication and the nested SUMMA algorithm. The prefix sum, data packing, and all-to-all-v operations are all part of tensor redistribution, which is a noticeable overall overhead. Tensor folding corresponds to local transposition of each tensor block.

Table 10.4.2: A performance breakdown of important kernels for a CCSD iteration done by CTF on a system with $n_o = 125$ occupied orbitals and $n_v = 600$ virtual orbitals on 256 nodes of Edison (XC30) and 1024 nodes of Mira (BG/Q). Complexity is in terms of p processors and M memory per processor.

kernel	BG/Q	XC30	complexity	architectural bounds
matrix mult.	49%	35%	$O(n_v^4 n_o^2/p)$	flops/mem bandwidth
broadcasts	24%	37%	$O(n_v^4 n_o^2/p\sqrt{M})$	multicast bandwidth
prefix sum	10%	4%	$O(p)$	allreduce bandwidth
data packing	4%	3%	$O(n_v^2 n_o^2/p)$	integer ops
all-to-all-v	3%	10%	$O(n_v^2 n_o^2/p)$	bisection bandwidth
tensor folding	6%	5%	$O(n_v^2 n_o^2/p)$	memory bandwidth
other	4%	6%		

Table 10.4.3: A performance breakdown of important kernels for a CCSDT iteration done by CTF on a system with 8 water molecules ($n_o = 40$ occupied orbitals) and a cc-pVDZ basis set ($n_v = 192$ virtual orbitals) on 256 nodes of Edison (XC30) and 1024 nodes of Mira (BG/Q). Complexity is in terms of p processors and M memory per processor.

kernel	BG/Q	XC30	complexity	architectural bounds
matrix mult.	20%	16%	$O(n_v^5 n_o^3/p)$	flops/mem bandwidth
broadcasts	16%	11%	$O(n_v^5 n_o^3/p\sqrt{M})$	multicast bandwidth
prefix sum	5%	3%	$O(p)$	allreduce bandwidth
data packing	9%	9%	$O(n_v^3 n_o^3/p)$	integer ops
all-to-all-v	19%	20%	$O(n_v^3 n_o^3/p)$	bisection bandwidth
tensor folding	26%	39%	$O(n_v^3 n_o^3/p)$	memory bandwidth
other	5%	2%		

Table 10.4.2 yields the somewhat misleading initial observation that nearly half the execution is spent in matrix multiplication on BG/Q, 14% more than the fraction spent in matrix multiplication on Edison. In fact, we observe this difference for two reasons, because the BG/Q run uses four times the number of processors causing there to be more redundant computation on padding in the decomposition, and because the matrix multiplication library being employed on BG/Q achieves a significantly lower fraction of peak than on Edison for the invoked problem sizes. Overall, the computational efficiency of our CCSD implementation is more favorable on Edison than on BG/Q.

For CCSDT, Aquarius and CTF maintain a lower percentage of peak than for CCSD, but achieves good parallel scalability. The lower fraction of peak is expected due to the CCSDT increased relative cost and frequency of transpositions and summations with respect to CCSD. In particular CCSD performs $O(n^{3/2})$ computation with $O(n)$ data while CCSDT performs $O(n^{4/3})$ computation with $O(n)$ data. The third-order excitation tensor $\mathbf{T_3}$ present only in CCSDT has 6 dimensions and its manipulation (packing, unpacking, transposing, and folding) becomes an overhead. We observe this trend in the CCSDT performance profile on Table 10.4.3, where tensor

transposition and redistribution take up a larger fraction of the total time than in the CCSD computation considered in Table 10.4.2. The 8-water CCSDT problem in Table 10.4.3 took 15 minutes on 1024 nodes of BG/Q and 21 minutes on 256 nodes of Edison. The strong scalability achievable for this problem is significantly better on Edison, increasing the number of nodes by four, BG/Q performs such a CCSDT iteration (using 4096 nodes) in 9 minutes while Edison computes it (using 1024 nodes) in 6 minutes.

10.5 Future Work

CTF provides an efficient distributed-memory approach to dense tensor contractions. The infrastructure presented in the chaptermay be extended conceptually and is being improved in practice by more robust performance modelling and mapping logic. Another promising direction is the addition of inter-contraction parallelism, allowing for smaller contractions to be scheduled concurrently. CTF already supports working on subsets of nodes and moving data between multiple instances of the framework, providing the necessary infrastructure for a higher level scheduling environment.

From a broader applications standpoint the addition of sparsity support to CTF is a particularly attractive future direction. Working support for distributed memory tensor objects would allow for the expression of a wide set of numerical schemes based on sparse matrices. Further tensors with banded sparsity structure can be decomposed cyclically so as to preserve band structure in the same way CTF preserves symmetry. Tensors with arbitrary sparsity can also be decomposed cyclically, though the decomposition may need to perform load balancing in the mapping and execution logic.

Cyclops Tensor Framework will also be integrated with a higher-level tensor expression manipulation framework as well as CC code generation methods. We have shown a working implementation of CCSD and CCSDT on top of CTF, but aim to implement more complex methods. In particular, we are targeting the CCSDTQ method, which employs tensors of dimension up to 8 and gets the highest accuracy of any desirable CC method (excitations past quadruples have a negligible contribution).

Chapter 11

Contracting Symmetric Tensors Using Fewer Multiplications

How many scalar multiplications are necessary to multiply a n-by-n symmetric matrix by a vector? Commonly, n^2 scalar multiplications are performed, since the symmetrically equivalent entries of the matrix are multiplied by different vector entries. This chapter considers a method for such symmetric-matrix-by-vector multiplication that performs only $n^2/2$ scalar multiplications to leading order, but at the overhead of extra addition operations. In fact, we give an algorithm for any fully-symmetrized contraction of n-dimensional symmetric tensors of order $s+v$ and $v+t$ (in this chapter we refer to tensor dimension as the size of the range of each tensor index and the order as the number of its indices), yielding a result of order $s+t$ (with a total of $\omega = s+v+t$ indices) that uses $\binom{n+\omega-1}{\omega} \approx n^\omega/\omega!$ multiplications to leading order instead of the usual $n^\omega/(s! \cdot t! \cdot v!)$. This cost corresponds to exploiting symmetry in all ω indices involved in the contraction, while in the standard method, only the symmetry in index groups which are preserved (three groups of s, v, and t indices) in the contraction equation is exploited. Our alternative algorithm forms different intermediate quantities, which have fully-symmetric structure, by accumulating some redundant terms, which are then cancelled out via low-order computations.

The technique we introduce is reminiscent of Gauss's trick, which reduces the number of scalar multiplications from 4 to 3 for complex multiplication, but requires more additions. It is also similar to fast matrix multiplication algorithms such as Strassen's [152] and more closely to simultaneous matrix multiplication $A \cdot B$, $C \cdot D$ discussed by Victor Pan [123]. What all of these methods have in common is the use of algebraic reorganizations of the stated problem which form intermediate quantities with a preferable computational structure. The symmetric tensor contraction algorithms presented in this chapter similarly forms intermediate tensors which include mathematically redundant subterms in order to maintain symmetric structure in the overall computation.

The symmetric tensor contraction algorithm we derive lowers the number of multiplications necessary significantly, but requires more additions per multiplication and more data movement per computation performed. The algorithm is therefore most beneficial when each scalar multiplication costs more than an addition, which is the case when each tensor entry is a complex number or a matrix/tensor. This tensor contraction algorithm also has several special cases rele-

vant for numerical matrix computations, including some of the Basic Linear Algebra Subroutines (BLAS) [108]. In particular, `symv`, `symm`, `syr2`, and `syr2k` can be done with half the multiplications. Furthermore, in complex arithmetic, the Hermitian matrix BLAS routines `hemm`, `her2k` and LAPACK routines such as `hetrd` can be done in 3/4 of the arithmetic operations overall. However, these reformulated algorithms are unable to make use of matrix multiplication subroutines, although in certain cases, the new algorithms can achieve lower communication costs then the standard methods. Also, since the algorithms require multiple additions for each multiplication, they are may not be able to fully exploit fused multiply-add units. Further, while numerically stable, the reformulated algorithms may incur somewhat larger numerical error for certain inputs and have slightly weaker error bounds.

The fast symmetric tensor contraction algorithm also generalizes to the antisymmetric case (when the value of the tensor changes sign with interchange of its indices), which yields applications for tensor computations in quantum chemistry, especially coupled-cluster [16, 77], which is where the largest potential speedups of our approach can be found. Computation of the commutator of symmetric matrices or anticommutator of antisymmetric matrices (Lie and Jordan rings over respective symmetric matrix groups [164]), can be done using a factor of 6X fewer multiplications and 1.5X fewer total operations than the standard method. Frequently in coupled-cluster and other physical tensor computations, the operand tensors are only partially symmetric, in which case the new algorithm can be nested over each symmetric subset of indices with each scalar multiplication at the lowest level becoming a multiplication of matrices or some other contraction over the nonsymmetric subset of indices. In such partially symmetric cases, if the contraction over nonsymmetric indices (e.g. matrix multiplication) is more expensive than addition (e.g. matrix addition), the fast algorithm's reduction in the number of multiplications immediately yields the same reduction in the leading order operation count of the overall operation. We show that for three typical coupled-cluster contractions taken from methods of three different orders, our algorithm achieves 2X, 4X, and 9X improvements in arithmetic and communication cost with respect to the traditional approach.

The rest of the chapter is organized as follows:

- Section 11.1 introduces our notation and the symmetric tensor contraction problem,
- Section 11.2 details the standard and new algorithm for symmetric tensor contractions and gives special cases of the latter for vectors and matrices,
- Section 11.3 proves the correctness and stability of the fast tensor contraction algorithm and derives its computation and communication costs,
- Section 11.4 illustrates how the algorithm can be adapted to antisymmetric and Hermitian tensors
- Section 11.5 uses the antisymmetric adaptation to show how a few sample coupled-cluster contractions can be significantly accelerated,

- Section 11.6 presents some conclusions and future work on the topic of fast symmetric tensor contraction algorithms.

11.1 Symmetric Tensor Contractions

Our notation in this chapter follows that of Chapter 10 (which is partially in line with Kolda and Bader [101]), but with some additional constructs. We denote a d-tuple (tensor indices) as $i\langle d\rangle = (i_1, \dots i_d)$, with $i\langle d\rangle j\langle f\rangle = (i_1, \dots i_d, j_1, \dots j_f)$. We refer to the first $g < d$ elements of $i\langle d\rangle$ as $i\langle g\rangle$. We employ one-sized tuples $i_1 = i\langle 1\rangle$, $i\langle d\rangle j_m = (i_1, \dots i_d, j_m)$, as well as zero-sized tuples $j\langle 0\rangle = \emptyset$, $i\langle d\rangle j\langle 0\rangle = i\langle d\rangle$. Given an order d tensor A, we will refer to its elements using the notation $A_{i\langle d\rangle} = A_{i_1,\dots i_d}$. We refer to an index permutation $\pi \in \Pi^d$ as any bijection f between the set $S = [1, d]$ and itself, so $\pi(i\langle d\rangle) = j\langle d\rangle$ if for all $k \in [1, d]$, $j_{f(k)} = i_k$. Each tensor index tuple, $i\langle d\rangle$, has $d!$ reorderings, to which we refer to as

$$\Pi(i\langle d\rangle) = \{j\langle d\rangle : \exists \pi \in \Pi^d, \pi(i\langle d\rangle) = j\langle d\rangle\}.$$

We define the disjoint partition set $\chi^p_q(k\langle r\rangle)$ as the set of all tuples of size p and q, which are disjoint subsets of $k\langle r\rangle$ and preserve the ordering of elements in $k\langle r\rangle$, in other words if k_i and k_j appear in the same tuple (partition) and $i < j$, then k_i must appear before k_j. We can define the set $\chi^p_q(k\langle r\rangle)$ inductively,

- base case for $r < p + q$:

$$\chi^p_q(k\langle r\rangle) = \emptyset,$$

  for $r = 0$ and for $p = q = 0$:

$$\chi^p_q(k\langle r\rangle) = \{(\emptyset, \emptyset)\}.$$

- inductive case for $\chi^p_q(k\langle r\rangle)$:
  - if $p > 0$ and $q > 0$,
$$\begin{aligned}\chi^p_q(k\langle r\rangle) =& \{(i\langle r_1\rangle k_r, j\langle r_2\rangle) : \exists (i\langle r_1\rangle, j\langle r_2\rangle) \in \chi^{p-1}_q(k\langle r-1\rangle)\} \\ &\cup \{(i\langle r_1\rangle, j\langle r_2\rangle k_r) : \exists (i\langle r_1\rangle, j\langle r_2\rangle) \in \chi^p_{q-1}(k\langle r-1\rangle)\} \\ &\cup \{(i\langle r_1\rangle, j\langle r_2\rangle) \quad : \exists (i\langle r_1\rangle, j\langle r_2\rangle) \in \chi^p_q(k\langle r-1\rangle)\}\end{aligned}$$
  - if $p = 0$,
$$\begin{aligned}\chi^0_q(k\langle r\rangle) =& \{(\emptyset, j\langle q-1\rangle k_r) : \exists (\emptyset, j\langle q-1\rangle) \in \chi^0_{q-1}(k\langle r-1\rangle)\} \\ &\cup \{(\emptyset, j\langle q\rangle) \quad : \exists (\emptyset, j\langle q\rangle) \quad \in \chi^0_q(k\langle r-1\rangle)\}\end{aligned}$$
  - if $q = 0$,
$$\begin{aligned}\chi^p_0(k\langle r\rangle) =& \{(i\langle p-1\rangle k_r, \emptyset) : \exists (i\langle p-1\rangle, \emptyset) \in \chi^{p-1}_0(k\langle r-1\rangle)\} \\ &\cup \{(i\langle p\rangle, \emptyset) \quad : \exists (i\langle p\rangle, \emptyset) \quad \in \chi^p_0(k\langle r-1\rangle)\}\end{aligned}$$

We denote all possible ordered subsets as $\chi^d(k\langle d+f\rangle) = \{i\langle d\rangle | \exists (i\langle d\rangle, j\langle f\rangle) \in \chi^d_f(k\langle d+f\rangle)\}$. When it is implicitly clear what partition is needed, we omit the superscript and subscript from χ completely, for example $i\langle d\rangle \in \chi(k\langle d+f\rangle)$ is equivalent to $i\langle d\rangle \in \chi^d(k\langle d+f\rangle)$ and $(i\langle d\rangle, j\langle f\rangle) \in \chi(k\langle d+f\rangle)$ is equivalent to $(i\langle d\rangle, j\langle f\rangle) \in \chi^d_f(k\langle d+f\rangle)$.

We note that for any $d, f \geq 0$, the union of all concatenations of all possible permutations of the ordered subsets produced by χ_f^d is equal to all possible permutations of the partitioned subset,

$$\Pi(k\langle d+f\rangle) = \{i'\langle d\rangle j'\langle f\rangle : \exists (i\langle d\rangle, j\langle f\rangle) \in \chi(k\langle d+f\rangle), i'\langle d\rangle \in \Pi(i\langle d\rangle), j'\langle f\rangle \in \Pi(j\langle f\rangle)\}.$$

Definition: For any $s, t, v \geq 0$, we define a **symmetrized contraction** between tensors, $C = A \odot B$, where A is of order $s+v$ with elements in Abelian group $(R_A, +)$, B is of order $t+v$ with elements in Abelian group $(R_B, +)$, into C is of order $s+t$ with elements in Abelian group $(R_C, +)$ (all with dimension/edge-length n), via a distributive (not necessarily associative) operator " $\cdot$ " $\in R_A \times R_B \to R_C$, as

$$C = A \odot B \;\equiv\; \forall i\langle s+t\rangle,\; C_{i\langle s+t\rangle} = \frac{1}{(s+t)!} \cdot \sum_{j\langle s\rangle l\langle t\rangle \in \Pi(i\langle s+t\rangle)} \left(\sum_{k\langle v\rangle} A_{j\langle s\rangle k\langle v\rangle} \cdot B_{l\langle t\rangle k\langle v\rangle} \right). \tag{11.1.1}$$

In the case of $v = 0$, the sum over $k\langle v\rangle$ disappears (one multiplication is done instead of a sum of them). Until Section 11.5, we will mostly focus on the case where $R = R_A = R_B = R_C$, so all the tensor elements are in some (possibly nonassociative) ring [137] $(R, +, \cdot)$. We will also concentrate our analysis and proofs on cases where at least two of s, t, v are nonzero, since otherwise one of the tensors is a scalar and the problem is trivial. The algorithms we present reduce to an equivalent form and have the same cost for the cases when only one of s, t, v is nonzero. The resulting tensor C computed via equation 11.1.1 is symmetric. For multiplication of a matrix A with a vector b ($s = 1, t = 0, v = 1$), equation 11.1.1 becomes $c = A \cdot b$. For the rank-two vector outer product of vectors a and b ($s = 1, t = 1, v = 0$), it becomes $C = \frac{1}{2}(a \cdot b^T + b \cdot a^T)$ and for the matrix multiplication of A and B ($s = 1, t = 1, v = 1$), it becomes $C = \frac{1}{2}(A \cdot B^T + B \cdot A^T)$.

Definition: When A and B are symmetric, we call a symmetrized contraction between A and B, a **fully-symmetric contraction**. For such a contraction it is no longer necessary to compute all possible orderings of the indices $j\langle s\rangle$, $l\langle t\rangle$, since they all yield the same A and B values respectively. So, in this case we can rewrite equation 11.1.1 to sum only over the ordered partitions $\chi_t^s(i\langle s+t\rangle)$ and scale by a smaller prefactor,

$$C = A \odot B \;=\; \forall i\langle s+t\rangle,\; C_{i\langle s+t\rangle} = \frac{1}{\binom{s+t}{s}} \cdot \sum_{(j\langle s\rangle, l\langle t\rangle) \in \chi(i\langle s+t\rangle)} \left(\sum_{k\langle v\rangle} A_{j\langle s\rangle k\langle v\rangle} \cdot B_{l\langle t\rangle k\langle v\rangle} \right). \tag{11.1.2}$$

Equation 11.1.2 yields an equivalent result C as equation 11.1.1, when A and B are symmetric, since

$$\forall j'\langle s\rangle l'\langle t\rangle \in \Pi(i\langle s+t\rangle), \quad \exists (j\langle s\rangle, l\langle t\rangle) \in \chi(i\langle s+t\rangle) \text{ and } \exists \pi_s \in \Pi^s, \pi_t \in \Pi^t, \text{ such that } \pi_s(j\langle s\rangle)\pi_t(l\langle t\rangle) = j'\langle s\rangle l'\langle t\rangle.$$

Further, $\forall k\langle v\rangle$, $A_{j\langle s\rangle k\langle v\rangle} \cdot B_{l\langle t\rangle k\langle v\rangle} = A_{j'\langle s\rangle k\langle v\rangle} \cdot B_{l'\langle t\rangle k\langle v\rangle}$ when

$$\exists \pi_s \in \Pi^s, \pi_t \in \Pi^t, \text{ such that } \pi_s(j\langle s\rangle)\pi_t(l\langle t\rangle) = j'\langle s\rangle l'\langle t\rangle.$$

We denote $\omega = s + t + v$ and assume $n \gg \omega$. We note that the symmetrized contraction operator $\odot$ is commutative so long as the underlying scalar product is commutative, meaning that $A \odot B = B \odot A$. However, $\odot$ is not associative, and in order to nest $\odot$ over different symmetric sets of indices or a single index, we do not require "$\cdot$" to be associative. We will return to and elaborate on this when we consider partially-symmetric tensors in Section 11.5. As an example of a symmetric contraction, in the case when $s = t = v = 1$ and scalar multiplication, "$\cdot$", is commutative, A and B are symmetric n-by-n matrices and $C = \frac{1}{2}(A \cdot B + B \cdot A)$, in which "$\cdot$" means the usual matrix product. We consider more examples of this problem and algorithms for them in Section 11.2.4.

We will revisit these definitions and adapt them to the case of antisymmetric tensors and Hermitian tensors, defining antisymmetrized contractions, and Hermitian contractions in Section 11.4.

11.2 Algorithms

We now discuss algorithms for evaluating equation 11.1.2. The first algorithm exploits the symmetry which is preserved in the contraction, as is commonly done. The main contribution of the chapteris the second 'fast' symmetric tensor contraction algorithm, which manages to additionally exploit permutational symmetry, which is not preserved in equation 11.1.2.

11.2.1 Nonsymmetric Tensor Contraction Algorithm

We first consider the cost of contracting nonsymmetric tensors A and B with all dimensions equal to n, where A is of order $s + v$ with elements in Abelian group $(R_A, +)$, B is of order $t + v$ with elements in Abelian group $(R_B, +)$, into C is of order $s + t$ with elements in Abelian group $(R_C, +)$.

Algorithm $\Upsilon^{(\mathbf{s,t,v})}$: For some permutation of the indices of A and B any contraction with a distributive operator "$\cdot$" : $R_A \times R_B \to R_C$ can be written as

$$\forall i\langle s+t\rangle, \;\; C_{i\langle s\rangle j\langle t\rangle} = \sum_{k\langle v\rangle} A_{i\langle s\rangle k\langle v\rangle} \cdot B_{k\langle v\rangle j\langle t\rangle}.$$

This contraction may be computed by a single matrix multiplication with matrices of dimensions n^s-by-n^v and n^v-by-n^t. We call this nonsymmetric tensor contraction algorithm $\Upsilon^{(s,t,v)}$.

For any pair of operators $\varrho = (\cdot, +)$, we associate the computational cost μ_ϱ with the distributive operator of the elements of A and B "$\cdot$" (usually multiplication), and we the associate computational cost ν_ϱ^C with the Abelian group operator of the elements of C "$+$" (usually addition). We count the cost of $\Upsilon^{(s,t,v)}$ as equal to the cost of multiplying matrices of dimensions n^s-by-n^v and n^v-by-n^t into a matrix with dimensions n^s-by-n^t, which is

$$F^\Upsilon(n, s, t, v, \varrho) = n^\omega \cdot (\nu_\varrho^C + \mu_\varrho),$$

where $\omega = s + t + v$. Although, in practice, transposing the tensor indices from any given layout into a distribution which allows direct application of matrix multiplication is non-trivial.

Theorem 11.2.1. *The communication cost of $\Upsilon^{(s,t,v)}(A,B)$, given a sequential machine with a cache of size $\hat{M}$, i.e. the number of words moved between cache and memory, under the assumption that no operands start in cache and all outputs are written to memory and that all elements have unit element size has the following bounds,*

$$\frac{2n^\omega}{\sqrt{\hat{M}}} + n^{s+t} + n^{s+v} + n^{t+v} \geq \hat{W}^\Psi(n,s,t,v,\hat{M}) \geq \max\left[\frac{2n^\omega}{\sqrt{\hat{M}}}, n^{s+t} + n^{s+v} + n^{t+v}\right].$$

Proof. The upper bound on the cost may be attained by the classical algorithm that computes matrix multiplication block-by-block with blocks of each matrix of dimension $\sqrt{\hat{M}/3}$.

The lower bound is asymptotically the same as the classical result by Hong and Kung [90], but includes constant factors rather than just an asymptotic bound. A lower bound with a constant factor was also given by [12], but with a smaller constant prefactor. Our proof of the lower bound with this constant factor is in Section 4.2.

11.2.2 Standard Symmetric Contraction Algorithm

The nonsymmetric algorithm may be used to compute equation 11.1.2, with only an additional step of symmetrization of the result of the multiplication between A and B. However, the nonsymmetric algorithm ignores symmetry in A and B, some of which may easily be exploited.

Algorithm $\Psi_\odot^{(\mathbf{s},\mathbf{t},\mathbf{v})}$: Typically, such symmetric contractions are done by forming the partially-symmetric intermediate tensor,

$$\bar{C}_{j\langle s\rangle l\langle t\rangle} = \sum_{k\langle v\rangle} A_{j\langle s\rangle k\langle v\rangle} \cdot B_{l\langle t\rangle k\langle v\rangle}. \tag{11.2.1}$$

$\bar{C}$ is symmetric in the index groups $j\langle s\rangle$ and $l\langle t\rangle$. C may subsequently be obtained from $\bar{C}$ by symmetrization of the $j\langle s\rangle$ and $l\langle t\rangle$ index groups,

$$C_{i\langle s+t\rangle} = \frac{1}{\binom{s+t}{s}} \sum_{(j\langle s\rangle, l\langle t\rangle)\subset\chi(i\langle s+t\rangle)} \bar{C}_{j\langle s\rangle l\langle t\rangle}. \tag{11.2.2}$$

We denote this 'standard' algorithm as $C = \Psi_\odot^{(s,t,v)}(A,B)$.

The formation of the intermediate tensor $\bar{C}$ may be reduced to a single matrix multiplication via *index folding*. In particular, each of the three index groups $l\langle t\rangle$, $j\langle s\rangle$, and $k\langle v\rangle$ may be linearized (folded) into a single larger index of sizes $\binom{n+t-1}{t}$, $\binom{n+s-1}{s}$, and $\binom{n+v-1}{v}$, respectively, and the resulting matrices, $\bar{A}$ and $\bar{B}$ can be multiplied nonsymmetrically. For example when $s = t = v = 1$, no folding is necessary and the matrices A and B are simply multiplied to obtain $\bar{C} = A \cdot B$, then each entry of C is obtained via symmetrization: $C_{ij} = \frac{1}{2} \cdot (\bar{C}_{ij} + \bar{C}_{ji})$.

Contracting over a folded $k\langle v\rangle$ index group requires a scaling by a factor of $v!$ for all contributions to the output, except those on a diagonal e.g. with $k_1 = k_2$. Therefore, each of $\binom{n+s-1}{s}\binom{n+t-1}{t}$

entries of $\bar{C}$ requires $\binom{n+v-1}{v}$ scalar multiplications, for a total of

$$\binom{n+s-1}{s}\binom{n+t-1}{t}\binom{n+v-1}{v} = n^{s+t+v}/(s!t!v!) + O(n^{s+t+v-1})$$

multiplications and the same number of additions. We do not count the cost of scaling by $\frac{1}{\binom{s+t}{s}}$, because the scaling is not necessary in all applications and could differ in cost from the multiplication of entries of A and B depending on the operator. Symmetrization of $\bar{C}$ requires

$$\binom{s+t}{s}\binom{n+s+t-1}{s+t} = n^{s+t}/(s!t!) + O(n^{s+t-1})$$

additions. Assuming $s, t, v \geq 1$, the cost of symmetrization of the result is low-order, and the cost is dominated by forming the intermediate tensor, $\bar{C}$.

Exploiting preserved symmetry (performing $\Psi_{\odot}^{(s,t,v)}(A, B)$) has been common practice in quantum chemistry tensor contraction implementations for decades, for instance it is explicitly discussed in [77], although the technique was probably employed in earlier coupled-cluster and Møller-Plesset perturbation-theory codes. The consideration and exploitation of such preserved symmetry has also been automated for arbitrary-order tensors by the Tensor Contraction Engine (TCE) [79] and Cyclops Tensor Framework (discussed in Chapter 10).

For any pair of operators $\varrho = (\cdot, +)$, we associate the computational cost μ_ϱ with the distributive operator of the elements of A and B "$\cdot$" (usually multiplication), and we the associate computational cost ν_ϱ^C with the Abelian group operator of the elements of C "$+$" (usually addition). The computational cost of the algorithm $\Psi_{\odot}^{(s,t,v)}(A, B)$ is then

$$F^{\Psi}(n, s, t, v, \varrho) = \frac{n^\omega}{s!t!v!} \cdot (\nu_\varrho + \mu_\varrho) + \frac{n^{s+t}}{s!t!} \cdot \nu_\varrho + O(n^{\omega-1} \cdot (\nu_\varrho + \mu_\varrho)).$$

Theorem 11.2.2. *The communication cost of $\Psi_{\odot}^{(s,t,v)}(A, B)$, given a sequential machine with a cache of size $\hat{M}$, i.e. the number of words moved between cache and memory, under the assumption that no operands start in cache and all outputs are written to memory and that all elements have unit element size has the following bounds,*

$$\frac{2n^\omega}{s!t!v!\sqrt{\hat{M}}} + \frac{n^{s+t}}{s!t!} + \frac{n^{s+v}}{s!v!} + \frac{n^{t+v}}{t!v!} \geq \hat{W}^{\Psi}(n, s, t, v, \hat{M})$$
$$\geq \max\left[\min\left(\frac{s!v!}{(s+v)!}, \frac{t!v!}{(t+v)!}, \frac{s!t!}{(s+t)!}\right)^{3/2} \cdot \frac{2n^\omega}{s!t!v!\sqrt{\hat{M}}}, \frac{n^{s+t}}{(s+t)!} + \frac{n^{s+v}}{(s+v)!} + \frac{n^{t+v}}{(t+v)!}\right].$$

Proof. The bounds for the standard algorithm are similar to the naive algorithm, since both are usually dominated by a matrix multiplication, although the standard algorithm requires smaller matrices.

The upper bound on the cost may be attained by performing the multiplication via the classical algorithm that computes matrix multiplication block-by-block with blocks of each matrix of

dimension $\sqrt{\hat{M}/3}$ and accumulating all partial summations directly to their symmetrized destination in the packed represenation of the tensor C.

The lower bound is asymptotically the same as the classical result by Hong and Kung [90], but includes constant factors and is decreased due to the additional symmetry of A, B, and C. A lower bound with a constant factor was also given by [12], but with a smaller constant prefactor.

In the standard algorithm for symmetric tensor contractions, $C = \Psi_{\odot}^{(s,t,v)}(A, B)$ where C is of order $s + t$, A of order $s + v$, and B of order $t + v$. The algorithm treats the tensors as matrices with dimensions $\binom{n+s-1}{s}$, $\binom{n+t-1}{t}$, and $\binom{n+v-1}{v}$, but these reduced matrix forms have fewer unique entries than nonsymmetric matrices of the same dimension due to symmetry. The lower bound for reading in the inputs from memory to cache and writing the output from cache to memory is

$$\hat{W}^{\Psi}(n, s, t, v, \hat{M}) \geq \frac{n^{s+t}}{(s+t)!} + \frac{n^{s+v}}{(s+v)!} + \frac{n^{t+v}}{(t+v)!},$$

which is lower than the corresponding lower bound for nonsymmetric matrix multiplication.

When these tensors all fit into cache, the above lower bound is attainable since the unique entries of A, B can all be read into cache then contracted and symmetrized into C on the fly. This cost is a factor of $(s + t)!/(s!t!)$ smaller than the IO complexity of reading in a nonsymmetric (folded) $\bar{A}$ of dimensions $n^s/s!$ and $n^t/t!$, and similarly for the costs associated with $\bar{B}$ and $\bar{C}$. When the tensors do not fit in cache, the symmetry is more difficult to exploit in a general fashion so our algorithm (upper bound) does not attempt to. However, it is at least possible to lower the communication cost by exploiting symmetry in one tensor if the other two tensors are vectors. For instance in multiplying a symmetric-matrix by a vector, we could pick out blocks from the lower triangle of the matrix, and multiply them by the two sets of vector entries, which the block and its symmetric reflection are multiplied by, contributing to two sets of vector entries in the output. However, this does not generalize easily to the $s, t, v \geq 1$ case, where reading in extra subsets of B and C counteracts with the cache reuse achieved by reading in a larger subset of B and C.

We do not attempt to prove that schedules cannot achieve extra reuse when $s, t, v \geq 1$, and instead simply prove a looser lower bound that yields a similar fractional symmetric prefactor for any s, t, v,

$$\hat{W}^{\Psi}(n, s, t, v, \hat{M}) \geq \max\left(\frac{s!v!}{(s+v)!}, \frac{t!v!}{(t+v)!}, \frac{s!t!}{(s+t)!}\right)^{3/2} \cdot \frac{2n^{s+t+v}}{s!t!v!\sqrt{\hat{M}}}.$$

So, whether this lower bound is attainable for $s, t, v \geq 1$ or whether a tighter bound may be proven remains an open question.

We obtain this lower bound by reducing from schedules for the standard symmetric tensor contraction algorithm to schedules for nonsymmetric matrix multiplication (with an appropriate prefactor). In particular, for $r = \max\left(\frac{(s+v)!}{s!v!}, \frac{(t+v)!}{t!v!}, \frac{(s+t)!}{s!t!}\right)$, we show that any schedule for the standard symmetric contraction algorithm with cost $\hat{W}(n, s, t, v, \hat{M})$ yields a schedule for matrix multiplication of arbitrary nonsymmetric matrices $\hat{A}$ and $\hat{B}$ which uses $r \cdot \hat{M}$ cache memory and

attains the cost $\bar{\hat{W}}(n^s/s!, n^t/t!, n^v/v!, r\cdot\hat{M}) \leq r\cdot\hat{W}(n,s,t,v,\hat{M})$, where $\bar{W}$ is the communication cost of nonsymmetric multiplication, for which we have a lower bound via Theorem 4.2.1.

