UC Berkeley
UC Berkeley Electronic Theses and Dissertations

Title
Tuning Hardware and Software for Multiprocessors

Permalink
https://escholarship.org/uc/item/1dg4w04b

Author
Mohiyuddin, Marghoob

Publication Date
2012

Peer reviewed|Thesis/dissertation

Tuning Hardware and Software for Multiprocessors

by

Marghoob Mohiyuddin

A dissertation submitted in partial satisfaction of the

requirements for the degree of

Doctor of Philosophy

in

Electrical Engineering and Computer Sciences

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Professor John Wawrzynek, Chair
Professor James Demmel
Professor Ming Gu

Spring 2012

Tuning Hardware and Software for Multiprocessors

Abstract

Tuning Hardware and Software for Multiprocessors

by

Marghoob Mohiyuddin

Doctor of Philosophy in Electrical Engineering and Computer Sciences

University of California, Berkeley

Professor John Wawrzynek, Chair

Technology scaling trends have enabled the exponential growth of computing power. However, the performance of communication subsystems scales less aggressively. This means that an application constrained by memory/interconnect performance will not be able to use the available computing power efficiently—in fact, technology scaling will make this efficiency even worse. This problem can be alleviated if algorithms minimize communication. To this end, we describe communication-avoiding algorithms and highly optimized implementations of a sparse linear algebra kernel called "matrix powers". Results show up to $2.3\times$ improvement in performance over the naïve algorithms on modern architectures. Our multi-core implementation of matrix powers enables us to develop a communication-avoiding iterative solver for sparse linear systems which is up to $2.1\times$ faster than a conventional Generalized Minimal Residual method (GMRES) implementation.

Another problem plaguing the supercomputer industry is the power bottleneck—power has, in fact, become the pre-eminent design constraint for future high-performance computing systems which is why computational efficiency is being emphasized over simply peak performance. Static benchmark codes have traditionally been used to find architectures optimal with respect to specific metrics. Unfortunately, because compilers generate suboptimal code, benchmark performance can be a poor indicator of the performance potential of architecture design points. Therefore, we present hardware/software co-tuning as a novel approach for system design. In co-tuning, traditional architecture space exploration is tightly coupled with software auto-tuning for delivering substantial improvements in area and power efficiency. We demonstrate co-tuning by exploring the parameter space of a Tensilica's Xtensa-based multi-processor running three of the most heavily used kernels in scientific computing, each with widely varying micro-architectural requirements: sparse matrix vector multiplication, stencil-based computations, and general matrix-matrix multiplication. Results demonstrate that co-tuning improves hardware area and power efficiency by up to $3\times$ and $2.4\times$ respectively.

زندگی ہو میری پروانے کی صورت یا رب!
علم کی شمع سے ہو مجھ کو محبت یا رب!
(اقبال)

O Lord, let me emulate the example of the moth!
O Lord, let me dote and dart on the candle flame of knowledge!
(excerpt from Allama Iqbal's "A Child's Prayer")

To Ammi, Abba, Nana and Shareen

Contents

Acknowledgments

It feels great to acknowledge the contributions of the people who made this thesis possible. First of all, I thank my advisor Professor John Wawrzynek for giving me enough freedom and flexibility in research. He was always very patient and understanding, especially during some tough times in research, and, for this, I am extremely grateful to him. I also thank Professor Jim Demmel and Professor Kathy Yelick for giving me the opportunity to work in the Berkeley Benchmarking and Optimization ("BeBOP") research group—I owe my continued interest in high-performance computing to this great experience. My Ph.D. research has been an amalgamation of ideas from Professor Wawrzynek and Professor Demmel, and I am extremely thankful to them for helping me out with finding an interesting thesis topic. I thank Professor Ming Gu for being on my thesis committee and providing me useful feedback.

I thank Professor Adnan Aziz, who was my M.S. advisor at the University of Texas at Austin, for motivating me to pursue a Ph.D. degree. He has been a great mentor and friend to me and his continued advice and support to this day is always appreciated.

During the course of my graduate studies, I had the opportunity of being a part of multiple research groups—the BeBOP, Reconfigurable, and ParLab research groups on campus. It has been a terrific experience working and interacting with so many talented graduate students and researchers. Among them, I would first like to thank Mark Hoemmen for not just being a great collaborator but a great friend too—working with him is always a pleasure. Special thanks to Dave Donofrio and Greg Gibeling for helping me with the hardware infrastructure—their contributions to GateLib and help with debugging hardware modules were very crucial. Many other students deserve thanks, in particular Grey Ballard, Erin Carson, Jike Chong, Kaushik Datta, Shauki Elassaad, Andrew Gearhart, Ankit Jain, Shoaib Kamil, Alex Krasnov, Nick Knight, Adam Megacz, Mark Murphy, Rajesh Nishtala, and Sam Williams.

I would like to acknowledge the researchers at the Future Technologies Group at Lawrence Berkeley National Laboratory (LBNL) and the National Energy Research Scientific Computing Center (NERSC), especially John Shalf, Leonid Oliker, Michael Wehner, Norman Miller, and Leroy Drummond. It was a great experience working as a Graduate Student Researcher (GSR) at LBNL and I really enjoyed being involved in the "Green Flash" project.

I acknowledge the use of computing facilities at UC Berkeley (the Millennium cluster) and the NERSC center. This research used resources of NERSC center, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231.

I thank the Stanford Smart Memories research group, led by Professor Mark Horowitz, for letting us use their Smart Memories simulator. Special thanks to Amin Firoozshahian and Alex Solomatnikov for helping us with setting up and running the simulator.

Many thanks to the UC Berkeley staff, especially Ruth Gjerde, Tamille Johnson, La Shana Polaris, Laura Rebusi, and Shirley Salanio, for helping me with the paperwork and meeting the administrative deadlines.

My graduate student life has been a fun and memorable one due to lots of wonderful people: Saurabh Amin, Richa Sharma, Huma Zaidi, Anurag Gupta, Vishnu Narayanan,

Chapter 1

Introduction

Although clock scaling stopped almost a decade ago, the semiconductor industry still follows Moore's law by the doubling the computational power per chip every technology generation. However, raw computational power doesn't necessarily translate to actual performance—the software needs to be designed in a way to make an efficient use of the underlying computational power. With a variety of machine configurations available in the market, the software needs to be customized (aka "tuned") for each target to get good performance. Traditionally, manual software-tuning along with using appropriate compiler flags, has been done to optimize the code for a target hardware setup. Due to the general-purpose nature of commonly available compilers (gcc, icc, llvm), naïve code can perform rather poorly even when using aggressive compiler flags. This makes software tuning a necessity for high-performance codes. However, manual tuning is prohibitively expensive due to the overwhelming number of target machines available today as well as possible in the future. Software auto-tuning is an approach to mitigate the problem of achieving portable performance. It relies on trying out different algorithms, code variants, data structures and empirically measuring the performance on the target machine—in the end, the best performing algorithm, code variant, data structure is chosen. Such an approach has yielded success stories in dense linear algebra (ATLAS [112]), sparse linear algebra (OSKI [106]), and signal processing (SPIRAL [82], FFTW [39]). More recent work has also focused on developing auto-tuning frameworks in order to ease the development of auto-tuned codes. In keeping with this, the unifying theme of this work is "tuning", i.e., adapting the software to the target hardware and, going one step further, we propose "hardware/software co-tuning" as an approach to adapt the software and the hardware to each other for even more efficient hardware/software solutions.

Another problem we note with the scaling trends is that different technology components scale differently—computational power scales more aggressively when compared to memory/network subsystem latency and bandwidth. We refer to the latter as the communication subsystem. Current technology trends show exponentially increasing gaps between computation, bandwidth and latency costs. A study [92] of high performance computing showed floating point speeds increasing historically at 59%/year, but interprocessor bandwidth improving only 26%/year, and interprocessor latency improving only 15%/year.

Indeed, on certain very large, distributed computing platforms (like the Grid) latencies are already speed-of-light limited and on the order of milliseconds, as opposed to fractions of nanoseconds for floating point operations. Similarly, memory (DRAM) bandwidth is improving only at 23%/year, and memory latency at 5.5%/year. For out-of-core algorithms, with disk bandwidth and latency limited by the rotational speed of disks, the gaps are even larger. Another study [78] observed that latency improves much more slowly than bandwidth across many technologies. These trends suggest that algorithms should be designed not to minimize arithmetic operations, as is traditional, but to minimize communication both within a local memory hierarchy and between processors. In this work, we consider the computations that arise in the communication-intensive Krylov subspace methods (KSMs) used to solve large sparse linear systems or large sparse eigenvalue problems. On current machines, KSMs are limited by memory and network performance, because they execute only a small constant number of arithmetic operations per communicated data value. We replace conventional KSMs with "communication avoiding" versions that send fewer messages and read data less frequently from slow memory at the cost of slightly more arithmetic.

1.1 Related Work

1.1.1 Communication-Avoiding Algorithms

There are two costs to implementing algorithms: "computation" and "communication". "Computation" is the useful work done and it involves processing of data. "Communication" is the movement of data across the system—this data movement can be between different levels of the memory hierarchy, between different processors in a parallel machine and between processors and coprocessors. Furthermore, the communication between different processors can be through the memory subsystem, an on-chip interconnect network or an inter-node interconnect. The cost of communication manifests as a hit on performance as well as energy consumption. Thus, it is desirable to minimize the cost of communication for an application. In fact, given the current technology scaling trends and the lag between communication and computation costs for units of data, it may even make sense to perform more computation if it cuts down the communication significantly. Thus, applications must be communication-avoiding, especially if communication can be a significant part of performance.

At a high-level, communication-avoidance has been a key component of software-tuning. A very good example of communication-avoidance strategies in software design is cache blocking optimization commonly seen in key scientific computing kernels [112, 106, 43, 96]. Cache blocking optimizations try to reuse data in the cache as much as possible which amounts to avoiding data movement across the memory subsystem as much as possible. Another low-level optimization which can be considered communication-avoiding is register tiling, which is performed in both dense and sparse linear algebra codes. In the case of dense linear algebra, register tiling improves register reuse while keeping the same computational complexity. As for sparse codes, register tiling may result in more computation being performed but cut down the memory footprint (and, hence, communication cost) to result

in an overall improvement in performance. Although, "cache blocking" and "register tiling" are optimizations commonly used in different computational kernels, they mean different low-level optimizations depending on the kernel.