Consider any schedule for the standard symmetric contraction algorithm, which computes $\bar{C} = \bar{A}\cdot\bar{B}$, where $\bar{A}$ and $\bar{B}$ are matrices corresponding to folded forms of A and B respectively, and $\bar{C}$ is subsequently accumulated (symmetrized) to form C. Such a schedule may employ multiplications of entries of the operands $A_{ij} \in \bar{A}$, which are equivalent to some other $A_{kl} \in \bar{A}$, $A_{ij} = A_{kl}$, due to being symmetric in A (the folding ignores some symmetries), and the same for B. We represent the schedule as an interleaved sequence S of 'read', 'write', 'compute', 'copy', and 'discard' operations. We define cache storage set M_l to be the elements stored in cache after the ith operation in the sequence S and let $M_0 = \emptyset$, since we assume the cache is empty at the start. The size of the cache storage set $\forall l \in [1, |S|]$, should not exceed the cache capacity $|M_l| \leq \hat{M}$. Any operations in sequence $s_l \in S$ must be one the following,

- read of an element of A from memory into cache, $s_l = (a, \text{read})$, where $a \in A$ and $a \notin M_{l-1}$, but subsequently $a \in M_l$,
- read of an element of B from memory into cache, $s_l = (b, \text{read})$, where $b \in B$ and $b \notin M_{l-1}$, but subsequently $b \in M_l$,
- computation of the form $\bar{c} = a \cdot b$ (corresponding to multiplication $\bar{A}_{ik} \cdot \bar{B}_{kj}$ into the entry $\bar{C}_{ij}$), $s_l = (\bar{c}, a, b, \text{compute})$, where $a \in A$, $b \in B$, and $a, b \in M_{l-1}$, while $\bar{c} \notin M_{l-1}$, but subsequently, $\bar{c} \in M_l$ (multiple partial sums of $\bar{C}_{ij}$ may not be in cache concurrently),
- computation of the form $\bar{c} = \bar{c} + a \cdot b$ (corresponding to multiplication $\bar{A}_{ik} \cdot \bar{B}_{kj}$ and its accumulation into the entry $\bar{C}_{ij}$), $s_l = (\bar{c}, a, b, \text{compute})$, where $a \in A$, $b \in B$, and $a, b, \bar{c} \in M_{l-1}$,
- computation of the form $c = c + \bar{c}$ (corresponding to symmetrization of $\bar{C}$ into C) $s_l = (c, \bar{c}, \text{compute})$, where $c, \bar{c} \in M_{l-1}$ and subsequently $\bar{c} \notin M_l$,
- copy of the form $c = \bar{c}$ (corresponding to symmetrization of $\bar{C}$ into C or a simple set when symmetrization of $\bar{C}$ is not necessary, namely when $\bar{C}$ is a vector) $s_l = (c, \bar{c}, \text{copy})$, where $\bar{c} \in M_{l-1}$ and subsequently $c \in M_l$ and $\bar{c} \notin M_l$,
- discard of values of A from cache, $s_l = (a, \text{discard})$, where $a \in A$ and $a \in M_{l-1}$, but subsequently $a \notin M_l$,
- discard of values of B from cache, $s_l = (b, \text{discard})$, where $b \in B$ and $b \in M_{l-1}$, but subsequently $b \notin M_l$,
- write/accumulate from cache to memory, $s_l = (c, \text{write})$, where $c \in C$ and $c \in M_{l-1}$ but subsequently $c \notin M_l$.

Each of the $\binom{n+s-1}{s}\binom{n+t-1}{t}\binom{n+v-1}{v}$ multiplications necessary for the standard algorithm must correspond to a 'compute' operation in the schedule. Each of $\binom{n+s-1}{s}\binom{n+t-1}{t}\binom{n+v-1}{v}+\binom{n+s-1}{s}\binom{n+t-1}{t}$

additions must correspond to a 'compute' operation or an accumulate ('write') operation in the schedule. All computed entries of C (results of compute operations), must also be written or accumulated to the output, and output entries may not be overwritten. The 'copy' operations are free as they do not correspond to any computation, but are simply a relabeling. We note that while the standard symmetric contraction algorithm must implicitly compute the multiplications necessary to form the $\bar{C}$ intermediate, under our model this schedule forces accumulation to memory of the intermediates and multiplications directly to the output C itself (since the entries of $\bar{C}$ must be eventually accumulated there anyway). We measure the communication cost of the schedule as the number of 'read' and 'write' operations in S, therefore accumulating directly to C is always more sensible than accumulating to some other intermediate. The formulation of this schedule is strict in the sense that it precludes recomputation of contributions to C as well as any other operations that don't contribute directly to algorithmic progress.

We now define a new schedule $\hat{S}$ from S, with cache store $\hat{M}_l$ after operation $s_l \in \hat{S}$, which computes matrix multiplication of two arbitrary (nonsymmetric) matrices $\hat{C} = \hat{A} \cdot \hat{B}$, where the dimensions of $\hat{A}$ are $\binom{n+s-1}{s}$ by $\binom{n+v-1}{v}$ and the dimensions of $\hat{B}$ are $\binom{n+v-1}{v}$ by $\binom{n+t-1}{t}$. The new schedule $\hat{S}$ mimics S by reading and discarding all symmetrically equivalent entries in A and B, while computing and writing multiplications which contribute to an entry of $\bar{C}$ to the same destination in $\hat{C}$. We say that a pair of entries in $\bar{A}_{ij}, \bar{A}_{kl} \in A$ are *symmetrically equivalent* if they are folded from entries $\bar{A}_{ij} = A_{i\langle s+v\rangle}$, $\bar{A}_{kl} = A_{j\langle s+v\rangle}$ and there exists a permutation π such that $i\langle s+v\rangle = \pi(j\langle s+v\rangle)$. There are sets of size up to $(s+v)!$ of symmetrically equivalent entries A, but only $(s+v)!/(s!v!)$ in $\bar{A}$, since the folded matrix $\bar{A}$ does not include entries that are symmetrically equivalent in permutation within $j\langle s\rangle$ and within $l\langle v\rangle$ for any entry $A_{j\langle s\rangle l\langle v\rangle}$. Similarly, for B, the folded matrix $\bar{B}$ contains sets of size up to $(t+v)!/(t!v!)$ of entries which are symmetrically equivalent due to the symmetry in B. Formally, for the ith operation s_l for $l = [1, |S|]$, we append the following operations to $\hat{S}$ (or ignore it if its not one of the below):

- if $s_l = (a, \text{read})$, where $a \in A$, we add up to $(s+v)!/(s!v!)$ read operations of all entries of $\hat{A}$ to $\hat{S}$ which have the same indices as those which are symmetrically equivalent to a in the matrix $\bar{A}$ (there may be fewer than $(s+v)!/(s!v!)$ if the entry is on some diagonal in A), thus for all 'compute' operations for which a can be an operand in S, we have loaded the corresponding entries in $\hat{A}$,

- if $s_l = (b, \text{read})$, where $b \in B$, we add up to $(t+v)!/(t!v!)$ read operations of all entries of $\hat{B}$ to $\hat{S}$ which have the same indices as those which are symmetrically equivalent to b in the matrix $\bar{B}$, thus for all 'compute' operations for which b can be an operand in S, we have loaded the corresponding entries in $\hat{B}$,

- if $s_l = (\bar{c}, a, b, \text{compute})$ of the form $\bar{c} = a \cdot b$ or of the form $\bar{c} = \bar{c} + a \cdot b$ and corresponds to multiplication $\bar{A}_{ik} \cdot \bar{B}_{kj}$, we add a compute operation $\hat{c} = \hat{c} + \hat{A}_{ik} \cdot \hat{B}_{kj}$ to $\hat{S}$ where $\hat{c}$ is a partial sum to be accumulated into $\hat{C}_{ij}$, if there is already an existing $\hat{c}$ partial sum of $\hat{C}_{ij}$ is in $\hat{M}_{l-1}$ and add operation $\hat{c} = \hat{A}_{ik} \cdot \hat{B}_{kj}$ to $\hat{S}$ if no partial sum of $\hat{C}_{ij}$ was in $\hat{M}_{l-1}$,

- if $s_l = (a, \text{discard})$, where $a \in A$, we add up to $(s+v)!/(s!v!)$ discard operations of all entries of $\hat{A}$ to $\hat{S}$ which have the same indices as those which are symmetrically equivalent to a in the matrix $\bar{A}$,
- if $s_l = (b, \text{discard})$, where $b \in B$, we add up to $(t+v)!/(t!v!)$ discard operations of all entries of $\hat{B}$ to $\hat{S}$ which have the same indices as those which are symmetrically equivalent to b in the matrix $\bar{B}$,
- if $s_l = (c, \text{write})$, where $c \in C$, we add up to $(s+t)!/(s!t!)$ write or accumulate operations all contributions to $\hat{C}$ in M_{l-1} which have the same indices as those which are symmetrically equivalent to the entry to which c contributes in $\hat{C}$ (which discards these partial sums from M_l).

The resulting schedule $\hat{S}$ computes the correct answer $\hat{C}$, as it performs the $\binom{n+s-1}{s}\binom{n+v-1}{v}\binom{n+t-1}{t}$ multiplications needed, as those must also be done to compute $\bar{C}$ by the standard symmetric contraction algorithm schedule. Further, it always has the needed operands to compute the multiplications, since whenever an entry $a \in \bar{A}$ or $b \in \bar{B}$ resides in S, all entries to which are symmetrically equivalent to a or b via the symmetry of A and B (for which a or b may act as operands in place of) are loaded from $\hat{A}$ or $\hat{B}$ by $\hat{S}$ and kept in memory until a or b is discarded by S. While the standard symmetric contraction schedule accumulates the partial sums of $\bar{C}$ to partial sums of C, prior to writing them to memory, the schedule $\hat{S}$ accumulates all the partial sums of $\hat{C}$ to memory whenever a partial sum of C with a symmetrically equivalent entry is written by S. All contributions to $\hat{C}$, will be written or accumulated to memory (accumulated if another contribution has already been written to this entry), since they correspond to a partial sum of $\bar{C}$ of the same index, which must be accumulated or copied into an entry of C, which must subsequently be written, and for each operation that writes an entry of C to memory, $\hat{S}$ writes to memory all entries of $\hat{C}$, that have the same index as any partial sum that could have been accumulated into this partial sum of C.

Further, the cache size necessary to compute $\hat{S}$ is no more than $r \cdot \hat{M} = \max\left(\frac{(s+v)!}{s!v!}, \frac{(t+v)!}{t!v!}, \frac{(s+t)!}{s!t!}\right) \cdot \hat{M}$, since for each entry of A present in cache at any given point in S, at most $(s+v)!/(s!v!)$ entries of $\hat{A}$ are present in $\hat{S}$, as up to this number of symmetrically equivalent entries are read and discarded, whenever the corresponding of A is read or discarded. A similar argument for entries of $\hat{B}$ yields the factor of $\frac{(t+v)!}{t!v!}$. For partial sums of $\hat{c} \in \hat{C}$, we notice that whenever some partial sum is in $\hat{S}$ there must either by a partial sum of $\bar{c} \in \bar{C}$ with the same index as $\hat{c}$ or a partial sum of $c \in C$ to which $\bar{c}$ was copied or accumulated into and which corresponds to an entry in C to which $\hat{c}$ is symmetrized into. Since there at most $(s+t)!/(s!t!)$ entries (and partial sums of unique index) in $\bar{C}$ that are symmetrized into the same entry of C and these are accumulated to memory (and therefore discarded), whenever the entry of C is accumulated, there can be no more than $(s+t)!/(s!t!)$ different partial sums present in $\hat{S}$ for each partial sum present in S. The number of reads and writes in $\hat{S}$ is by a similar argument no greater than a factor of r of those in S. Therefore, we've created a valid schedule for matrix multiplication of arbitrary matrices with cost no more than $\bar{\hat{W}}(n^s/s!, n^t/t!, n^v/v!, r \cdot \hat{M}) \leq r \cdot \hat{W}^{\Psi}(n, s, t, v, \hat{M})$. Now, since, we have a lower bound for

nonsymmetric matrix multiplication, we can use Theorem 4.2.1 to assert that,

$$\begin{aligned}\hat{W}^{\Psi}(n,s,t,v,\hat{M}) &\geq \frac{1}{r}\cdot\bar{\hat{W}}(n^s/s!,n^t/t!,n^v/v!,r\cdot\hat{M})\\ &\geq \frac{1}{r}\cdot\frac{2n^{s+t+v}}{s!t!v!\sqrt{r\cdot\hat{M}}}\\ &= \frac{1}{r^{3/2}}\cdot\frac{2n^{s+t+v}}{s!t!v!\sqrt{\hat{M}}}\\ &= \min\left(\frac{s!v!}{(s+v)!},\frac{t!v!}{(t+v)!},\frac{s!t!}{(s+t)!}\right)^{3/2}\cdot\frac{2n^{s+t+v}}{s!t!v!\sqrt{\hat{M}}}.\end{aligned}$$

Combining the above lower bound, with the trivial lower bound from reading the inputs and outputs, we obtain the desired,

$$\begin{aligned}\hat{W}(n,s,t,v,\hat{M}) \geq \max\Bigg[&\min\left(\frac{s!v!}{(s+v)!},\frac{t!v!}{(t+v)!},\frac{s!t!}{(s+t)!}\right)^{3/2}\frac{2n^{s+t+v}}{s!t!v!\sqrt{\hat{M}}},\\ &\frac{n^{s+t}}{(s+t)!}+\frac{n^{s+v}}{(s+v)!}+\frac{n^{t+v}}{(t+v)!}\Bigg].\end{aligned}$$

We note that a similar reduction may be used to show that the multiplication of nonsymmetric matrices whose entries may overlap (e.g. $A\cdot A$) has cost no less than $\frac{\sqrt{2}}{4}$ of $\bar{\hat{W}}(n,m,k,\hat{M})$ (where it is assumed the entries A and B are disjoint/independent).

□

11.2.3 The Fast Symmetric Contraction Algorithm

Our main result is an algorithm that performs any such contraction using only $n^{\omega}/\omega!$ multiplications to leading order, which is a factor of $\frac{(s+t+v)!}{s!t!v!}$ fewer than the standard method. However, the algorithm performs more additions per multiplication than the standard method.

Algorithm $\boldsymbol{\Phi}_{\odot}^{(\mathbf{s},\mathbf{t},\mathbf{v})}$: The new algorithm is based on the idea of computing a fully symmetric intermediate tensor $\hat{Z}$ of order ω, with elements corresponding to the result of a scalar multiplication. Due to its symmetry this tensor has $\binom{n+\omega-1}{\omega}$ unique entries, which are $\forall i\langle\omega\rangle$,

$$\hat{Z}_{i\langle\omega\rangle} = \left(\sum_{j\langle s+v\rangle\in\chi(i\langle\omega\rangle)} A_{j\langle s+v\rangle}\right)\cdot\left(\sum_{l\langle t+v\rangle\in\chi(i\langle\omega\rangle)} B_{l\langle t+v\rangle}\right). \tag{11.2.3}$$

This tensor need not, and should not be stored explicitly due to the associated memory overhead. Instead, each unique multiplication corresponding to one of the unique entries of $\hat{Z}$, should be accumulated to $\binom{\omega}{s+t}$ entries of its 'contracted form', an order $s+t$ fully symmetric tensor Z, whose elements $\forall i\langle s+t\rangle$ are

$$Z_{i\langle s+t\rangle} = \sum_{k\langle v\rangle}\hat{Z}_{i\langle s+t\rangle k\langle v\rangle}. \tag{11.2.4}$$

The tensor Z contains all the terms needed for the output C, but also some extra terms. Fortunately, all of these extra terms may be computed with low-order cost. We form two other fully symmetric tensors of order $s+t$: V and W, which contain all of these unwanted terms and are subsequently used to cancel them out. To form the tensor V efficiently, we compute contracted forms of A and B, $A^{(p)}$ and $B^{(q)}$, $\forall p, q \in [1, v]$. The elementwise equations for these tensors are

$$\forall p \in [1, v],\ \forall i\langle s+v-p\rangle,\ A^{(p)}_{i\langle s+v-p\rangle} = \sum_{k\langle p\rangle} A_{i\langle s+v-p\rangle k\langle p\rangle},$$
$$\forall q \in [1, v],\ \forall i\langle t+v-q\rangle,\ B^{(q)}_{i\langle t+v-q\rangle} = \sum_{k\langle q\rangle} B_{i\langle t+v-q\rangle k\langle q\rangle}.$$

Now, we define the elements of V in terms of the above intermediates (also with $A^{(0)} = A$ and $B^{(0)} = B$), $\forall i\langle s+t\rangle$,

$$V_{i\langle s+t\rangle} = \sum_{r=0}^{v-1} \binom{v}{r} \cdot \sum_{p=\max(0,v-t-r)}^{v-r} \binom{v-r}{p} \cdot \sum_{q=\max(0,v-s-r)}^{v-p-r} \binom{v-p-r}{q} \cdot n^{v-p-q-r} \cdot \sum_{k\langle r\rangle} \left[\sum_{j\langle s+v-p-r\rangle \in \chi(i\langle s+t\rangle)} \left(A^{(p)}_{j\langle s+v-p-r\rangle k\langle r\rangle} \right) \right] \cdot \left[\sum_{l\langle t+v-q-r\rangle \in \chi(i\langle s+t\rangle)} \left(B^{(q)}_{l\langle t+v-q-r\rangle k\langle r\rangle} \right) \right]. \tag{11.2.5}$$

The tensor W employs a different set of intermediates $U^{(r)}$, $\forall r \in [1, \min(s,t)]$, which are of order $s+t-r$. These intermediates have r indices, which are shared between A and B and are not contracted over. The equations for forming the elements of these tensors are

$$\forall r \in [1, \min(s,t)],\ \ \forall i\langle s+t-2r\rangle,$$
$$U^{(r)}_{m\langle r\rangle i\langle s+t-2r\rangle} = \sum_{(j\langle s-r\rangle, l\langle t-r\rangle) \in \chi(i\langle s+t-2r\rangle)} \left(\sum_{k\langle v\rangle} A_{m\langle r\rangle j\langle s-r\rangle k\langle v\rangle} \cdot B_{m\langle r\rangle l\langle t-r\rangle k\langle v\rangle} \right), \tag{11.2.6}$$

Each $U^{(r)}$ tensor is symmetric in the $m\langle r\rangle$ index group and separately in the $i\langle s+t-2r\rangle$ index group. Using these tensors, we can now form our last intermediate, W, $\forall i\langle s+t\rangle$,

$$W_{i\langle s+t\rangle} = \sum_{r=1}^{\min(s,t)} \left(\sum_{(m\langle r\rangle, h\langle s+t-2r\rangle) \in \chi(i\langle s+t\rangle)} U^{(r)}_{m\langle r\rangle h\langle s+t-2r\rangle} \right) \tag{11.2.7}$$

As shown in the proof of correctness in Section 11.3.1, the V and W tensors contain all the terms which separate Z from the C we seek to compute (equation 11.1.2). So, we form the final output of the algorithm $\hat{C}$ (shown in Section 11.3.1 to be algebraically equivalent to C) via the summation, $\forall i\langle s+t\rangle$,

$$\hat{C}_{i\langle s+t\rangle} = \frac{1}{\binom{s+t}{s}} \cdot \left(Z_{i\langle s+t\rangle} - V_{i\langle s+t\rangle} - W_{i\langle s+t\rangle} \right). \tag{11.2.8}$$

We designate this algorithm as a function $\hat{C} = \Phi_{\odot}^{(s,t,v)}(A, B)$. The memory footprint necessary for this algorithm is to leading order the same as that of the standard algorithm ($\hat{C} = \Psi_{\odot}^{(s,t,v)}(A, B)$), both are dominated by the space necessary to store the input and output tensors. Intermediates, such as $\bar{C}$ in the standard algorithm, and Z, V, and W in the fast algorithm, may be accumulated directly to the output (C and $\hat{C}$, respectively), and therefore require no extra storage. Similarly, for each element of $U^{(r)}$, it is possible to compute all elements of W to which it contributes and discard it. However, a low-order amount of extra storage may be required for $A^{(p)}$ and $B^{(q)}$ since elements of W depend on different combinations of entries of these two intermediates. However, in practice implementations of the standard algorithm frequently compute $\bar{C}$ explicitly for simplicity, and it seems unlikely that the storage necessary for $A^{(p)}$, $B^{(q)}$ would be problematic. Further, the only time $\Omega(n)$ combinations of $A^{(p)}$ and $B^{(q)}$ need to be multiplied together is when $p, q \geq 1$, in which case it should be possible to recompute one of the two redundantly on the fly, without raising the leading order cost, if a constant amount of auxiliary storage is really a requirement.

The next section gives some basic but important special cases of this algorithm for $s, t, v \in \{0, 1\}$. Following these examples, we return to analysis of the general form, proving correctness in Section 11.3.1, numerical stability in Section 11.3.2, and analyzing the computational and communication costs in Section 11.3.3.

11.2.4 Examples (Special Cases)

To give an intuitive interpretation of the general algorithm, as well as to cover some basic applications, we now give a few special cases of it. We consider multiplication of symmetric matrix with a vector ($s = 1, v = 1, t = 0$), the symmetric rank-2 outer product ($s = 1, v = 0, t = 1$), and the symmetrized multiplication of symmetric matrices, which is known as Jordan ring matrix multiplication ($s = v = t = 1$). As before we will consider all vector/matrix/tensor entries to be elements of a possibly nonassociative ring over the set R with operations $\varrho = (+, \cdot)$ the costs of which are ν_ϱ, μ_ϱ respectively.

Multiplication of a Symmetric Matrix by a Vector

Consider two vectors of n elements b and c as well as an n-by-n symmetric matrix A. We seek to compute

$$\forall i, \;\; c_i = \sum_k A_{ik} \cdot b_k,$$

which is just equation 11.1.2 with $s = 1, v = 1, t = 0$. The cost of the traditional algorithm (equation 11.2.1), $\Psi_{\odot}^{(1,0,1)}(A, b)$ which simply treats A as nonsymmetric is $F_{\text{symv}}^{\Psi}(\varrho, n) = \mu_\varrho \cdot n^2 + \nu_\varrho \cdot n^2$. Our new algorithm for this problem, $\Phi_{\odot}^{(1,0,1)}(A, b)$ computes c as follows,

$$\begin{aligned}
&\forall i, j, \quad \hat{Z}_{ij} = A_{ij} \cdot (b_i + b_j), \\
&\forall i, \qquad Z_i = \sum_k \hat{Z}_{ik}, \qquad A_i^{(1)} = \sum_k A_{ik}, \qquad V_i = A_i^{(1)} \cdot b_i, \qquad c_i = Z_i - V_i.
\end{aligned}$$

Since $(b_i + b_j)$ and subsequently $\hat{Z}_{ij}$ are symmetric in i, j, Z_i can be computed in $n^2/2$ multiplications to leading order, for a total cost of

$$F^{\Phi}_{\text{symv}}(\varrho, n) = \mu_\varrho \cdot \left(\frac{1}{2}n^2 + \frac{3}{2}n\right) + \nu_\varrho \cdot \left(\frac{5}{2}n^2 + \frac{3}{2}n\right).$$

Now, if we consider multiplication of A by K vectors (equivalently by an n-by-K matrix), the cost of computing $A^{(1)}$ is amortized over the K matrix-vector multiplications, yielding the overall cost,

$$F^{\Phi}_{\text{symm}}(\varrho, n, K) = \mu_\varrho \cdot \left(\frac{1}{2}Kn^2 + \frac{3}{2}Kn\right) + \nu_\varrho \cdot \left(\frac{3}{2}Kn^2 + n^2 + \frac{3}{2}Kn\right).$$

The fast algorithm for the `symm` problem is useful whenever $F^{\Phi}_{\text{symm}}(\varrho, n, K) < K \cdot F^{\Psi}_{\text{symv}}(\varrho, n)$, which is the case whenever $\mu_\varrho > \nu_\varrho$ and $n, K \gg 1$. That condition is true for the cases where the scalars are complex, where $\mu_\varrho \geq 3 \cdot \nu_\varrho$ (assuming multiplication of two real numbers costs at least as much as addition). There are two use-cases for the complex-problem, one where the A is complex and symmetric, e.g. the Discrete Fourier Transform (DFT) matrix, for which case the described algorithm applies without modification (outside of the real addition and multiplications becoming complex additions and multiplications), and the second, more common case where A is a Hermitian matrix. The latter case corresponds to the BLAS routine `hemm`, and our algorithm may be adapted to handle it as described in Section 11.4.3. In both cases, the fast algorithm yields at least a 4/3 reduction in cost.

The fast algorithm is also easily adaptable to the case when A is a sparse symmetric matrix with nnz non-zeros. Instead of the usual nnz multiplications and nnz additions, the fast algorithm requires $\frac{1}{2}$nnz multiplications and $\frac{5}{2}$nnz additions to leading order (or $\frac{3}{2}$nnz leading order additions when computation of $A^{(1)}$ may be amortized).

However, we note that for certain choices of A and b the fast algorithm incurs more error (see Section 11.3.4).

Symmetrized Vector Outer Product

We now consider a rank-2 outer product of vectors a and b of length n to form an n-by-n symmetric matrix C, computing $\forall i, j$,

$$C_{ij} = \frac{1}{2}(a_i \cdot b_j + a_j \cdot b_i).$$

Solving this problem corresponds to equation 11.1.2 with $s = 1$, $v = 0$, and $t = 1$. For floating point real numbers this corresponds to BLAS routine `syr2`. The standard method $\Psi^{(1,1,0)}_{\odot}$ computes the unique part of C using n^2 scalar multiplications and n^2 scalar additions, so $F^{\Psi}_{\text{syr2}}(\varrho, n) = F^{\Psi}_{\text{symv}}(\varrho, n)$. Our new algorithm for this problem, $\Phi^{(1,1,0)}_{\odot}(a, b)$ computes C as follows,

$$\forall i, j, \quad Z_{ij} = \hat{Z}_{ij} = (a_i + a_j) \cdot (b_i + b_j), \quad U^{(1)}_i = a_i \cdot b_i, \quad W_{ij} = U^{(1)}_i + U^{(1)}_j, \quad C_{ij} = \frac{1}{2}(Z_{ij} - W_{ij}).$$

The number of multiplications needed for the new algorithm is the same as in the previous case of a symmetric matrix multiplied by a vector, since $\hat{Z}$ is again symmetric and be computed via the $\binom{n+1}{2}$ multiplications necessary to form its unique elements. However, the number of additions is slightly less than the `symv` case,

$$F^{\Phi}_{\text{syr2}}(\varrho, n) = \mu_\varrho \cdot \left(\frac{1}{2}n^2 + \frac{3}{2}n\right) + \nu_\varrho \cdot (2n^2 + 2n).$$

Now we consider the case of a rank-$2K$ outer product where instead of vectors a and b, we have n-by-K matrices A and B, which corresponds to the BLAS routine `syr2K`. In this case, we can compute W with low-order cost, yielding a reduction in the number of additions needed. The resulting new algorithm has the same leading order cost as `symm`,

$$F^{\Phi}_{\text{syr2K}} = (\varrho, n, K) = \mu_\varrho \cdot \left(\frac{1}{2}Kn^2 + \frac{3}{2}Kn\right) + \nu_\varrho \cdot \left(\frac{3}{2}Kn^2 + n^2 + \frac{5}{2}Kn + n\right).$$

Furthermore, to leading order, F^{Φ}_{syr2K} has the same relation to F^{Ψ}_{syr2K} as F^{Φ}_{symm} had to F^{Ψ}_{symm}.

This algorithm may be adapted to the case of the Hermitian outer product $a \cdot b^* + a \cdot b^*$, where $*$ denotes the Hermitian adjoint (conjugate) of the vector. The adaptation of the fast algorithm to this operation, which also corresponds to BLAS routine `her2`, is described in a general fashion in Section 11.4.3 and performs the same number of multiplication and addition operations (of complex numbers) as F^{Φ}_{syr2}. The extension to `her2K` has the same number of complex operations as F^{Φ}_{syr2K} and is therefore 4/3 cheaper in leading order cost than the standard algorithm for `her2K`. So, since we now know how to compute `hemm` and `her2K` in 3/4 of the operations to leading order, we can compute the reduction to tridiagonal form for the Hermitian eigensolve (LAPACK routine `hetrd`), which can be formulated so that the cost is dominated by these two subroutines, in 3/4 of the usual number of operations to leading order.

Jordan Ring Matrix Multiplication

As a last example, we consider the case of 11.1.2 where $s = t = v = 1$, where all terms of the $\Phi^{(1,1,1)}_{\odot}(A, B)$ take part. Given symmetric matrices A, B of dimension n, we compute, $\forall i, j$,

$$C_{ij} = \frac{1}{2} \cdot \sum_k \left(A_{ik} \cdot B_{jk} + A_{jk} \cdot B_{ik}\right).$$

This problem is again a special case of equation 11.1.2, this time with $s, v, t = 1$. The traditional algorithm (equation 11.2.1), $\Psi^{(1,1,1)}_{\odot}(A, B)$ simply multiplies A and B as nonsymmetric matrices and symmetrizes the product, for a total cost of

$$F^{\Psi}_{\text{syrmm}}(\varrho, n) = \mu_\varrho \cdot n^3 + \nu_\varrho \cdot n^3 + O((\mu_\varrho + \nu_\varrho) \cdot n^2).$$

$\Phi^{(1,1,1)}_{\odot}(A, B)$ computes C by doing only $n^3/6$ leading order multiplications required to compute the third order symmetric tensor $\hat{Z}$, $\forall i, j, k$,

$$\hat{Z}_{ijk} = (A_{ij} + A_{ik} + A_{jk}) \cdot (B_{ij} + B_{ik} + B_{jk}).$$

Subsequently, the rest of the work can be done in a low-order number of multiplications as follows, $\forall i, j$,

$$
\begin{aligned}
Z_{ij} &= \sum_k Z_{ijk}, \qquad A_i^{(1)} = \sum_k A_{ik}, \qquad B_i^{(1)} = \sum_k B_{ik}, \\
V_{ij} &= n \cdot A_{ij} \cdot B_{ij} + \left(A_i^{(1)} + A_j^{(1)}\right) \cdot B_{ij} + A_{ij} \cdot \left(B_i^{(1)} + B_j^{(1)}\right), \\
U_l &= \sum_k A_{lk} \cdot B_{lk}, \qquad W_{ij} = U_i + U_j, \quad C_{ij} = \frac{1}{2}(Z_{ij} - V_{ij} - W_{ij}).
\end{aligned}
$$

Due to the symmetry in computation of Z_{ij}, and the fact that all other terms are low-order, the cost of this reformulation is

$$
F^{\Phi}_{\text{syrmm}}(\varrho, n) = \mu_\varrho \cdot \frac{1}{6} n^3 + \nu_\varrho \cdot \frac{7}{6} n^3 + O((\mu_\varrho + \nu_\varrho) \cdot n^2),
$$

so this form requires a factor of 6 fewer multiplications and a factor of $3/2$ fewer total operations than F^{Ψ}_{syrmm} to leading order. However, we note that for certain choices of A and B the fast algorithm incurs more error (see Section 11.3.4).

A special case of Jordan ring matrix multiplication is squaring a symmetric matrix ($B = A$ and a prefactor of $1/2$). For this case, a typical approach might only exploit the symmetry of the output (only computing the unique $\binom{n+1}{2}$ entries of the output), performing of $n^3/2$ multiplications and additions to leading order, while the fast algorithm, $A^2 = \Phi_{\odot}^{(1,1,1)}(A, A)$, would need $n^3/6$ multiplications and $5n^3/6$ additions to leading order (since the $2n^3/6$ additions for B may now be avoided). While the total number of operations is the same, a factor of three fewer multiplications are needed by the fast algorithm.

11.2.5 Nesting Algorithms

The three general algorithms defined in this section, $\Upsilon^{(s,t,v)}$, $\Psi_{\odot}^{(s,t,v)}$, and $\Phi_{\odot}^{(s,t,v)}$ all perform contractions where multiplication is any distributive (not necessarily associative) operator " $\cdot$ " : $R_A \times R_B \to R_C$. Further the three algorithms themselves are distributive operators mapping from the set of $s + v$-order tensors (R_A) and the set of $v + t$-order tensors (R_B) onto the set of $s + t$-order tensors (R_C). The only requirements of R_A, R_B, and R_C is that they are an Abelian group over addition ($+$), which the set of tensors of any given order satisfies. Therefore, we may nest the application of any of the three algorithms by applying it to tensors of tensors and letting "$\cdot$" be another tensor contraction.

Further, we may define any tensor of order larger than one as a nest of multiple tensors of lower order, for instance we may refer to a matrix as a vector of vectors. However, such nested forms lose the notion of symmetry if we break up an index group, for instance by referring to a symmetric matrix as a vector of vectors. However, the nested form makes a lot of sense for partially-symmetric tensors, e.g. a fourth order tensor with two disjoint symmetric groups of two indices, may be referred to as a symmetric matrix of symmetric matrices, a representation which reflects the full symmetry of the fourth order tensor.

We write the nested application of algorithms as concatenated calls to the nested algorithms, with the leftmost being at the root level of recursion and the rightmost algorithm executing the actual scalar multiplications needed to compute the contraction. For instance we might nest the symmetric tensor contraction algorithm as

$$C = \Psi_{\odot}^{(s_1,t_1,v_1)} \Psi_{\odot}^{(s_2,t_2,v_2)} \Upsilon^{(s_3,t_3,v_3)}(A, B),$$

and similarly the fast symmetric tensor contraction algorithm as

$$C = \Phi_{\odot}^{(s_1,t_1,v_1)} \Phi_{\odot}^{(s_2,t_2,v_2)} \Upsilon^{(s_3,t_3,v_3)}(A, B).$$

These expressions signify that the nested algorithm would contract A of order $\sum_{i=1}^{3} s_i + v_i$ and symmetric in two sets of indices of size $s_1 + v_1$ and $s_2 + v_2$, B of order $\sum_{i=1}^{3} t_i + v_i$ and symmetric in two sets of indices of size $t_1 + v_1$ and $t_2 + v_2$, and C of order $\sum_{i=1}^{3} s_i + t_i$, symmetrized among index groups $s_1 + t_1$ and $s_2 + t_2$. Any partially-symmetric contraction can be decomposed in such a nested fashion, although in some cases symmetric groups of indices need to be split up among nested levels and effectively ignored. Such nested forms will be especially useful for the coupled-cluster contractions discussed in Section 11.5. Further, the fact that we can nest $\Phi_{\odot}^{(s,t,v)}$ on top of $\Upsilon^{(s',t',v')}$, will allow the application of the fast algorithm, $\Phi_{\odot}^{(s,t,v)}$ to scenarios where addition is much cheaper than multiplications, such as when $s', t', v' \geq 1$ for $\Upsilon^{(s',t',v')}$.

11.3 Analysis

We now return to analyzing the new algorithm, $\Phi(A, B)$ in its general form. We give a proof of correctness, numerical stability, and quantify the cost of the algorithm in terms of the number of operations and the amount of communication necessary.

11.3.1 Proof of Correctness

Theorem 11.3.1. *If, given A and B, $\hat{C} = \Phi_{\odot}^{(s,t,v)}(A, B)$ is computed in exact arithmetic then it is equivalent to $C = \hat{C}$ where $C = A \odot B$ is given by equation 11.1.2.*

Proof. To show algebraic equivalence of the formulation in equations 11.2.3–11.2.8 to equation 11.1.2 we show that V and W (equations 11.2.5 and 11.2.7) cancel the extra terms included in Z (equation 11.2.4), leaving exactly the terms needed by equation 11.1.2. There are a total of $n^v \cdot \binom{s+t+v}{s+v} \cdot \binom{s+t+v}{t+v}$ products $A_{j\langle s+v\rangle} \cdot B_{l\langle t+v\rangle}$ appearing in the expansion of equation 11.2.4. We partition these subterms constituting Z according to which $k\langle v\rangle$ indices (based on subscript m of k_m rather than the value that k_m takes on) show up in the $j\langle s+v\rangle$ indices of the A operand and the $l\langle t+v\rangle$ indices of the B operand:

- $x\langle a\rangle \in \chi(k\langle v\rangle)$, the $a \leq v$ sized subset of $k\langle v\rangle$ chosen by $j\langle s+v\rangle \in \chi(i\langle s+t\rangle k\langle v\rangle)$ to appear in the A operand,

- $y\langle b\rangle \in \chi(k\langle v\rangle)$, the $b \leq v$ sized subset of $k\langle v\rangle$ chosen by $l\langle t+v\rangle \in \chi(i\langle s+t\rangle k\langle v\rangle)$ to appear in the B operand.

Now we consider all possible cases of overlap of the indices appearing in $x\langle a\rangle$ and $y\langle b\rangle$,

- let $d\langle r\rangle \in \chi(k\langle v\rangle)$, for $r \leq \min(a,b) \leq v$ be the subset of $k\langle v\rangle$ that appears both in $x\langle a\rangle$ and $y\langle b\rangle$,
- let $e\langle p\rangle \in \chi(k\langle v\rangle)$, for $p \leq a-r$ be the subset of $k\langle v\rangle$ that appears exclusively in $x\langle a\rangle$ (and not in $y\langle b\rangle$),
- let $f\langle q\rangle \in \chi(k\langle v\rangle)$, for $q \leq b-r \leq v-p-r$ be the subset of $k\langle v\rangle$ that appears exclusively in $y\langle b\rangle$ (and not in $x\langle a\rangle$),
- let $g\langle v-q-p-r\rangle \in \chi(k\langle v\rangle)$ be the remaining subset of $k\langle v\rangle$, which appear in neither $x\langle a\rangle$ nor $y\langle b\rangle$.

We now partition Z according to the above partitions of $k\langle v\rangle$, $\forall i\langle s+t\rangle$,

$$Z_{i\langle s+t\rangle} = \sum_{k\langle v\rangle} \sum_{r=0}^{v} \sum_{(d\langle r\rangle, u\langle v-r\rangle)\in\chi(k\langle v\rangle)} \sum_{p=\max(0,v-t-r)}^{v-r} \sum_{(e\langle p\rangle, w\langle v-r-p\rangle)\in\chi(u\langle v-r\rangle)} \sum_{q=\max(0,v-s-r)}^{v-r-p} \sum_{(f\langle q\rangle, g\langle v-r-p-q\rangle)\in\chi(w\langle v-r-p\rangle)} \left[\left[\sum_{j\langle s+v-p-r\rangle\in\chi(i\langle s+t\rangle)} A_{j\langle s+v-p-r\rangle d\langle r\rangle e\langle p\rangle}\right] \cdot \left[\sum_{l\langle t+v-q-r\rangle\in\chi(i\langle s+t\rangle)} B_{l\langle t+v-q-r\rangle d\langle r\rangle f\langle q\rangle}\right]\right].$$

We note the fact that the index variables in $k\langle v\rangle$ are indistinguishable (i.e. they are dummy integration variables), which implies that for a given r, p, q no matter how the $k\langle v\rangle$ is partitioned amongst $d\langle r\rangle$, $e\langle p\rangle$, $f\langle q\rangle$, and $g\langle v-r-p-q\rangle$, the result is the same. This allows us to replace $k\langle v\rangle$ and its partitions with new sets of variables and multiply by prefactors corresponding to the number of selections possible for each partition, $\forall_{i\langle s+t\rangle}$,

$$Z_{i\langle s+t\rangle} = \sum_{r=0}^{v} \binom{v}{r} \sum_{d\langle r\rangle} \sum_{p=\max(0,v-t-r)}^{v-r} \binom{v-r}{p} \sum_{e\langle p\rangle} \sum_{q=\max(0,v-s-r)}^{v-r-p} \binom{v-r-p}{q} \sum_{f\langle q\rangle} \sum_{g\langle v-r-p-q\rangle} \left[\sum_{j\langle s+v-p-r\rangle\in\chi(i\langle s+t\rangle)} A_{j\langle s+v-p-r\rangle d\langle r\rangle e\langle p\rangle}\right] \cdot \left[\sum_{l\langle t+v-q-r\rangle\in\chi(i\langle s+t\rangle)} B_{l\langle t+v-q-r\rangle d\langle r\rangle f\langle q\rangle}\right].$$

Since B is independent of $e\langle p\rangle$ and A is independent of $f\langle q\rangle$, we can use distributivity to bring these summations inside the parentheses. Further, since both A and B are independent of the

indices $g\langle v-r-p-q\rangle$, we can replace this summation with a prefactor of $n^{v-r-p-q}$ (the size of the range of $g\langle v-r-p-q\rangle$), $\forall_{i\langle s+t\rangle}$,

$$Z_{i\langle s+t\rangle} = \sum_{r=0}^{v} \binom{v}{r} \sum_{p=\max(0,v-t-r)}^{v-r} \binom{v-r}{p} \sum_{q=\max(0,v-s-r)}^{v-r-p} \binom{v-r-p}{q} n^{v-r-p-q} \sum_{d\langle r\rangle} \Bigg[\Bigg[\sum_{j\langle s+v-p-r\rangle \in \chi(i\langle s+t\rangle)} \left(\sum_{e\langle p\rangle} A_{j\langle s+v-p-r\rangle d\langle r\rangle e\langle p\rangle} \right) \Bigg] \cdot \Bigg[\sum_{l\langle t+v-q-r\rangle \in \chi(i\langle s+t\rangle)} \left(\sum_{f\langle q\rangle} B_{l\langle t+v-q-r\rangle d\langle r\rangle f\langle q\rangle} \right) \Bigg] \Bigg].$$

Renaming the index sets and isolating $r = v, p = 0, q = 0$, allows us to extract the V term (equation 11.2.5), $\forall i\langle s+t\rangle$,

$$Z_{i\langle s+t\rangle} = \sum_{r=0}^{v-1} \binom{v}{r} \cdot \sum_{p=\max(0,v-t-r)}^{v-r} \binom{v-r}{p} \cdot \sum_{q=\max(0,v-s-r)}^{v-p-r} \binom{v-p-r}{q} \cdot n^{v-p-q-r} \sum_{k\langle r\rangle} \Bigg[\Bigg[\sum_{j\langle s+v-p-r\rangle m\langle p\rangle k\langle r\rangle \in \chi(i\langle s+t\rangle)} \left(\sum_{m\langle p\rangle} A_{j\langle s+v-p-r\rangle m\langle p\rangle k\langle r\rangle} \right) \Bigg] \cdot \Bigg[\sum_{l\langle t+v-q-r\rangle m\langle q\rangle k\langle r\rangle \in \chi(i\langle s+t\rangle)} \left(\sum_{m\langle q\rangle} B_{l\langle t+v-q-r\rangle m\langle q\rangle k\langle r\rangle} \right) \Bigg] \Bigg]$$
$$+ \sum_{k\langle v\rangle} \left(\sum_{j\langle s\rangle k\langle v\rangle \in \chi(i\langle s+t\rangle)} A_{j\langle s\rangle k\langle v\rangle} \right) \cdot \left(\sum_{l\langle t\rangle k\langle v\rangle \in \chi(i\langle s+t\rangle)} B_{l\langle t\rangle k\langle v\rangle} \right),$$

$$Z_{i\langle s+t\rangle} = V_{i\langle s+t\rangle} + \sum_{k\langle v\rangle} \left(\sum_{j\langle s\rangle k\langle v\rangle \in \chi(i\langle s+t\rangle)} A_{j\langle s\rangle k\langle v\rangle} \right) \cdot \left(\sum_{l\langle t\rangle k\langle v\rangle \in \chi(i\langle s+t\rangle)} B_{l\langle t\rangle k\langle v\rangle} \right)$$

We have now isolated the subterms where both A and B are contracted over the desired full set of $k\langle v\rangle$ indices into the latter term in the equation above. However, this latter term still contains subterms where the same outer (or noncontracted) indices appear in both A and B, which we want to discard, since equation 11.1.2 contains only disjoint partitions of the $i\langle s+t\rangle$ index set among A and B. We obtain W (equation 11.2.7) via enumerating all such remaining unneeded terms by the number of outer indices $r \in [1, \min(s,t)]$ which appear in both operands A and B, then summing over the disjoint partitions of the remaining $s-t-2r$ indices into $s-r$ indices which appear exclusively in term A and $t-r$ indices which appear exclusively in term B. This leaves r indices which appear in neither A nor B, which is the reason that the term V can be computed in low order

via U (equation 11.2.6). So now, $\forall i\langle s+t\rangle$, we have,

$$Z_{i\langle s+t\rangle} = V_{i\langle s+t\rangle} + \sum_{(j\langle s\rangle, l\langle t\rangle)\in\chi(i\langle s+t\rangle)} \left(\sum_{k\langle v\rangle} A_{j\langle s\rangle k\langle v\rangle} \cdot B_{l\langle t\rangle k\langle v\rangle}\right) +$$

$$\sum_{r=1}^{\min(s,t)} \left[\sum_{(m\langle r\rangle, h\langle s+t-2r\rangle)\in\chi(i\langle s+t\rangle)} \left[\sum_{(j\langle s-r\rangle, l\langle t-r\rangle)\in\chi(h\langle s+t-2r\rangle)} \left(\sum_{k\langle v\rangle} A_{m\langle r\rangle j\langle s-r\rangle k\langle v\rangle} \cdot B_{m\langle r\rangle l\langle t-r\rangle k\langle v\rangle}\right)\right]\right]$$

$$= V_{i\langle s+t\rangle} + \binom{s+t}{s} \cdot C_{i\langle s+t\rangle} + W_{i\langle s+t\rangle}$$

Plugging the result into equation 11.2.8, we see that $C = \hat{C}$.

□

11.3.2 Numerical Stability Analysis

We derive error bounds in terms of $\gamma_n = \frac{n\epsilon}{1-n\epsilon}$, where ϵ is the machine precision. We assume the error made in scalar additions, subtractions, and multiplications to be bounded equivalently $|fl\,(a\cdot b) - a\cdot b| \le |a\cdot b|\cdot\epsilon$ and $|fl\,(a\pm b) - (a\pm b)| \le |a\pm b|\cdot\epsilon$. The forward error bound for the standard symmetric contraction algorithm (equations 11.1.2) arises directly from matrix multiplication where $n^v/v! + O(n^{v-1})$ scalar intermediates contribute to each entry of the output, with an extra factor of $\binom{s+t}{s}$ incurred from the symmetrization of the partially-symmetric intermediate $\bar{C}$ (the result of the matrix multiplication). We use the $||X||_{\max}$ norm, which is the largest magnitude element in X, and therefore the bound for $\bar{C}$ grows by the same factor as the error, yielding the overall error bound,

$$||fl\left(\Psi_{\odot}^{(s,t,v)}(A,B)\right) - C||_{\max} \le \gamma_m \cdot m \cdot ||A||_{\max} \cdot ||B||_{\max} \text{ where } m = \frac{n^v}{v!}\binom{s+t}{s} + O(n^{v-1}).$$

To bound the error of the fast algorithm, we start with a lemma, which simplifies the error analysis of multiple intermediate tensors.

Lemma 11.3.2. *Consider a computation of a scalar from l-by-m matrix A and k-by-g matrix B of the form,*

$$c = \sum_{g=1}^{m}\left(\sum_{i=1}^{l} A_{ig}\right)\cdot\left(\sum_{j=1}^{k} B_{jg}\right),$$

where $||A||_{\max} \le \alpha$ and $||B||_{\max} \le \beta$. The floating point error of c is bound by

$$|fl\,(c) - c| \le \gamma_{m+l+k}\cdot m\cdot l\cdot k\cdot\alpha\cdot\beta + O(\epsilon^2).$$

Proof. The magnitude of the exact value $a_g = \sum_{i=1}^{l} A_{ig}$ is at most $l\cdot\alpha$, so the floating point error incurred is bounded by $\gamma_l\cdot l\cdot\alpha$. Similarly, for $b_g = \sum_{j=1}^{k} B_{jg}$ the floating point error is at most

$\gamma_k \cdot k \cdot \beta$. Therefore, we can obtain a bound on the floating point error for c via

$$|fl\,(c) - c| = \left| fl\left(\sum_{g=1}^{m}(a_g + \delta_g^A \cdot l \cdot \alpha) \cdot (b_g + \delta_g^B \cdot k \cdot \beta)\right) - c \right|,$$

where $|\delta_g^A| \leq \gamma_l$ and $|\delta_g^B| \leq \gamma_k$,

$$\begin{aligned}|fl\,(c) - c| &\leq \gamma_m \cdot m \cdot (l \cdot \alpha) \cdot (k \cdot \beta) + m \cdot (\gamma_l \cdot l \cdot \alpha) \cdot (k \cdot \beta) + m \cdot (l \cdot \alpha) \cdot (\gamma_k \cdot k \cdot \beta) + O(\epsilon^2) \\ &\leq \gamma_{m+l+k} \cdot m \cdot l \cdot k \cdot \alpha \cdot \beta + O(\epsilon^2)\end{aligned}$$

□

The following theorem bounds the forward error of the computation done by the fast symmetric tensor contraction algorithm. The bound for the fast algorithm is higher than for the standard algorithm, due to errors accumulated in the computation of the intermediate terms. In the theorem and derivation we employ the convention that $\binom{n}{n'} = 0$ for any $n' > n$.

Theorem 11.3.3. *The $fl\left(\Phi_{\odot}^{(s,t,v)}(A,B)\right)$ tensor computed via equations 11.2.3–11.2.8 in floating point arithmetic satisfies the following error bound with respect to $C = A \odot B$ the exact solution to equation 11.1.2,*

$$\left|fl\left(\hat{C}_{i\langle s+t\rangle}\right) - C_{i\langle s+t\rangle}\right| \leq (\gamma_{3m} \cdot \bar{z} + \gamma_{\hat{v}\cdot m} \cdot \hat{v} \cdot \bar{v} + \gamma_{\hat{w}\cdot m} \cdot \hat{w}) \cdot m \cdot ||A||_{max} \cdot ||B||_{max}$$

where $\bar{z} = \binom{\omega}{t} \cdot \binom{\omega}{s}$, $\hat{v} = \binom{s+t}{s}$, $\bar{v} = \binom{s+t}{s+v} + \binom{s+t}{t+v}$, $\hat{w} = 2^{s+t} \cdot \binom{s+t}{s}$, *and* $m = \binom{n+v-1}{v} + O(n^{v-1})$.

Proof. We bound the error of the algorithm using the following variables to denote the maximum magnitude values in A and B, $\alpha = ||A||_{\max}$ and $\beta = ||B||_{\max}$. We first bound the error incurred by any entry of Z (equation 11.2.4), where each left operand of each scalar multiplication requires $k_Z = \binom{\omega}{t}$ additions of A entries and each right operand requires $l_Z = \binom{\omega}{s}$ of B entries. Each entry of Z is a sum of $m_Z = \binom{n+v-1}{v}$ such multiplications, which allows us to apply Lemma 11.3.2 to obtain a bound the error on entries of Z as, $\forall i\langle s+t\rangle$,

$$\left|fl\left(Z_{i\langle s+t\rangle}\right) - Z_{i\langle s+t\rangle}\right| \leq \gamma_{m_Z+l_Z+k_Z} \cdot m_Z \cdot l_Z \cdot k_Z \cdot \alpha \cdot \beta + O(\epsilon^2).$$

We set $\bar{z} = k_Z \cdot l_Z = \binom{\omega}{t} \cdot \binom{\omega}{s}$ and $m = m_Z$. We further simplify the expression via $k_Z + l_Z + m \leq 3m$, which assumes $m \geq k_Z$ and $m \geq l_Z$, since it is expected that $n \gg \omega$, obtaining $\forall i\langle s+t\rangle$,

$$\left|fl\left(Z_{i\langle s+t\rangle}\right) - Z_{i\langle s+t\rangle}\right| \leq \gamma_{3m} \cdot \bar{z} \cdot m \cdot \alpha \cdot \beta + O(\epsilon^2).$$

We now bound the error contribution of V by first considering the error for a given p, q, r. For each scalar multiplication, the left operand is a sum of $k_V = \binom{s+t}{s+v-p-r}\binom{n+p-1}{p}$ entries of A (where we've included the sum necessary to compute the needed entry of $A^{(p)}$), and the right operand is a

sum of $l_V = \binom{s+t}{t+v-q-r}\binom{n+q-1}{q}$ entries of B (where we've included the sum necessary to compute the needed entry of $B^{(q)}$). The contribution of the particular choice of p, q, r to a given entry of V is a sum of $m_V = \binom{n+r-1}{r}$ such multiplications. Therefore, we can apply Lemma 11.3.2 to bound the error of such a contribution $c_{p,q,r}$ as

$$|fl\left(c_{p,q,r}\right) - c_{p,q,r}| \leq \gamma_{m_V+k_V+l_V} \cdot m_V \cdot k_V \cdot l_V \cdot \alpha \cdot \beta + O(\epsilon^2).$$

We now note that since $r \leq v-1$, m_V cannot exceed $O(n^{v-1})$. Similarly, $k_V = O(n^{v-1})$ whenever $p < v$ and $l_V = O(n^{v-1})$ whenever $q < v$. Since $p + q + r \leq v$, we are left with only two possible contributions which contribute to the leading order error, $O(\gamma_{n^v} \cdot n^v \cdot \alpha \cdot \beta)$, namely $c_{v,0,0}$ and $c_{0,v,0}$. For $c_{v,0,0}$, (for which it must be the case that $v \leq s$), $k_V = \binom{s+t}{s}\binom{n+v-1}{v}$, while $l_V = \binom{s+t}{t+v}$, and $m_V = 1$. For $c_{0,v,0}$, (for which it must be the case that $v \leq t$), $k_V = \binom{s+t}{s+v}$, while $l_V = \binom{s+t}{s}\binom{n+v-1}{v}$, and $m_V = 1$. We can therefore, bound the total error on any entry of V as, $\forall i \langle s+t \rangle$,

$$\left|fl\left(V_{i\langle s+t\rangle}\right) - V_{i\langle s+t\rangle}\right| \leq \gamma_{\hat{v}\cdot m} \cdot \hat{v} \cdot \bar{v} \cdot m \cdot \alpha \cdot \beta + O(\gamma_{\hat{v}\cdot m/n} \cdot m \cdot \alpha \cdot \beta),$$

where $\hat{v} = \binom{s+t}{s}$, $\bar{v} = \binom{s+t}{s+v} + \binom{s+t}{t+v}$, and as before $m = \binom{n+v-1}{v}$.

Finally, we bound the error of the entries of W. We start with an error bound on the entries of $U^{(r)}$ which is $\forall r \in [1, \min(s,t)]$, $\forall i \langle s+t-2r \rangle$,

$$\left|fl\left(U^{(r)}_{m\langle r\rangle i\langle s+t-2r\rangle}\right) - U^{(r)}_{m\langle r\rangle i\langle s+t-2r\rangle}\right| \leq \gamma_{\hat{u}\cdot m} \cdot \hat{u} \cdot m \cdot \alpha \cdot \beta,$$

where $\hat{u} = \binom{s+t-2r}{s-r}$ and $m = \binom{n+v-1}{v}$. We bound the error on elements of W in terms of the error of $U^{(r)}$,

$$\left|fl\left(W_{i\langle s+t\rangle}\right) - W_{i\langle s+t\rangle}\right| \leq \gamma_{\hat{w}\cdot m} \cdot \hat{w} \cdot m \cdot \alpha \cdot \beta$$

where $\hat{w} = \sum_{r=1}^{\min(s,t)} \binom{s+t}{r}\binom{s+t-r}{r}\binom{s+t-2r}{s-r}$ and $m = \binom{n+v-1}{v}$. Applying the identity $\binom{g}{h}\binom{g-h}{k} = \binom{g}{k}\binom{g-k}{h}$ repeatedly to $\hat{w}$, we simplify the expression to

$$\hat{w} = \sum_{r=1}^{\min(s,t)} \binom{s+t}{s}\binom{s}{r}\binom{t}{r} \leq 2^{s+t} \cdot \binom{s+t}{s},$$

because $\sum_{r=1}^{\min(s,t)} \binom{s}{r} \cdot \binom{t}{r}$ counts the number of subsets of $s+t$ objects with equal numbers taken from the s and from the t, which is fewer than the total number of subsets 2^{s+t}.

We can now combine the error bounds from Z, V, and W (substituting $\alpha = ||A||_{\max}$ and $\beta = ||B||_{\max}$ back in and approximating $\binom{n+v-1}{v} = n^v/v! + O(n^{v-1})$) to obtain an error bound on $\hat{C} = fl\left(\Phi^{(s,t,v)}_{\odot}(A,B)\right)$,

$$\left|fl\left(\hat{C}_{i\langle s+t\rangle}\right) - C_{i\langle s+t\rangle}\right| \leq (\gamma_{3m} \cdot \bar{z} + \gamma_{\hat{v}\cdot m} \cdot \hat{v} \cdot \bar{v} + \gamma_{\hat{w}\cdot m} \cdot \hat{w}) \cdot m \cdot ||A||_{\max} \cdot ||B||_{\max}$$

where $\bar{z} = \binom{\omega}{t} \cdot \binom{\omega}{s}$, $\hat{v} = \binom{s+t}{s}$, $\bar{v} = \binom{s+t}{s+v} + \binom{s+t}{t+v}$, $\hat{w} = 2^{s+t} \cdot \binom{s+t}{s}$, and $m = \binom{n+v-1}{v} + O(n^{v-1})$.

Overall, these stability bounds show that the fast algorithm is numerically stable, as they are only larger than the standard algorithm by factors related to s, t, v, which are small constants. This observation is verified by our numerical tests, although for certain specially-picked tensors, the fast algorithm does incur more error than the standard algorithm (see Section 11.3.4).

11.3.3 Execution Cost Analysis

We begin by quantifying the number of arithmetic operations necessary for the new algorithm.

Theorem 11.3.4. *The floating point cost of algorithm $\hat{C} = \Phi_{\odot}^{(s,t,v)}(A, B)$ is, in the case of $v > 0$, in terms of the cost of multiplication of elements of A and B, μ_ϱ, and in terms of the cost of additions of elements of A, ν_ϱ^A, the cost of additions of elements of B, ν_ϱ^B, and the cost of additions of elements of C, ν_ϱ^C,*

$$F^{\Phi}(n,s,t,v,\varrho) = \frac{n^\omega}{\omega!}\cdot\left[\mu_\varrho + \binom{\omega}{t}\cdot\nu_\varrho^A + \binom{\omega}{s}\cdot\nu_\varrho^B + \binom{\omega}{v}\cdot\nu_\varrho^C\right] + \frac{n^{s+v}}{(s+v)!}\cdot\nu_\varrho^A + \frac{n^{t+v}}{(t+v)!}\cdot\nu_\varrho^B + \frac{n^{s+t}}{(s+t)!}\cdot\nu_\varrho^C + O(n^{\omega-1}\cdot(\mu_\varrho+\nu_\varrho^A+\nu_\varrho^B+\nu_\varrho^C)).$$

and in the case of $v = 0$ and $s, t > 0$,

$$F^{\Phi}(n,s,t,v,\varrho) \leq \frac{n^\omega}{\omega!}\cdot\left[\mu_\varrho + \binom{\omega}{t}\cdot\nu_\varrho^A + \binom{\omega}{s}\cdot\nu_\varrho^B + \binom{\omega}{v}\cdot\nu_\varrho^C\right] + \frac{1}{2}\binom{2(s+t)}{s+t}\frac{n^{s+t}}{(s+t)!}\cdot\nu_\varrho^C + O(n^{\omega-1}\cdot(\mu_\varrho+\nu_\varrho^A+\nu_\varrho^B+\nu_\varrho^C)).$$

Proof.

The number of multiplications necessary to compute Z via equation 11.2.4 is $n^\omega/\omega!$ to leading order. Naively, n^v multiplications are necessary for each of n^{s+t} elements of Z, for a total of n^ω. However, we can exploit the fact that the tensor of multiplications in equation 11.2.3 is the same no matter in which of $w!$ orders the indices $i\langle\omega\rangle$ appear. In particular, we can show that the tensor $\hat{Z}$ of order ω is fully symmetric and therefore has only $\binom{n+\omega-1}{\omega}$ unique entries. Since A and B are both fully symmetric, we can show that each operand stays the same under permutation of any pair of indices i_p and i_q in the $i\langle\omega\rangle$ index group. The terms in the summation forming each operand (we consider A but the same holds for B) fall into three cases:

- Both indices i_p and i_q in the pair appear in the term, in which case the term stays the same since the tensor is symmetric.
- One of two indices appear in some term, without loss of generality let i_p appear and not i_q, after permutation we can write this term as $A_{j\langle s+v-1\rangle i_p}$ where $j\langle s+v-1\rangle \in i\langle\omega\rangle$ and $j\langle s+v-1\rangle$ does not include i_p or i_q. Now, we can assert there is another term in the summation of the form $A_{j\langle s+v-1\rangle i_q}$, since $\chi(i\langle\omega\rangle)$ yields all possible ordered subsets of $i\langle\omega\rangle$, which must include an ordered index set containing the distinct indices $j\langle s+v-1\rangle i_q$.

- If neither of the two i_q and i_p indices appear, the term stays the same trivially.

The number of multiplications necessary to compute each one of V and U is $O(n^{\omega-1})$ and therefore low-order. Each multiplication in $\hat{Z}$ requires the addition of $\binom{\omega}{s+v} = \binom{\omega}{t}$ elements of A and $\binom{\omega}{t+v} = \binom{\omega}{s}$ elements of B. These summations may in some cases be amortized by reuse of partial sums to compute different elements of $\hat{Z}$, which is an optimization we ignore. Further each element unique multiplication (element of $\hat{Z}$) is accumulated to $\binom{\omega}{s+t} = \binom{\omega}{v}$ elements of Z. The accumulation of the $\binom{v+\omega-1}{\omega}$ entries of $\hat{Z}$, whose $\binom{\omega}{v}$ permutations, of the form $\hat{Z}_{i\langle s+t\rangle k\langle v\rangle}$ are added into $Z_{i\langle s+t\rangle}$ need to be multiplied by a scalar factor of r, where r is the number of possible unique order permutations of the index set $k\langle v\rangle$. When $k\langle v\rangle$ contains $l \leq v$ different values with multiplicities $m_1, m_2, \ldots m_l$, where $\sum_{i=1}^{l} m_i = v$, the scalar $r = \binom{v}{m_1, m_2, \ldots m_l}$ is a multinomial, which means that $r = v!/\prod_{i=1}^{l} m_i!$. This scalar factor is needed since for all unique $k'\langle v\rangle$, the entries $\hat{Z}_{i\langle s+t\rangle k'\langle v\rangle}$ where $k\langle v\rangle = \pi(k'\langle v\rangle)$ for some permutation π are part of the sum (equation 11.2.4) which contribute to $Z_{i\langle s+t\rangle}$, and all of these entries are the same due to the symmetry of $\hat{Z}$. One way to compute only the unique entries of $\hat{Z}$ and accumulate these accordingly to Z is to only compute those in normal order, i.e. $\hat{Z}_{i\langle\omega\rangle}$ for all $i\langle\omega\rangle = (i_1, i_2, \ldots i_\omega)$ such that $i_1 \leq i_2 \leq \ldots i_\omega$ We demonstrate this in Algorithm 11.3.1.

Algorithm 11.3.1 $Z \leftarrow$ Normal-ordered-Z-computation(A, B, n)

```
1:
2: % for all i⟨s + t⟩ in normal order...
3: for i_1 = 1 to n do
4:     for i_2 = i_1 to n do
5:         ⋱
6:         for i_{s+t} = i_{s+t−1} to n do
7:             Z_{i⟨s+t⟩} := 0
8:
9: % for all i⟨ω⟩ in normal order...
10: for i_1 = 1 to n do
11:     for i_2 = i_1 to n do
12:         ⋱
13:         for i_ω = i_{ω−1} to n do
14:             a_op = Σ_{j⟨s+v⟩∈χ(i⟨ω⟩)} A_{j⟨s+v⟩}
15:             b_op = Σ_{l⟨t+v⟩∈χ(i⟨ω⟩)} B_{l⟨t+v⟩}
16:             Ẑ_{i⟨ω⟩} = a_op · b_op
17:             for all (j⟨s + t⟩, k⟨v⟩) ∈ χ(i⟨ω⟩) do
18:                 Let r = v!/∏_{i=1}^{l} m_i! where k⟨v⟩ contains l ≤ v sets of identical entries of size
    m_i for i = [1, l]
19:                 Z_{j⟨s+t⟩} := Z_{j⟨s+t⟩} + r · Ẑ_{i⟨ω⟩}
```

Using Algorithm 11.3.1 (which does not exploit amortized summations), the cost of the additions needed to compute Z is no more than

$$\frac{n^\omega}{\omega!} \cdot \left[\binom{\omega}{t} \cdot \nu_\varrho^A + \binom{\omega}{s} \cdot \nu_\varrho^B + \binom{\omega}{v} \cdot \nu_\varrho^C \right].$$

The additions necessary to compute V, U, and W then accumulate then to C are low order with respect to those associated with Z, whenever $s, t, v > 0$. However, when $v = 0$, V does not need to be computed, but the number of additions necessary to compute W is not low order. When $v = 0$, $U^{(r)}$ is of order $s + t - r$ (and so is its computation cost), which is less than the order of the overall computation $O(n^{s+t})$, however, when it is accumulated into W, which is of order $s+t$, the number of additions necessary per element of W is $\binom{s+t}{r} \cdot \binom{s+t-r}{r}$, which is the number of permutations in the sum in equation 11.2.7. Over all r the number of additions necessary to compute W from each $U^{(r)}$ is

$$\begin{aligned}\binom{n+s+t-1}{s+t} \cdot \sum_{r=1}^{\min(s,t)} \binom{s+t}{r} \cdot \binom{s+t-r}{r} &\leq \binom{n+s+t-1}{s+t} \cdot \sum_{r=1}^{\min(s,t)} \binom{s+t}{r}^2 \\ &\leq \frac{1}{2}\binom{n+s+t-1}{s+t}\binom{2(s+t)}{s+t},\end{aligned}$$

where the upper bound is not particularly tight, especially when s and t are different. All of the above additions for W incur a cost of ν_ϱ^C. When $s = 0$ or $t = 0$, we do not need to compute W, but the number of additions needed to compute $A^{(1)}$ (in the case of $t = 0$) and $B^{(1)}$ (in the case of $s = 0$) is not lower order. To leading the costs of these additions are $\nu_\varrho^A \cdot n^{s+v}/(s+v)!$ and $\nu_\varrho^B \cdot n^{t+v}/(t+v)!$ in the two respective cases. All these costs are included in the equation for F^Φ in the theorem. □

We proceed to quantify the communication cost of the algorithm by specifying a lower and upper bound on the cost of a communication-optimal schedule.