There has been theoretical work in bounding the cost of communication [47] and developing communication-efficient algorithms [9]. The authors in [47] were able to bound the communication cost of a cache-blocked dense matrix-matrix multiplication as a function of the cache size. It was shown that the multiplication of two dense $n \times n$ matrices using the conventional $\Theta(n^3)$ algorithm on a machine with a fast memory of size M, requires $\Omega(n^3/\sqrt{M})$ words of data movement between fast and slow memories. This lower bound is tight as "blocked" algorithms were shown which attain the aforementioned lower bound. Irony et al. [51] extended the communication bounds to parallel machines and showed that the lower bound can be expressed as $\Omega\left(\frac{\#\text{arithmetic operations}}{\sqrt{M}}\right)$, where M is the size of fast memory for the sequential machine model and size of the local memory for the parallel machine model. Ballard et al. [9] extended the communication lower bound results to all direct methods of linear algebra—examples include LU factorization, Cholesky factorization, QR factorization, etc. It is also interesting to note that some of the these bounds for dense and sparse linear algebra operations easily translate to related graph algorithms, e.g., all-pairs shortest paths. In addition, Ballard et al. [9] list algorithms which attain the lower bounds.

1.1.2   Software Auto-Tuning

Software tuning is usually required for high-performance codes because general-purpose compilers are usually unable to generate good machine code due to their general nature. Traditionally, careful hand-tuning of software [54] and hand-coded assembly have been used to get high-performance for critical sections of high-performance applications. Hand-tuning software is not only time consuming but also does not scale well as applications need to be deployed on new machines. Furthermore, with the complex interplay between software and the underlying hardware, it is often difficult to predict performance on a target machine given a particular piece of code. Auto-tuning has emerged as an approach to mitigate this problem of tuning software. The idea is fairly simple: write code which automatically adapts to the target hardware. This necessitates that tuning "knobs" be built into the code—the values of these "knobs" which result in high performance, are determined empirically by actually trying out different values and choosing the best performing one. In a way, auto-tuning of software aims to achieve portable performance using minimal writing of code. An auto-tuned software performs part of the tuning at install-time (done once per target machine) and the rest at run-time (which depends on the inputs to the software). A typical auto-tuner achieves its goal of performance portability by employing the following:

- Parameterized code generators: Given that there may be several ways of writing the code for the same computations and different code variants may perform well on different target machines, code generators are used to generate the different possibilities. At install-time, these code generators are used to generate different code variants and the

performance of these code variants is empirically measured to determine the best one. A very prominent example of such a code generator is from ATLAS [112], which is an auto-tuned library for dense linear algebra. ATLAS's code generator generates different variants for the dense matrix-matrix computational kernel. The code generator is heavily parameterized—examples of some parameters include the matrix dimensions, loop unroll factor, number of matrix entries to keep in registers, and matrix data layout (row major or column major format).

- Parameterized routines: While code generators account for different ways of writing code, parameterized routines are used to account for making run-time decisions for performance. These run-time decisions may include the choice of data structures, algorithms, core computational codes, etc. As with code generators, the right parameters are determined empirically by trying out different parameter values at run-time or at install-time. One good example of such a run-time parameter is the data structure used to represent a sparse matrix in memory—this is a key decision to be make in case of a sparse linear algebra library like OSKI [106]. A sparse matrix can be represented in many ways and the optimal representation (with respect to performance) depends on the sparsity pattern of the matrix which is only known at runtime.

- Search heuristic: An auto-tuned software may have several tuning knobs built into it and an exhaustive exploration of the parameter space can be impractical and more so if the exploration has to be performed at run-time. This necessitates the integration of an efficient search strategy in the auto-tuner. The search heuristic must be able to find parameter values yielding good performance in as little time as possible.

There are several success stories involving high-performance libraries which make use of auto-tuning and, in fact, address a variety of computations:

- Dense linear algebra: One of the early success stories for auto-tuned software has been ATLAS [112], which can be seen as a successor to PHiPAC [11, 12]. ATLAS aims to provide a set of high-performance routines for dense linear algebra. At the heart of ATLAS is a high-performance routine for multiplying two dense matrices (General Matrix Matrix Multiply or GEMM). ATLAS uses an auto-tuner to figure out a good performing implementation of GEMM on a target machine. It is interesting that while the simplest implementation of GEMM (which is basically three nested loops in C) performs very poorly due to low data reuse, auto-tuned implementations can achieve close to machine peak performance. Almost all the tuning in for GEMM is done at install-time as the performance only depends on the matrix dimensions.

- Sparse linear algebra: The OSKI library [106] attempts the same by targeting sparse linear algebra. One of the key computational kernels in OSKI is Sparse Matrix Vector Multiplication (SpMV) which is really hard to optimize as the performance is determined heavily by the sparsity pattern of the matrix [30]. OSKI performs part of the tuning at install-time and the remaining is done at run-time when the matrix structure is known.

- Discrete Fourier Transforms: FFTW [39] is one of the most popular libraries for computing the DFT (Discrete Fourier Transform). Given the dimensions of the DFT, FFTW explores possible ways of computing the DFT—this includes the different ways the FFT recursion can be done. Since the DFT dimensions are known at run-time, the first call to the DFT computation can be expensive as auto-tuning is performed to try out different alternatives and measure their performance. Since an exhaustive exploration of the search space can be expensive, FFTW includes a fast mode where a reasonably performing DFT solution is computed quickly. Note that FFTW performs all of its tuning at run-time.

- Signal processing: SPIRAL [82] is one example of a library targeting signal processing applications. Given a digital signal processing (DSP) computation expressed as a formula, SPIRAL uses auto-tuning to generate target-optimized code implementing the DSP computation. These formulas are expressed in a special "tensor-product" language and all the tuning is done at compile-time. Like FFTW, SPIRAL can also be used to compute the DFT but unlike FFTW, it can handle a wider range of DSP computations.

In addition to the above mentioned libraries, significant efforts have been made in advancing the state of the art for auto-tuning for individual kernels. Independently of OSKI, auto-tuned SpMV implementations have been developed targeting multi-cores [113, 114, 16] and, more recently, GPUs [20]. In a similar vein, Volkov et al. [103] implemented several key linear algebra kernels on GPUs and use auto-tuning and overlapping CPU and GPU computations to achieve near peak performance. The PLASMA (Parallel Linear Algebra for Scalable Multi-core Architectures) and MAGMA (Matrix Algebra on GPU and Multics Architecture) projects [1] are recent collaborative efforts to develop auto-tuned frameworks for dense linear algebra on multi-cores as well as hybrid systems involving both multi-cores as well as GPUs. Datta et al. [25] performed an extensive study comparing auto-tuned implementations of the 7-point stencil from the heat equation on structured meshes, on several platforms ranging from multi-cores, IBM Cell to GPUs. Kamil et al. [55] generalize the stencil optimization work to develop a framework for generating auto-tuned implementations of arbitrary stencils specified in a high-level language—this approach is similar to the SPIRAL framework. Performance comparison against the naïve implementations on several platforms showed up to 22× speedups.

There has also been work towards developing general frameworks for auto-tuning:

- The POET (Parameterized Optimizations for Empirical Tuning) language [121] was developed to specify code and the allowed transformations. This lets developers write domain-specific source code generators which simplifies auto-tuner design. The output source code generated by the POET compiler can be fed to a low level compiler as part of the auto-tuning process.

- The ROSE compiler project [88] allows one to be able to build custom source-to-source generators—this is achieved by ROSE allowing the user to manipulate the abstract syntax tree (AST) of the source code and then using the AST to generate ("unparse")

the source code. This can be used to target whole applications instead of limiting to specific kernels. As a proof of concept, ROSE was used as part of PERI [67] to identify performance bottlenecks and auto-tune the SMG2000 [15] (Semicoarsening Multigrid Solver) benchmark from DOE which contains about 28k lines of code—the application was sped up by $1.8\times$ after a bottleneck stencil kernel was auto-tuned.

- CHiLL [19] (Composing High-Level Loops) is a compiler framework for describing and loop transformations. Polyhedral representations of nested loops are used along with scripts describing the set of allowed transformations in order to generate the loop code variants to test for performance tuning. Example loop transformations include loop unrolling, loop nest permutation, tiling, loop splitting, and copy of data to temporary locations. As a demonstration, CHiLL scripts were used to auto-tune different dense linear algebra kernels, e.g., matrix matrix multiplication, LU factorization, and triangular solve. Performance improvements close to hand-tuned code were reported.

- PERI (Performance Engineering Research Institute) [7] is a major collaborative effort to automate the process of tuning whole applications (in contrast to just tuning specific kernels and providing them as libraries). In a way, PERI integrates code profilers, source code generators, source-to-source translators, and search heuristics to formalize and implement a methodology for systematic tuning of whole applications.

- The SEJITS (Selective Embedded Just-In-Time Specialization) [17] project is an attempt to bridge the gap between writing productive code in high level languages like Python and Ruby and the performance of auto-tuned codes written in low level languages like C. One of the key ideas in SEJITS is providing *specializers* which describe source code transformations from a high level language to a low level language. The specializers, if provided, accelerate the critical portions of the high level code by generating and executing auto-tuned low level implementations.

As we can see, although auto-tuning is still an emerging field, significant progress has been made from the initial work involving custom auto-tuners for specific computational kernels, to more recent work involving developing frameworks which enable development of auto-tuners in a more systematic way. Since our goal was simply to demonstrate the effectiveness of our communication-avoiding algorithms, for the purpose of this work, we use an almost exhaustive search in our auto-tuned implementation of matrix powers.

1.1.3   Hardware Design Space Exploration

Hardware design space exploration (DSE) is typically performed to determine the optimal hardware parameters for a given set of benchmark applications [23, 6, 117]. In fact, most of the work in computer architecture evaluates hardware design points using benchmark codes [75, 66, 64, 76]. Figure 1.1 shows a high-level view of such DSE studies—different hardware configurations are evaluated using a set of benchmark applications. The DSE may be constrained (by chip area, cost, power, energy) in order to study the trade-offs involved and answer questions like:

- Should more chip area be devoted to caches?

- Should more money be spent on memory bandwidth?

- Should low voltage be used to trade off speed for power?

Kumar et al. [60] showed that cores, caches and interconnect need to be *co-designed* in order to get good performance or energy efficiency. This is due to the non-negligible cost of the interconnect in terms of area, performance and energy. Interestingly, due to the high cost of the interconnect for sharing caches, the authors found that the theoretical performance benefits go away when constraints of area and power are applied. Thus, it is better to simulate the system as a whole rather than independent components.