Theorem 11.3.5. *The minimum communication cost of $\Phi_\odot^{(s,t,v)}(A, B)$ given a cache of size $\hat{M}$ is*

$$\begin{aligned}&\max\left(\left\lfloor \binom{n+\omega-1}{\omega} / (2\hat{M})^{\omega/\max(s+v,v+t,s+t)} \right\rfloor \cdot \hat{M}, \frac{n^{s+t}}{(s+t)!} + \frac{n^{s+v}}{(s+v)!} + \frac{n^{t+v}}{(t+v)!} \right) \\ &\leq \hat{W}^\Phi(n, s, t, v, \hat{M}) \leq \frac{n^\omega \cdot \left[\binom{\omega}{s} + \binom{\omega}{t} + \binom{\omega}{v}\right]^{1/\max(s+v,v+t,s+t)}}{\omega! \cdot \hat{M}^{-1+\omega/\max(s+v,v+t,s+t)}} + O(n^{s+t} + n^{s+v} + n^{t+v} + n^{\omega-1}),\end{aligned}$$

assuming that all elements of A, B, and C are of unit size.

Proof.

Upper bound: An upper bound on the communication cost of the algorithm may be obtained by providing a suitable execution schedule. Consider the following schedule, parameterized by b, which will be picked so that $b \ll n$, and where we assume b divides into n (otherwise the algorithm

can be executed for a problem of size $n+(n \bmod b)$ padded with zeros, with no greater asymptotic cost since $b \ll n$). We define a tensor $\tilde{A}$ whose elements are blocks of A of order $s+v$ and with dimension b. Each dimension of the tensor $\tilde{A}$ itself is n/b. Its elements are defined as follows $\forall I\langle s+v\rangle \in [0, n/b-1]^{s+v}$,

$$\tilde{A}^{I\langle s+v\rangle} = \{A_{i\langle s+v\rangle} \mid \forall\, i\langle s+v\rangle = (b \cdot I\langle s+v\rangle + \hat{i}\langle s+v\rangle), \hat{i}\langle s+v\rangle \in [1,b]^{s+v}\}$$

Similarly, we define blocked versions of B and $\hat{Z}$ (where $\hat{Z}$ is as before the tensor of multiplications needed to compute Z) as

$$\begin{aligned}
&\forall I\langle t+v\rangle \in [0, n/b-1]^{t+v},\\
&\qquad \tilde{B}^{I\langle t+v\rangle} = \{B_{i\langle t+v\rangle} \mid \forall\, i\langle t+v\rangle = (b \cdot I\langle t+v\rangle + \hat{i}\langle t+v\rangle), \hat{i}\langle t+v\rangle \in [1,b]^{t+v}\},\\
&\forall I\langle \omega\rangle \in [0, n/b-1]^{\omega},\\
&\qquad \tilde{Z}^{I\langle \omega\rangle} = \{\hat{Z}_{i\langle \omega\rangle} \mid \forall\, i\langle \omega\rangle \qquad = (b \cdot I\langle \omega\rangle + \hat{i}\langle \omega\rangle), \qquad \hat{i}\langle \omega\rangle \in [1,b]^{\omega}\}.
\end{aligned}$$

We can now reference the elements of the tensors A, B, and $\hat{Z}$ by the block to which they belong along with the offset inside the block, as $\forall i\langle\omega\rangle \in [1,n]^{\omega},\ \exists I\langle\omega\rangle \in [0, n/b-1]^{\omega}, \hat{i}\langle\omega\rangle \in [1,b]^{\omega}$, such that $i\langle\omega\rangle = b \cdot I\langle\omega\rangle + \hat{i}\langle\omega\rangle$, so

$$\hat{Z}_{i\langle\omega\rangle} = \hat{Z}_{b\cdot I\langle\omega\rangle + \hat{i}\langle\omega\rangle} = \tilde{Z}^{I\langle\omega\rangle}_{\hat{i}\langle\omega\rangle}.$$

Now, we express the multiplications necessary to compute $\hat{Z}$ according to the blocks of A and B whose input they require starting with equation 11.2.3, $\forall i\langle\omega\rangle = b \cdot I\langle\omega\rangle + \hat{i}\langle\omega\rangle$,

$$\begin{aligned}
\hat{Z}_{i\langle\omega\rangle} &= \left(\sum_{j\langle s+v\rangle \in \chi(i\langle\omega\rangle)} A_{j\langle s+v\rangle}\right) \cdot \left(\sum_{l\langle t+v\rangle \in \chi(i\langle\omega\rangle)} B_{l\langle t+v\rangle}\right)\\
&= \tilde{Z}^{I\langle\omega\rangle}_{\hat{i}\langle\omega\rangle} = \left(\sum_{(b\cdot J\langle s+v\rangle + \hat{j}\langle s+v\rangle) \in \chi(b\cdot I\langle\omega\rangle + \hat{i}\langle\omega\rangle)} \tilde{A}^{J\langle s+v\rangle}_{\hat{j}\langle s+v\rangle}\right) \cdot \left(\sum_{(b\cdot L\langle t+v\rangle + \hat{l}\langle t+v\rangle) \in \chi(b\cdot I\langle\omega\rangle + \hat{i}\langle\omega\rangle)} \tilde{B}^{L\langle t+v\rangle}_{\hat{l}\langle t+v\rangle}\right),
\end{aligned} \tag{11.3.1}$$

where in the partitioning $(b\cdot J\langle s+v\rangle + \hat{j}\langle s+v\rangle) \in \chi(b\cdot I\langle\omega\rangle + \hat{i}\langle\omega\rangle)$, each value in $J\langle s+v\rangle$ takes on a value of $I\langle\omega\rangle$ of the same index (position in the tuple) as the position of the value of $\hat{i}\langle s+v\rangle$ which is taken on by $\hat{j}\langle s+v\rangle$. The same applies to $(b\cdot L\langle t+v\rangle + \hat{l}\langle t+v\rangle) \in \chi(b\cdot I\langle\omega\rangle + \hat{i}\langle\omega\rangle)$. So for a given block (given value of $I\langle\omega\rangle$), the number of blocks of A and B which contain elements that are necessary to compute $\tilde{Z}^{I\langle\omega\rangle}_{\hat{i}\langle\omega\rangle}$ for all $\hat{i}\langle\omega\rangle$, is equal to the number of unique values $J\langle s+v\rangle$ and $L\langle s+v\rangle$ can take on. Since the partition function acts independently on these block indices ($J\langle s+v\rangle$ and $L\langle s+v\rangle$ are subsets of $I\langle\omega\rangle$) and the inner-block indices ($\hat{j}\langle s+v\rangle$ and $\hat{l}\langle s+v\rangle$ are subsets of $\hat{i}\langle\omega\rangle$), the values which $J\langle s+v\rangle$ can take on are given by the set $\chi^{s+v}(I\langle\omega\rangle)$, while the values which $L\langle t+v\rangle$ can take on are given by the set $\chi^{t+v}(I\langle\omega\rangle)$. These sets have sizes

$\binom{\omega}{s+v} = \binom{\omega}{t}$ and $\binom{\omega}{t+v} = \binom{\omega}{s}$, respectively. Therefore, to compute all elements of a block $\hat{Z}^{I\langle\omega\rangle}$ for any $I\langle\omega\rangle$, the size of the inputs needed (the number of reads from memory needed) is

$$\hat{W}_{\text{in}} = \binom{\omega}{t} \cdot b^{s+v} + \binom{\omega}{s} \cdot b^{t+v},$$

since the size of each block of A (element of $\tilde{A}$) is b^{s+v} while the size of each block of B (element of $\tilde{B}$) is b^{t+v}.

Z and subsequently C is computed from $\hat{Z}$ via equation 11.2.4. We transform equation 11.2.4 (with Z replaced by Q for notational convenience) into a blocked version by expressing it in terms of the elements of $\tilde{Z}$ and $\tilde{Q}$, where the block-tensor $\tilde{Q}$ is defined as

$$\begin{aligned}
&\forall I\langle s+t\rangle \in [0, n/b-1]^{s+t}, \\
&\quad \tilde{Q}^{I\langle s+t\rangle} = \{Q_{i\langle s+t\rangle} \mid \forall\, i\langle s+t\rangle = (b \cdot I\langle s+t\rangle + \hat{i}\langle s+t\rangle), \hat{i}\langle s+t\rangle \in [1,b]^{s+t}\}.
\end{aligned}$$

Equation 11.2.4 becomes $\forall i\langle s+t\rangle = b \cdot I\langle s+t\rangle + \hat{i}\langle s+t\rangle$,

$$\begin{aligned}
Q_{i\langle s+t\rangle} &= \sum_{k\langle v\rangle} \hat{Z}_{i\langle s+t\rangle k\langle v\rangle} \\
&= \tilde{Q}^{I\langle s+t\rangle}_{\hat{i}\langle s+t\rangle} = \sum_{b\cdot K\langle v\rangle + \hat{k}\langle v\rangle} \hat{Z}^{I\langle s+t\rangle K\langle v\rangle}_{\hat{i}\langle s+t\rangle \hat{k}\langle v\rangle}.
\end{aligned} \tag{11.3.2}$$

We now show that by exploiting symmetry in $\hat{Z}$ only a fraction of the blocks (elements of $\tilde{Z}$) actually need to be computed, with each one being accumulated into $\binom{\omega}{v}$ blocks of Q (elements of $\tilde{Q}$) corresponding to the blocks $\tilde{Q}^{I\langle s+t\rangle}$ given by the partitions $(I\langle s+t\rangle, K\langle v\rangle) \in \chi(J\langle\omega\rangle)$. The symmetry of $\hat{Z}$ was formally demonstrated in the proof of Theorem 11.3.4. It follows that the elements of block $\tilde{Z}^{I\langle\omega\rangle}$ are the same (but differently ordered) as those of $\tilde{Z}^{J\langle\omega\rangle}$ whenever $I\langle\omega\rangle$ and $J\langle\omega\rangle$ are equivalent after some permutation of the block indices $J\langle\omega\rangle = \pi(I\langle\omega\rangle)$. This is true since for any element of the first block, $\tilde{Z}^{I\langle\omega\rangle}_{\hat{i}\langle\omega\rangle} = \hat{Z}_{b\cdot I\langle\omega\rangle + \hat{i}\langle\omega\rangle}$, there is an equivalent element $\tilde{Z}^{J\langle\omega\rangle}_{\pi(\hat{i}\langle\omega\rangle)} = \tilde{Z}^{\pi(I\langle\omega\rangle)}_{\pi(\hat{i}\langle\omega\rangle)} = \hat{Z}_{\pi(b\cdot I\langle\omega\rangle + \hat{i}\langle\omega\rangle)}$, due to the fact that $\hat{Z}$ is symmetric (symmetry means that for any permutation of indices π', $\hat{Z}_{i\langle\omega\rangle} = \hat{Z}_{\pi'(i\langle\omega\rangle)}$). Now, since $Q = Z$ is symmetric, it similarly follows that the elements of each block $\tilde{Q}^{I\langle s+t\rangle}$ are the same as of any block $\tilde{Q}^{J\langle s+t\rangle}$ when $I\langle s+t\rangle$ and $J\langle s+t\rangle$ are equivalent after permutation. Therefore, it suffices to compute only blocks $\tilde{Q}^{I\langle s+t\rangle}$ where $I\langle s+t\rangle$ is in normal order ($I_1 \leq I_2 \leq \ldots I_{s+t}$). Further, these blocks of Q can be computed from only the blocks of $\tilde{Z}^{J\langle\omega\rangle}$ for which $J\langle\omega\rangle$ is in normal order. In particular, whenever our schedule computes each normal-ordered block, $\tilde{Z}^{J\langle\omega\rangle}$, it accumulates the block to all normal-ordered blocks of Q to which certain permutations of the indices $J\langle\omega\rangle$ contributes. These permutations correspond to partitions of the index set $J\langle\omega\rangle$ into $I\langle s+t\rangle$ and $K\langle v\rangle$, which are given by the set $\chi^{s+t}_v(J\langle\omega\rangle)$, the size of which is $\binom{\omega}{v}$. So, for each block of $\hat{Z}$ the schedule computes, it accumulates to memory (which we allow to be done with a unit memory transfer cost)

$$\hat{W}_{\text{out}} = \binom{\omega}{v} \cdot b^{s+t},$$

elements of Q, since each block of Q is of size b^{s+t}.

Algorithm 11.3.2 $\tilde{Q} \leftarrow$ symmetric-schedule$(\tilde{A}, \tilde{B}, s, t, v)$

for all $I\langle s+t\rangle \in [0, n/b-1]^{\omega}$, such that $I_1 \leq I_2 \leq \ldots I_{s+t}$ **do**
  Initialize block $\tilde{Q}^{I\langle s+t\rangle}$ to zero
for all $J\langle\omega\rangle \in [0, n/b-1]^{\omega}$, such that $J_1 \leq J_2 \leq \ldots J_{\omega}$ **do**
  Load into cache all blocks $\tilde{A}^{I\langle s+v\rangle}$ for all $I\langle s+v\rangle \in \chi(J\langle\omega\rangle)$
  Load into cache all blocks $\tilde{B}^{I\langle t+v\rangle}$ for all $I\langle t+v\rangle \in \chi(J\langle\omega\rangle)$
  Compute all elements in block $\tilde{Z}^{J\langle\omega\rangle}$ via equation 11.3.1
  for all $(I\langle s+t\rangle, K\langle v\rangle) \in \chi(J\langle\omega\rangle)$ **do**
    Reorder the elements of $\tilde{Z}^{J\langle\omega\rangle}$ to obtain the block $\tilde{Z}^{I\langle s+t\rangle K\langle v\rangle}$
    Let r be the number of unique possible orderings of the set of values in $K\langle v\rangle$
    for all $\hat{i}\langle s+t\rangle \in [1, b]^{s+t}$ **do**
      Accumulate $\tilde{Q}^{I\langle s+t\rangle}_{\hat{i}\langle s+t\rangle} := \tilde{Q}^{I\langle s+t\rangle}_{\hat{i}\langle s+t\rangle} + r \cdot \sum_{\hat{k}\langle v\rangle} \tilde{Z}^{I\langle s+t\rangle K\langle v\rangle}_{\hat{i}\langle s+t\rangle \hat{k}\langle v\rangle}$

We formulate the above description into Algorithm 11.3.2. We argue that at termination the normal-ordered blocks of Q (elements of $\tilde{Q}$) which Algorithm 11.3.2 computes are equivalent to equation 11.3.2. Each element, $\tilde{Q}^{I\langle s+t\rangle}_{\hat{i}\langle s+t\rangle}$ is a sum of n^v elements of $\hat{Z}$, which belong to elements of $(n/b)^v$ blocks, $\tilde{Z}^{I\langle s+t\rangle K\langle v\rangle}$, for the $(n/b)^v$ values that $K\langle v\rangle$ can take on. We can observe that the contributions to $\tilde{Q}^{I\langle s+t\rangle}_{\hat{i}\langle s+t\rangle}$ of $\tilde{Z}^{I\langle s+t\rangle K\langle v\rangle}$ and $\tilde{Z}^{I\langle s+t\rangle K'\langle v\rangle}$ for any permutationally equivalent $K'\langle v\rangle = \pi(K\langle v\rangle)$ are the same due to the symmetry in $\hat{Z}$, since

$$\begin{aligned}
\sum_{\hat{k}\langle v\rangle} \tilde{Z}^{I\langle s+t\rangle K'\langle v\rangle}_{\hat{i}\langle s+t\rangle \hat{k}\langle v\rangle} &= \sum_{\hat{k}\langle v\rangle} \hat{Z}_{b\cdot I\langle s+t\rangle K'\langle v\rangle + \hat{i}\langle s+t\rangle \hat{k}\langle v\rangle} \\
&= \sum_{\hat{k}\langle v\rangle} \hat{Z}_{(b\cdot I\langle s+t\rangle + \hat{i}\langle s+t\rangle)(K'\langle v\rangle + \hat{k}\langle v\rangle)} \\
&= \sum_{\pi(\hat{k}\langle v\rangle)} \hat{Z}_{(b\cdot I\langle s+t\rangle + \hat{i}\langle s+t\rangle)\pi(K'\langle v\rangle + \hat{k}\langle v\rangle)} \\
&= \sum_{\hat{l}\langle v\rangle} \hat{Z}_{(b\cdot I\langle s+t\rangle + \hat{i}\langle s+t\rangle)(K\langle v\rangle + \hat{l}\langle v\rangle)} \\
&= \sum_{\hat{l}\langle v\rangle} \tilde{Z}^{I\langle s+t\rangle K\langle v\rangle}_{\hat{i}\langle s+t\rangle \hat{l}\langle v\rangle}
\end{aligned}$$

This equivalence justifies the multiplication by a factor of r in Algorithm 11.3.2 which is the number of unique possible orderings of $K\langle v\rangle$, since by the argument above the contributions of all these blocks for each element of Q is the same. This factor is different depending on how many values in the index set $K\langle v\rangle$ are the same (diagonals need to be multiplied by different factors, as is also done in the standard algorithm, $\Psi^{(s,t,v)}_{\odot}(A, B)$, when $v > 1$). When $K\langle v\rangle$ contains $l \leq v$

different values with multiplicities $m_1, m_2, \ldots m_l$, $\sum_{i=1}^{l} m_i = v$, r is a multinomial, which means that $r = v!/\prod_{i=1}^{l} m_i!$.

We can then argue that all $(n/b)^v$ blocks which contribute to each element of Q in equation 11.3.2 are accounted for, since for each index to $\tilde{Z}$ in equation 11.3.2, $I\langle s+t\rangle K\langle v\rangle$, we know that

- $I\langle s+t\rangle$ will be in normal-order since we only compute blocks of Q whose indices are in normal order,
- there exists a block $\tilde{Z}^{I\langle s+t\rangle K'\langle v\rangle}$ where $K'\langle v\rangle = \pi(K\langle v\rangle)$ for some permutation π such that $K'\langle v\rangle$ is in normal order and whose contribution is the same as that of $\tilde{Z}^{I\langle s+t\rangle K\langle v\rangle}$ due to the symmetry argument given above, and there are r such blocks which share the same normal order after permutation (distinct possible orderings of $K\langle v\rangle$),
- there exists a block $\tilde{Z}^{J\langle\omega\rangle}$ where $J\langle\omega\rangle$ is in normal order and permutationally equivalent to $I\langle s+t\rangle K'\langle v\rangle$ ($\exists\pi$, such that $\pi(I\langle s+t\rangle K'\langle v\rangle) = J\langle\omega\rangle$),
- Algorithm 11.3.2 computes block $\tilde{Z}^{J\langle\omega\rangle}$ since it computes all blocks of $\hat{Z}$ with a normal-ordered index,
- Algorithm 11.3.2 adds the correct elements of the block $\tilde{Z}^{J\langle\omega\rangle}$ to all elements in $\tilde{Q}^{I\langle s+t\rangle}$ since $(I\langle s+t\rangle, K'\langle v\rangle) \in \chi(J\langle\omega\rangle)$ with the desired prefactor of r accounting for all distinct possible orderings of $K'\langle v\rangle$.

We can quantify the communication cost of Algorithm 11.3.2 by multiplying the number of blocks of $\hat{Z}$ it computes by the size of the inputs $\hat{W}_{\text{in}}$ needed to compute the block and adding this to the size of the outputs $\hat{W}_{\text{out}}$ to which the result of the block is accumulated. Only the blocks of $\hat{Z}$ which have normal-ordered indices are computed, and the total number of such blocks is $\binom{n/b+\omega-1}{\omega}$. Assuming accumulation to memory and reads from memory have unit cost and that the inputs and outputs all fit into cache, we then obtain the following expression for the communication cost of Algorithm 11.3.2,

$$\begin{aligned}\hat{W}_Z(n,s,t,v,b) &= \binom{n/b+\omega-1}{\omega}(\hat{W}_{\text{in}} + \hat{W}_{\text{out}}) \\ &= \frac{n^\omega}{\omega!\cdot b^\omega}\left[\binom{\omega}{t}\cdot b^{s+v} + \binom{\omega}{s}\cdot b^{t+v} + \binom{\omega}{v}\cdot b^{s+t}\right] + O(n^{\omega-1})\end{aligned}$$

The amount of cache used by the algorithm throughout the computation of each block of $\hat{Z}$ is

$$\hat{W}_{\text{in}} + \hat{W}_{\text{out}} = \binom{\omega}{t}\cdot b^{s+v} + \binom{\omega}{s}\cdot b^{t+v} + \binom{\omega}{v}\cdot b^{s+t},$$

which we need to be less than the size of the cache $\hat{M}$. Therefore, we pick b to be

$$b = \left(\hat{M}/\left[\binom{\omega}{s} + \binom{\omega}{t} + \binom{\omega}{v}\right]\right)^{1/\max(s+v,v+t,s+t)}$$

Substituting this b into $\hat{W}_Z(n,s,t,v,b)$, we obtain the following upper bound on the communication cost of Algorithm 11.3.2

$$\begin{aligned}\hat{W}_Z(n,s,t,v,\hat{M}) \leq & \frac{n^\omega \cdot \left[\binom{\omega}{s}+\binom{\omega}{t}+\binom{\omega}{v}\right]^{1/\max(s+v,v+t,s+t)}}{\omega! \cdot \hat{M}^{\omega/(\max(s+v,v+t,s+t))}} \cdot \hat{M} + O(n^{\omega-1}) \\ = & \frac{n^\omega \cdot \left[\binom{\omega}{s}+\binom{\omega}{t}+\binom{\omega}{v}\right]^{1/\max(s+v,v+t,s+t)}}{\omega! \cdot \hat{M}^{-1+\omega/\max(s+v,v+t,s+t)}} + O(n^{\omega-1})\end{aligned}$$

We now consider the communication costs associated with computing the other intermediates, $A^{(r)}$, $B^{(r)}$, V, $U^{(r)}$, and W. As we noted in the proof of Theorem 11.3.4 the number of operations required to compute these is $O(n^{\omega-1})$ whenever $s,v,t>0$, so the communication cost in this case is also bound by $O(n^{\omega-1})$. When $v=0$, computing W is no longer a low-order cost, but we can bound it by $O(n^{s+t})$ (since this is a bound on the number of operations needed to compute W), which also bounds the cost of writing the results into the output C. In the cases when either $s=0$ or $t=0$, the computation of $B^{(1)}$ and $A^{(1)}$, respectively, become leading order terms. We can, however, also bound these costs by the size of the inputs $O(n^{s+v}+n^{t+v})$. Combining these bounds with the bound for Algorithm 11.3.2, which computes $Z=Q$, yields the following upper bound on the communication cost of the fast tensor contraction algorithm:

$$\hat{W}^{\Phi}(n,s,t,v,\hat{M}) = \frac{n^\omega \cdot \left[\binom{\omega}{s}+\binom{\omega}{t}+\binom{\omega}{v}\right]^{1/\max(s+v,v+t,s+t)}}{\omega! \cdot \hat{M}^{-1+\omega/\max(s+v,v+t,s+t)}} + O(n^{s+t}+n^{s+v}+n^{t+v}+n^{\omega-1}).$$

Lower bound: Now we prove the communication lower bound on algorithm $\Phi_{\odot}^{(s,t,v)}(A,B)$. We apply the lower bound technique from [37], which relies on the Hölder inequality [84] and its generalization [22]. Similar lower-bound-motivated generalizations of these inequalities were also provided by Tiskin [156]. Specifically, we employ Theorem 6.6 from Section 6.3 of [37], which applies to programs which are loop nests where in the innermost loop arrays are accessed based on subsets of the loop indices. Algorithm 11.3.1 is exactly this type of program, with ω nested loops and with $\binom{\omega}{t}$ accesses to the A array, $\binom{\omega}{s}$ accesses to the B array, and $\binom{\omega}{v}$ accesses to the Z array in the innermost loop. Each of these $\binom{\omega}{t}+\binom{\omega}{s}+\binom{\omega}{v}$ accesses is a projection or homomorphism in the language of [37], which is a mapping from the index set $i\langle\omega\rangle$ to a subset thereof. We enumerate these homomorphisms for accesses to A by enumerating the set $\chi^{s+v}(i\langle\omega\rangle)$, with $\phi_j^A(i\langle\omega\rangle)$ being the jth member of the set for $j\in[1,\binom{\omega}{t}]$. We similarly enumerate the access to B by enumerating the set $\chi^{t+v}(i\langle\omega\rangle)$, with $\phi_j^B(i\langle\omega\rangle)$ being the jth member of the set for $j\in[1,\binom{\omega}{s}]$. Lastly, we enumerate the accesses (accumulates) to Z by enumerating the set $\chi^{s+t}(i\langle\omega\rangle)$, with $\phi_j^B(i\langle\omega\rangle)$ being the jth member of the set for $j\in[1,\binom{\omega}{v}]$. Now, we ignore two of the three of these sets of projections (ignoring some dependencies still allows us to obtain a valid lower bound in this case), namely we pick $r=\min(s,v,t)$, then if $r=t$ we take the projections from A, otherwise if $r=s$ we take the projections from B, and if $r<s,t$, so $r=v$, we take the projections from Z. We end up with projections $\phi_j(i\langle\omega\rangle)$, which enumerate the set $\chi^{\omega-r}(i\langle\omega\rangle)$ for $j\in[1,\binom{\omega}{r}]$.

We would like to use the implication of Theorem 3.2 of [37], which states that if the projections ϕ_j for $j\in[1,\binom{\omega}{r}]$ satisfy a certain constraint for parameters $s_j\in[0,1]$, then given any set of index

tuples (loop iterations) $E = \{i\langle\omega\rangle \in [1,n]^\omega\}$,

$$|E| \leq \prod_{j=1}^{\binom{\omega}{r}} |\phi_j(E)|^{s_j} \tag{11.3.3}$$

So, the exponents s_j bound the amount of iterations which may be computed with a given set of inputs or outputs based on the projection of the iteration set onto the inputs or outputs. The constraint which binds the choice of s_j is significantly simplified from Theorem 3.2 of [37] by Theorem 6.6 of [37], which applies for our program. In our case, the constraint is (taking the form from the proof of the theorem),

$$\forall k \in [1,\omega],\ \sum_{j=1}^{\binom{\omega}{r}} s_j \cdot \Delta_{jk} \geq 1, \tag{11.3.4}$$

where Δ is an $\binom{\omega}{r}$-by-ω matrix, with $\Delta_{jk} = 1$ if $i_k \in \phi_j(i\langle\omega\rangle)$ and $\Delta_{jk} = 0$ otherwise. In our case, Δ has $\omega - r$ ones per row, since each ϕ_j takes a distinct subset of $\omega - r$ indices. Further, since each index appears in the same number of subsets (elements of the set $\chi^{\omega-r}(i\langle\omega\rangle)$), the number of ones per column in Δ is balanced and therefore equal to the number of ones per row multiplied by the number of rows and divided by the number of columns,

$$\sum_{j=1}^{\binom{\omega}{r}} \Delta_{jk} = (\omega - r) \cdot \binom{\omega}{r} / \omega = \binom{\omega - 1}{r - 1}.$$

So, we can satisfy the constraint in equation 11.3.4 by selecting $s_j = 1/\binom{\omega-1}{r-1}$ for all $j \in [1, \binom{\omega}{r}]$.

Now, for any schedule (ordering) of the $\binom{n+\omega-1}{\omega}$ iterations of Algorithm 11.3.1, we subdivide the iterations into subsets E_i for $i \in [1, f]$ where $f = \lceil \binom{n+\omega-1}{\omega} / (2\hat{M})^{\omega/(\omega-r)} \rceil$ and the first $f - 1$ sets are of size $(2\hat{M})^{\omega/(\omega-r)}$ while the last remainder subset of size

$$|E_f| = \binom{n+\omega-1}{\omega} - (2\hat{M})^{\omega/(\omega-r)} \cdot \left\lfloor \binom{n+\omega-1}{\omega} / (2\hat{M})^{\omega/(\omega-r)} \right\rfloor \leq (2\hat{M})^{\omega/(\omega-r)},$$

which may be zero. Now, for each E_i we have the bound from equation 11.3.3,

$$\begin{aligned} |E_i| &\leq \prod_{j=1}^{\binom{\omega}{r}} |\phi_j(E_i)|^{s_j} \\ &\leq \prod_{j=1}^{\binom{\omega}{r}} |\phi_j(E_i)|^{1/\binom{\omega-1}{r-1}}. \end{aligned}$$

If we are considering projections onto Z (when $r = v < s, t$), then the size of each $\phi_j(E_i)$ would be bound by the cache size $\hat{M}$ (partial sums kept in cache at the end of segment) added to the number of writes/accumulates done to memory $\hat{W}_i$ during segment i for $i < f$, since we assume no recomputation, meaning all partial sums computed in the last chunk have to be written to memory during the last chunk. However, if we are considering projections onto A or B, the size of each $\phi_j(E_i)$ would be bound by the cache size $\hat{M}$ (operands from the previous segment) added to the number of reads from memory $\hat{W}_i$ done during chunk i for all $i > 1$ and exclusively $\hat{W}_i$ for $i = 1$, since we assume the cache begins empty. We note that in these latter cases (operand projections), we can consider the schedule ordering in reverse so that the remainder chunk be the last chunk in reverse order (first chunk actually executed), the bounds on $\phi_j(E_i)$ become the same as in the case when we are considering projections of the partial sums to Z. So, without loss of generality, we assume the bounds $\phi_j(E_i) \leq \hat{M} + \hat{W}_i$ for all $i < f$ and $\phi_j(E_f) \leq \hat{W}_f$. We now seek to obtain a lower bound on communication cost by obtaining a lower bound on $\sum_{i=1}^{f} \hat{W}_i$. We ignore the remainder chunk and obtain a bound for each $i \in [1, f-1]$,

$$\begin{aligned}
|E_i| &\leq \prod_{j=1}^{\binom{\omega}{r}} |\phi_j(E_i)|^{1/\binom{\omega-1}{r-1}}, \\
(2\hat{M})^{\omega/(\omega-r)} &\leq (\hat{M} + \hat{W}_i)^{\binom{\omega}{r}/\binom{\omega-1}{r-1}} \\
&= (\hat{M} + \hat{W}_i)^{\omega/(\omega-r)} \\
2\hat{M} &\leq \hat{M} + \hat{W}_i, \\
\hat{W}_i &\geq \hat{M}
\end{aligned}$$

We can then obtain a lower bound on the communication cost for Algorithm 11.3.1, by summing over the $f - 1$ iterations of full chunks,

$$\begin{aligned}
\hat{W}^{\Phi}(n, s, t, v, \hat{M}) \geq (f-1) \cdot \hat{M} &= \left\lfloor \binom{n+\omega-1}{\omega} / (2\hat{M})^{\omega/(\omega-r)} \right\rfloor \cdot \hat{M} \\
&= \left\lfloor \binom{n+\omega-1}{\omega} / (2\hat{M})^{\omega/\max(s+v, v+t, s+t)} \right\rfloor \cdot \hat{M}.
\end{aligned}$$

We augment this lower bound, by combining it with a lower bound that is based purely on the size of A and B, each of whose entries must be read into cache at least once and C, whose entries must be written to memory at least once, obtaining,

$$\begin{aligned}
\hat{W}^{\Phi}(n, s, t, v, \hat{M}) \geq \max \Bigg(& \left\lfloor \binom{n+\omega-1}{\omega} / (2\hat{M})^{\omega/\max(s+v, v+t, s+t)} \right\rfloor \cdot \hat{M}, \\
& \frac{n^{s+t}}{(s+t)!} + \frac{n^{s+v}}{(s+v)!} + \frac{n^{t+v}}{(t+v)!} \Bigg).
\end{aligned}$$

□

We observe that communication cost of the fast algorithm achieves asymptotically less reuse than the standard algorithm when $s, t, v > 0$ and s, t, v are unequal, because in these cases the exponent on $\hat{M}$ corresponding to the cache reuse obtained for each element by the fast algorithm is $-1 + \omega/\max(s+v, t+v, s+t) < 1/2$ (the standard algorithm gets $\hat{M}^{1/2}$ reuse in these case). In the cases when one of s, t, v is zero, reading in one of the input tensors or writing the output tensor becomes the asymptotically dominant cost for both algorithms. When $s = t = v > 0$, $\Psi_{\odot}^{(s,t,v)}(A, B)$ achieves the same asymptotic reuse factor of $\hat{M}^{1/2}$ and performs less communication by a constant factor due to performing a factor of $\omega!/(s!t!v!)$ less operations, albeit the benefit in communication does not achieve this full constant factor reduction (there is an $\omega!$ in the denominator, but additional, smaller constant factors in the numerator) since it requires more communication per computation. However, in many of the key applications of the algorithm, where each scalar multiplication done by the fast algorithm is an asymptotically more expensive operation than each scalar addition (e.g. a matrix multiplication), the reduction in the number of multiplications induces a reduction in communication cost by the same factor, since the underlying scalar operation (e.g. matrix multiplication) may achieve a factor of $\hat{M}^{1/2}$ reuse itself. The communication cost analysis done in this section does not correctly capture the cost in such cases, since it assumes scalars are of unit size that is much smaller than the cache size.

The extension of these communication results to the parallel case is straight-forward since the computation of each scalar multiplication in the algorithm may be done in parallel, as with matrix multiplication. We state a memory-independent parallel communication lower bound in Theorem 11.3.6, whose proof demonstrates that the dependency graph of $\Phi^{(s,t,v)}$ (computed as Algorithm 11.3.1) is structurally an $(\omega, \max(s+v, v+t, s+t))$-lattice hypergraph by the definition in Chapter 3.

Theorem 11.3.6. *The minimum communication cost of a load-balanced parallelization of the fast algorithm $\Phi_{\odot}^{(s,t,v)}(A, B)$ on a homogeneous machine with p processors each with local memory M is*

$$W^{\Phi}(n, s, t, v, p) = \Omega\left(\frac{n^{\max(s+v,v+t,s+t)}}{p^{\max(s+v,v+t,s+t)/\omega}}\right).$$

Proof. Consider any dependency graph $G_{\Phi} = (V_{\Phi}, E_{\Phi})$ that computes Algorithm 11.3.1. It must compute all multiplications $\hat{Z}_{i\langle\omega\rangle}$ for all ordered $i\langle\omega\rangle$. Each $\hat{Z}_{i\langle\omega\rangle}$ is a dependent on all $A_{j\langle s+v\rangle}$ where $j\langle s+v\rangle \in i\langle\omega\rangle$ and on all $B_{l\langle t+v\rangle}$ where $l\langle t+v\rangle \in i\langle\omega\rangle$, and contributes to the sum forming (is a dependency of) $Z_{j\langle s+t\rangle}$ ($C_{j\langle s+t\rangle}$) for all $j\langle s+t\rangle \in i\langle\omega\rangle$. We can therefore define a parent $(\omega, \max(s+v, v+t, s+t))$-lattice hypergraph of breadth n (as defined in Section 3.3) $H_{\Phi} = (V_{\Phi}^H, E_{\Phi}^H)$ where the vertices contain all entries $\hat{Z}_{i\langle\omega\rangle} \in V_{\Phi}^H$ with $i_1 < \cdots < i_{\omega}$. Each hyperedge $e_{j\langle\max(s+v,v+t,s+t)\rangle} \in E_{\Phi}^H$ corresponds to the highest order projection (if $s+v$ largest then A, otherwise if $v+t$ largest then B, and otherwise if $s+t$ is largest then C) and contains $e_{j\langle\max(s+v,v+t,s+t)\rangle} = \{\hat{Z}_{i\langle\omega\rangle} \in V_{\Phi}^H : j\langle\max(s+v, v+t, s+t)\rangle \subset i\langle\omega\rangle\}$. These hyperedges are edge disjoint connected partitions, since each corresponds either to all dependents of an operand tensor entry or to all dependencies of an output entry $Z_{j\langle s+t\rangle}$, which must be combined via some reduction tree (the intermediate sums in the reduction tree are ignored in a par-

ent hypergraph as in Chapter 5). Since we assume the computation is load balanced and V_Φ^H contains $\Theta(n^\omega)$ vertices, some processor must compute no more than $\lfloor |V_\Phi^H|/4 \rfloor$ vertices and at least $\lfloor |V_\Phi^H|/p \rfloor$. By Theorem 3.3.1, the minimum $\frac{1}{p}$-$\frac{1}{4}$-balanced hyperedge cut of H_Φ is of size $\epsilon_p(H_\Phi) = \Omega(n^{\max(s+v,v+t,s+t)}/p^{\max(s+v,v+t,s+t)/\omega})$. Further by Theorem 3.3.2, since H_Φ is a parent hypergraph of any dependency graph G_Φ and has degree at most $\binom{\omega}{\max(s+v,v+t,s+t)}$ the minimum $\frac{1}{p}$-$\frac{1}{4}$-balanced vertex separator of any G_Φ is of size at least

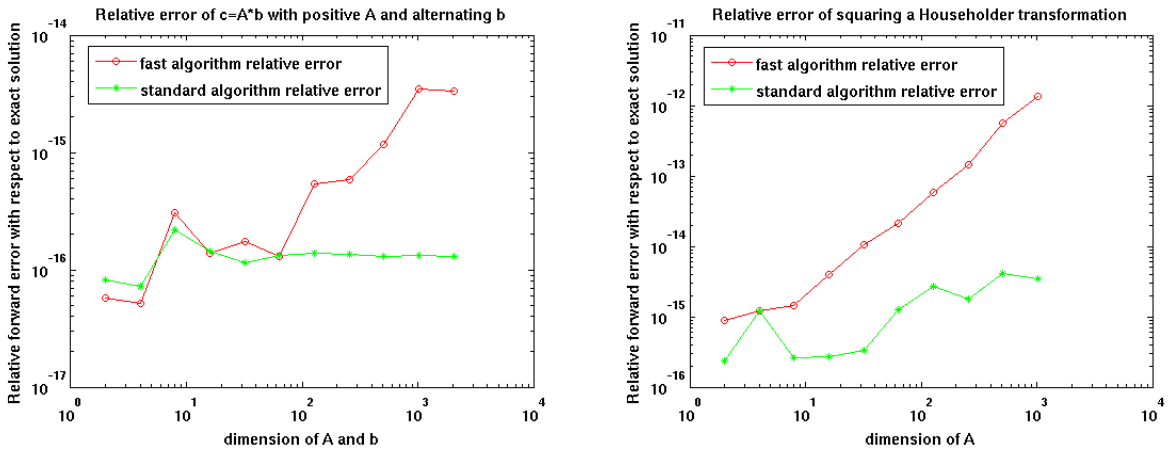
$$\chi_{p,4}(G_\Phi)(n,s,t,v) = \Omega(n^{\max(s+v,v+t,s+t)}/p^{\max(s+v,v+t,s+t)/\omega}).$$

Since any communicated set is a $\frac{1}{p}$-$\frac{1}{4}$-balanced vertex separator of G_Φ (by definition in Section 2.1.1), the lower bound on separator size is also a lower bound on communication cost for any load-balanced execution of $\Phi^{(s,t,v)}$,

$$W^\Phi(n,s,t,v,p) = \Omega\left(\frac{n^{\max(s+v,v+t,s+t)}}{p^{\max(s+v,v+t,s+t)/\omega}}\right).$$

□

11.3.4 Numerical Tests



(a) Multiplication of a symmetric matrix by a vector (b) Symmetrized multiplication of symmetric matrices

Figure 11.1: Numerical stability tests stressing the potential error arising from the intermediates formed by the fast algorithm.

We identify two scenarios in which the fast symmetric tensor contraction algorithm incurs higher numerical floating point error than the standard algorithm. In the first case, we test the algorithms for multiplying a symmetric matrix by a vector (Section 11.2.4). We tested two cases, an A matrix with values in the range $[-.5, .5]$ and a random A matrix with values in the range

$[0, 1]$. In both cases, we pick entries of vector b to be in the range $[-.5, .5]$. We quantify the relative difference between the standard and the fast symmetric algorithm and the analytically obtained exact solution. In particular we compute

$$\frac{||fl\,(c) - c||_2}{||A||_2 \cdot ||b||_2},$$

where $fl\,(c)$ is the answer numerically computed by $\Psi(A, b)$ and $\Phi(A, b)$. In the first case, the error was roughly proportional to that of running two versions of the standard algorithm with sums performed in different order, suggesting that the symmetric algorithm does not incur a noticeable amount of extra error. However, in the latter case (A strictly positive), the error for the symmetric algorithm is larger and growing with matrix dimension. This error arises from the fact that entries of the V intermediate are dependent on partial sums of A ($V_i = (\sum_j A_{ij}) \cdot b_i$), which are large due to the fact that A is positive. Since, V is designed to cancel terms in Z, an error is incurred due to cancellation, as the magnitude of the result is small with respect to the magnitude of V. Figure 11.1(a) displays an adaptation of this test, with A picked to be positive, but non-random, in particular $A_{ij} = (2i + j - 2 \mod n) \cdot .3 + .001$ and b picked to be alternating, in particular $b_i = .1$ for $i \in \{2, 4, \ldots\}$ and $b_i = -.1$ for $i \in \{1, 3, \ldots\}$. These non-random values were picked to obtain an exact result as a basis of comparison, as in this case, $c = A \cdot b$, has entries $c_i = -.015 \cdot n$ for all i.

The second scenario in which the fast symmetric algorithm incurs more error is specially designed to make the magnitude of the U intermediate large with respect to the answer. Figure 11.1(b) demonstrates this scenario for the Jordan ring matrix multiplication algorithm (Section 11.2.4). We first observed a difference in relative error between the two algorithms for random matrices when $B = A$ (the difference was negligible when A and B are picked randomly independently). In order to compare with respect to an exact solution, we set $A = B = Q = (I - 2uu^T)$, where Q is a Householder transformation computed from a random vector. Since Q is symmetric and orthogonal $A \cdot B + B \cdot A = Q \cdot Q + Q \cdot Q = 2I$. Thus we again compare the numerically computed solution to the analytically obtained exact solution, calculating the relative error as

$$\frac{||fl\,(C) - C||_2}{||C||_2}.$$

We see that the fast algorithm, $\Phi_{\odot}^{(1,1,1)}(A, A)$ incurs somewhat more error than the standard algorithm, $\Psi_{\odot}^{(1,1,1)}(A, A)$, because of the cancellation error incurred due to the U intermediate being large. This intermediate is large because $U_l = \sum_k A_{lk} \cdot B_{lk}$ is a sum of squares when $B = A$.

These numerical tests demonstrate that the fast algorithm has different numerical properties from the standard algorithm and can incur more error when the cancelled terms are large. However, these tests were artificially picked to maximize the error, and typically the difference of the error between the two algorithms is negligible. Further, the error incurred by the fast algorithm is not especially large in absolute magnitude even in these specially picked cases. Overall, these tests lead us to conclude that the fast algorithm should be used with some caution and awareness of its numerical characteristics, but is likely fit for deployment in a general numerical library.

11.4 Antisymmetric and Hermitian Adaptations

We now consider the case when the operands and/or the output are antisymmetric. The value of an antisymmetric tensor element changes sign if a pair of its indices are permuted e.g. $A_{i\ldots j\ldots k\ldots} = -A_{i\ldots k\ldots j\ldots}$. This implies that given a tuple of indices $i\langle d\rangle$ and a reordered set of the same indices $j\langle d\rangle$ the value of $A_{i\langle d\rangle} = (-1)^{\delta(i\langle d\rangle, j\langle d\rangle)} A_{j\langle d\rangle}$, where $\delta(i\langle d\rangle, j\langle d\rangle)$ is the number of pairwise permutations of indices necessary to transform $j\langle d\rangle$ back into $i\langle d\rangle$. In other words if the number of permutations is odd, there is a change of sign and if the number of permutations is even, the sign stays the same and the elements are equivalent. It does not matter whether δ corresponds to the fewest number of pairwise permutations necessary to transform $i\langle d\rangle$ into $j\langle d\rangle$ or to the length of any sequence of pairwise permutations, which transforms between the two indices, since only the parity matters.

We will need to perform summations over index partitions where terms have different signs depending on the particular partitioning of the index set. We will write such antisymmetric summations as

$$\widehat{\sum_{(j\langle s\rangle, l\langle t\rangle)\in\chi(i\langle s+t\rangle)}} f(j\langle s\rangle, l\langle t\rangle) \equiv \sum_{(j\langle s\rangle, l\langle t\rangle)\in\chi(i\langle s+t\rangle)} (-1)^{\delta(j\langle s\rangle l\langle t\rangle, i\langle s+t\rangle)} f(j\langle s\rangle, l\langle t\rangle),$$

where $\delta(j\langle s\rangle l\langle t\rangle, i\langle s+t\rangle)$ is the number of pairwise index permutations needed to transform $j\langle s\rangle l\langle t\rangle$ into $i\langle s+t\rangle$ (based on index label not index value), and $f(j\langle s\rangle, l\langle t\rangle)$ is an arbitrary function. We will define the summation over index subsets accordingly,

$$\widehat{\sum_{j\langle s\rangle\in\chi(i\langle s+t\rangle)}} f(j\langle s\rangle) \equiv \widehat{\sum_{(j\langle s\rangle, l\langle t\rangle)\in\chi(i\langle s+t\rangle)}} f(j\langle s\rangle).$$

We will first consider contractions that are symmetrized and so contracted via equation 11.1.1 but involve antisymmetric operands. For certain values of s, t, v, such contractions yield a zero result. We will then introduce an antisymmetrized version of equation 11.1.1 and consider contractions of both symmetric and antisymmetric tensors, again for certain s, t, v the contractions yield a zero result. Depending on the values of s, t, v and choices of which operands are symmetric/antisymmetric, as well as where the contraction is symmetrized and antisymmetrized, the contractions discussed will either

- have a zero result,
- have a nonzero result and be computable via the standard symmetric contraction algorithm but not the fast algorithm,
- have a nonzero result and be computable via either the standard or the fast symmetric contraction algorithm.

We will then move on to the Hermitian tensors, which may be represented as a pair of tensors: one real symmetric tensor and one pure imaginary antisymmetric tensor. We will also introduce

Hermitian contractions which are the analogue of symmetrized and antisymmetrized contractions. It will be possible to compute Hermitian contractions for all values of s, t, v via both the standard and fast symmetric contraction algorithms. The Hermitian algorithms will be primarily performed via the use of symmetrized and antisymmetrized contractions of symmetric and antisymmetric tensors. Table 11.4.1 summarizes the results of this section in terms of whether each antisymmetric/symmetric contraction is zero, computable by standard algorithm, computable fast algorithm, and whether it to what term (real or imaginary or none) it contributions in the case of Hermitian contractions.

Table 11.4.1: Status of different cases of (s, t, v) for symmetrized $(\odot)$ and antisymmetrized $(\otimes)$ contractions for symmetric (SY) and antisymmetric (AS) operands. The 'Hermitian' column lists whether the particular contraction contributes to the real or imaginary or no term of the output tensor in a Hermitian contraction. Zero entries mean the contraction always evaluates to zero. Presence of Ψ or Φ means that Ψ or Φ may be used as the contraction algorithm.

	Hermitian	$s > 1$	$t > 1$	$v > 1$
SY $\odot$ SY	real	Ψ, Φ	Ψ, Φ	Ψ, Φ
SY $\odot$ AS	none	Ψ	0	0
AS $\odot$ SY	none	0	Ψ	0
AS $\odot$ AS	real	0	0	Ψ, Φ
SY $\otimes$ SY	none	0	0	Ψ
SY $\otimes$ AS	imaginary	0	Ψ, Φ	0
AS $\otimes$ SY	imaginary	Ψ, Φ	0	0
AS $\otimes$ AS	none	Ψ	Ψ	Ψ

11.4.1 Symmetrized Contractions of Antisymmetric Tensors

In this section, we consider all valid cases of contraction between antisymmetric or symmetric operands that are symmetrized into a symmetric output tensor. Our main application, the coupled-cluster method, does not perform symmetrized contractions. However, one important use-case covered by this section is squaring an antisymmetric matrix, which results in a symmetric matrix. Different cases are valid for different values for s, t, v and the fast algorithm works for a subset of the valid contractions.

Symmetrized Contraction of an Antisymmetric Tensor with a Symmetric Tensor

We first consider the case with $(s, t, v) = (1, t, 1)$ where A is an antisymmetric matrix and B is a symmetric tensor of order $t+1$. The standard algorithm $\Psi_{\odot}^{(1,t,1)}$ works for $(s, t, v) = (1, t, 1)$ for *any* matrix A, because $A_{j\langle 1\rangle k\langle 1\rangle}$ is trivially symmetric under all permutations of the single index $j\langle 1\rangle$ and of the single index $k\langle 1\rangle$. The fast algorithm $\Phi_{\odot}^{(1,t,1)}(A, B)$ does not handle the antisymmetry of a matrix A since when $t > 0$, the algorithm $\Phi_{\odot}^{(1,t,1)}(A, B)$ would be symmetrizing an antisymmetric

matrix A, which gives the needed entries within Z the incorrect relative signs. When $t = 0$, there is no symmetrization done, and this case is equivalent to the antisymmetrized contraction of an antisymmetric tensor with a symmetric tensor, which we discuss in Section 11.4.2.

If we consider contraction of an antisymmetric tensor A of order $s + v > 2$ with a symmetric tensor B, then either $s > 1$ or $v > 1$. When $s > 1$, an antisymmetric pair of indices in A is symmetrized to compute C, which yields the trivial result $C = 0$. Otherwise, when $v > 1$, a pair of antisymmetric indices in A is contracted with a pair of symmetric indices in B, again yielding a trivial result, $C = 0$.

Symmetrized Contraction of Antisymmetric Tensors

We first consider the contraction of two antisymmetric tensors A and B via equation 11.1.2 into a vector or a symmetric matrix. This type of contraction is invalid if a pair of indices from A or B also appears in C, since this would force two indices in C to be antisymmetric. Therefore, we consider only the cases when $s, t \leq 1$, namely when C is a vector or a symmetric matrix. The case when only one of s, t is equal to 1 (the output is a vector) is most relevant to quantum-chemistry theory, while the case when $s = t = 1$ (the output is a symmetric matrix) is not common in quantum-chemistry theory. However, a good motivation for the latter case is that when $s = t = v = 1$, so A is an antisymmetric matrix, and when $B = A$, this contraction corresponds to squaring an antisymmetric matrix, the result of which, $C = \frac{1}{2}(A \cdot A^T + A^T \cdot A) = -A^2$, is symmetric (as in the case of the symmetric matrix square discussed at the end of Section 11.2.4, a factor of 3 fewer multiplications is needed for the fast algorithm, but the same number of additions and multiplications in total).

The standard algorithm, $\Psi_{\odot}^{(s,t,v)}(A, B)$ works effectively without modification, by folding A and B into matrices with dimensions k and $\binom{n}{v}$, where $k \in \{1, n\}$ depending on whether $s = 1$ or $s = 0$ for A and depending on whether $t = 1$ or $t = 0$ for B. If $s = t = 1$, the result $\bar{C}$ is symmetrized as before, $\forall i, j,\ \ C_{ij} = \bar{C}_{ij} + \bar{C}_{ji}$.

We adapt the fast algorithm $\Phi_{\odot}^{(s,t,v)}(A, B)$ to handle symmetrized contraction of antisymmetric tensors with $s, t \leq 1$ and arbitrary v as follows, $\forall i\langle\omega\rangle$,

$$\hat{Z}_{i\langle\omega\rangle} = \left(\widehat{\sum_{j\langle s+v\rangle \in \chi(i\langle\omega\rangle)}} A_{j\langle s+v\rangle}\right) \cdot \left(\widehat{\sum_{l\langle t+v\rangle \in \chi(i\langle\omega\rangle)}} B_{l\langle t+v\rangle}\right). \tag{11.4.1}$$

The summation over $\hat{Z}$ (analogous to equation 11.2.4) is performed with $\widehat{\sum}$ in place of $\sum$ and is multiplied by a sign prefactor,

$$Z_{i\langle s+t\rangle} = (-1)^{st+sv+tv} \sum_{k\langle v\rangle} \left(\widehat{\sum_{j\langle s+v\rangle \in \chi(i\langle s+t\rangle k\langle v\rangle)}} A_{j\langle s+v\rangle}\right) \cdot \left(\widehat{\sum_{l\langle t+v\rangle \in \chi(i\langle s+t\rangle k\langle v\rangle)}} B_{l\langle t+v\rangle}\right). \tag{11.4.2}$$

Looking at the definition of V (equation 11.2.5), we note that $s \geq p$ or $r \geq v - s - t$ (otherwise the sum over q has null range), and further that if $s, t \leq 1$, then $p, q \leq 1$. So, in this case we need

to compute only $A^{(1)}$ and $B^{(1)}$ ($A^{(0)} = A$ and $B^{(0)} = B$),

$$\forall i\langle s+v-1\rangle,\ A^{(1)}_{i\langle s+v-1\rangle} = \sum_k A_{i\langle s+v-1\rangle,k},$$
$$\forall i\langle t+v-1\rangle,\ B^{(1)}_{i\langle t+v-1\rangle} = \sum_k B_{i\langle t+v-1\rangle,k}.$$

Now, we can compute V as, $\forall i\langle s+t\rangle$,

$$V_{i\langle s+t\rangle} = \sum_{r=\max(0,v-s-t)}^{v-1} \binom{v}{r} \cdot \sum_{p=\max(0,v-t-r)}^{v-r} \binom{v-r}{p} \cdot \sum_{q=\max(0,v-s-r)}^{v-p-r} \binom{v-p-r}{q} \cdot n^{v-p-q-r} \cdot$$
$$(-1)^{st+(p+q)v+pq} \sum_{k\langle r\rangle} \left(\widehat{\sum_{j\langle s+v-p-r\rangle k\langle r\rangle \in \chi(i\langle s+t\rangle)}} A^{(p)}_{j\langle s+v-p-r\rangle k\langle r\rangle} \right) \cdot \left(\widehat{\sum_{l\langle t+v-q-r\rangle k\langle r\rangle \in \chi(i\langle s+t\rangle)}} B^{(q)}_{l\langle t+v-q-r\rangle k\langle r\rangle} \right). \tag{11.4.3}$$

The tensor W will only exist when $s = t = 1$, in which case we can compute,

$$\forall m,\ \ U^{(1)}_m = \left(\sum_{k\langle v\rangle} A_{m,k\langle v\rangle} \cdot B_{m,k\langle v\rangle} \right),$$
$$\forall i\langle 2\rangle,\ \ W_{i\langle 2\rangle} = -\sum_{m\in\chi(i\langle 2\rangle)} U^{(1)}_m.$$

The result C is just the sum of these intermediates again as in equation 11.2.8.

We now prove that the above method is correct following the proof of correctness for the purely symmetric fast algorithm (Theorem 11.3.1) except this time propagating signs from the inner terms. We partition the index set $k\langle v\rangle$ as described in that proof to obtain the following partition of Z, taking the sign of each operand from equation 11.4.2, $\forall i\langle s+t\rangle$,

$$Z_{i\langle s+t\rangle} = (-1)^{st+sv+tv} \sum_{k\langle v\rangle} \sum_{r=0}^{v} \sum_{(d\langle r\rangle, u\langle v-r\rangle)\in\chi(k\langle v\rangle)} \sum_{p=\max(0,v-t-r)}^{v-r} \sum_{(e\langle p\rangle, w\langle v-r-p\rangle)\in\chi(u\langle v-r\rangle)}$$
$$\sum_{q=\max(0,v-s-r)}^{v-r-p} \sum_{(f\langle q\rangle, g\langle v-r-p-q\rangle)\in\chi(w\langle v-r-p\rangle)} \Bigg[$$
$$\left(\sum_{(j\langle s+v-p-r\rangle, h\langle t-v+p+r\rangle)\in\chi(i\langle s+t\rangle)} (-1)^{\delta(j\langle s+v-p-r\rangle d\langle r\rangle e\langle p\rangle w\langle t\rangle,\ i\langle s+t\rangle k\langle v\rangle)} A_{j\langle s+v-p-r\rangle d\langle r\rangle e\langle p\rangle} \right)$$
$$\cdot \left(\sum_{(l\langle t+v-q-r\rangle, h'\langle s-v+q+r\rangle)\in\chi(i\langle s+t\rangle)} (-1)^{\delta(l\langle t+v-q-r\rangle d\langle r\rangle f\langle q\rangle v\langle s\rangle,\ i\langle s+t\rangle k\langle v\rangle)} B_{l\langle t+v-q-r\rangle d\langle r\rangle f\langle q\rangle} \right) \Bigg],$$

where $w\langle t\rangle$ are the t indices of $i\langle s+t\rangle k\langle v\rangle$ not appearing in the A operand, namely the ordered index set $f\langle q\rangle g\langle v-r-p-q\rangle h\langle t-v+p+r\rangle$, and $v\langle s\rangle$ are the s indices not appearing in the B

operand, namely the ordered index set $e\langle p\rangle g\langle v-r-p-q\rangle h'\langle s-v+q+r\rangle$. Noting that $s,t\leq 1$, which implies that the ordering of the (size at most one) index set does not matter, we can set

$$w\langle t\rangle = f\langle q\rangle g\langle v-r-p-q\rangle h\langle t-v+p+r\rangle$$

and $v\langle s\rangle = e\langle p\rangle g\langle v-r-p-q\rangle h'\langle s-v+q+r\rangle$. Now we manipulate the sign permutation factor on A by rotating the index set $h\langle t-v+p+r\rangle$ (of size at most one), v permutations to the left, yielding,

$$\begin{aligned}
&(-1)^{\delta(j\langle s+v-p-r\rangle d\langle r\rangle e\langle p\rangle f\langle q\rangle g\langle v-r-p-q\rangle h\langle t-v+p+r\rangle,\, i\langle s+t\rangle k\langle v\rangle)}\\
&\quad=(-1)^{(t-v+p+r)v}(-1)^{\delta(j\langle s+v-p-r\rangle h\langle t-v+p+r\rangle d\langle r\rangle e\langle p\rangle f\langle q\rangle g\langle v-r-p-q\rangle,\, i\langle s+t\rangle k\langle v\rangle)}\\
&\quad=(-1)^{(t-v+p+r)v}(-1)^{\delta(j\langle s+v-p-r\rangle h\langle t-v+p+r\rangle,\, i\langle s+t\rangle)}(-1)^{\delta(d\langle r\rangle e\langle p\rangle f\langle q\rangle g\langle v-r-p-q\rangle,\, k\langle v\rangle)}.
\end{aligned}$$

Performing the same manipulation on the sign factor of B, yields,

$$\begin{aligned}
&(-1)^{\delta(l\langle t+v-q-r\rangle d\langle r\rangle f\langle q\rangle e\langle p\rangle g\langle v-r-p-q\rangle h'\langle s-v+q+r\rangle,\, i\langle s+t\rangle k\langle v\rangle)}\\
&\quad=(-1)^{(s-v+q+r)v}(-1)^{\delta(l\langle t+v-q-r\rangle h'\langle s-v+q+r\rangle d\langle r\rangle f\langle q\rangle e\langle p\rangle g\langle v-r-p-q\rangle,\, i\langle s+t\rangle k\langle v\rangle)}\\
&\quad=(-1)^{(s-v+q+r)v}(-1)^{\delta(l\langle t+v-q-r\rangle h'\langle s-v+q+r\rangle,\, i\langle s+t\rangle)}(-1)^{\delta(d\langle r\rangle f\langle q\rangle e\langle p\rangle g\langle v-r-p-q\rangle,\, k\langle v\rangle)}.
\end{aligned}$$

The order of $f\langle p\rangle$ and $e\langle p\rangle$ may be swapped with a prefactor of $(-1)^{pq}$, so that the last terms in the above sign factors for A and B become the same, yielding, $\forall i\langle s+t\rangle$,

$$\begin{aligned}
&Z_{i\langle s+t\rangle} = (-1)^{st+sv+tv}\\
&\sum_{k\langle v\rangle}\sum_{r=0}^{v}\sum_{(d\langle r\rangle,u\langle v-r\rangle)\in\chi(k\langle v\rangle)}\sum_{p=\max(0,v-t-r)}^{v-r}\sum_{(e\langle p\rangle,w\langle v-r-p\rangle)\in\chi(u\langle v-r\rangle)}\sum_{q=\max(0,v-s-r)}^{v-r-p}\\
&\sum_{(f\langle q\rangle,g\langle v-r-p-q\rangle)\in\chi(w\langle v-r-p\rangle)}(-1)^{(t-v+p+r)v}(-1)^{(s-v+q+r)v}(-1)^{pq}(-1)^{2\delta(d\langle r\rangle e\langle p\rangle f\langle q\rangle g\langle v-r-p-q\rangle,\, k\langle v\rangle)}\Bigg[\\
&\left(\sum_{(j\langle s+v-p-r\rangle,h\langle t-v+p+r\rangle)\in\chi(i\langle s+t\rangle)}(-1)^{\delta(j\langle s+v-p-r\rangle h\langle t-v+p+r\rangle,\, i\langle s+t\rangle)}A_{j\langle s+v-p-r\rangle d\langle r\rangle e\langle p\rangle}\right)\\
&\cdot\left(\sum_{(l\langle t+v-q-r\rangle,h'\langle s-v+q+r\rangle)\in\chi(i\langle s+t\rangle)}(-1)^{\delta(l\langle t+v-q-r\rangle h'\langle s-v+q+r\rangle,\, i\langle s+t\rangle)}B_{l\langle t+v-q-r\rangle d\langle r\rangle f\langle q\rangle}\right)\Bigg],
\end{aligned}$$

Now, we absorb the factors of $(-1)^{\delta(j\langle s+v-p-r\rangle h\langle t-v+p+r\rangle,\, i\langle s+t\rangle)}$ and $(-1)^{\delta(l\langle t+v-q-r\rangle h'\langle s-v+q+r\rangle,\, i\langle s+t\rangle)}$ to transform the summations from $\sum$ to $\widehat{\sum}$, as well as combine the four exponents of -1 which

were factored out of the innermost summation,

$$
\begin{aligned}
Z_{i\langle s+t\rangle} = (-1)^{st+sv+tv} & \\
\sum_{k\langle v\rangle} \sum_{r=0}^{v} & \sum_{(d\langle r\rangle, u\langle v-r\rangle)\in\chi(k\langle v\rangle)} \quad \sum_{p=\max(0,v-t-r)}^{v-r} \quad \sum_{(e\langle p\rangle, w\langle v-r-p\rangle)\in\chi(u\langle v-r\rangle)} \\
& \sum_{q=\max(0,v-s-r)}^{v-r-p} \quad \sum_{(f\langle q\rangle, g\langle v-r-p-q\rangle)\in\chi(w\langle v-r-p\rangle)} (-1)^{(s+t-2v+p+q+2r)v+pq} \Bigg[\\
& \left(\widehat{\sum_{(j\langle s+v-p-r\rangle, h\langle t-v+p+r\rangle)\in\chi(i\langle s+t\rangle)}} A_{j\langle s+v-p-r\rangle d\langle r\rangle e\langle p\rangle} \right) \\
& \cdot \left(\widehat{\sum_{(l\langle t+v-q-r\rangle, h'\langle s-v+q+r\rangle)\in\chi(i\langle s+t\rangle)}} B_{l\langle t+v-q-r\rangle d\langle r\rangle f\langle q\rangle} \right) \Bigg].
\end{aligned}
$$

Now we combine the factors of $(-1)^{st+sv+tv}$ and $(-1)^{(s+t-2v+p+q+2r)v+pq}$, obtaining

$$
\begin{aligned}
Z_{i\langle s+t\rangle} = & \sum_{k\langle v\rangle} \sum_{r=0}^{v} \sum_{(d\langle r\rangle, u\langle v-r\rangle)\in\chi(k\langle v\rangle)} \sum_{p=\max(0,v-t-r)}^{v-r} \sum_{(e\langle p\rangle, w\langle v-r-p\rangle)\in\chi(u\langle v-r\rangle)} \sum_{q=\max(0,v-s-r)}^{v-r-p} \\
& \sum_{(f\langle q\rangle, g\langle v-r-p-q\rangle)\in\chi(w\langle v-r-p\rangle)} (-1)^{st+(p+q)v+pq} \Bigg[\\
& \left(\widehat{\sum_{j\langle s+v-p-r\rangle\in\chi(i\langle s+t\rangle)}} A_{j\langle s+v-p-r\rangle d\langle r\rangle e\langle p\rangle} \right) \cdot \left(\widehat{\sum_{l\langle t+v-q-r\rangle\in\chi(i\langle s+t\rangle)}} B_{l\langle t+v-q-r\rangle d\langle r\rangle f\langle q\rangle} \right) \Bigg].
\end{aligned}
$$

By the same argument as in the symmetric correctness proof (Theorem 11.3.1), we assert that for a particular choice of p, q, r the $k\langle v\rangle$ indices are indistinguishable, which allows us to replace sums with scalar prefactors. We again also isolate the $r = v$ term, yielding

$$
\begin{aligned}
Z_{i\langle s+t\rangle} = & \sum_{r=\max(0,v-s-t)}^{v-1} \binom{v}{r} \cdot \sum_{p=\max(0,v-t-r)}^{v-r} \binom{v-r}{p} \cdot \sum_{q=\max(0,v-s-r)}^{v-p-r} \binom{v-p-r}{q} \cdot n^{v-p-q-r} \cdot \\
& (-1)^{st+(p+q)v+pq} \sum_{k\langle r\rangle} \left(\widehat{\sum_{j\langle s+v-p-r\rangle\in\chi(i\langle s+t\rangle)}} A^{(p)}_{j\langle s+v-p-r\rangle k\langle r\rangle} \right) \cdot \left(\widehat{\sum_{l\langle t+v-q-r\rangle\in\chi(i\langle s+t\rangle)}} B^{(q)}_{l\langle t+v-q-r\rangle k\langle r\rangle} \right) \\
& + (-1)^{st} \sum_{k\langle v\rangle} \left(\widehat{\sum_{j\langle s\rangle\in\chi(i\langle s+t\rangle)}} A_{j\langle s\rangle k\langle v\rangle} \right) \cdot \left(\widehat{\sum_{l\langle t\rangle\in\chi(i\langle s+t\rangle)}} B_{l\langle t\rangle k\langle v\rangle} \right).
\end{aligned}
$$

The first of these two terms is just V, and plugging this in we obtain

$$
Z_{i\langle s+t\rangle} = V_{i\langle s+t\rangle} + (-1)^{st} \sum_{k\langle v\rangle} \left(\widehat{\sum_{j\langle s\rangle\in\chi(i\langle s+t\rangle)}} A_{j\langle s\rangle k\langle v\rangle} \right) \cdot \left(\widehat{\sum_{l\langle t\rangle\in\chi(i\langle s+t\rangle)}} B_{l\langle t\rangle k\langle v\rangle} \right).
$$

Again following the correctness proof for the symmetric case (Theorem 11.3.1), we subdivide the latter term into subterms where $j\langle s\rangle$ and $l\langle t\rangle$ are disjoint and when they have overlap. Since in this case $s, t \leq 1$, we only need to consider the $r = 1$ case (which only exists when $s = t = 1$ since $r = \min(s, t)$, which we signify by multiplying the term by st),

$$\begin{aligned}\forall i\langle s+t\rangle,\ \ Z_{i\langle s+t\rangle} = {} & V_{i\langle s+t\rangle} \\ & + (-1)^{st} \sum_{(j\langle s\rangle, l\langle t\rangle)\in\chi(i\langle s+t\rangle)} \left(\sum_{k\langle v\rangle} (-1)^{\delta(j\langle s\rangle l\langle t\rangle,\ i\langle s+t\rangle)} A_{j\langle s\rangle k\langle v\rangle} \cdot (-1)^{\delta(l\langle t\rangle j\langle s\rangle,\ i\langle s+t\rangle)} B_{l\langle t\rangle k\langle v\rangle} \right) \\ & + st \cdot (-1)^{st} \left[\sum_{(m,h)\in\chi(i\langle 2\rangle)} \left(\sum_{k\langle v\rangle} (-1)^{\delta(mh,\ i\langle 2\rangle)} A_{mk\langle v\rangle} \cdot (-1)^{\delta(mh,\ i\langle 2\rangle)} B_{mk\langle v\rangle} \right) \right]\end{aligned}$$

Absorbing the permutation factors into the antisymmetric tensors' indices we obtain $\forall i\langle s+t\rangle$,

$$\begin{aligned}Z_{i\langle s+t\rangle} = {} & V_{i\langle s+t\rangle} + \sum_{(j\langle s\rangle, l\langle t\rangle)\in\chi(i\langle s+t\rangle)} \left(\sum_{k\langle v\rangle} A_{j\langle s\rangle k\langle v\rangle} \cdot B_{l\langle t\rangle k\langle v\rangle} \right) \\ & - st \cdot \left[\sum_{m\in\chi(i\langle 2\rangle)} \left(\sum_{k\langle v\rangle} A_{mk\langle v\rangle} \cdot B_{mk\langle v\rangle} \right) \right] \\ = {} & V_{i\langle s+t\rangle} + \binom{s+t}{s} \cdot C_{i\langle s+t\rangle} + W_{i\langle s+t\rangle},\end{aligned}$$

which concludes the correctness proof.