Li et al. [66] performed a DSE study under thermal (temperature) and physical (pin bandwidth, chip area, power) constraints. To cut down the simulation time, the authors decoupled core simulation from interconnect and memory subsystem simulation. Single core simulations were used to generate traces which were used for the interconnect and memory simulations. Thus, multi-core simulation results were actually an extrapolation of results from single core simulations and interconnect and memory subsystem simulations. As expected, CPU-bound workloads favored a higher core count while memory-bound workloads favored larger cache area on chip. Interestingly, thermal constraints dominated physical constraints and simpler core designs were found to be favorable for more power-efficient solutions.

Leverich et al. [64] performed a DSE study to compare cache-based multiprocessors with local store-based ones with respect to energy consumption and performance. The multiprocessor was designed using Tensilica's embedded processor XTensa [98] with VLIW features. Multiple parameters like core counts, bandwidth, and frequency were varied. It was found that with features like hardware prefetching and non-allocating stores enabled, cache-based configurations performed close to local store-based configurations. Given that local store-based configurations require extra programming effort, cache-based configurations had an overall advantage. The authors hand-tuned some of the benchmarks and found significant improvement in performance, thus demonstrating the effectiveness of software tuning.

Musoll et al. [76] used DSE to compare multi-threaded with out-of-order single-threaded superscalar architectures. Hardware multi-threading was shown to be effective at reducing power consumption as well as area when compared to a superscalar for the same performance.

Seo et al. [89] proposed a new software-managed cache design called the extended set-index cache (ESC). The cache design attempts to have the tag search performance of a set-associative cache and the miss rate of a fully-associative cache. Interestingly, the hardware adapts the cache parameters depending on the code being run—this can depend on which part of which application is being run. It was found that the hardware adaptation at run-time improved performance significantly for the benchmark applications as there was no one size fits all solution for the cache parameters.

In a study closest to co-tuning, Shrivastava et al. [90] perform DSE for horizontally partitioned cache architectures. The compiler used the hardware configuration information to
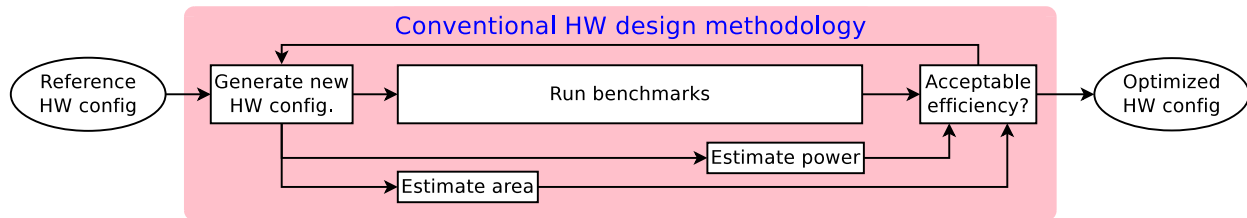
Figure 1.1: Traditional hardware design space exploration

generate code tuned to minimize energy consumption. The authors found that incorporating the compiler to generate target specific code gave much better results when compared to the traditional approach of using fixed codes for all hardware design points.

One major issue with DSE studies is the size of the design space [27, 49, 36, 3, 62, 52], especially since software simulation of hardware is traditionally used to perform such studies. To cut down the simulation time, various techniques have been tried to make the DSE tractable, ranging from relying on "industry experience, intuition and literature surveys" [27] to statistical sampling for creating trained models using artificial neural networks [49], or cubic splines [62]. Random sampling of the design space is used to train the statistical models. Dubach et al. [36] use linear regression to predict the performance of a new application using the trained model and performance on a few sampled design points. More recently, Azizi et al. [3] use posynomial[1] functions for the trained models. Given these statistical techniques, only a small part of the design space need to be actually simulated for training the statistical model. Once the training is done, the model can be used to predict the performance over the design space. Since our work explored limited design spaces, we employed an exhaustive search rather than the sophisticated statistical modeling techniques currently used in exploring large design spaces. We note that hardware design spaces can be explored more quickly using FPGA[2]-acceleration [97, 22]–the acceleration can easily be two orders of magnitude. This speeding up of hardware simulation also enables exploration of the larger design space which can also include the software design space. Using hardware to accelerate also has ramifications with respect to the power consumption–with the speedups involved, a single FPGA board can easily replace a cluster of machines and yet achieve the same simulation throughput for a much lower power consumption. Chapter 3 covers FPGA-acceleration of hardware simulation in more detail.

1.2 Contributions of This Work

In light of the key issues discussed previously, two major contributions are made in this thesis:

- We describe communication-minimizing algorithms, models and implementation of

---

[1]A posynomial is a function of the form $f(x_1, x_2, \ldots, x_n) = \sum_{k=1}^{K} c_k x_1^{a_{1,k}} \cdots x_n^{a_{n,k}}$ where the coordinates x_i and coefficients c_k are positive real numbers, and the exponents $a_{i,k}$ are real numbers.

[2]An FPGA (Field Programmable Gate Array) is a configurable hardware and can be used to quickly implement a digital circuit

"matrix powers"—a linear algebra kernel. Furthermore, in keeping with the overarching theme of tuning, our implementation of the "matrix powers" kernel is auto-tuned. We also demonstrate the effectiveness of the matrix powers implementation in an iterative solver for large sparse systems—results (see Section 2.11) show that performance improves by up to 2.2 over the naïve implementation. Parts of this work have appeared in [31, 32, 73].

- We describe "hardware/software co-tuning" (henceforth, referred to as simply co-tuning) as a methodology for hardware design and demonstrate its effectiveness on a set of well-known kernels from scientific computing and a hardware architecture—results (see Sections 3.5 and 3.7). Parts of this work have appeared in [74, 111]—Sections 3.6-3.7, however, are new.

The rest of the thesis describes the aforementioned contributions (see Section 1.2) in detail. Chapter 2 covers the matrix powers kernel in detail. Chapter 3 discusses co-tuning. Finally, Chapter 4 covers conclusions and future work.

Chapter 2

The Matrix Powers Kernel

2.1 Background

Krylov subspace methods (KSMs) are iterative methods for computing solutions of sparse linear systems (solving for x in $Ax = b$) and eigenvalue problems (solving for x in $Ax = \lambda x$), particularly when the problems are large. Formally, the Krylov subspace for a matrix A and a vector v is the following subspace:

$$\mathcal{K}_k(A, v) = \text{span}\{v, Av, \ldots, A^k v\}, \qquad k \in \mathbb{N}.$$

A KSM minimizes the residual $\|Ay - b\|$ by computing better approximations to the solution every iteration. At every iteration, the KSM expands the set of basis vectors for the Krylov subspace—the basis is used to approximate the solution or the eigenvalues and eigenvectors. Conventional algorithms for KSMs spend most of the time in communication-limited kernels like sparse matrix-vector multiply (SpMV), vector dot products and sums. Given the data dependencies in each iteration, only limited performance gains can be made by overlapping communication with computation. Given that KSMs are an important component of scientific and engineering applications, making them communicate less can have significant impact in improving the performance of those applications.

Conventional implementations of KSMs involve the following computational kernels:

- SpMVs (sparse matrix-vector products): given a sparse matrix A, a vector x, compute $A \cdot x$ or $A^T \cdot x$.

- AXPYs (vector-vector sums): given vectors x and y, scalar α, compute $\alpha \cdot x + y$.

- Dot products: given vectors x and y, compute $y^* \cdot x$[1]

A typical implementation would alternate SpMV with AXPYs and dot products. Note that all the above mentioned kernels are communication-limited as they perform very few computations per data word read from memory. Therefore, both the computation and communication cost of k iterations grows at least proportionally to k.

---

[1]For a vector v of complex values, y^* denotes the conjugate transpose of v.

The Generalized Minimal Residual method (GMRES) of Saad and Schultz [86] is a Krylov subspace method for solving a non-symmetric square system of linear equations $Ax = b$. The standard implementation of GMRES alternates between using a sparse matrix-vector product to generate a new Krylov basis vector, and using BLAS 1[2] - based Modified Gram-Schmidt to orthogonalize that vector against all the previously generated and orthogonalized basis vectors. A number of authors proposed performing GMRES in a different way [108, 28, 53, 4, 38]. Begin with a starting vector v_1, and then generate k more vectors v_2, ..., v_{k+1} so that they form a basis of the *Krylov subspace*

$$\text{span}\{v_1, v_2, \ldots, v_{k+1}\} = \text{span}\{v_1, Av_1, A^2v_1, \ldots, A^jv_1\} \quad \text{for } j = 1, \ldots, k.$$

Then, use a QR factorization[3] to orthogonalize the basis vectors. They become therefore identical to the basis vectors that standard GMRES would generate (modulo a unitary column scaling). Finally, use the R factor to reconstruct the $k + 1$ by k upper Hessenberg matrix[4] from standard GMRES, compute a new approximate solution, and restart if the desired accuracy is not yet reached. Other authors developed similar algorithms, generally called "s-step Krylov methods," for conjugate gradient iteration and other Krylov iterations for symmetric matrices [21, 100]. The above variants of GMRES all require restarting after each group of k steps.

In this work, we reorganize KSMs so that the communication cost is nearly as small as theoretically possible. Specifically, we implement a communication-avoiding version of GMRES called CA-GMRES (communication-avoiding GMRES) to demonstrate the effectiveness of our approach. To this end, communication-avoiding analogues of some computational kernels were introduced:

- *Matrix powers*: Given a matrix A in sparse format, a vector x, an integer $k > 0$, compute the k vectors $[p_1(A)x, p_2(A)x, \ldots, p_k(A)x]$, where $p_i(A)$ is a polynomial of degree i, $1 \leq i \leq k$. This kernel replaces the SpMV kernel and performs multiple iterations instead of one iteration at a time which is the case with SpMV.

- *Tall Skinny QR* (TSQR): Perform the QR factorization of an $m \times n$ dense matrix A with $m \gg n$.

- *Block Gram-Schmidt* (BGS): Perform Modified Gram-Schmidt (MGS) orthogonalization but instead of processing a column at a time, process a block of columns at a time to improve performance. It is not really a new kernel but its use in combination with TSQR is the novel idea here.

In related work [46], we describe numerically stable reorganizations of typical KSMs, such as GMRES and CG, that leverage a matrix powers kernel (or small variations) to

---

[2]The Basic Linear Algebra Subprograms (BLAS) [61], an interface for dense linear algebra computations, such as dot products (BLAS 1), vector sums (BLAS 1), matrix-vector multiply (BLAS 2), and matrix-matrix multiply (BLAS 3).

[3]Given a matrix A, its QR factorization is matrices Q and R such that $A = QR$, Q is orthogonal and R is upper triangular.

[4]An upper Hessenberg matrix has zero entries below the first sub-diagonal.

advance k steps in one iteration. That work also describes support for preconditioning, which is essential to KSMs in practice.

We focus here on the matrix powers kernel with nearly minimal communication time. On a parallel computer, this means with latency costs that are independent of k. In fact if the sparse matrix has a suitable (and common) sparsity structure described below, we will see that the latency cost of matrix powers is just $O(1)$. On a sequential computer, this means that latency and bandwidth costs are independent of k. Said another way, both the matrix A and k output vectors will need to be moved between fast and slow memory just $1 + o(1)$ times (1 time is obviously the minimum), not k times. Since bandwidth is always the bottleneck in the sequential case, our approach is always an improvement. Our communication *avoiding* approach complements and is more powerful than communication *overlap* techniques, which can at best halve the running time. Avoiding communication can achieve up to k-fold speedups when communication is dominant, and can be combined with overlap for an additional performance boost.

Specifically, we consider two flavors of the matrix powers kernel:

- Given a matrix A, a vector x, a positive integer k, compute $[x, Ax, A^2x, \ldots, A^kx]$. This is a simplification of the general form. When the basis vectors of the Krylov subspace are computed in this way, we refer to it as the monomial basis. For the purpose of illustration and explanation, this flavor will be used throughout this chapter.

- Given a matrix A, a vector x, a positive integer k, compute $[p_1(A)x, \ldots, p_k(A)x]$, where $p_i(A) = (A - \lambda_1 I) \cdots (A - \lambda_i I)$. When this form is used, we refer to it as the Newton basis. Our multi-core implementation of CA-GMRES uses this form.

2.2   Related Work

The optimizations described in this work belong to a collection of techniques for improving the performance of applying a stencil repeatedly to a regular discrete domain, or multiplying a vector repeatedly by a sparse matrix. They, in turn, are a subset of various methods known as *tiling* or *blocking*. They all involve decompositions of the d-dimensional domain into d-dimensional sub-domains, and rearranging the order of arithmetic operations in order to exploit the parallelism and/or temporal locality implicit in those sub-domains.

Tiling research falls into three general categories. The first encompasses performance-oriented implementations and practical performance models. See, for example, [81, 79, 50, 115, 84, 71, 93, 116, 35, 95, 34, 119, 104, 57, 56]. The second category consists of theoretical algorithms and asymptotic performance analyses. These are based on sequential or parallel processing models which account for the memory hierarchy and/or inter-processor communication costs. Works that specifically discuss stencils or more general sparse matrices include [47], [63], and [100]. The third category contains suggested applications that call for repeated application of a stencil (resp. sparse matrix) to a domain (resp. vector). See, for example, [102, 85, 21, 28, 4, 95].

The idea of using redundant computation to avoid communication or slow memory accesses in stencil codes may be as old as out-of-core (OOC) stencil codes themselves. Leiserson

et al. cite a reference from 1963 [63, 81]. Nevertheless, many tilings do not involve redundant computation. For example, Douglas et al. describe a parallel tiling algorithm that works on the interiors of the tiles in parallel, and then finishes the boundaries sequentially [35]. Many sequential tilings do not require redundant computations [56]; our SA1 algorithm does not.

However, at least in the parallel case, tilings with redundant computation have the advantage of requiring only a single round of messages, if the stencil is applied several times. The latency penalty is thus independent of the number of applications, though the bandwidth requirements increase. Furthermore, Strout et al. point out that the sequential fill-in of boundary regions suggested by Douglas et al. suffers from poor locality [95]. Most importantly, redundant computation to save messages is becoming more and more acceptable, given the exponential divergence in performance between latency, bandwidth, and floating-point rate.

Extensions of stencil tiling to more general sparse matrices require runtime analysis of the sparse matrix structure, often using a graph partitioner. Finding an optimal partition is an NP-complete problem which must be approximated in practice, at nontrivial cost. Theoretical algorithms for the out-of-core sequential case already existed (see e.g., [63]), but Douglas et al. were apparently the first to attempt an implementation of parallel tiling of a general sparse matrix, in the context of repeated applications of a multigrid smoother [35]. This was extended by Strout et al. [94] into a sequential cache optimization which resembles our SA1 (discussed in Section 2.5) algorithm.

Our work differs from existing approaches in many ways. First, we developed our methods in tandem with an algorithmic justification: communication-avoiding or "s-step" Krylov subspace methods [46]. Toledo had suggested an s-step variant of conjugate gradient iteration, based on a generalization of algorithm PA1 (discussed in Section 2.4), but he did not supply an implementation for matrices more general than tridiagonal matrices [100]. We have a full implementation of PA1 for general sparse matrices, and have detailed theoretical models showing performance increases on a wide variety of platforms.

Douglas et al. and Strout developed their matrix powers kernel for classical iterations like Gauss-Seidel [35, 95]. However, these iterations' most common use in modern linear solvers are as multigrid smoothers. The payoff of applying a smoother k times in a row decreases rapidly with k; this is, in fact, why multigrid is used, rather than classical iterations such as Jacobi or Gauss-Seidel. Douglas et al. acknowledge that usually $1 \leq k \leq 5$ [35]. In contrast, communication-avoiding Krylov subspace methods are potentially much more scalable in k. Saad also suggested applying something like a matrix powers kernel to polynomial preconditioning, but here again, increasing the degree of the polynomial preconditioner has a decreasing payoff, in terms of the number of CG iterations required for convergence [85].

We have also expanded the space of possible algorithms by including algorithms PA2 (see Section 2.4) and SA2 (see Section 2.5). PA2 avoids some redundant computation, but offers less opportunity for overlapping communication and computation. SA2 extends SA1 for the case in which the vectors (as well as the matrix) do not fit entirely in fast memory. As far as we can tell, PA2 and SA2 are novel. In addition, we compose the parallel algorithms with sequential algorithms to implement matrix powers on multi-cores (discussed in Section 2.9).

2.3 Model Problems

Our techniques work for general sparse matrices, but the case of regular d-dimensional meshes with $(2b+1)^d$-point stencils illustrates potential performance gains for a representative class of matrices. By a *regular mesh*, we mean a simply connected domain in $\mathbb{R}^d$ ($d = 1$, 2, 3, ...) where the vertices are evenly spaced at the grid points. The connectivity pattern of a regular mesh is described by the *stencil*, which shows for each point in the mesh, how it is connected to its neighbors. For example, on a 2D mesh, a *5-point stencil* means that each point is connected to its east, south, west, and north immediate neighbors. A *9-point stencil* for a 2D mesh means that each point is connected to its 8 immediate neighbors, i.e., east, southeast, south, southwest, west, northwest, north, and northeast neighbors. In these cases, we say that the bandwidth b of the stencil is 1, since each point is only connected to its immediate neighbors. Generalizing this, we can have $(2b+1)^d$-point stencils where each point is connected to neighbors within a radius of b points. Since a sparse matrix can be thought of as representing a directed graph, we use the terms sparse matrix and graphs interchangeably. When meshes are being discussed, we may also refer to them simply by their stencil pattern.

Here, we assume a symmetric pattern (but arbitrary non-symmetric matrix entries) and describe the pattern in terms of its undirected graph.

We consider the following model problems:

1. 1D mesh with n unknowns and $(2b+1)$-point stencil;[5]

2. 2D mesh with n^2 unknowns and $(2b+1)^2$-point stencil, in nested dissection[6] ordering;

3. 3D mesh with n^3 unknowns and $(2b+1)^3$-point stencil, in nested dissection ordering.

We call the *surface* of a mesh the number of points on the partition boundary. For an $n \times n$ (2D) mesh partitioned in to p equal sized squares with a 5-point stencil, the surface is $4\frac{n}{p^{1/2}}$. For an $n \times n \times n$ (3D) mesh partitioned in to p equal sized cubes with a 7-point stencil, the surface is $6\frac{n^2}{p^{2/3}}$. The *volume* of a mesh is the total number of points in each processor's partition, namely $\frac{n^2}{p}$ and $\frac{n^3}{p}$ in the 2D and 3D cases. The *surface-to-volume ratio* of a mesh is therefore $4\frac{p^{1/2}}{n}$ in the 2D case and $6\frac{p^{1/3}}{n}$ in the 3D case. We assume all the above roots (like $p^{1/3}$) and fractions (like $\frac{n}{p^{1/2}}$) are integers, for simplicity.

The surface-to-volume ratio of a partition of a general sparse matrix is defined analogously. All our algorithms work best when the surface-to-volume ratio is small, as is the case for meshes with large n and sufficiently smaller p.

In the parallel case, we use the usual mapping where row blocks j of A, x and $y = Ax$ are all assigned to processor j. As stated above, the latency cost of our version of a KSM will be *independent* of k, requiring just 1 message between each processor and its "neighbors",

---

[5]We consider tridiagonal matrices in order to illustrate our techniques most clearly, not because they are computationally challenging for computing products.

[6]Nested dissection is a heuristic for partitioning a sparse matrix [41, 69, 42]—the corresponding graph is partitioned recursively to minimize the number of vertices connecting different partitions.

i.e., those processors owning components of x needed to compute the local components of $y = Ax$. The bandwidth and computation costs will be nearly minimal, increasing only by lower order terms (depending on the surface-to-volume ratio) compared to a conventional implementation. For example, suppose that the sparsity pattern of our matrix is that of a 27 point stencil operating on an $n \times n \times n$ mesh, and that we assign $\frac{n}{p^{1/3}} \times \frac{n}{p^{1/3}} \times \frac{n}{p^{1/3}}$ "cubes" of mesh points to each of p processors. Then using our approach drops the number of messages per processor from $26k$ to 26, while only increasing the number of words communicated per processor from $\frac{6kn^2}{p^{2/3}}$ to $\frac{6kn^2}{p^{2/3}} \cdot (1 + \frac{2kp^{1/3}}{n})$ and arithmetic operations per processor from $\frac{53kn^3}{p}$ to $\frac{53kn^3}{p} \cdot (1 + \frac{1.5kp^{1/3}}{n})$. In both cases, the factor $\frac{p^{1/3}}{n}$ is proportional to the surface-to-volume ratio, which will be small for problems of interest.

In the sequential case, our algorithm will mimic the parallel algorithm, processing the matrix block by block. As stated above, all the data (matrix and vectors) will only have to move from slow to fast memory $1 + o(1)$ times in order to implement k steps of a KSM, not move k times. In other words the number of slow memory accesses (the latency cost) and the bandwidth cost will exceed their minimal values by this $1 + o(1)$ factor. The $o(1)$ term will be proportional to the surface-to-volume ratio.