We now argue that the cost of this algorithm, $\Phi_{\odot}^{(s,t,v)}(A, B)$ for antisymmetric A and B is no greater than the cost of $\Phi_{\odot}^{(s,t,v)}(A', B')$ for symmetric A' and B' with the same dimensions. Since the sign factors do not affect the number of operations, it suffices to show that $\hat{Z}$ is antisymmetric, and can therefore be computed in $n^{\omega}/\omega!$ operations to leading order. Since A and B are both fully antisymmetric, we can show that each operand of $\hat{Z}$ in equation 11.4.1 is antisymmetric under permutation of any pair of indices i_p and i_q in the $i\langle\omega\rangle$ index group, which implies that entries of $\hat{Z}$ are symmetric under any permutations of i_p and i_q since the signs from the operands cancel. The terms in the summation forming each operand (we consider A but the same holds for B) fall into two cases (note that the third case, in Section 11.3.3 no longer occurs since we assume $s, t \leq 1$).

In the first case, both indices i_p and i_q will appear in the term, in which case the term is antisymmetric since the tensor is antisymmetric.

In the second case, if $t = 1$, only one of two indices may appear in some term, without loss of generality let i_p appear and not i_q, the indices of the term are some permutation of $j\langle s+v-1\rangle i_p$ where $j\langle s+v-1\rangle \in i\langle\omega\rangle$ and $j\langle s+v-1\rangle$ does not include i_p or i_q. Now, we can assert there is another term in the summation whose indices are some permutation of $j\langle s+v-1\rangle i_q$, since $\chi(i\langle\omega\rangle)$ yields all possible ordered subsets of $i\langle\omega\rangle$, which must include an ordered index set containing the distinct indices $j\langle s+v-1\rangle i_q$. We now argue that the signs of these two

terms are different. Let the ordered indices of the first term (which include i_p) be $l\langle s+v\rangle$ and the ordered indices of the second term be $m\langle s+v\rangle$. Recalling that $t = 1$ (so $\omega = s+v+1$), the signs of these terms assigned to them by the sum ($\widehat{\sum}$) are decided by the parities of $\delta(l\langle s+v\rangle i\langle p\rangle,\ i\langle s+v+1\rangle) = (-1)^{s+v-p}$ and $\delta(m\langle s+v\rangle i\langle q\rangle,\ i\langle s+v+1\rangle) = (-1)^{s+v-q}$, so the signs are the same if p, q are both even or both odd. Now since A is antisymmetric, and assuming without loss of generality that $p > q$, we can permute its indices to write the value of the term as $A_{l\langle s+v\rangle} = (-1)^{s+v-q-1}A_{j\langle s+v-1\rangle i_q}$ (since $j\langle s+v-1\rangle$ does not contain i_p) and $A_{m\langle s+v\rangle} = (-1)^{s+v-p}A_{j\langle s+v-1\rangle i_p}$ (since $q < p$ and so $j\langle s+v-1\rangle$ contains all indices i_k in $i\langle\omega\rangle$ with $k > p$). Therefore, if we permute i_p with i_q the two values of the terms take on an opposite sign only if p and q are both even or both odd. Therefore, in all cases, either the value or the sign assigned by the summation ($\widehat{\sum}$) to the terms is different or the values of the terms (based on which order the indices appear in the operand) take on opposite signs when i_p is permuted with i_q, and since the pairing is unique, the overall operand is antisymmetric. As an example, for $s = v = t = 1$, the antisymmetric left operand is $(A_{ij} - A_{ik} + A_{jk})$, if we permute i with j, A_{ik} is swapped with A_{jk} which have had different signs assigned to them by the summation. However, if we swap i with k, A_{ij} takes on the opposite sign, $A_{kj} = -A_{jk}$, and A_{jk} takes on the opposite sign $A_{ji} = -A_{ij}$.

11.4.2 Antisymmetrized Contractions of Antisymmetric Tensors

Definition: We define an **antisymmetrized contraction** between tensors A and B by analogue to equation 11.1.1,

$$C = A \otimes B \ \equiv\ \forall i\langle s+t\rangle,\ C_{i\langle s+t\rangle} = \frac{1}{(s+t)!} \cdot \widehat{\sum_{j\langle s\rangle l\langle t\rangle \in \Pi(i\langle s+t\rangle)}} \left(\sum_{k\langle v\rangle} A_{j\langle s\rangle k\langle v\rangle} \cdot B_{l\langle t\rangle k\langle v\rangle} \right). \tag{11.4.4}$$

The result of an antisymmetrized contraction is an antisymmetric tensor C.

Antisymmetrized Contraction of Symmetric Tensors

We first consider the case where A and B are both symmetric. The standard algorithm can do such contractions for $s = t = 1$ and arbitrary v, since the intermediate is a nonsymmetric matrix, which can be symmetrized. Equation 11.1.1 evaluates to $C = 0$ for either $s > 1$ or $t > 1$ since a symmetry would be 'preserved' between a symmetric tensor and antisymmetric tensor, so adding the permutations in $\Pi(i\langle s+t\rangle)$ would cancel each other out. The fast algorithm $\Psi_{\odot}^{(s,t,v)}$ cannot handle the $s = t = 1$ case of the antisymmetrized product of two symmetric tensors, since the tensor Z is symmetric and contains the wrong relative signs of the needed (antisymmetrized) entries.

Antisymmetrized Contraction of Antisymmetric Tensors

The case when both operands A and B are antisymmetric and the result is antisymmetrized is well-defined for any $s, t, v \geq 1$. Equation 11.1.2 takes the following form for this case

$$C_{i\langle s+t\rangle} = \frac{1}{\binom{s+t}{s}} \cdot \widehat{\sum_{(j\langle s\rangle, l\langle t\rangle)\in\chi(i\langle s+t\rangle)}} \left(\sum_{k\langle v\rangle} A_{j\langle s\rangle k\langle v\rangle} \cdot B_{l\langle t\rangle k\langle v\rangle} \right). \tag{11.4.5}$$

Algorithm $\Psi_{\otimes}^{(\mathbf{s},\mathbf{t},\mathbf{v})}$: The standard algorithm works essentially without modification from the symmetric case, for this antisymmetric case. A partially antisymmetric intermediate $\bar{C}$ is formed with dimensions $\binom{n}{s}$ by $\binom{n}{t}$ via a matrix multiplication that sums over a dimension of length $\binom{n}{v}$. The result is then antisymmetrized, $\forall i\langle s+t\rangle$

$$C_{i\langle s+t\rangle} = \frac{1}{\binom{s+t}{s}} \cdot \widehat{\sum_{(j\langle s\rangle, l\langle t\rangle)\in\chi(i\langle s+t\rangle)}} \bar{C}_{j\langle s\rangle l\langle t\rangle}.$$

We call this standard algorithm for antisymmetrized contractions

$$C = A \otimes B = \Psi_{\otimes}^{(s,t,v)}(A, B).$$

This antisymmetric case is in a sense even simpler than the symmetric case, since the diagonals are zero and can be ignored, while in the symmetric case they have to be handled with care (scaled by different factors).

The fast algorithm, $\Psi_{\odot}^{(s,t,v)}(A, B)$ cannot be adapted to handling such contractions, because when $s = t = 1$, if we try to antisymmetrize the operands of $\hat{Z}$ (equation 11.2.3 with uses of $\sum$ replaced by $\widehat{\sum}$), the resulting matrix would be symmetric, and we want to compute an antisymmetrized rather than a symmetrized product. Further, when either s or t is at least two, at least one of the operands is no longer antisymmetric, since it will contain terms in the sum that have some pair of indices i, j appear in the tensor (and therefore be antisymmetric), and terms where both i and j are not present, which are effectively symmetric in this pair of indices. Therefore, the resulting Z would be neither symmetric nor antisymmetric.

However, such antisymmetric contractions with $s, t, v \geq 1$ do not occur in coupled-cluster theory or any other quantum chemistry applications we are aware of. This curious fact can be shown diagrammatically or noted as a consequence of raised and lowered index notation (discussed in detail in Section 11.5.1), as only indices of a single orientation may be antisymmetric, while only indices of different orientation may be contracted together. So, all contractions which arise in our quantum chemistry applications have one of $s = 0$ or $t = 0$ or $v = 0$. Those antisymmetric contractions where two of s, t, v are at least two (and the other zero) are also incompatible with the fast symmetric contraction algorithm, for the same reason as discussed above. However, while a few such cases are present in the high-order CCSDTQ [102] method, the predominant number of tensor contraction cases in coupled-cluster theory have s, t, v be some permutation of $(0, 1, k)$ for some $k \geq 1$. In these cases, one of the A, B, and C tensors is a vector and

the other two tensors are of order k and $k+1$. These cases may be executed using the fast algorithm as we detail in the next two sections. We first consider the cases where A and B are antisymmetric tensors of order k and $k+1$ and C is a vector or a symmetric matrix (so when $(s,t,v) \in \{(0,1,k),(1,0,k),(1,1,k)\}$), then consider the case where C and either A or B are antisymmetric tensors of order k and $k+1$, while one of the operands is a vector or a symmetric matrix (so when $(s,t,v) \in \{(k,0,1),(0,k,1),(1,k,0),(k,1,0)\}$). The fast tensor contraction algorithm works for all of these cases, since none of the antisymmetric tensors has fewer than $\omega-1$ indices allowing $\hat{Z}$ to be formed with appropriate symmetry or antisymmetry.

Antisymmetrized Contraction of an Antisymmetric Tensor with a Symmetric Matrix or a Vector

In the previous section, we considered a contraction of two antisymmetric operands into a vector or a symmetric matrix. In this section, we consider the contraction of a single antisymmetric operand with another operand that is either a symmetric matrix or a vector into an output that is an antisymmetric tensor. We consider only the case where A is antisymmetric and B is a symmetric matrix or vector, as the fact that the multiplication operator is commutative permutes the roles of the operands (switch A and B) in any contraction where A is a symmetric matrix or vector and B is antisymmetric. We limit the consideration to symmetric matrices B rather than symmetric tensors of higher order, since if B has order three or more, it must either share two indices with C or share two indices with A. In the first case, the contraction evaluates to zero since C is antisymmetric in these indices while B is symmetric in these indices. In the latter case, the contraction evaluates to zero, since all entries of the output must sum over a pair of indices which are antisymmetric in A and symmetric in B, e.g.

$$\sum_{ij} A_{ij} \cdot B_{ij} = \sum_{ji} A_{ji} \cdot B_{ji} = \sum_{ij} -A_{ij} \cdot B_{ji} = -\sum_{ij} A_{ij} \cdot B_{ij} = 0.$$

So, throughout this section we consider only $t, v \leq 1$.

Equation 11.1.2 for the contraction of antisymmetric A with vector or symmetric matrix B takes the same antisymmetrized form $\forall i \langle s+t \rangle$

$$C_{i\langle s+t\rangle} = \frac{1}{\binom{s+t}{s}} \cdot \widehat{\sum_{(j\langle s\rangle, l\langle t\rangle) \in \chi(i\langle s+t\rangle)}} \left(\sum_{k\langle v\rangle} A_{j\langle s\rangle k\langle v\rangle} \cdot B_{l\langle t\rangle k\langle v\rangle} \right).$$

The standard algorithm $\Psi_{\otimes}^{(s,t,v)}(A,B)$ forms a partially antisymmetric intermediate $\bar{C}$, which has s antisymmetric indices inherited from A and $t \leq 1$ non-symmetric indices from B, then antisymmetrizes the result. The cost of the standard algorithm is to leading order the same as before.

Algorithm $\mathbf{\Phi_{\otimes}^{(s,t,v)}}$: The fast algorithm $\Phi_{\odot}^{(s,t,v)}(A,B)$ may be adapted to handle this case by forming an antisymmetric $\hat{Z}$ tensor, which is a point-wise product of an antisymmetric tensor of operands of A and a symmetric tensor of operands of B. We call this adapted algorithm $C = A \otimes B = \Phi_{\otimes}^{(s,t,v)}(A,B)$ and define it for arbitrary s and $t, v \leq 1$, but assert that it also works for

arbitrary t and $s, v \leq 1$, by symmetry of operands. The equation is

$$\hat{Z}_{i\langle\omega\rangle} = \left(\widehat{\sum_{j\langle s+v\rangle \in \chi(i\langle\omega\rangle)}} A_{j\langle s+v\rangle}\right) \cdot \left(\sum_{l\langle t+v\rangle \in \chi(i\langle\omega\rangle)} B_{l\langle t+v\rangle}\right). \tag{11.4.6}$$

The summation over $\hat{Z}$ (analogous to equation 11.2.4) is performed with $\widehat{\sum}$ in place of $\sum$ and is multiplied by a sign prefactor,

$$Z_{i\langle s+t\rangle} = (-1)^{tv} \sum_{k\langle v\rangle} \left(\widehat{\sum_{j\langle s+v\rangle \in \chi(i\langle s+t\rangle k\langle v\rangle)}} A_{j\langle s+v\rangle}\right) \cdot \left(\sum_{l\langle t+v\rangle \in \chi(i\langle s+t\rangle k\langle v\rangle)} B_{l\langle t+v\rangle}\right). \tag{11.4.7}$$

In defining V (equation 11.2.5), we note that we have restricted the case to $v \leq 1$, which considerably simplifies the expression. If $v = 0$, V does not need to be computed at all (is zero), and when $v = 1$ we compute $A^{(1)}$ and $B^{(1)}$,

$$\forall i\langle s+v-1\rangle,\ A^{(1)}_{i\langle s+v-1\rangle} = \sum_k A_{i\langle s+v-1\rangle,k},$$
$$\forall i\langle t+v-1\rangle,\ B^{(1)}_{i\langle t+v-1\rangle} = \sum_k B_{i\langle t+v-1\rangle,k}.$$

Now, if $v = 1$, we can compute V as (multiplying by v to signify it is nonzero only if $v = 1$), $\forall i\langle s+t\rangle$,

$$V_{i\langle s+t\rangle} = v \cdot (-1)^{p-1} \sum_{p=\max(0,v-t)}^{1} \sum_{q=\max(0,v-s)}^{1-p} n^{v-p-q} \cdot \left[\left(\widehat{\sum_{j\langle s+1-p\rangle \in \chi(i\langle s+t\rangle)}} A^{(p)}_{j\langle s+1-p\rangle}\right) \cdot \left(\sum_{l\langle t+1-q\rangle \in \chi(i\langle s+t\rangle)} B^{(q)}_{l\langle t+1-q\rangle}\right)\right].$$

The tensor W is also simplified, since we consider only $t \leq 1$, and W exists only when $s, t > 0$. So it suffices to consider only $t = 1$ (we multiply by t for this reason) and compute, $\forall i\langle s-1\rangle$,

$$U^{(1)}_{mi\langle s-1\rangle} = \sum_{k\langle v\rangle} A_{mi\langle s-1\rangle k\langle v\rangle} \cdot B_{mk\langle v\rangle},$$
$$W_{i\langle s+1\rangle} = t \cdot \widehat{\sum_{(m,h\langle s-1\rangle)\in\chi(g\langle s\rangle)(g\langle s\rangle,l)\in\chi(i\langle s+1\rangle)}} U^{(1)}_{mh\langle s-1\rangle}.$$

The result C is just the sum of these intermediates again as in equation 11.2.8.

We now prove that the above method is correct following the proof of correctness for the purely symmetric fast algorithm (Theorem 11.3.1) except this time propagating signs from the A

antisymmetric summation. We partition the index set $k\langle v\rangle$ as described in that proof to obtain the following partition of Z, taking the sign of each operand from equation 11.4.7, $\forall i\langle s+t\rangle$,

$$\begin{aligned}Z_{i\langle s+t\rangle} =&(-1)^{tv}\sum_{k\langle v\rangle}\sum_{r=0}^{v}\sum_{(d\langle r\rangle,u\langle v-r\rangle)\in\chi(k\langle v\rangle)}\sum_{p=\max(0,v-t-r)}^{v-r}\sum_{(e\langle p\rangle,w\langle v-r-p\rangle)\in\chi(u\langle v-r\rangle)}\\ &\sum_{q=\max(0,v-s-r)}^{v-r-p}\sum_{(f\langle q\rangle,g\langle v-r-p-q\rangle)\in\chi(w\langle v-r-p\rangle)}\Bigg[\\ &\left(\sum_{(j\langle s+v-p-r\rangle,h\langle t-v+p+r\rangle)\in\chi(i\langle s+t\rangle)}(-1)^{\delta(j\langle s+v-p-r\rangle d\langle r\rangle e\langle p\rangle w\langle t\rangle,\, i\langle s+t\rangle k\langle v\rangle)}A_{j\langle s+v-p-r\rangle d\langle r\rangle e\langle p\rangle}\right)\\ &\cdot\left(\sum_{l\langle t+v-q-r\rangle\in\chi(i\langle s+t\rangle)}B_{l\langle t+v-q-r\rangle d\langle r\rangle f\langle q\rangle}\right)\Bigg],\end{aligned}$$

where $w\langle t\rangle$ are the t indices of $i\langle s+t\rangle k\langle v\rangle$ not appearing in the A operand, namely the ordered index set $f\langle q\rangle g\langle v-r-p-q\rangle h\langle t-v-p-r\rangle$. Noting that $t\leq 1$, which implies that the ordering of the (size at most one) index set does not matter, we can set

$$w\langle t\rangle = f\langle q\rangle g\langle v-r-p-q\rangle h\langle t-v+p+r\rangle.$$

Now we manipulate the sign permutation factor on A by rotating the index set $h\langle t-v+p+r\rangle$ (of size at most one), v permutations to the left, yielding,

$$\begin{aligned}&(-1)^{\delta(j\langle s+v-p-r\rangle d\langle r\rangle e\langle p\rangle f\langle q\rangle g\langle v-r-p-q\rangle h\langle t-v+p+r\rangle,\, i\langle s+t\rangle k\langle v\rangle)}\\ &\quad=(-1)^{(t-v+p+r)v}(-1)^{\delta(j\langle s+v-p-r\rangle h\langle t-v+p+r\rangle,\, i\langle s+t\rangle)}(-1)^{\delta(d\langle r\rangle e\langle p\rangle f\langle q\rangle g\langle v-r-p-q\rangle,\, k\langle v\rangle)}.\end{aligned}$$

Plugging this in and factoring out $(-1)^{(t-v+p+r)v}(-1)^{\delta(d\langle r\rangle e\langle p\rangle f\langle q\rangle g\langle v-r-p-q\rangle,k\langle v\rangle)}$ we obtain $\forall i\langle s+t\rangle$,

$$\begin{aligned}Z_{i\langle s+t\rangle} =&(-1)^{tv}\sum_{k\langle v\rangle}\sum_{r=0}^{v}\sum_{(d\langle r\rangle,u\langle v-r\rangle)\in\chi(k\langle v\rangle)}\sum_{p=\max(0,v-t-r)}^{v-r}\sum_{(e\langle p\rangle,w\langle v-r-p\rangle)\in\chi(u\langle v-r\rangle)}\sum_{q=\max(0,v-s-r)}^{v-r-p}\\ &\sum_{(f\langle q\rangle,g\langle v-r-p-q\rangle)\in\chi(w\langle v-r-p\rangle)}(-1)^{(t-v+p+r)v}(-1)^{\delta(d\langle r\rangle e\langle p\rangle f\langle q\rangle g\langle v-r-p-q\rangle,k\langle v\rangle)}\Bigg[\\ &\left(\sum_{(j\langle s+v-p-r\rangle,h\langle t-v+p+r\rangle)\in\chi(i\langle s+t\rangle)}(-1)^{\delta(j\langle s+v-p-r\rangle h\langle t-v+p+r\rangle,\, i\langle s+t\rangle)}A_{j\langle s+v-p-r\rangle d\langle r\rangle e\langle p\rangle}\right)\\ &\cdot\left(\sum_{l\langle t+v-q-r\rangle\in\chi(i\langle s+t\rangle)}B_{l\langle t+v-q-r\rangle d\langle r\rangle f\langle q\rangle}\right)\Bigg]\end{aligned}$$

Now, we absorb the factors of $(-1)^{\delta(j\langle s+v-p-r\rangle h\langle t-v+p+r\rangle,\, i\langle s+t\rangle)}$ and to transform the summation over A operands from $\sum$ to $\widehat{\sum}$,

$$
\begin{aligned}
=&(-1)^{tv}\sum_{k\langle v\rangle}\sum_{r=0}^{v}\sum_{(d\langle r\rangle,u\langle v-r\rangle)\in\chi(k\langle v\rangle)}\sum_{p=\max(0,v-t-r)}^{v-r}\sum_{(e\langle p\rangle,w\langle v-r-p\rangle)\in\chi(u\langle v-r\rangle)}\sum_{q=\max(0,v-s-r)}^{v-r-p}\\
&\sum_{(f\langle q\rangle,g\langle v-r-p-q\rangle)\in\chi(w\langle v-r-p\rangle)}(-1)^{(t-v+p+r)v}(-1)^{\delta(d\langle r\rangle e\langle p\rangle f\langle q\rangle g\langle v-r-p-q\rangle,\, k\langle v\rangle)}\Bigg[\\
&\left(\widehat{\sum_{j\langle s+v-p-r\rangle\in\chi(i\langle s+t\rangle)}}A_{j\langle s+v-p-r\rangle d\langle r\rangle e\langle p\rangle}\right)\cdot\left(\sum_{l\langle t+v-q-r\rangle\in\chi(i\langle s+t\rangle)}B_{l\langle t+v-q-r\rangle d\langle r\rangle f\langle q\rangle}\right)\Bigg].
\end{aligned}
$$

We now separate the $r=v$ term, which is the only term when $v=0$, and the $r=0$ term which is auxiliary only when $v=1$, as we signify by multiplying the term by v, yielding $\forall i\langle s+t\rangle$,

$$
\begin{aligned}
Z_{i\langle s+t\rangle}=&(-1)^{tv}\sum_{k\langle v\rangle}(-1)^{tv}\left[\left(\widehat{\sum_{j\langle s\rangle\in\chi(i\langle s+t\rangle)}}A_{j\langle s\rangle k\langle v\rangle}\right)\cdot\left(\sum_{l\langle t\rangle\in\chi(i\langle s+t\rangle)}B_{l\langle t\rangle k\langle v\rangle}\right)\right]\\
&+v\cdot(-1)^{t}\sum_{k}\sum_{p=\max(0,1-t)}^{1}\sum_{(e\langle p\rangle,w\langle v-p\rangle)\in\chi(k)}\sum_{q=\max(0,1-s)}^{1-p}\\
&\sum_{f\langle q\rangle\in\chi(w\langle v-p\rangle)}(-1)^{t-1+p}\left[\left(\widehat{\sum_{j\langle s+v-p\rangle\in\chi(i\langle s+t\rangle)}}A_{j\langle s+v-p\rangle e\langle p\rangle}\right)\cdot\left(\sum_{l\langle t+v-q\rangle\in\chi(i\langle s+t\rangle)}B_{l\langle t+v-q\rangle f\langle q\rangle}\right)\right]
\end{aligned}
$$

By the same argument as in the symmetric correctness proof (Theorem 11.3.1), we assert that for a particular choice of p,q,r the $k\langle v\rangle$ indices are indistinguishable, which allows us to replace sums with scalar prefactors. We again also isolate the $r=v$ term, yielding

$$
\begin{aligned}
Z_{i\langle s+t\rangle}=&\sum_{k\langle v\rangle}\left(\widehat{\sum_{j\langle s\rangle\in\chi(i\langle s+t\rangle)}}A_{j\langle s\rangle k\langle v\rangle}\right)\cdot\left(\sum_{l\langle t\rangle\in\chi(i\langle s+t\rangle)}B_{l\langle t\rangle k\langle v\rangle}\right)\\
&+v\cdot(-1)^{p+1}\sum_{p=\max(0,1-t)}^{1}\sum_{q=\max(0,1-s)}^{1-p}n^{v-p-q}\cdot\\
&\left(\widehat{\sum_{j\langle s+v-p\rangle\in\chi(i\langle s+t\rangle)}}A^{(p)}_{j\langle s+v-p\rangle}\right)\cdot\left(\sum_{l\langle t+v-q\rangle\in\chi(i\langle s+t\rangle)}B^{(q)}_{l\langle t+v-q\rangle}\right)\\
=&\sum_{k\langle v\rangle}\left(\widehat{\sum_{j\langle s\rangle\in\chi(i\langle s+t\rangle)}}A_{j\langle s\rangle k\langle v\rangle}\right)\cdot\left(\sum_{l\langle t\rangle\in\chi(i\langle s+t\rangle)}B_{l\langle t\rangle k\langle v\rangle}\right)+V_{i\langle s+t\rangle}.
\end{aligned}
$$

Now, we've isolated V and proceed to separate C from W in the first term, where C again corresponds to the cases when $j\langle s\rangle$ is disjoint from $l\langle t\rangle$ and W captures all cases of overlap between

the two. Since, $t \leq 1$, they can overlap only in one index and only when $t = 1$, we label this index m and multiply the term by a factor of t to signify that it disappears for $t = 0$. We also need to carefully handle the missing index (labeled l) that does not appear in neither operand in the W term, as it affects the sign,

$$\begin{aligned}
Z_{i\langle s+t\rangle} =& V_{i\langle s+t\rangle} + \sum_{k\langle v\rangle} \sum_{(j\langle s\rangle, l\langle t\rangle)\in\chi(i\langle s+t\rangle)} (-1)^{\delta(j\langle s\rangle l\langle t\rangle,\, i\langle s+t\rangle)} A_{j\langle s\rangle k\langle v\rangle} \cdot B_{l\langle t\rangle k\langle v\rangle} \\
&+ t \cdot \sum_{k\langle v\rangle} \sum_{(m,h\langle s-1\rangle)\in\chi(g\langle s\rangle)} \sum_{(g\langle s\rangle,l)\in\chi(i\langle s+t\rangle)} (-1)^{\delta(mh\langle s-1\rangle l,\, i\langle s+1\rangle)} A_{mh\langle s-1\rangle k\langle v\rangle} \cdot B_{mh\langle s-1\rangle k\langle v\rangle} \\
=& V_{i\langle s+t\rangle} + C_{i\langle s+t\rangle} + t \cdot \sum_{k\langle v\rangle} \sum_{(m,h\langle s-1\rangle)\in\chi(g\langle s\rangle)} \sum_{(g\langle s\rangle,l)\in\chi(i\langle s+t\rangle)} \Bigg[\\
&(-1)^{\delta(mh\langle s-1\rangle,\, g\langle s\rangle)} (-1)^{\delta(g\langle s\rangle l,\, i\langle s+1\rangle)} A_{mh\langle s-1\rangle k\langle v\rangle} \cdot B_{mh\langle s-1\rangle k\langle v\rangle} \Bigg] \\
=& V_{i\langle s+t\rangle} + C_{i\langle s+t\rangle} + t \cdot \widehat{\sum_{(m,h\langle s-1\rangle)\in\chi(g\langle s\rangle)}} \widehat{\sum_{(g\langle s\rangle,l)\in\chi(i\langle s+1\rangle)}} U^{(1)}_{mh\langle s-1\rangle}, \\
=& V_{i\langle s+t\rangle} + \binom{s+t}{s} \cdot C_{i\langle s+t\rangle} + W_{i\langle s+t\rangle}
\end{aligned}$$

Since $\binom{s+t}{s} C = Z - V - W$, this demonstrates correctness of this antisymmetric adaptation $\Phi^{(s,t,v)}_{\otimes}(A, B)$. The algorithm has the same leading order costs as in the case of symmetric tensors, since it computes the same intermediates only with different signs and since the intermediates have the same number of non-zeros. The antisymmetry of Z holds by applying the antisymmetric argument from Section 11.4.1 to the A operand and the symmetric argument from Section 11.3.3 to the B operand, as the point-wise product of an antisymmetric and a symmetric tensor preserves the sign and is therefore antisymmetric.

11.4.3 Hermitian Adaptation

We define a complex tensor of order d as $A = \mathrm{Re}(A) + i \cdot \mathrm{Im}(A)$, where $\mathrm{Re}(A)$ (real part) and $\mathrm{Im}(A)$ (complex part) are tensors with elements in Abelian group $(R_A, +)$. Given another complex tensor $B = \mathrm{Re}(B) + i \cdot \mathrm{Im}(B)$ with elements in $(R_B, +)$, Abelian group $(R_C, +)$, and a distributive operator, "$\cdot$" $\in R_A \times R_B \to R_C$, the imaginary unit $i = \sqrt{-1}$ defines the operator (which following convention we overload) "$\cdot$" $\in (R_A \times R_A) \times (R_B \times R_B) \to (R_C \times R_C)$, specifically for any $a = a_1 + i \cdot a_2 \in (R_A \times R_A)$, $b = b_1 + i \cdot b_2 \in (R_B \times R_B)$, $a \cdot b$ yields $c \in (R_C \times R_C)$, where

$$c = c_1 + i \cdot c_2 = a \cdot b = (a_1 \cdot b_1 - a_2 \cdot b_2) + i \cdot (a_1 \cdot b_2 + a_2 \cdot b_1).$$

We denote the conjugate of an element of any complex tensor, $A_{i\langle v\rangle} = \mathrm{Re}(A)_{i\langle v\rangle} + i \cdot \mathrm{Im}(A)_{i\langle v\rangle}$, as $A^*_{i\langle v\rangle} = \mathrm{Re}(A)_{i\langle v\rangle} - i \cdot \mathrm{Im}(A)_{i\langle v\rangle}$.