We contrast our approach of *avoiding* communication with the complementary approach of *overlapping* communication and computation. The latter approach can at best halve the running time, whereas avoiding communication can potentially achieve up to k-fold speedups when communication is dominant. Furthermore, we can use overlapping to accelerate our algorithms as well.

We present both detailed performance models and measurements from different implementations. We model matrices with the sparsity patterns of both a 2D and 3D stencil on a variety of parallel and sequential computers. The two parallel computers modeled are a Petaflop machine consisting of 8100 50 GFlop/s processors connected over a fast network, and a Grid consisting of 125 1 TFlop/s processors connected over the internet. The speedup over a conventional algorithm depends on whether it is a 2D or 3D problem, the width of the stencil (e.g., 5 point, 9 point etc.), the problem size, and how much computation and communication can be overlapped. Table 2.1 summarizes the maximum speedups modeled below for matrices whose graphs are 9 point 2D stencils and 27 point 3D stencils (but stored as general sparse matrices). For Peta, the best speedups were for smaller n in the range studied, because communication was more dominant; maximum speedups fell as n increased and the problem became computation bound. Also for Peta, non-overlapping computation made latency more important, and so our approach to avoiding latency yielded larger speedups. On the Grid, for the lower n in the range modeled, it was fastest to use just one processor because communication was so expensive. But as n grew, it eventually became effective to use parallelism, and close to this transition point our approach yielded large speedups.

The two "sequential" machines modeled are:

1. Uniprocessor with DRAM as fast memory and a single disk as slow memory (called OOC for Out-Of-Core).

2. A multi-core chip with on-chip cache as fast memory and DRAM as slow memory (called "CacheBlocked"). Although the machine is parallel, we only address how to

Machine	Matrix	Range of n	Overlap	Max Modeled Speedup
Peta	2D	2^{10} to 2^{22}	Yes	6.9
			No	15.1
	3D	2^9 to 2^{14}	Yes	1.02
			No	3.56
Grid	2D	2^{10} to 2^{22}	Yes	22.22
			No	15.63
	3D	2^9 to 2^{14}	Yes	4.41
			No	7.79

Table 2.1: Performance modeling summary for parallel machines. "Overlap" indicates whether communication is overlapped with computation.

Machine	Matrix	Range of n	Modeled Speedup	% Peak
OOC	2D	2^{14} to 2^{25}	10.2	17%
	3D	2^8 to 2^{17}	[7.39,9.51]	[14%, 18%]
CacheBlocked	2D	2^8 to 2^{19}	[2.45,2.58]	[62%, 65%]
	3D	2^8 to 2^{12}	[1.34,1.36]	38%

Table 2.2: Performance modeling summary for sequential machine model.

avoid off-chip latency and bandwidth to DRAM.

We only modeled the non-overlapping case, with modeled speedups as shown in the table below. In contrast to the last table, Table 2.2 shows the range of speedups attained over all problem sizes n, since bandwidth is always the bottleneck, so significant speedups were attained for all problems sizes. Here, "% Peak" is the ratio of the (modeled) running time of the algorithm on a zero latency / infinite bandwidth machine to the (modeled) true time. The closer this is to 100%, the more completely the algorithm masks the cost of slow memory access. On OOC, we see that we get high speedups, though we are not near peak. On CacheBlocked our speedups are more modest, but still good, and we are closer to peak.

We now describe the organization of the rest of the chapter. Section 2.4 describes parallel algorithms for distributed memory systems, whereas Section 2.5 covers sequential algorithms. We discuss the results of asymptotic analysis of our proposed algorithms in Section 2.6. Section 2.7 covers the results of performance modeling for different machine models. Section 2.8 describes an actual out-of-core implementation, which achieves a speedup of 3.2×, and is 16% as fast as it would be if run on a machine with zero disk latency and infinite disk bandwidth, up from 5%. This is both a good speedup, and shows that we are within 16% of peak. We also describe our performance model, which uses measured machine parameters and agree with measured performance closely. Section 2.9 adapts the parallel and sequential algorithms for shared memory multi-core machines and Section 2.10 describes a multi-core implementation of matrix powers. Section 2.11 describes the integration of matrix powers in communication-avoiding GMRES (CA-GMRES) and performance results.

2.4 Distributed Memory Parallel Algorithms for Matrix Powers

We consider the conventional parallel algorithm, as well as our two new approaches:

Conventional Parallel Approach (PA0). The algorithm runs in k steps, where step j computes $x^{(j)} = A^j x$ from $x^{(j-1)} = A^{j-1} x$ by each processor receiving messages with the needed remotely stored entries of $x^{(j-1)}$ and computing its local components of $x^{(j)}$.

Parallel Approach 1 (PA1). We begin the computation of all *locally computable* components of $[Ax, ..., A^k x]$, and simultaneously begin sending all the components of x needed by the neighboring processors to compute the remaining components of $[Ax, ..., A^k x]$. *Locally computable* entries are ones which can be computed locally without needing any entry from any other processor. When the locally computable components are complete, we block until the remote components of x arrive. This maximizes the potential overlap of computation and communication, but does not minimize redundant work, as we will see in PA2.

Parallel Approach 2 (PA2). We compute the set of local values of $[Ax, ..., A^k x]$ needed by the neighboring processors, so as to minimize redundant computation. Then we send these values to the neighboring processors, and simultaneously compute the remaining locally computable values. When all the locally computable values are complete, we block until the remote components of $[Ax, ..., A^k x]$ arrive, and complete the work. This minimizes redundant work, but permits slightly less overlap of computation and communication.

The difference between PA1 and PA2 will become clearer when we explain them for the 1D mesh.

We will estimate the cost of our parallel algorithms by measuring five quantities:

1. number of floating point operations per processor

2. number of floating point numbers communicated per processor (the "bandwidth cost")

3. number of messages sent per processor (the "latency cost"),

4. total memory required per processor for the matrix, and

5. total memory required per processor for the vectors.

Now we argue informally why either approach PA1 or PA2 approximately minimizes communication. We assume that there is no cancellation in any of the powers A^j or $A^j x$ that would make them sparser than if all their nonzero entries were nonnegative, and that there are no algebraic relations among entries of A and/or x. Thus the complexity only depends on the sparsity pattern, and for simplicity of notation we assume all the nonzero

entries of A and x are positive. In particular the set $\mathcal{D}$ of processors owning entries of x on which block row i of $[Ax, A^2x, ..., A^kx]$ depends may be described as the set of processors owning those x_j where block row i of $A + A^2 + \cdots + A^k$ has a nonzero j-th column. In both algorithms PA1 and PA2, the processor owning row block i receives exactly one message from each processor in $\mathcal{D}$, which minimizes latency. Furthermore, PA1 only sends those entries of x in each message on which the answer depends, which minimizes the bandwidth cost. PA2 sends the same amount of data although different values so as to minimize redundant computation.

The rest of this section is organized as follows:

- Subsection 2.4.1 describes PA1 and PA2 in more detail for 1D meshes (band matrices).

- Subsection 2.4.2 describes PA1 and PA2 in more detail for 2D and 3D meshes.

- Subsection 2.4.3 presents a tabular summary of all the operation counts for meshes, and discusses how they specialize to stencil matrices, where each row of the matrix has identical nonzero entries (modulo boundaries).

- Subsection 2.4.4 describes PA1 and PA2 on general sparse matrices.

2.4.1   1D meshes

We begin by considering a tridiagonal matrix (i.e., with bandwidth $b = 1$). In the conventional parallel algorithm each processor executes the code in Algorithm 2.1 in order to compute $x^{(j)} = A^j x^{(0)}$ for $j = 1$ to k.

Algorithm 2.1 PA0: conventional parallel approach for 1D mesh with $b = 1$

    {processor q owns $x_{s_q}^{(0)}, ..., x_{e_q}^{(0)}$, boundaries $q = 1$ and $q = p$ ignored}
    for $j = 1, 2, \ldots, k$ **do**
        start sending $x_{s_q}^{(j-1)}$ to processor $q - 1$
        start sending $x_{e_q}^{(j-1)}$ to processor $q + 1$
        start receiving $x_{s_q-1}^{(j-1)}$ from processor $q - 1$
        start receiving $x_{e_q+1}^{(j-1)}$ from processor $q + 1$
        compute $x_{s_q+1}^{(j)} = (Ax^{(j-1)})_{s_q+1}, \ldots, x_{e_q-1}^{(j)} = (Ax^{(j-1)})_{e_q-1}$
        wait for messages to arrive
        compute $x_{s_q}^{(j)} = (Ax^{(j-1)})_{s_q}$ and $x_{e_q}^{(j)} = (Ax^{(j-1)})_{e_q}$

The computational cost is clearly $2k$ messages, $2k$ words sent, and $5k\frac{n}{p}$ flops (3 multiplies and 2 additions per vector component computed). The memory required per processor is $3\frac{n}{p}$ matrix entries and $(k+1)\frac{n}{p} + 2$ vector entries (for the local components of $[x, Ax, .., A^kx]$ and for the values on neighboring processors).

To explain PA1, consider Figure 2.1. Each row of circles represents the entries of A^jx, for $j = 0$ to $j = 8$. A subset of 30 components of each vector is shown, owned by 2
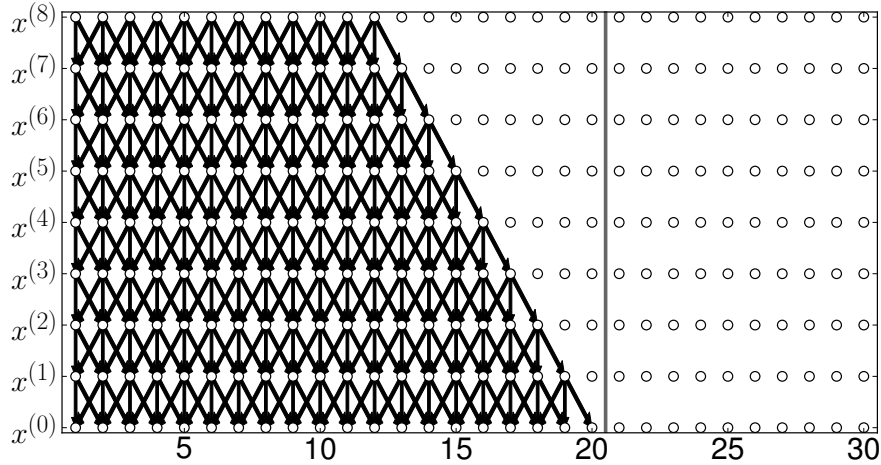
Figure 2.1: Locally Computable components of $[Ax, ..., A^8x]$ for tridiagonal matrix

processors, one to the left of the vertical gray line, and one to the right. (There are further components and processors not shown, to the left and to the right of the ones in the figure). The diagonal and vertical lines show the dependencies: the three lines below each circle (component i of A^jx) connect to the circles on which its value depends (components $i-1$, i and $i+1$ of $A^{j-1}x$). Figure 2.1 shows the local dependencies of the left processor, i.e., all the circles (vector components) that can be computed without communicating with the right processor. The remaining circles without attached lines to the left of the vertical gray line require information from the right processor to be computed.