An order d complex tensor is Hermitian if $\mathrm{Re}(A)$ is a symmetric tensor and $\mathrm{Im}(A)$ is an antisymmetric tensor. This implies that for any odd permutation $i\langle d\rangle = \pi(j\langle d\rangle)$ (so with odd $\delta(i\langle d\rangle,\ j\langle d\rangle)$), $H_{i\langle d\rangle} = H^*_{j\langle d\rangle}$, and for any even permutation $i\langle d\rangle = \pi'(k\langle d\rangle)$ (so with even $\delta(i\langle d\rangle,\ j\langle d\rangle)$), $H_{i\langle d\rangle} = H_{k\langle d\rangle}$.

We say a complex tensor $\bar{H}$ is partially-Hermitian, if $\mathrm{Re}(\bar{H})$ and $\mathrm{Im}(\bar{H})$ are partially-symmetric and partially-antisymmetric, respectively, in the same index sets. We combine the symmetrized and antisymmetrized summation notation over tuple partitions to yield a notation for a sum which yields a Hermitian tensor result H when applied to any non-Hermitian order d tensor $\bar{H}$,

$$H_{i\langle d\rangle} = \sum_{j\langle d\rangle\in\Pi(i\langle d\rangle)}^{*} \bar{H}_{i\langle d\rangle} \equiv \sum_{j\langle d\rangle\in\Pi(i\langle d\rangle)} \mathrm{Re}(\bar{H})_{j\langle d\rangle} + \widehat{\sum}_{j\langle d\rangle\in\Pi(i\langle d\rangle)} \mathrm{Im}(\bar{H})_{j\langle d\rangle}$$

We also define a sum which yields a Hermitian tensor H when applied to a partially-Hermitian tensor order $s+t$ $\bar{H}$ (Hermitian in index groups $j\langle s\rangle$ and $l\langle t\rangle$),

$$H_{i\langle s+t\rangle} = \sum_{(j\langle s\rangle,l\langle t\rangle)\in\chi(i\langle s+t\rangle)}^{*} \bar{H}_{j\langle s\rangle l\langle t\rangle} \equiv \sum_{(j\langle s\rangle,l\langle t\rangle)\in\chi(i\langle s+t\rangle)} \mathrm{Re}(\bar{H})_{j\langle s\rangle l\langle t\rangle} + \widehat{\sum_{(j\langle s\rangle,l\langle t\rangle)\in\chi(i\langle s+t\rangle)}} \mathrm{Im}(\bar{H})_{j\langle s\rangle l\langle t\rangle}.$$

Definition: Using the above summation notation, we now define a **Hermitian contraction** by analogue of equation 11.1.1 and of equation 11.4.4. For complex tensors A and B,

$$C = A \times B \ \equiv\ \forall i\langle s+t\rangle,\ C_{i\langle s+t\rangle} = \frac{1}{(s+t)!} \cdot \sum_{j\langle s\rangle l\langle t\rangle\in\Pi(i\langle s+t\rangle)}^{*} \left(\sum_{k\langle v\rangle} A_{j\langle s\rangle k\langle v\rangle} \cdot B_{l\langle t\rangle k\langle v\rangle} \right). \tag{11.4.8}$$

The resulting complex tensor C is Hermitian.

Hermitian Contractions with Zero Terms

When A and B are Hermitian and when $v > 1$, the result of the Hermitian contraction has

$$\mathrm{Im}(C) = \mathrm{Re}(A) \otimes \mathrm{Im}(B) + \mathrm{Im}(A) \otimes \mathrm{Re}(B) = 0$$

since $\forall i\langle s+t\rangle$,

$$\mathrm{Im}(C)_{i\langle s+t\rangle} = \frac{1}{(s+t)!} \cdot \widehat{\sum_{j\langle s\rangle,l\langle t\rangle\in\Pi(i\langle s+t\rangle)}} \left(\sum_{k\langle v\rangle} \mathrm{Re}(A)_{j\langle s\rangle k\langle v\rangle} \cdot \mathrm{Im}(B)_{l\langle t\rangle k\langle v\rangle} + \sum_{k\langle v\rangle} \mathrm{Im}(A)_{j\langle s\rangle k\langle v\rangle} \cdot \mathrm{Re}(B)_{l\langle t\rangle k\langle v\rangle} \right),$$

so each element is a sum over a $v > 1$ set of indices of two products of a symmetric and an anti-symmetric operands. Each of the two products is anti-symmetric in the $k\langle v\rangle$ index set, so summing

over this set yields a zero result. Therefore, when $v > 1$, it suffices to compute $\mathrm{Re}(C)$, which may be done by computing

$$\mathrm{Re}(C) = \mathrm{Re}(A) \odot \mathrm{Re}(B) - \mathrm{Im}(A) \odot \mathrm{Im}(B).$$

The latter term, $\mathrm{Im}(A) \odot \mathrm{Im}(B)$ is only nonzero when $s, t \leq 1$. This is because $\odot$ is the symmetrized contraction, and if $s, t \geq 1$, at least one anti-symmetric index pair is preserved in $\mathrm{Im}(A) \odot \mathrm{Im}(B)$, which means the term goes to zero once fully symmetrized via equation 11.4.8. For the cases of $v > 1$, it is possible to use the standard algorithm $\Psi_{\odot}^{(s,t,v)}(\mathrm{Re}(A), \mathrm{Re}(B)) - \Psi_{\odot}^{(s,t,v)}(\mathrm{Im}(A), \mathrm{Im}(B))$ or to use the fast algorithm $\Phi_{\odot}^{(s,t,v)}(\mathrm{Re}(A), \mathrm{Re}(B)) - \Phi_{\odot}^{(s,t,v)}(\mathrm{Im}(A), \mathrm{Im}(B))$, in both cases not computing the imaginary part contraction if either $s, t > 1$. It does not make sense to construct an explicit version of $\Psi_{\times}^{(s,t,v)}$ and $\Phi_{\times}^{(s,t,v)}$ for the case with $v > 1$, since the imaginary and real parts act disjointly of each other.

When $v \leq 1$ and both $s, t > 1$, we again have $\mathrm{Im}(C) = 0$, since $\widehat{\sum}_{j\langle s\rangle, l\langle t\rangle \in \Pi(i\langle s+t\rangle)}$ leads to a sum over a product of a symmetric pair of indices with an antisymmetric pair of indices in both contributions to $\mathrm{Im}(C)$, which are then both zero. When $t \leq 1$ and $s > 1$ as well as when $s \leq 1$ and $t > 1$, one of the two terms contributing to $\mathrm{Im}(C)$ is zero and the other needs to be computed, in both cases $\mathrm{Im}(A) \odot \mathrm{Im}(B) = 0$. For instance, when $t, v \leq 1$ and $s > 1$, we have $\mathrm{Re}(A) \otimes \mathrm{Im}(B) = 0$, but need to compute $\mathrm{Im}(A) \otimes \mathrm{Re}(B)$ (an antisymmetrized product of an antisymmetric tensor with a vector or matrix). In all cases, we can employ both the standard and the fast algorithm to compute the two needed contractions, the first being fully symmetric,

$$\mathrm{Re}(C) = \mathrm{Re}(A) \odot \mathrm{Re}(B) = \Psi_{\odot}^{(s,t,v)}(\mathrm{Re}(A), \mathrm{Re}(B)) = \Phi_{\odot}^{(s,t,v)}(\mathrm{Re}(A), \mathrm{Re}(B)),$$

and the second being a contraction of an antisymmetric tensor and a symmetric matrix or vector when $t \leq 1$,

$$\mathrm{Im}(C) = \mathrm{Im}(A) \otimes \mathrm{Re}(B) = \Psi_{\otimes}^{(s,t,v)}(\mathrm{Im}(A), \mathrm{Re}(B)) = \Phi_{\otimes}^{(s,t,v)}(\mathrm{Im}(A), \mathrm{Re}(B)),$$

or when $s \leq 1$,

$$\mathrm{Im}(C) = \mathrm{Re}(A) \otimes \mathrm{Im}(B) = \Psi_{\otimes}^{(s,t,v)}(\mathrm{Re}(A),\ \mathrm{Im}(B)) = \Phi_{\otimes}^{(s,t,v)}(\mathrm{Re}(A), \mathrm{Im}(B)),$$

which can both be done via the fast algorithm adaptation algorithm in Section 11.4.2.

Hermitian Contractions without Zero Terms

As we demonstrated in the previous subsection, pieces of the Hermitian contraction contributing to both the imaginary and real parts evaluate to zero in all cases except when all $s, t, v \leq 1$. We now focus in on these cases.

Definition: When A and B are Hermitian, equation 11.4.8 can for $s, t, v \in \{0, 1\}$ be computed via the analogue of equation 11.1.2 and equation 11.4.5, which we call a **Hermitian contraction**,

$$C = A \times B \ \equiv \ \forall i\langle s+t\rangle,\ C_{i\langle s+t\rangle} = \frac{1}{\binom{s+t}{s}} \cdot \sum_{(j\langle s\rangle, l\langle t\rangle) \in \chi(i\langle s+t\rangle)}^{*} \left(\sum_{k\langle v\rangle} A_{j\langle s\rangle k\langle v\rangle} \cdot B_{l\langle t\rangle k\langle v\rangle} \right). \tag{11.4.9}$$

For $s, t, v \in \{0, 1\}$ we expect each of the two contributions to both the real and imaginary parts of C to be nonzero. We could compute $C = A \times B$ by four applications of either the standard or the fast algorithm,

$$\begin{aligned}
C =& A \times B \\
=& \Psi_{\odot}^{(s,t,v)}(\mathrm{Re}(A),\ \mathrm{Re}(B)) - \Psi_{\odot}^{(s,t,v)}(\mathrm{Im}(A),\ \mathrm{Im}(B)) \\
&+ i\left(\Psi_{\otimes}^{(s,t,v)}(\mathrm{Re}(A),\ \mathrm{Im}(B)) + \Psi_{\otimes}^{(s,t,v)}(\mathrm{Im}(A),\ \mathrm{Re}(B))\right) \\
=& \Phi_{\odot}^{(s,t,v)}(\mathrm{Re}(A),\ \mathrm{Re}(B)) - \Phi_{\odot}^{(s,t,v)}(\mathrm{Im}(A),\ \mathrm{Im}(B)) \\
&+ i\left(\Phi_{\otimes}^{(s,t,v)}(\mathrm{Re}(A),\ \mathrm{Im}(B)) + \Phi_{\otimes}^{(s,t,v)}(\mathrm{Im}(A),\ \mathrm{Re}(B))\right).
\end{aligned}$$

We can also explicitly define complex algorithms $\Psi_{\times}^{(s,t,v)}$ and $\Phi_{\times}^{(s,t,v)}$ for these cases. However, we would gain nothing out of $\Psi_{\times}^{(s,t,v)}$, since it does an addition for each multiplication, and since each complex multiplication costs six real multiplications and each complex addition costs two real additions, the cost would go up by a factor of four, which is the same as the cost as two invocations of both $\Psi_{\odot}^{(s,t,v)}$ and $\Psi_{\otimes}^{(s,t,v)}$ on reals as done above. However, an explicit fast algorithm, $\Phi_{\times}^{(s,t,v)}$ would be expected to yield an improvement due to the fact that it performs more additions than multiplications, and complex additions are a third of the cost of complex multiplications. We could achieve much of this savings by amortizing the cost of computing the operands to $\hat{Z}$ (equation 11.2.3) as well as the operands to other intermediates, since each operand will appear in one call to $\Phi_{\odot}^{(s,t,v)}$ and another call to $\Phi_{\otimes}^{(s,t,v)}$. For instance the left operand to $\hat{Z}$, $\Phi_{\odot}^{(s,t,v)}(\mathrm{Re}(A),\ \mathrm{Re}(B))$ is the same as in $\Phi_{\otimes}^{(s,t,v)}(\mathrm{Re}(A),\ \mathrm{Im}(B))$. Further, we also want to amortize the cost of the accumulation of $\hat{Z}$ to Z by adding together the real and imaginary parts of $\hat{Z}$ a priori, for instance we can immediately subtract the $\hat{Z}$ computed by $\Phi_{\odot}^{(s,t,v)}(\mathrm{Im}(A),\ \mathrm{Im}(B))$ from that computed by $\Phi_{\odot}^{(s,t,v)}(\mathrm{Re}(A),\ \mathrm{Re}(B))$.

In the case when $s = t = v = 1$, the computation of Z dominates, and the number of additions needed to leading order to compute all four $\hat{Z}$ are: $n^3/3$ for operands formed from $\mathrm{Re}(A)$, $n^3/3$ for operands formed from $\mathrm{Im}(A)$, and $2n/3$ more from Z similarly, for a total of $4n^3/3$ additions to compute the operands. To compute Z from $\hat{Z}$, we could add together the resulting $\hat{Z}$ for the imaginary and real parts with total cost $n^3/3$ then accumulate the results with cost $2n^3/3$, for a total of $4n^3/3 + n^3/3 + 2n^3/3 = 7n^3/3$, which is twice as few adds as the naive applications of $\Phi_{\odot}^{(1,1,1)}$ and $\Phi_{\otimes}^{(1,1,1)}$ (4X times the number of additions in F_{syrmm}^{Φ} from Section 11.2.4).

The other cases we need to consider are $s, t, v \in \{(1, 0, 1), (0, 1, 1), (1, 1, 0)\}$, the latter two of which (matrix times vector multiplication and vector times matrix) are effectively identical by symmetry of operands. The $(s, t, v) = (1, 0, 1)$ case for a symmetric matrix, $\Phi_{\odot}^{(1,0,1)}$ was given in Section 11.2.4. We give the Hermitian matrix case $\Phi^{(1,0,1)}$ below (the problem is defined the same

way),

$$\forall i,j, \quad \hat{Z}_{ij} = A_{ij} \cdot (b_i^* + b_j),$$
$$\forall i, \quad Z_i = \sum_k \hat{Z}_{ik}, \quad A_i^{(1)} = \sum_k A_{ik}, \quad V_i = A_i^{(1)} \cdot b_i^*, \quad c_i = Z_i - V_i.$$

The algorithm's correctness is evident by inspection, and it suffices to argue that $\hat{Z}$ is Hermitian. We have $\forall i,j$,

$$\hat{Z}_{ij}^* = A_{ij}^* \cdot (b_i^* + b_j)^* = A_{ji} \cdot (b_i + b_j^*) = \hat{Z}_{ji}.$$

The costs for this case are therefore the same as in Section 11.2.4, but with a different μ_ϱ and μ_ϱ.

The last case we need to consider is $s,t,v = (1,1,0)$, namely the Hermitian rank-2 outer product. Where for two complex vectors a and b, we want to compute

$$C_{ij} = \frac{1}{2}(a_i \cdot b_j^* + a_j^* \cdot b_i).$$

By analogy to Section 11.2.4 $\Phi_\times^{1,1,0}$ is

$$\forall i,j,\ Z_{ij} = \hat{Z}_{ij} = (a_i + a_j^*) \cdot (b_i + b_j^*),\ U_i^{(1)} = a_i \cdot b_i,\ W_{ij} = U_i^{(1)} + (U_j^{(1)})^*,\ C_{ij} = \frac{1}{2}(Z_{ij} - W_{ij}).$$

Correctness comes from the fact that

$$W_{ij} = U_i^{(1)} + (U_j^{(1)})^* = a_i \cdot b_i + (a_j \cdot b_j)^* = a_i \cdot b_i + a_j^* \cdot b_j^*,$$

which is exactly the part of $\hat{Z}$ that differentiates it from C. $\hat{Z}$ is Hermitian due to the fact that $\forall i,j$,

$$\hat{Z}_{ij}^* = [(a_i + a_j^*) \cdot (b_i + b_j^*)]^* = (a_i^* + a_j) \cdot (b_i^* + b_j) = \hat{Z}_{ji}.$$

Therefore, the costs are the same as in Section 11.2.4, except with different μ_ϱ and ν_ϱ. As noted in that section, the fast that the multiplications cost more in the Hermitian case, allows the routines `hemm`, `her2K`, and `hetrd` to be done via $\Phi_\times^{(1,1,0)}$, $\Phi_\times^{(1,0,1)}$, and $\Phi_\times^{(0,1,1)}$ with 3/4 of the operations of the standard methods.

11.5 Applications

In this section, we consider tensor contractions which arise in quantum chemistry calculations. In this context, the tensors are most often antisymmetric (skew-symmetric), because they represent interactions of electrons, which like all fermions follow the Pauli exclusion principle, meaning the (electronic) wave-function is antisymmetric with respect to particle interchange.

11.5.1 Coupled-Cluster Contractions

We now consider some antisymmetric coupled-cluster [16] contractions. We employ some notation which is standard in this field to express the contractions in familiar form for quantum chemist readers. For instance, we will use raised index notation, which is used for book-keeping of index symmetry and meaning in physics calculations. Some rules we will follow include contracting raised indices of one tensor with lowered indices in another, and expressing antisymmetric index groups amongst indices belonging within the sets $\{a, b, c, d\}$ (occupied orbitals) or $\{i, j, k, l\}$ (virtual orbitals) and all indices in the group appearing as lowered or raised within the tensor. Additionally, we will consider different spin-cases, with spin-α indices labeled normally as $\{a, b, c, d\}$, $\{i, j, k, l, m\}$, and spin-β indices labeled with bars as $\{\bar{a}, \bar{b}, \bar{c}, \bar{d}\}$, $\{\bar{i}, \bar{j}, \bar{k}, \bar{l}, \bar{m}\}$. Indices with different spin are not antisymmetric unlike indices of the same spin and type.

We start with a contraction from the CCSD method, where the fourth order tensor T_2 (with elements referred to as e.g. T_{ij}^{ab} for the pure spin-α case) is contracted with a fourth-order two-electron integral tensor V into an fourth-order intermediate Z (this is a contraction of leading order cost within CCSD),

$$Z_{i\bar{c}}^{a\bar{k}} = \sum_{bj} T_{ij}^{ab} \cdot V_{b\bar{c}}^{j\bar{k}},$$

where the T_2 'amplitude' tensor is antisymmetric in a, b and i, j (the fact that this symmetry is present and no other symmetries are present in the contraction could actually be inferred directly from the notation rules we described). We will use nested contraction algorithms as described in Section 11.2.5 to express algorithms for this problem. For simplicity, we will ignore the antisymmetry i, j and exploit only a, b (although both partial symmetries could be exploited recursively as done in the next contractions). This tensor represents two-electron excitations, and we will refer to it as T_2. The antisymmetry in T_2 is not preserved in the contraction, so the cost of the standard algorithm nested with a nonsymmetric contraction,

$$\Psi_{\otimes}^{(0,1,1)} \Upsilon^{(2,1,1)}(T_2, V) = \Upsilon^{(2,2,2)}(T_2, V)$$

would be $2n^6$ operations. We can apply the algorithm $\Phi_{\otimes}^{(0,1,1)}(A, B)$ to reduce this operation count by a factor of two, by defining vectors $\bar{v}$ and $\bar{z}$, the elements of which are order 3 tensors, and antisymmetric matrix $\bar{T}_2$, each element of which is itself an antisymmetric matrix. We can then assign

$$\forall a, b, i, j, \;\; (\bar{T}^{ab})_{ij} = T_{ij}^{ab},$$

as well as

$$\forall j, \bar{c}, b, k, \;\; (\bar{v}_b)_{\bar{c}}^{j\bar{k}} = V_{b\bar{c}}^{j\bar{k}},$$

and similarly for $\bar{z}$. Now we can apply $\Phi_{\otimes}^{(0,1,1)}(\bar{v}, \bar{T}_2)$ with $s = 0, v = 1, t = 1$, where each element-wise multiplication is a contraction over the remaining four indices, which we refer to as the nested contraction

$$\Phi_{\otimes}^{(0,1,1)} \Upsilon^{(2,1,1)}(T_2, V).$$

Since the number of element-wise multiplications is decreased by a factor of $\omega = 2$ and the additions of elements of $\bar{z}$, $\bar{v}$, and $\bar{T}_2$ cost n^3, n^3, and $\binom{n+1}{2}$, respectively, while the element-wise application of "$\cdot$" costs $O(n^4)$, the overall operation count for the contraction via $\Phi_{\otimes}^{(0,1,1)}\Upsilon^{(2,1,1)}(T_2, V)$ is n^6 rather than $2n^6$.

In the higher-order CCSDT [121] method, another amplitude tensor of order 6 is computed, which we call T_3. We refer to the elements of T_3 as T_{ijk}^{abc} (in the pure spin-α case). The T_3 tensor is antisymmetric in two index groups of up to three indices (depending on spin), namely the a, b, c indices and i, j, k indices. One contraction between T_2 and a fourth-order intermediate W contributing to T_3, which appears both in the CCSDT method, and its perturbatively performed subset CCSD(T) [133] (it is leading order in the latter method) is

$$T_{ij\bar{k}}^{ab\bar{c}} = \sum_{(a,b)\in\chi(a,b)} \sum_{(i,j)\in\chi(i,j)} \sum_{\bar{l}} T_{i\bar{l}}^{a\bar{c}} \cdot W_{j\bar{k}}^{\bar{l}b}.$$

The T_3 output tensor is antisymmetric in i, j and in a, b in this contraction. Since these symmetries are not preserved in the contraction, it would cost $2n^7$ operations if computed via the standard algorithm,

$$T_3 = \Psi_{\otimes}^{(1,1,0)}\Psi_{\otimes}^{(1,1,0)}\Upsilon^{(1,1,1)}(T_2, W).$$

Instead, we can use two nested levels of the fast algorithm as

$$T_3 = \Phi_{\otimes}^{(1,1,0)}\Phi_{\otimes}^{(1,1,0)}\Upsilon^{(1,1,1)}(T_2, W),$$

which performs the contraction as follows,

$$((T^{ab})_{ij})_{\bar{k}}^{\bar{c}} = \sum_{\bar{l}} ((T^{a})_{i})_{\bar{l}}^{\bar{c}} \cdot ((V^{b})_{j})_{\bar{k}}^{\bar{l}}.$$

The fast symmetric contraction algorithm is applied over the a, b indices with $s = 1, t = 1, v = 0$, then over the i, j indices with $s = 1, t = 1, v = 0$, then (for each scalar multiplication in the fast algorithm) over the nonsymmetric $\bar{c}, \bar{k}, \bar{l}$ indices. This amounts to performing a nonsymmetric matrix multiplication at the bottom level, and the factor of two reduction in multiplications in each of the two levels of the fast symmetric algorithm, results in a 4X reduction in overall operation count needed for the contraction.

Lastly, we analyze a contraction from the CCSDTQ [102] method, which additionally appears in the CCSDT(Q) [18] perturbatively performed subset of CCSDTQ, and is in the leading order in the CCSDT(Q). The contraction we analyze is similar in form to the CCSDT one above, except it contributes to the T_4 amplitude tensor which is computed in CCSDTQ and is a result of a contraction between T_2 and a sixth order intermediate X,

$$T_{ijk\bar{l}}^{abc\bar{d}} = \sum_{(a,(b,c))\in\chi(a,b,c)} \sum_{(i,(j,k))\in\chi(i,j,k)} \sum_{\bar{m}} T_{i\bar{m}}^{a\bar{d}} \cdot X_{jk\bar{l}}^{\bar{m}bc}.$$

In this contraction, there are again two antisymmetric groups, but this time of size 3, a, b, c and i, j, k. Further, the antisymmetries between b and c and between j and k are preserved since X and

T are antisymmetric in these index groups. Therefore, the standard contraction algorithm,

$$T_4 = \Psi_{\otimes}^{(2,1,0)} \Psi_{\otimes}^{(1,2,0)} \Upsilon^{(1,1,1)}(T_2, X),$$

would save a factor of four from symmetry, yielding $n^9/2$ operations to leading order. As done for the CCSDT contraction, we can again use two nested levels of the fast symmetric algorithm and perform a nonsymmetric matrix multiplication at the third, bottom level. We can express this nesting as

$$T_4 = \Phi_{\otimes}^{(2,1,0)} \Phi_{\otimes}^{(1,2,0)} \Upsilon^{(1,1,1)}(T_2, X),$$

where the fast algorithm effectively computes

$$((T^{abc})_{ijk})_{\bar{l}}^{\bar{d}} = \sum_{\bar{m}} ((T^a)_i)_{\bar{m}}^{\bar{d}} \cdot ((X^{bc})_{jk})_{\bar{l}}^{\bar{m}}.$$

The fast algorithm is applied with $s = 1, t = 2, v = 0$ over the a, b, c indices, each scalar multiplication therein becoming another symmetric contraction with $s = 1, t = 2, v = 0$ over the i, j, k indices and each scalar multiplication at this second level being a nonsymmetric contraction over $\bar{d}, \bar{m}, \bar{l}$. Since the number of scalar multiplications is reduced by a factor of $(s + v + t)! = 3! = 6$ at each level, the overall cost is reduced by 36 ($n^9/18$ operations), which is 9X faster than the standard method.

11.6 Conclusions

We conclude that the new fast symmetric tensor contraction algorithm is of interest for consideration in proving the performance of the complex BLAS and LAPACK routines, but more importantly has significant potential for accelerating coupled-cluster methods and possibly other quantum chemistry computations. The cost quantification, numerical error analysis, and adaptation to antisymmetric and Hermitian cases demonstrate that the technique is robust. While we implemented test cases of the fast symmetric algorithm and its adaptations on top of Cyclops Tensor Framework (see Chapter 10), we have not tested the performance of the algorithm, which needs to be evaluated in a variety of contexts (for instance both as a sequential optimization to complex BLAS routines as well as nested use inside coupled-cluster contractions). We also leave it as future work to consider the best ways of applying the new algorithm to the full set of coupled-cluster equations for various methods, as well as implementation thereof.

Chapter 12

Future Work

In this last chapter we discuss future directions for the work presented in all previous chapters; some of these directions were already mentioned in the conclusion sections of those chapters. Generally, we have used asymptotic analysis of communication costs in a number of sections and did not consider overlap between communication and computation, which are natural directions for improvement of the work. While we carefully modelled collective communication on torus networks, extensions of the models to other network topologies remain future work.

Our communication lower bound techniques should extend in a general way to iterative methods (Krylov subspace methods or semiring-based graph algorithms) on more general classes of graphs than those we considered in Chapter 7. In particular, we expect that algorithms on graphs with hierarchical community structure [66] should yield exponential dependency bubble expansion. Further, while we demonstrated the applicability of the lower bounds to Gaussian elimination, we did not explicitly derive the lower bounds for algorithms for QR and the symmetric eigenvalue problem. We gave algorithms that achieved costs, which match those of Gaussian elimination, and as these problems are generally harder, we expect that the lower bounds should apply to these two problems and other dense matrix factorizations. The difficulty in extending the bounds to QR lays in the variety of approaches to QR factorization (Givens rotations and Householder as well as blocked versions of these), which all yield different algorithmic dependency graphs. With some assumptions such as forward progress, which were made in [12] to obtain bandwidth lower bounds for QR, we conjecture that the concept of parent hypergraphs and path-expanders given in Chapter 3 are extensible to these algorithms.

The adaptation and evaluation of the 2.5D symmetric eigensolver algorithms to algorithms for the singular value decomposition problem also remains future work. Further, for QR factorization, we did not give an implementation of the proposed 2.5D algorithm, so this remains an interesting direction for future work. Similarly for the symmetric eigenvalue problem, we did not give an implementation of the 2.5D successive band reduction algorithm (Algorithm 7.3.1), which is expected to be useful due to its good theoretical memory bandwidth and interprocessor bandwidth communication efficiency. However, since the algorithms requires more reduction stages, and therefore more work, realizing a practical benefit may be challenging on current architectures (on future architectures we expect communication to be even more expensive relative to computation).

Eigenvector computation by back transformation was also not considered in this thesis and remains an important direction for future work on the symmetric eigenvalue decomposition problem.

We also did not give any implementation of Algorithm 8.4.1 for computation of Krylov subspace methods. In principle, this algorithm should be quite beneficial for certain problem sizes (when the whole problem does not fit into the caches of all processors, but is also not too much larger than the sum of these cache sizes). Another remaining question is whether the memory-bandwidth efficient algorithm can be adapted and employed in methods that depend on the entire Krylov basis rather than just the last vector.

While we implemented and benchmarked the recursive all-pairs shortest-paths algorithm in Chapter 9, we did so for dense graphs, while sparse graphs are the more important problem (although the path matrix is still dense for a connected path graph). However, even for dense graphs, there remain interesting direction for future work in the analysis and implementation of the path-doubling algorithm presented by Tiskin [157]. This algorithm is more synchronization efficient than the recursive Floyd-Warshall-based algorithm we analyzed. Tiskin's adaptation of path-doubling also does no extra computation asymptotically and seems to be promising from a practical perspective. A theoretical question is whether this path-doubling technique can also speed up shortest-path computations on sparse graphs.

Much future work also remains in the tensor work described in Chapters 10 and 11. For Cyclops Tensor Framework, some of the most important future directions are support for sparse tensors, improvement of performance models and performance prediction, as well as implementation of other coupled-cluster and benchmarking of methods such as CCSDTQ. The framework also does not support the fast symmetric contraction algorithms, which are presented in Chapter 11, for which no high performance implementations exist (although we tested prototypes of the algorithm for correctness). The development of applications on top of the fast symmetric contraction algorithm (particularly coupled-cluster methods) also remains a promising piece of future work.

Bibliography

[1] R. C. Agarwal, S. M. Balle, F. G. Gustavson, M. Joshi, and P. Palkar. A three-dimensional approach to parallel matrix multiplication. *IBM J. Res. Dev.*, 39:575–582, September 1995.

[2] A. Aggarwal, A. K. Chandra, and M. Snir. Communication complexity of PRAMs. *Theoretical Computer Science*, 71(1):3 – 28, 1990.

[3] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley Longman, Boston, MA, USA, 1974.

[4] A. Alexandrov, M. F. Ionescu, K. E. Schauser, and C. Scheiman. LogGP: incorporating long messages into the LogP model – one step closer towards a realistic model for parallel computation. In *Proceedings of the Seventh Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '95, pages 95–105, New York, NY, USA, 1995. ACM.

[5] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. D. Croz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov, and D. Sorensen. *LAPACK Users' Guide*. SIAM, Philadelphia, PA, USA, 1992.

[6] T. Auckenthaler, H.-J. Bungartz, T. Huckle, L. Krämer, B. Lang, and P. Willems. Developing algorithms and software for the parallel solution of the symmetric eigenvalue problem. *Journal of Computational Science*, 2(3):272 – 278, 2011. Social Computational Systems.

[7] G. Ballard, J. Demmel, L. Grigori, M. Jacquelin, H. D. Nguyen, and E. Solomonik. Reconstructing Householder vectors from tall-skinny QR. Technical report, EECS Department, University of California, Berkeley, 2013.

[8] G. Ballard, J. Demmel, L. Grigori, M. Jacquelin, H. D. Nguyen, and E. Solomonik. Reconstructing Householder vectors from tall-skinny QR. In *Proceedings of the 2014 IEEE 28th International Parallel and Distributed Processing Symposium*, IPDPS '14, pages 1159–1170, Washington, DC, USA, 2014. IEEE Computer Society.

[9] G. Ballard, J. Demmel, O. Holtz, B. Lipshitz, and O. Schwartz. Brief announcement: Strong scaling of matrix multiplication algorithms and memory-independent communication lower bounds. In *Proceedings of the 24th ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '12, pages 77–79, New York, NY, USA, 2012. ACM.

[10] G. Ballard, J. Demmel, O. Holtz, B. Lipshitz, and O. Schwartz. Communication-optimal parallel algorithm for Strassen's matrix multiplication. In *Proceedings of the 24th ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '12, pages 193–204, New York, NY, USA, 2012. ACM.

[11] G. Ballard, J. Demmel, O. Holtz, and O. Schwartz. Graph expansion and communication costs of fast matrix multiplication: regular submission. In *Proceedings of the 23rd ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '11, pages 1–12, New York, NY, USA, 2011. ACM.

[12] G. Ballard, J. Demmel, O. Holtz, and O. Schwartz. Minimizing communication in linear algebra. *SIAM J. Mat. Anal. Appl.*, 32(3), 2011.

[13] G. Ballard, J. Demmel, and N. Knight. Communication avoiding successive band reduction. *SIGPLAN Not.*, 47(8):35–44, Feb. 2012.