Figure 2.2 shows how to compute these remaining circles using PA1. The dependencies are again shown by diagonal and vertical lines below each circle, but now dependencies on data formally owned by the right processor are shown in red. All these values in turn depend on the $k = 8$ leftmost value of $x^{(0)}$ owned by the right processor, shown as blue circles in the bottom row. By sending these values from the right processor to the left processor, the left processor can compute all the circles whose dependencies are shown in Figure 2.2. The black circles indicate computations ideally done only by the left processor, and the red circles show redundant computations, i.e., ones also done by the right processor. Algorithm 2.2 summarizes this discussion.

The memory required by PA1 is $(k+4)\frac{n}{p}$ as in PA0 plus $2k$ more words for vector entries plus $6(k-1)$ more words for matrix entries, altogether $8k-6$ more than PA0. PA1's other costs are 2 messages (versus $2k$ for the conventional method), $2k$ words sent (same as the conventional method), and $5k\frac{n}{p} + 5k(k-1)$ flops, or roughly $5k^2$ more flops than the conventional method. This can also be described as an increase in flops by a factor $1 + k/(\frac{n}{p})$.

Note that we are assuming that $k < \frac{n}{p}$, so that only data from neighboring processors is needed, rather than more distant processors. Indeed, we expect that $k \ll \frac{n}{p}$ in practice, which will mean that the number of extra flops (not to mention extra memory) will be negligible. We continue to make this assumption later without repeating it, and use it to
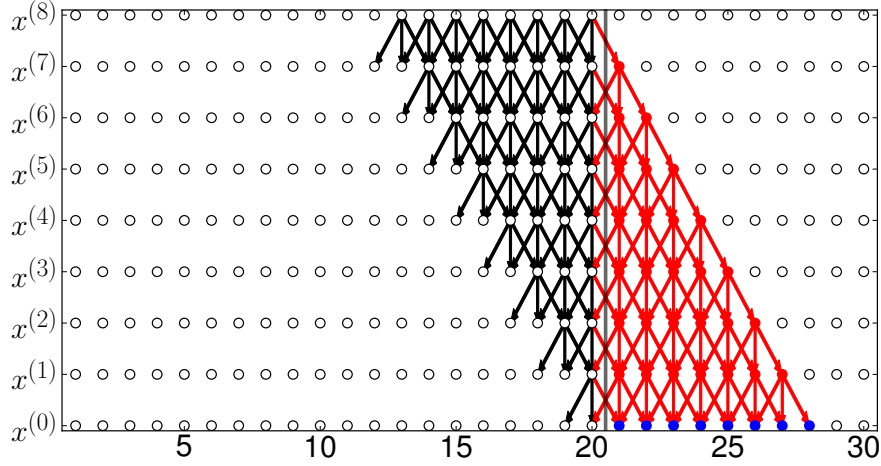
Figure 2.2: Remote dependencies in PA1 for $[Ax, ..., A^8x]$ for tridiagonal matrix

---

Algorithm 2.2 PA1: Parallel Approach 1 for 1D mesh with $b = 1$

{processor q owns $x_{s_q}^{(0)}, \ldots, x_{e_q}^{(0)}$; ignore boundaries as in PA0}

start sending $x_{s_q}^{(0)}, \ldots, x_{s_q+k-1}^{(0)}$ to processor $q - 1$

start sending $x_{e_q-k+1}^{(0)}, \ldots, x_{e_q}^{(0)}$ to processor $q + 1$

start receiving $x_{s_q-k}^{(0)}, \ldots, x_{s_q-1}^{(0)}$ from processor $q - 1$

start receiving $x_{e_q+1}^{(0)}, \ldots, x_{e_q+k}^{(0)}$ from processor $q + 1$

for $j = 1, \ldots, k$ **do**

   compute locally dependent components of $A^j x^{(0)}$ as shown in Figure 2.1

wait for receives to finish

for $j = 1, \ldots, k$ **do**

   compute remaining red and black components of $A^j x^{(0)}$ as shown in Figure 2.2

---

simplify some expressions in Table 2.3 in Section 2.4.3.

Now consider PA2, illustrated in Figure 2.3. We note that the blue and black circles owned by the right processor and attached to blue lines can be computed locally by the right processor. The 8 solid blue circles can then be sent to the left processor to compute the remaining circles connected to black and/or red lines. This saves the redundant work represented by the blue circles, but leaves the redundant work to compute the red circles, about half the redundant work of PA1. Algorithm 2.3 summarizes this discussion.

The memory required by PA2 is $(k+4)\frac{n}{p}$ as in PA0 plus $2k$ more words for vector entries plus $6\lfloor \frac{k}{2} \rfloor$ more words for matrix entries, altogether roughly $5k$ more words than PA0. The other costs of PA2 are 2 messages (versus $2k$ for the conventional method), $2k$ words sent (same as the conventional method), and $5k\frac{n}{p} + 10\lfloor \frac{k}{2} \rfloor(\lfloor \frac{k}{2} \rfloor + odd(k))$ flops, where $odd(k) = 1$ if k is odd and $odd(k) = 0$ if k is even. In other words, PA2 takes roughly $\frac{5}{2}k^2$ more flops than the conventional method, half as many extra flops as PA1.

We will not always draw the corresponding detailed pictures or algorithms for the other
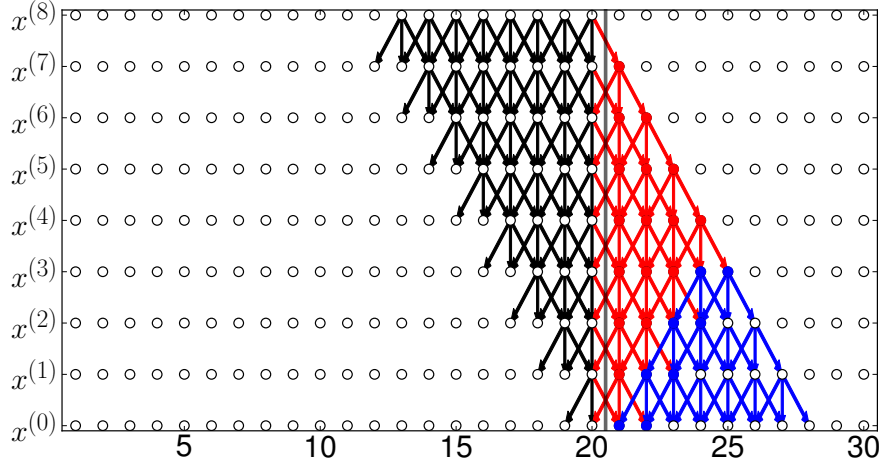
Figure 2.3: Remote dependencies in PA2 for $[Ax, ..., A^8x]$ for tridiagonal matrix

---

Algorithm 2.3 PA2: Parallel approach 2 for 1D mesh with $b = 1$

{assume processor q owns $x^{(0)}_{s_q}, ..., x^{(0)}_{e_q}$; ignore boundaries as in PA0}
compute blue circles corresponding to own blue triangles in Figure 2.3
start sending appropriate blue circles to processor $q - 1$
start sending appropriate blue circles to processor $q + 1$
start receiving appropriate blue circles from processor $q - 1$
start receiving appropriate blue circles from processor $q + 1$
compute locally dependent components of $A^j x^{(0)}$ as shown in Figure 2.1
wait for messages to arrive
compute remaining red and black components of $A^j x^{(0)}$ as shown in Figure 2.3

---

meshes, but the same kinds of analyses apply. Nor will we compute the exact expressions for the number of extra flops, but rather approximate the number of mesh points in the black red and blue regions (pyramids, triangles, tetrahedra, and higher dimensional polyhedra) that arise by computing the leading terms of the volumes of these geometric objects. The next sections will sketch these results, and Section 2.4.3 will summarize all the results in a table.

Now we briefly address 1D meshes with bandwidth $b > 1$, i.e., band matrices. The work per mesh point is $2b + 1$ multiplications and $2b$ additions, or $4b + 1$ flops in all per mesh point, or $(4b + 1)nk/p$ flops for the conventional method. A total of $2kb$ words are communicated with processors to the left and right, in $2k$ messages. The memory required per processor is $(k + 1)\frac{n}{p} + 2b$ words for the vectors and $(2b + 1)\frac{n}{p}$ words for the matrix entries, or $(2b + k + 2)\frac{n}{p} + 2b$ words in all.

Now consider PA1. To compute the extra flops, we must count the number of mesh points in the region corresponding to the red triangle in Figure 2.2, namely $bk(k - 1)/2$. To get the number of extra flops, this is multiplied by $4b + 1$. The number of messages is again 2, containing $2kb$ words in all. The number of words of memory required is $(k + 1)\frac{n}{p} + 2kb$

for the vectors and $(2b+1)(\frac{n}{p}+2kb)$ for the matrix entries, or $(2b+k+2)\frac{n}{p}+kb(4b+4)$ words in all.

Now consider PA2. The region corresponding to the blue region in Figure 2.3 has again about half the number of mesh points as the region corresponding to the red region in Figure 2.2, roughly $bk(k-1)/4$. To get the number of extra flops, this is again multiplied by $4b+1$, and by 2, for the right and left boundaries. The number of words of memory required for vectors is the same as PA1, and slightly smaller for matrix entries, $(2b+1)(\frac{n}{p}+kb)$.

2.4.2   2D and 3D meshes

2D mesh with a 5 point stencil graph

We consider multiplying by a matrix whose graph is the 5-point stencil, i.e., with North, South, East, West (NSEW) connections on an n-by-n grid of unknowns partitioned on $p^{1/2}$-by-$p^{1/2}$ grid of processors. We assume $p^{1/2}|n$, so that each processor owns a $\frac{n}{p^{1/2}}$-by-$\frac{n}{p^{1/2}}$ square of grid points (vector components), and their corresponding matrix rows, and that $k < \frac{n}{p^{1/2}}$. We expect that $k \ll \frac{n}{p^{1/2}}$ in practice.