[14] E. Bampis, C. Delorme, and J.-C. König. Optimal schedules for d-D grid graphs with communication delays. In C. Puech and R. Reischuk, editors, *STACS 96*, volume 1046 of *Lecture Notes in Computer Science*, pages 655–666. Springer Berlin Heidelberg, 1996.

[15] M. Barnett, D. G. Payne, R. A. van de Geijn, and J. Watts. Broadcasting on meshes with worm-hole routing. Technical report, Austin, TX, USA, 1993.

[16] R. J. Bartlett. Many-body perturbation theory and coupled cluster theory for electron correlation in molecules. 32(1):359–401, 1981.

[17] R. J. Bartlett and M. Musiał. Coupled-cluster theory in quantum chemistry. 79(1):291–352, 2007.

[18] R. J. Bartlett, J. Watts, S. Kucharski, and J. Noga. Non-iterative fifth-order triple and quadruple excitation energy corrections in correlated methods. *Chemical Physics Letters*, 165(6):513–522, 1990.

[19] G. Baumgartner, A. Auer, D. Bernholdt, A. Bibireata, V. Choppella, D. Cociorva, X. Gao, R. Harrison, S. Hirata, S. Krishnamoorthy, S. Krishnan, C. Lam, Q. Lu, M. Nooijen, R. Pitzer, J. Ramanujam, P. Sadayappan, and A. Sibiryakov. Synthesis of high-performance parallel programs for a class of ab initio quantum chemistry models. *Proceedings of the IEEE*, 93(2):276 –292, Feb. 2005.

[20] R. Bellman. On a routing problem. Technical report, DTIC Document, 1956.

[21] M. A. Bender, G. S. Brodal, R. Fagerberg, R. Jacob, and E. Vicari. Optimal sparse matrix dense vector multiplication in the I/O-model. *Theory of Computing Systems*, 47(4):934–962, 2010.

[22] J. Bennett, A. Carbery, M. Christ, and T. Tao. The Brascamp–Lieb inequalities: finiteness, structure and extremals. *Geometric and Functional Analysis*, 17(5):1343–1415, 2008.

[23] J. Berntsen. Communication efficient matrix multiplication on hypercubes. *Parallel Computing*, 12(3):335–342, 1989.

[24] G. Bilardi and F. P. Preparata. Processor–time tradeoffs under bounded-speed message propagation: Part II, lower bounds. *Theory of Computing Systems*, 32(5):531–559, 1999.

[25] C. Bischof, B. Lang, and X. Sun. Algorithm 807: The SBR Toolbox – Software Successive Band Reduction. *ACM Trans. Math. Soft.*, 26(4):602–616, Dec 2000.

[26] C. Bischof, B. Lang, and X. Sun. A Framework for Symmetric Band Reduction. *ACM Trans. Math. Soft.*, 26(4):581–601, Dec 2000.

[27] C. H. Bischof and X. Sun. On orthogonal block elimination. Technical Report MCS-P450-0794, Argonne National Laboratory, Argonne, IL, 1994.

[28] L. S. Blackford, J. Choi, A. Cleary, E. D'Azeuedo, J. Demmel, I. Dhillon, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley. *ScaLAPACK User's Guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1997.

[29] J. Brickell, I. S. Dhillon, S. Sra, and J. A. Tropp. The metric nearness problem. *SIAM J. Matrix Anal. Appl.*, 30:375–396, 2008.

[30] A. Buluç, J. R. Gilbert, and C. Budak. Solving path problems on the GPU. *Parallel Computing*, 36(5-6):241 – 253, 2010.

[31] E. J. Bylaska et. al. NWChem, a computational chemistry package for parallel computers, version 6.1.1, 2012.

[32] L. E. Cannon. *A cellular computer to implement the Kalman filter algorithm*. PhD thesis, Bozeman, MT, USA, 1969.

[33] E. Carson, N. Knight, and J. Demmel. Avoiding communication in nonsymmetric Lanczos-based Krylov subspace methods. *SIAM Journal on Scientific Computing*, 35(5):S42–S61, 2013.

[34] T. M. Chan. More algorithms for all-pairs shortest paths in weighted graphs. *SIAM Journal on Computing*, 39(5):2075–2089, 2010.

[35] D. Chen, N. A. Eisley, P. Heidelberger, R. M. Senger, Y. Sugawara, S. Kumar, V. Salapura, D. L. Satterfield, B. Steinmacher-Burow, and J. J. Parker. The IBM Blue Gene/Q interconnection network and message unit. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pages 26:1–26:10, New York, NY, USA, 2011. ACM.

[36] R. A. Chowdhury and V. Ramachandran. The cache-oblivious Gaussian elimination paradigm: theoretical framework, parallelization and experimental evaluation. In *Proceedings of the Nineteenth Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '07, pages 71–80, New York, NY, USA, 2007. ACM.

[37] M. Christ, J. Demmel, N. Knight, T. Scanlon, and K. Yelick. Communication lower bounds and optimal algorithms for programs that reference arrays–part 1. *arXiv preprint arXiv:1308.0068*, 2013.

[38] D. Coppersmith and S. Winograd. Matrix multiplication via arithmetic progressions. In *STOC '87: Proceedings of the Nineteenth Annual ACM Conference on Theory of Computing*, pages 1–6, New York, NY, USA, 1987. ACM Press.

[39] T. D. Crawford and H. F. Schaefer. An introduction to coupled cluster theory for computational chemists. volume 14, chapter 2, pages 33–136. VCH Publishers, New York, 2000.

[40] D. Culler, R. Karp, D. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian, and T. von Eicken. LogP: towards a realistic model of parallel computation. In *Proceedings of the Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPOPP '93, pages 1–12, New York, NY, USA, 1993. ACM.

[41] A. R. Curtis, T. Carpenter, M. Elsheikh, A. López-Ortiz, and S. Keshav. REWIRE: an optimization-based framework for data center network design. In *INFOCOM*, 2012.

[42] R. D. da Cunha, D. Becker, and J. C. Patterson. New parallel (rank-revealing) QR factorization algorithms. In *Euro-Par 2002 Parallel Processing*, pages 677–686. Springer, 2002.

[43] E. Dekel, D. Nassimi, and S. Sahni. Parallel matrix and graph algorithms. *SIAM Journal on Computing*, 10(4):657–675, 1981.

[44] D. Delling, A. V. Goldberg, A. Nowatzyk, and R. F. Werneck. Phast: Hardware-accelerated shortest path trees. *Parallel and Distributed Processing Symposium, International*, 0:921–931, 2011.

[45] J. Demmel. Trading off parallelism and numerical stability. In M. Moonen, G. Golub, and B. De Moor, editors, *Linear Algebra for Large Scale and Real-Time Applications*, volume 232 of *NATO ASI Series*, pages 49–68. Springer Netherlands, 1993.

[46] J. Demmel, D. Eliahu, A. Fox, S. Kamil, B. Lipshitz, O. Schwartz, and O. Spillinger. Communication-optimal parallel recursive rectangular matrix multiplication. In *IEEE International Symposium on Parallel Distributed Processing (IPDPS)*, 2013.

[47] J. Demmel, L. Grigori, M. Hoemmen, and J. Langou. Communication-optimal parallel and sequential QR and LU factorizations. *SIAM Journal on Scientific Computing*, 34(1):A206–A239, 2012.

[48] J. Demmel, L. Grigori, M. F. Hoemmen, and J. Langou. Communication-optimal parallel and sequential QR and LU factorizations. Technical Report UCB/EECS-2008-89, EECS Department, University of California, Berkeley, Aug 2008.

[49] J. Demmel, M. Hoemmen, M. Mohiyuddin, and K. Yelick. Avoiding communication in sparse matrix computations. In *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, pages 1–12. IEEE, 2008.

[50] E. Deumens, V. F. Lotrich, A. Perera, M. J. Ponton, B. A. Sanders, and R. J. Bartlett. Software design of ACES III with the super instruction architecture. *Wiley Interdisciplinary Reviews: Computational Molecular Science*, 1(6):895–901, 2011.

[51] I. S. Dhillon, B. N. Parlett, and C. Vömel. The design and implementation of the MRRR algorithm. *ACM Trans. Math. Softw.*, 32(4):533–560, Dec. 2006.

[52] J. Dongarra, M. Faverge, T. Hérault, M. Jacquelin, J. Langou, and Y. Robert. Hierarchical QR factorization algorithms for multi-core clusters. *Parallel Computing*, 39(4-5):212–232, Apr. 2013.

[53] J. J. Dongarra, D. C. Sorensen, and S. J. Hammarling. Block reduction of matrices to condensed forms for eigenvalue computations. *Journal of Computational and Applied Mathematics*, 27(1):215–227, 1989.

[54] J. J. Dongarra and R. A. van de Geijn. Reduction to condensed form for the eigenvalue problem on distributed memory architectures. *Parallel Computing*, 18(9):973 – 982, 1992.

[55] E. Elmroth and F. Gustavson. New serial and parallel recursive QR; factorization algorithms for SMP systems. In B. Kågström, J. Dongarra, E. Elmroth, and J. Wasniewski, editors, *Applied Parallel Computing Large Scale Scientific and Industrial Problems*, volume 1541 of *Lecture Notes in Computer Science*, pages 120–128. Springer Berlin / Heidelberg, 1998. 10.1007/BFb0095328.

[56] E. Epifanovsky, M. Wormit, T. Kuś, A. Landau, D. Zuev, K. Khistyaev, P. Manohar, I. Kaliman, A. Dreuw, and A. I. Krylov. New implementation of high-level correlated methods using a general block-tensor library for high-performance electronic structure calculations. *Journal of Computational Chemistry*, 2013.

[57] A. Faraj, S. Kumar, B. Smith, A. Mamidala, and J. Gunnels. MPI collective communications on the Blue Gene/P supercomputer: Algorithms and optimizations. In *High Performance Interconnects, 2009. HOTI 2009. 17th IEEE Symposium on*, 2009.

[58] J. G. Fletcher. A more general algorithm for computing closed semiring costs between vertices of a directed graph. *Communications of the ACM*, 23(6):350–351, 1980.

[59] R. W. Floyd. Algorithm 97: Shortest path. *Commun. ACM*, 5:345–, June 1962.

[60] V. Fock. Näherungsmethode zur Lösung des quantenmechanischen Mehrkörperproblems. *Zeitschrift für Physik*, 61(1-2):126–148, 1930.

[61] C. Fonseca Guerra, J. G. Snijders, G. te Velde, and E. J. Baerends. Towards an order-N DFT method. *Theoretical Chemistry Accounts*, 99(6):391–403, 1998.

[62] L. R. Ford. Network flow theory. 1956.

[63] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran. Cache-oblivious algorithms. In *Proceedings of the 40th Annual Symposium on Foundations of Computer Science*, FOCS '99, page 285, Washington, DC, USA, 1999. IEEE Computer Society.

[64] X. Gao, S. Krishnamoorthy, S. Sahoo, C.-C. Lam, G. Baumgartner, J. Ramanujam, and P. Sadayappan. Efficient search-space pruning for integrated fusion and tiling transformations. In *Languages and Compilers for Parallel Computing*, volume 4339 of *Lecture Notes in Computer Science*, pages 215–229. Springer Berlin / Heidelberg, 2006.

[65] R. Geisberger, P. Sanders, D. Schultes, and D. Delling. Contraction hierarchies: faster and simpler hierarchical routing in road networks. In *Proceedings of the 7th International Conference on Experimental Algorithms*, WEA'08, pages 319–333, Berlin, Heidelberg, 2008. Springer-Verlag.

[66] M. Girvan and M. E. Newman. Community structure in social and biological networks. *Proceedings of the National Academy of Sciences*, 99(12):7821–7826, 2002.

[67] G. Golub and C. Van Loan. *Matrix Computations*. Johns Hopkins Studies in the Mathematical Sciences. Johns Hopkins University Press, 2012.

[68] G. H. Golub, R. J. Plemmons, and A. Sameh. Parallel block schemes for large-scale least-squares computations. *High-Speed Computing: Scientific Application and Algorithm Design*, pages 171–179, 1986.

[69] L. Grigori, J. W. Demmel, and H. Xiang. Communication avoiding Gaussian Elimination. pages 29:1–29:12, 2008.

[70] L. Grigori, J. W. Demmel, and H. Xiang. CALU: A communication optimal LU factorization algorithm. *SIAM Journal on Matrix Analysis and Applications*, 32(4):1317–1350, 2011.

[71] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI: Portable parallel programming with the message-passing interface*. MIT Press, Cambridge, MA, USA, 1994.

[72] B. C. Gunter and R. A. Van De Geijn. Parallel out-of-core computation and updating of the qr factorization. *ACM Trans. Math. Softw.*, 31(1):60–78, Mar. 2005.

[73] M. B. Habbal, H. N. Koutsopoulos, and S. R. Lerman. A decomposition algorithm for the all-pairs shortest path problem on massively parallel computer architectures. *Transportation Science*, 28(4):292–308, 1994.

[74] M. Hanrath and A. Engels-Putzka. An efficient matrix-matrix multiplication based antisymmetric tensor contraction engine for general order coupled cluster. *The Journal of Chemical Physics*, 133(6), 2010.

[75] R. Haring, M. Ohmacht, T. Fox, M. Gschwind, D. Satterfield, K. Sugavanam, P. Coteus, P. Heidelberger, M. Blumrich, R. Wisniewski, A. Gara, G.-T. Chiu, P. Boyle, N. Chist, and C. Kim. The IBM Blue Gene/Q compute chip. *Micro, IEEE*, 32(2):48 –60, March-April 2012.

[76] D. R. Hartree. The wave mechanics of an atom with a non-coulomb central field. Part I. Theory and methods. *Mathematical Proceedings of the Cambridge Philosophical Society*, 24:89–110, 1 1928.

[77] M. Head-Gordon, J. A. Pople, and M. J. Frisch. MP2 energy evaluation by direct methods. *Chemical Physics Letters*, 153(6):503 – 506, 1988.

[78] M. Heath and C. Romine. Parallel solution of triangular systems on distributed-memory multiprocessors. *SIAM Journal on Scientific and Statistical Computing*, 9(3):558–588, 1988.

[79] S. Hirata. Tensor Contraction Engine: Abstraction and automated parallel implementation of configuration-interaction, coupled-cluster, and many-body perturbation theories. *The Journal of Physical Chemistry A*, 107(46):9887–9897, 2003.

[80] R. W. Hockney. The communication challenge for MPP: Intel Paragon and Meiko CS-2. *Parallel Computing*, 20(3):389 – 398, 1994.

[81] M. Hoemmen. *Communication-avoiding Krylov subspace methods*. PhD thesis, EECS Department, University of California, Berkeley, Apr 2010.

[82] M. Hoemmen. A communication-avoiding, hybrid-parallel, rank-revealing orthogonalization method. In *Parallel Distributed Processing Symposium (IPDPS), 2011 IEEE International*, pages 966–977, 2011.

[83] P. Hohenberg and W. Kohn. Inhomogeneous electron gas. *Phys. Rev.*, 136:B864–B871, Nov 1964.

[84] O. Hölder. Über einen Mittelwertsatz. *Nachrichten von der König. Gesellschaft der Wissenschaften und der Georg-Augusts-Universität zu Göttigen*, pages 38–47, 1889.

[85] S. Holzer and R. Wattenhofer. Optimal distributed all pairs shortest paths and applications. In *Proceedings of the 2012 ACM Symposium on Principles of Distributed Computing*, pages 355–364. ACM, 2012.

[86] IBM Journal of Research and Development. Overview of the IBM Blue Gene/P project. *IBM J. Res. Dev.*, 52:199–220, January 2008.

[87] D. Irony and S. Toledo. Trading replication for communication in parallel distributed-memory dense solvers. *Parallel Processing Letters*, 71:3–28, 2002.

[88] D. Irony, S. Toledo, and A. Tiskin. Communication lower bounds for distributed-memory matrix multiplication. *Journal of Parallel and Distributed Computing*, 64(9):1017 – 1026, 2004.

[89] J. Jenq and S. Sahni. All pairs shortest paths on a hypercube multiprocessor. In *ICPP '87: Proc. of the Intl. Conf. on Parallel Processing*, pages 713–716, 1987.

[90] H. Jia-Wei and H. T. Kung. I/O complexity: The red-blue pebble game. In *Proceedings of the thirteenth Annual ACM Symposium on Theory of Computing*, STOC '81, pages 326–333, New York, NY, USA, 1981. ACM.

[91] T. Joffrain, T. M. Low, E. S. Quintana-Ortí, R. v. d. Geijn, and F. G. V. Zee. Accumulating Householder transformations, revisited. *ACM Trans. Math. Softw.*, 32(2):169–179, June 2006.

[92] D. B. Johnson. Efficient algorithms for shortest paths in sparse networks. *Journal of the ACM*, 24(1):1–13, 1977.

[93] S. L. Johnsson. Minimizing the communication time for matrix multiplication on multiprocessors. *Parallel Comput.*, 19:1235–1257, November 1993.

[94] S. L. Johnsson and C.-T. Ho. Optimum broadcasting and personalized communication in hypercubes. *IEEE Trans. Comput.*, 38:1249–1268, September 1989.

[95] L. V. Kale and S. Krishnan. CHARM++: A portable concurrent object oriented system based on C++. In *Proceedings of the Eighth Annual Conference on Object-Oriented Programming Systems, Languages, and Applications*, OOPSLA '93, pages 91–108, New York, NY, USA, 1993. ACM.

[96] M. Kállay and P. R. Surján. Higher excitations in coupled-cluster theory. *The Journal of Chemical Physics*, 115(7):2945, 2001.

[97] D. Kats and F. R. Manby. Sparse tensor framework for implementation of general local correlation methods. *The Journal of Chemical Physics*, 138(14):–, 2013.

[98] J. Kim, W. J. Dally, S. Scott, and D. Abts. Technology-driven, highly-scalable dragonfly topology. In *Proceedings of the 35th Annual International Symposium on Computer Architecture*, ISCA '08, pages 77–88, Washington, DC, USA, 2008. IEEE Computer Society.

[99] P. Knowles and N. Handy. A new determinant-based full configuration interaction method. *Chemical Physics Letters*, 111(4-5):315 – 321, 1984.

[100] W. Kohn and L. J. Sham. Self-consistent equations including exchange and correlation effects. *Phys. Rev.*, 140:A1133–A1138, Nov 1965.

[101] T. Kolda and B. Bader. Tensor decompositions and applications. *SIAM Review*, 51(3):455–500, 2009.

[102] S. A. Kucharski and R. J. Bartlett. Coupled-cluster methods that include connected quadruple excitations, T4: CCSDTQ-1 and Q (CCSDT). *Chemical Physics Letters*, 158(6):550–555, 1989.

[103] S. Kumar, G. Dozsa, G. Almasi, P. Heidelberger, D. Chen, M. E. Giampapa, B. Michael, A. Faraj, J. Parker, J. Ratterman, B. Smith, and C. J. Archer. The deep computing messaging framework: generalized scalable message passing on the Blue Gene/P supercomputer. In *Proceedings of the 22nd Annual International Conference on Supercomputing*, ICS '08, pages 94–103, New York, NY, USA, 2008. ACM.

[104] V. Kumar and V. Singh. Scalability of parallel algorithms for the all-pairs shortest-path problem. *J. Parallel Distrib. Comput.*, 13:124–138, 1991.

[105] P.-W. Lai, K. Stock, S. Rajbhandari, S. Krishnamoorthy, and P. Sadayappan. A framework for load balancing of tensor contraction expressions via dynamic task partitioning. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '13, pages 13:1–13:10, New York, NY, USA, 2013. ACM.

[106] B. Lang. A parallel algorithm for reducing symmetric banded matrices to tridiagonal form. *SIAM Journal on Scientific Computing*, 14(6):1320–1338, 1993.

[107] R. C. Larson and A. R. Odoni. *Urban Operations Research*. Prentice-Hall, Englewood Cliffs, NJ, USA, 1981.

[108] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh. Basic Linear Algebra Subprograms for Fortran usage. *ACM Transactions on Mathematical Software (TOMS)*, 5(3):308–323, 1979.

[109] C. E. Leiserson, S. Rao, and S. Toledo. Efficient out-of-core algorithms for linear relaxation using blocking covers. In *Foundations of Computer Science, 1993. Proceedings., 34th Annual Symposium on*, pages 704–713. IEEE, 1993.

[110] R. J. Lipton, D. J. Rose, and R. E. Tarjan. Generalized nested dissection. *SIAM J. Numer. Analysis*, 16:346–358, 1979.

[111] L. Loomis and H. Whitney. An inequality related to the isoperimetric inequality. 1949.

[112] V. Lotrich, N. Flocke, M. Ponton, B. A. Sanders, E. Deumens, R. J. Bartlett, and A. Perera. An infrastructure for scalable and portable parallel programs for computational chemistry. In *Proceedings of the 23rd International Conference on Supercomputing*, ICS '09, pages 523–524, New York, NY, USA, 2009. ACM.

[113] W. F. McColl and A. Tiskin. Memory-efficient matrix multiplication in the BSP model. *Algorithmica*, 24:287–297, 1999.

[114] P. McKinley, Y.-J. Tsai, and D. Robinson. Collective communication in wormhole-routed massively parallel computers. *Computer*, 28(12):39 –50, Dec. 1995.

[115] U. Meyer and P. Sanders. Δ-stepping: a parallelizable shortest path algorithm. *J. Algorithms*, 49(1):114–152, 2003.

[116] M. Mohiyuddin, M. Hoemmen, J. Demmel, and K. Yelick. Minimizing communication in sparse matrix solvers. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, page 36. ACM, 2009.

[117] M. Mohri. Semiring frameworks and algorithms for shortest-distance problems. *Journal of Automata, Languages and Combinatorics*, 7(3):321–350, 2002.

[118] C. Møller and M. S. Plesset. Note on an approximation treatment for many-electron systems. *Physical Review*, 46(7):618, 1934.

[119] H. J. Monkhorst. Calculation of properties with the coupled-cluster method. *International Journal of Quantum Chemistry*, 12(S11):421âĂŞ432, 1977.

[120] J. Nieplocha, R. J. Harrison, and R. J. Littlefield. Global arrays: A nonuniform memory access programming model for high-performance computers. *The Journal of Supercomputing*, 10:169–189, 1996.

[121] J. Noga and R. J. Bartlett. The full CCSDT model for molecular electronic structure. *The Journal of chemical physics*, 86(12):7041–7050, 1987.

[122] J. Olsen, B. O. Roos, P. Jørgensen, and H. J. A. Jensen. Determinant based configuration interaction algorithms for complete and restricted configuration interaction spaces. *The Journal of Chemical Physics*, 89(4):2185–2192, 1988.

[123] V. Pan. *How to Multiply Matrices Faster*. Springer-Verlag New York, Inc., New York, NY, USA, 1984.

[124] C. Papadimitriou and J. Ullman. A communication-time tradeoff. *SIAM Journal on Computing*, 16(4):639–646, 1987.

[125] J.-S. Park, M. Penner, and V. K. Prasanna. Optimizing graph algorithms for improved cache performance. *IEEE Transactions on Parallel and Distributed Systems*, 15(9):769–782, 2004.

[126] J. A. Parkhill and M. Head-Gordon. A sparse framework for the derivation and implementation of fermion algebra. *Molecular Physics*, 108(3-4):513–522, 2010.

[127] B. N. Parlett. *The symmetric eigenvalue problem*. SIAM, 1980.

[128] R. G. Parr and Y. Weitao. *Density-functional theory of atoms and molecules*. Oxford University Press ; Clarendon Press, New York; Oxford, 1989.

[129] J. Pješivac-Grbović, T. Angskun, G. Bosilca, G. E. Fagg, E. Gabriel, and J. J. Dongarra. Performance analysis of MPI collective operations. *Cluster Computing*, 10:127–143, June 2007.

[130] J. A. Pople and R. K. Nesbet. Self-consistent orbitals for radicals. *Journal of Chemical Physics*, 22(3):571, 1954.

[131] J. Poulson, B. Maker, J. R. Hammond, N. A. Romero, and R. van de Geijn. Elemental: A new framework for distributed memory dense matrix computations. *ACM Transactions on Mathematical Software*. in press.

[132] C. Puglisi. Modification of the Householder method based on compact WY representation. *SIAM Journal on Scientific and Statistical Computing*, 13(3):723–726, 1992.

[133] K. Raghavachari, G. W. Trucks, J. A. Pople, and M. Head-Gordon. A fifth-order perturbation comparison of electron correlation theories. 157:479–483, May 1989.

[134] S. Rajbhandari, A. Nikam, P.-W. Lai, K. Stock, S. Krishnamoorthy, and P. Sadayappan. Framework for distributed contractions of tensors with symmetry. *Preprint, Ohio State University*, 2013.

[135] C. C. J. Roothaan. New developments in molecular orbital theory. *Reviews of Modern Physics*, 23(2):69 – 89, 1951.

[136] D. J. Rose and R. E. Tarjan. Algorithmic aspects of vertex elimination. In *Proceedings of Seventh Annual ACM Symposium on Theory of Computing*, STOC '75, pages 245–254, New York, NY, USA, 1975. ACM.

[137] R. D. Schafer. *An introduction to nonassociative algebras*, volume 22. Courier Dover Publications, 1966.

[138] A. C. Scheiner, G. E. Scuseria, J. E. Rice, T. J. Lee, and H. F. Schaefer. Analytic evaluation of energy gradients for the single and double excitation coupled cluster (CCSD) wave function: Theory and application. *Journal of Chemical Physics*, 87(9):5361, 1987.

[139] R. Schreiber and C. Van Loan. A storage-efficient WY representation for products of Householder transformations. *SIAM Journal on Scientific and Statistical Computing*, 10(1):53–57, 1989.

[140] R. Seidel. On the all-pairs-shortest-path problem in unweighted undirected graphs. *Journal of Computer and System Sciences*, 51(3):400–403, 1995.

[141] Y. Shao, L. F. Molnar, Y. Jung, J. Kussmann, C. Ochsenfeld, S. T. Brown, A. T. B. Gilbert, L. V. Slipchenko, S. V. Levchenko, D. P. O'Neill, R. A. D. Jr, R. C. Lochan, T. Wang, G. J. O. Beran, N. A. Besley, J. M. Herbert, C. Y. Lin, T. V. Voorhis, S. H. Chien, A. Sodt, R. P. Steele, V. A. Rassolov, P. E. Maslen, P. P. Korambath, R. D. Adamson, B. Austin, J. Baker, E. F. C. Byrd, H. Dachsel, R. J. Doerksen, A. Dreuw, B. D. Dunietz, A. D. Dutoi, T. R. Furlani, S. R. Gwaltney, A. Heyden, S. Hirata, C.-P. Hsu, G. Kedziora, R. Z. Khalliulin, P. Klunzinger, A. M. Lee, M. S. Lee, W. Liang, I. Lotan, N. Nair, B. Peters, E. I. Proynov, P. A. Pieniazek, Y. M. Rhee, J. Ritchie, E. Rosta, C. D. Sherrill, A. C. Simmonett, J. E. Subotnik, H. L. W. Iii, W. Zhang, A. T. Bell, A. K. Chakraborty, D. M. Chipman, F. J. Keil, A. Warshel, W. J. Hehre, H. F. S. Iii, J. Kong, A. I. Krylov, P. M. W. Gill, and M. Head-Gordon. Advances in methods and algorithms in a modern quantum chemistry program package. *Physical Chemistry Chemical Physics*, 8(27):3172–3191, 2006.

[142] S. S. Shende and A. D. Malony. The TAU parallel performance system. *International Journal of High Performance Computing Applications*, 20(2):287–311, Summer 2006.

[143] E. Solomonik, A. Bhatele, and J. Demmel. Improving communication performance in dense linear algebra via topology aware collectives. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pages 77:1–77:11, New York, NY, USA, 2011. ACM.

[144] E. Solomonik, A. Buluc, and J. Demmel. Minimizing communication in all-pairs shorest-paths. In *IEEE International Symposium on Parallel Distributed Processing (IPDPS)*, 2013.

[145] E. Solomonik, E. Carson, N. Knight, and J. Demmel. *Tradeoffs Between Synchronization, Communication, and Computation in Parallel Linear Algebra Computations*, pages 307–318. SPAA '14. ACM, New York, NY, USA, 2014.

[146] E. Solomonik and J. Demmel. Communication-optimal 2.5D matrix multiplication and LU factorization algorithms. In *Springer Lecture Notes in Computer Science, Proceedings of Euro-Par, Bordeaux, France*, Aug 2011.

[147] E. Solomonik and J. Demmel. Communication-optimal parallel 2.5D matrix multiplication and LU factorization algorithms. Technical Report UCB/EECS-2011-10, EECS Department, University of California, Berkeley, Feb 2011.

[148] E. Solomonik, D. Matthews, J. R. Hammond, J. F. Stanton, and J. Demmel. A massively parallel tensor contraction framework for coupled-cluster computations. *Journal of Parallel and Distributed Computing*, 2014.

[149] D. Sorensen. Analysis of pairwise pivoting in Gaussian elimination. *Computers, IEEE Transactions on*, C-34(3):274 –278, March 1985.

[150] J. F. Stanton and R. J. Bartlett. The equation of motion coupled-cluster method. a systematic biorthogonal approach to molecular excitation energies, transition probabilities, and excited state properties. *Journal of Chemical Physics*, 98(9):7029, 1993.

[151] J. F. Stanton and J. Gauss. Analytic second derivatives in high-order many-body perturbation and coupled-cluster theories: computational considerations and applications. *International Reviews in Physical Chemistry*, 19(1):61–95, 2000.

[152] V. Strassen. Gaussian elimination is not optimal. *Numerische Mathematik*, 13(4):354–356, 1969.

[153] X. Sun and C. Bischof. A basis-kernel representation of orthogonal matrices. *SIAM Journal on Matrix Analysis and Applications*, 16(4):1184–1196, 1995.

[154] Y. Tang, R. A. Chowdhury, B. C. Kuszmaul, C.-K. Luk, and C. E. Leiserson. The Pochoir stencil compiler. In *Proceedings of the Twenty-Third Annual ACM Symposium on Parallelism in Algorithms and Architectures*, pages 117–128. ACM, 2011.

[155] R. Thakur, R. Rabenseifner, and W. Gropp. Optimization of collective communication operations in MPICH. *International Journal of High Performance Computing Applications*, 19(1):49–66, Spring 2005.

[156] A. Tiskin. *The Design and Analysis of Bulk-Synchronous Parallel Algorithms*. PhD thesis, University of Oxford, 1998.

[157] A. Tiskin. All-pairs shortest paths computation in the BSP model. In F. Orejas, P. Spirakis, and J. van Leeuwen, editors, *Automata, Languages and Programming*, volume 2076 of *Lecture Notes in Computer Science*, pages 178–189. Springer Berlin / Heidelberg, 2001.

[158] A. Tiskin. Communication-efficient parallel generic pairwise elimination. *Future Generation Computer Systems*, 23(2):179 – 188, 2007.

[159] S. Toledo. Locality of reference in LU decomposition with partial pivoting. *SIAM Journal of Matrix Analysis and Applications*, 18(4):1065–1081, 1997.

[160] J. Torrellas. Architectures for extreme-scale computing, Nov. 2009.

[161] J. D. Ullman and M. Yannakakis. High probability parallel transitive-closure algorithms. *SIAM Journal of Computing*, 20:100–125, February 1991.

[162] L. G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, 1990.

[163] R. A. Van De Geijn and J. Watts. SUMMA: Scalable Universal Matrix Multiplication Algorithm. *Concurrency: Practice and Experience*, 9(4):255–274, 1997.

[164] V. S. Varadarajan. *Lie groups, Lie algebras, and their representations*, volume 102. Prentice-Hall Englewood Cliffs, NJ, 1974.

[165] J. Čížek. On the correlation problem in atomic and molecular systems. calculation of wavefunction components in ursell-type expansion using quantum-field theoretical methods. *The Journal of Chemical Physics*, 45(11):4256–4266, 1966.

[166] J. Čížek and J. Paldus. Correlation problems in atomic and molecular systems III. rederivation of the coupled-pair many-electron theory using the traditional quantum chemical methods. *International Journal of Quantum Chemistry*, 5(4):359–379, 1971.

[167] S. Warshall. A theorem on boolean matrices. *J. ACM*, 9:11–12, January 1962.

[168] Y. Yamamoto, 2012. Personal communication.

[169] Y. Yamamoto. Aggregation of the compact WY representations generated by the TSQR algorithm, 2012. Conference talk presented at SIAM Applied Linear Algebra.

[170] K. Yelick, D. Bonachea, W.-Y. Chen, P. Colella, K. Datta, J. Duell, S. L. Graham, P. Hargrove, P. Hilfinger, P. Husbands, et al. Productivity and performance using partitioned global address space languages. In *Proceedings of the 2007 International Workshop on Parallel Symbolic Computation*, pages 24–32. ACM, 2007.

[171] U. Zwick. All pairs shortest paths using bridging sets and rectangular matrix multiplication. *Journal of the ACM*, 49(3):289–317, May 2002.