Figure 2.4 shows the remote domain of dependence for a single processor (demarcated by green lines as before) owning a 10-by-10 grid of unknowns (the black circles). When $k=3$, the results of $[Ax,..,A^k x]$ will depend on the remote values shown by blue circles (the same notation as Figure 2.2). Unlike Figure 2.2, Figure 2.4 does not show circles for components of $A^j x$ for $j > 0$, but rather a projected view. Figure 2.5 shows a 3D view analogous to Figure 2.2.

The number of messages decreases from $4k$ for the conventional algorithm to 8 instead of to 4 for PA1, because communication is required with the corner neighbors (NW, SW, NE and SE), as well as side neighbors (N, S, E and W). The volume of communication also grows slightly to include the triangles of size $k-1$ owned by the 4 corner neighbors. The number of flops grows roughly by the factor $1 + 2k/(\frac{n}{p^{1/2}})$. When $k \ll \frac{n}{p^{1/2}}$, this increase is quite small. It is a little harder to visualize the regions of redundant computations than in the 1D mesh case: In the side neighbors, the red circles denoting redundant computations form a prism with triangular cross section and volume (and number of contained points) proportional to $k\frac{n}{p^{1/2}}$, and in the corner neighbors the red circles form a pyramid with volume (and number of contained points) proportional to k^3. The memory requirement for the conventional algorithm is $(k+1)\frac{n^2}{p}+4\frac{n}{p^{1/2}}$ vector entries and $5\frac{n^2}{p}$ matrix entries; for PA1 it increases by an additional $4k\frac{n}{p^{1/2}}+2k^2$ vector entries and 5 times as many matrix entries.

Figure 2.6 shows the dependencies for PA2 applies to the 2D mesh with a 5 point stencil graph. As in Figure 2.3, there are black circles that are the desired (or initial) values, red circles representing redundant work, blue circles denoting work that was redundant in PA1 but saved by PA2, and solid blue circles that are to be communicated. In the side processors, the regions of redundant computations again form prisms with triangular cross sections, with half the cross sectional area of PA1. Thus, like the 1D case, this means about half the redundant work is saved. It is somewhat more difficult to see what is happening in the corner processors. The pyramid of redundant operations from PA1 has a smaller
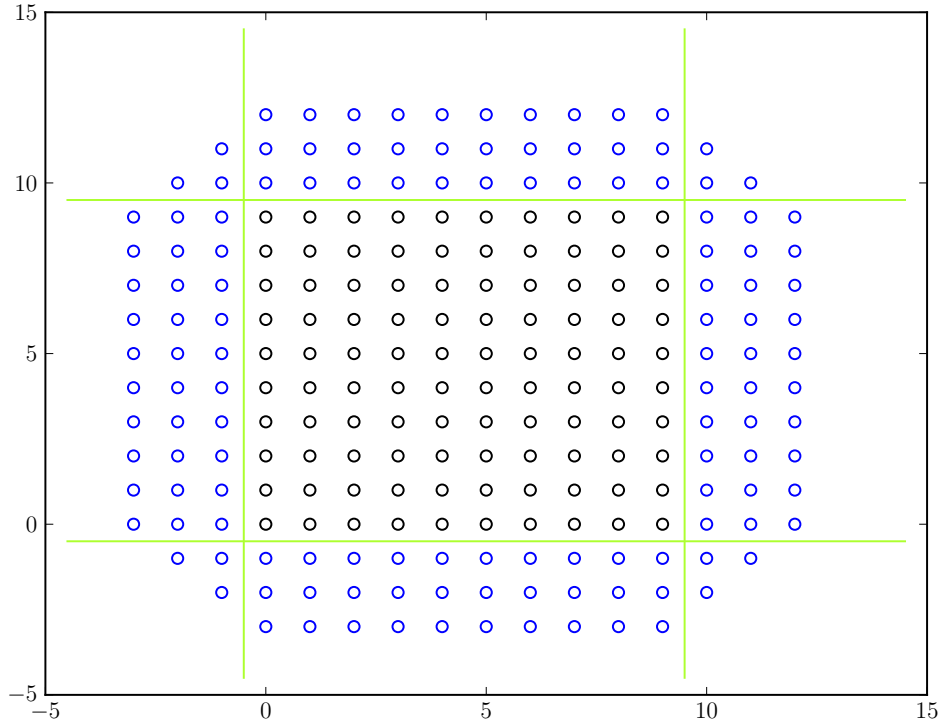
Figure 2.4: Remote dependencies in PA1 for $[Ax, ..., A^3x]$ for 2D mesh with 5 point stencil, projected view

tetrahedron of locally computable components subtracted from it; geometrical symmetry considerations indicate that this saves about $1/3$ of the redundant operations in the corners. The number of extra words of memory required for matrix entries decreases by nearly a factor of 2.

2D mesh with 9 point stencil graph

We consider multiplying by a matrix whose graph is the 9-point stencil, i.e., with N, S, E, W, NE, SE, SW, and NW connections on an n-by-n grid of unknowns partitioned on $p^{1/2}$-by-$p^{1/2}$ grid of processors. We assume $p^{1/2}$ divides n evenly.

Figures 2.7, 2.8, and 2.9 are analogous to Figures 2.4, 2.5, and 2.6, respectively. A similar counting exercise leads to the entries in Table 2.3.

2D mesh with $(2b + 1)^2$ point stencil graph

We consider multiplying by a matrix A whose graph is a $(2b + 1)^2$ point stencil, i.e., where each vertex has connections to other vertices within b to the left, right, up or down. The 9 point stencil graph of the last section is the special case $b = 1$. This can be thought of as a generalization of a band matrix to 2D; when exhibited in natural order the matrix has $2b + 1$ bands, each of which is $2b + 1$ entries wide.
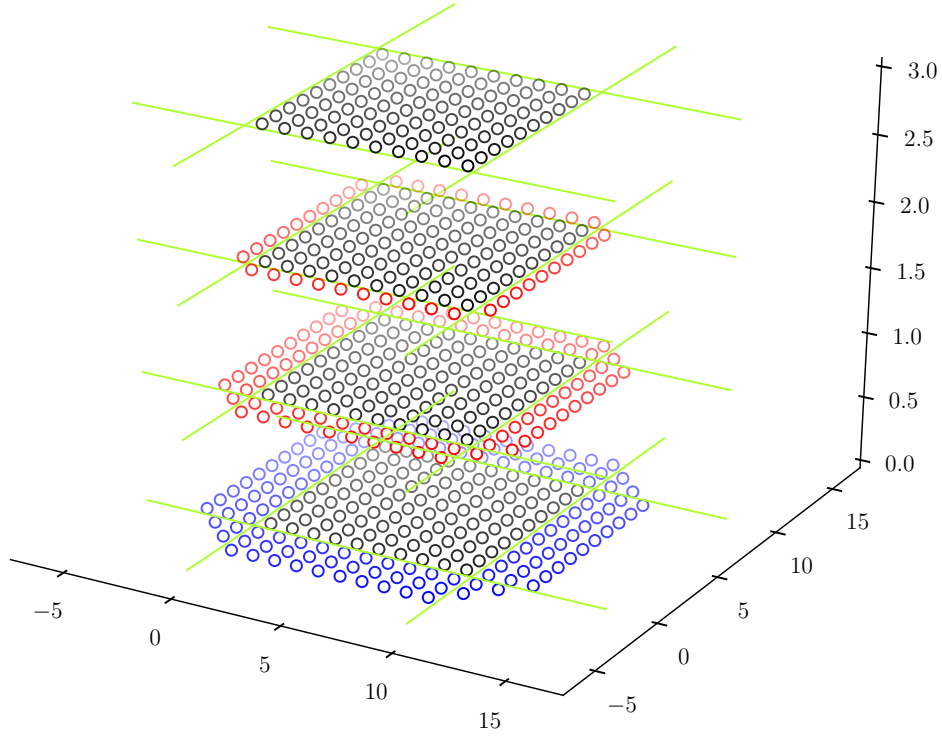
Figure 2.5: Remote dependencies in PA1 for $[Ax, ..., A^3x]$ for 2D mesh with 5 point stencil, 3D view

The conventional algorithm for multiplying Ax requires 8 messages. The 4 to the side processors each receive $b\frac{n}{p^{1/2}}$ vector entries, and the 4 to the corner processors each receive b^2 vector entries. The number of flops is $2(2b + 1)^2 - 1$ per vector entry for a total of $(8b^2 + 8b + 1)\frac{n^2}{p}$.

Similar counting exercises lead to the other entries in Table 2.3.

3D meshes, with 7 point, 27 point and $(2b+1)^3$ point stencils graphs

We first consider multiplying by a matrix whose graph is the 7-point stencil on an n-by-n-by-n grid of unknowns partitioned on a $p^{1/3}$-by-$p^{1/3}$-by-$p^{1/3}$ grid of processors. We assume $p^{1/3}|n$.

Table 2.3 entries are estimated as follows. There are 26 neighbors of a processor, so 26 messages need to be exchanged. The conventional algorithm will exchange $\frac{n^2}{p^{2/3}}$ boundary values with each of its 6 "face neighbors" at each step, for a total of $6k\frac{n^2}{p^{2/3}}$ words sent. The conventional algorithm will also do 13 flops to compute each of the $\frac{n^3}{p}$ components it owns at each step, for a total of $13k\frac{n^2}{p}$ flops.

The other entries of Table 2.3 are based on surface-to-volume analogies to the earlier cases.
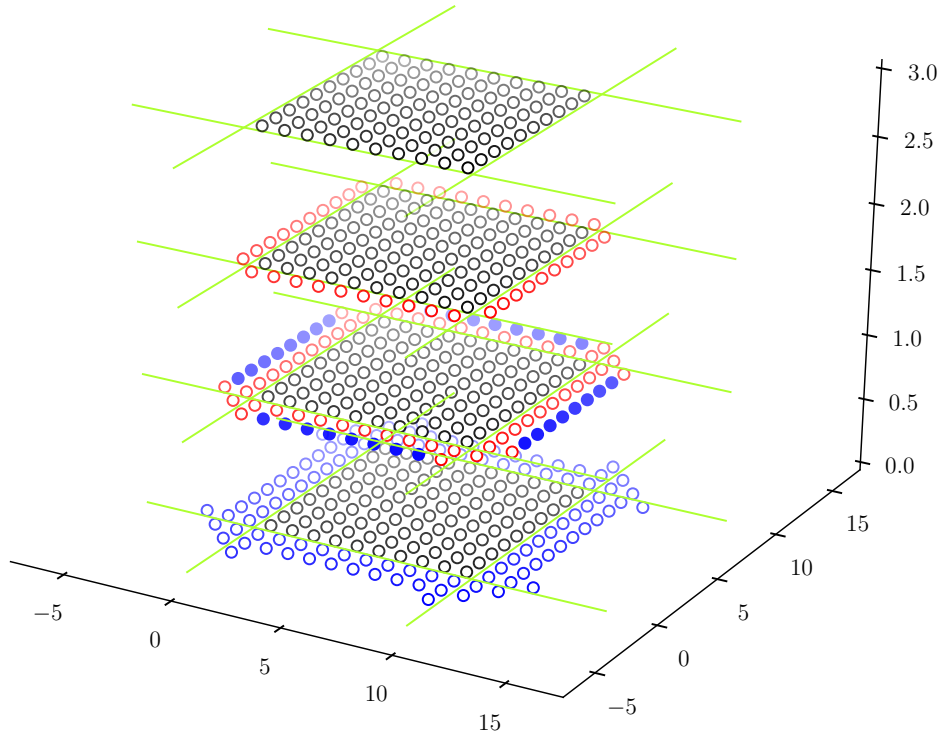
Figure 2.6: Remote dependencies in PA2 for $[Ax, ..., A^3x]$ for 2D mesh with 5 point stencil, 3D view

2.4.3   Summary of Parallel Complexity on Meshes

PA1 and PA2 can be extended to higher dimensions and different mesh bandwidths (and sparse matrices in general). There, the pictures of which regions are communicated and which are computed redundantly become more complicated, higher-dimensional polyhedra, but the essential algorithms remain the same. Table 2.3 summarizes all of the resulting costs for 1D, 2D, and 3D meshes.

In Table 2.3 which shows the summary for Parallel Algorithms, "Mess" is the number of messages sent per processor, "Words" is the total size of these messages, "Flops" is the number of floating point operations, "MMem" is the amount of memory needed per processor for the matrix entries, and "VMem" is the amount of memory needed per processor for the vector entries. Lower order terms are sometimes omitted for clarity.
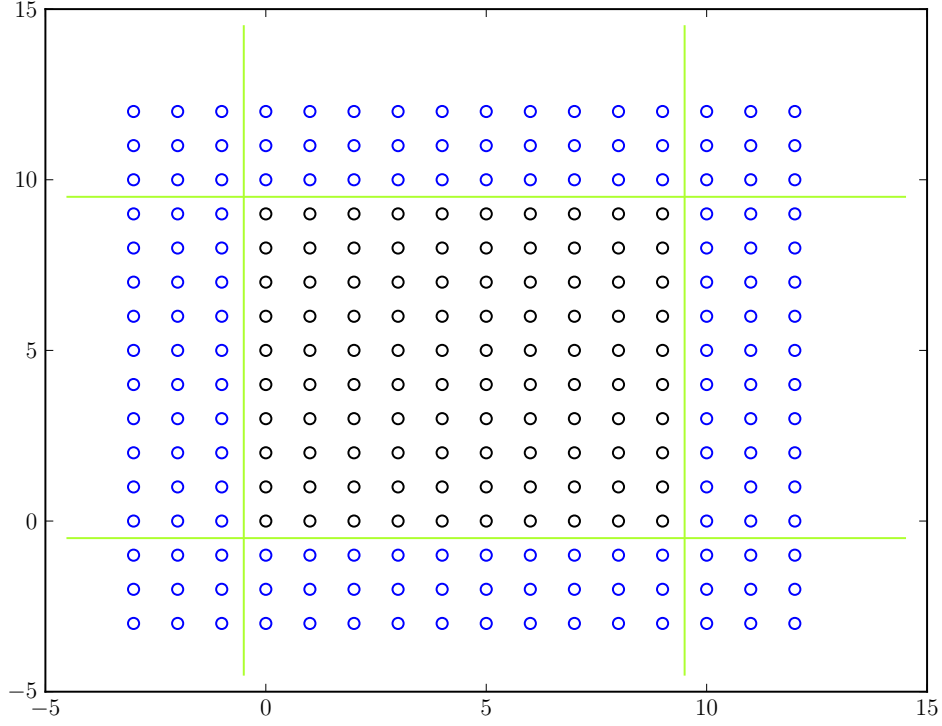
Figure 2.7: Remote dependencies in PA1 for $[Ax, ..., A^3x]$ for 2D mesh with 9 point stencil, projected view

Problem	Costs	Conventional Approach	Parallel Approach 1	Parallel Approach 2
1D mesh $b \geq 1$	Mess	$2k$	2	2
	Words	$2bk$	$2bk$	$2bk$
	Flops	$(4b+1)k\frac{n}{p}$	$(4b+1)(k\frac{n}{p}+bk^2)$	$(4b+1)(k\frac{n}{p}+\frac{bk^2}{2})$
	MMem	$(2b+1)\frac{n}{p}$	$(2b+1)\frac{n}{p}+bk(4b+2)$	$(2b+1)\frac{n}{p}+bk(2b+1)$
	VMem	$(k+1)\frac{n}{p}+2b$	$(k+1)\frac{n}{p}+2bk$	$(k+1)\frac{n}{p}+2bk$
2D mesh $(2b+1)^2$ pt stencil	Mess	$8k$	8	8
	Words	$4bk(\frac{n}{p^{1/2}}+b)$	$4bk(\frac{n}{p^{1/2}}+bk)$	$4bk(\frac{n}{p^{1/2}}+1.5bk)$
	Flops	$(8b^2+8b+1)k\frac{n^2}{p}$	$(8b^2+8b+1)\cdot$ $(k\frac{n^2}{p}+2bk^2\frac{n}{p^{1/2}}+\frac{4}{3}b^2k^3)$	$(8b^2+8b+1)\cdot$ $(k\frac{n^2}{p}+bk^2\frac{n}{p^{1/2}}+b^2k^3)$
	MMem	$(2b+1)^2\frac{n^2}{p}$	$(2b+1)^2(\frac{n^2}{p}+4bk\frac{n}{p^{1/2}}+4b^2k^2)$	$(2b+1)^2(\frac{n^2}{p}+2bk\frac{n}{p^{1/2}}+b^2k^2)$
	VMem	$(k+1)\frac{n^2}{p}+4b\frac{n}{p^{1/2}}$ $+4b^2$	$(k+1)\frac{n^2}{p}+4bk\frac{n}{p^{1/2}}$ $+4b^2k^2$	$(k+1)\frac{n^2}{p}+4bk\frac{n}{p^{1/2}}$ $+6b^2k^2$
3D mesh $(2b+1)^3$ pt stencil	Mess	$26k$	26	26
	Words	$6bk\frac{n^2}{p^{2/3}}+12b^2k\frac{n}{p^{1/3}}+O(b^3k)$	$6bk\frac{n^2}{p^{2/3}}+12b^2k^2\frac{n}{p^{1/3}}+O(b^3k^3)$	$6bk\frac{n^2}{p^{2/3}}+12b^2k^2\frac{n}{p^{1/3}}+O(b^3k^3)$
	Flops	$(2(2b+1)^3-1)k\frac{n^3}{p}$	$(2(2b+1)^3-1)\cdot$ $(k\frac{n^3}{p}+3bk^2\frac{n^2}{p^{2/3}}+O(b^2k^3\frac{n}{p^{1/3}}))$	$(2(2b+1)^3-1)\cdot$ $(k\frac{n^3}{p}+\frac{3}{2}bk^2\frac{n^2}{p^{2/3}}+O(b^2k^3\frac{n}{p^{1/3}}))$
	MMem	$(2b+1)^3\frac{n^3}{p}$	$(2b+1)^3\cdot$ $(\frac{n^3}{p}+6bk\frac{n^2}{p^{2/3}}+O(b^2k^2\frac{n}{p^{1/3}}))$	$(2b+1)^3\cdot$ $(\frac{n^3}{p}+3bk\frac{n^2}{p^{2/3}}+O(b^2k^2\frac{n}{p^{1/3}}))$
	VMem	$(k+1)\frac{n^3}{p}+6b\frac{n^2}{p^{2/3}}$ $+O(b^2\frac{n}{p^{1/3}})$	$(k+1)\frac{n^3}{p}+6bk\frac{n^2}{p^{2/3}}$ $+O(b^2k^2\frac{n}{p^{1/3}})$	$(k+1)\frac{n^3}{p}+6bk\frac{n^2}{p^{2/3}}$ $+O(b^2k^2\frac{n}{p^{1/3}})$

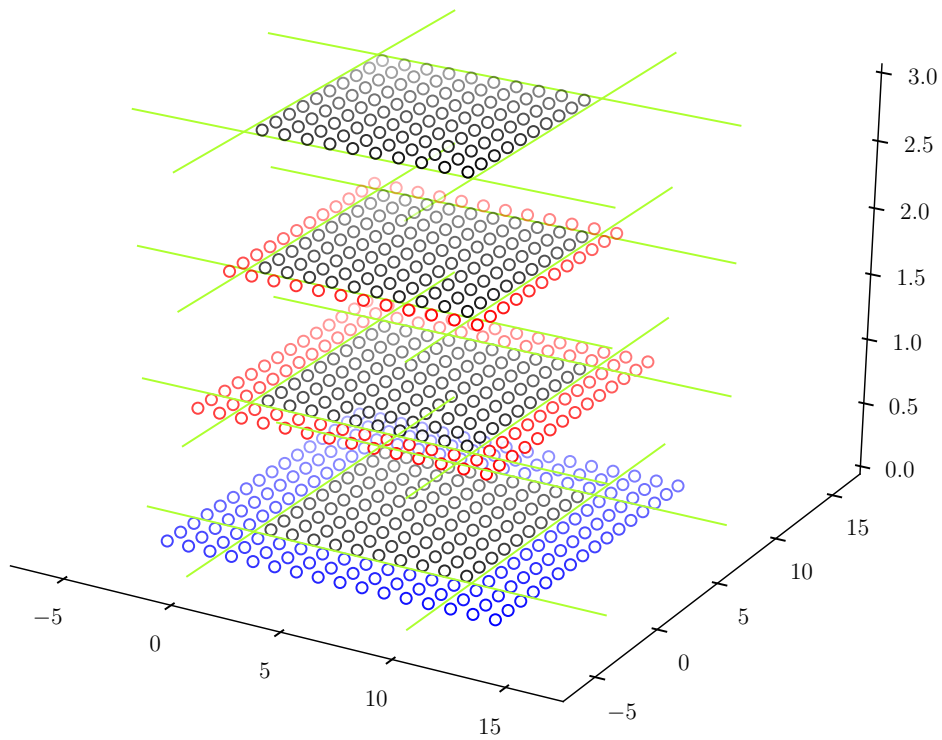Table 2.3: Summary for parallel algorithms (some lower order terms omitted)

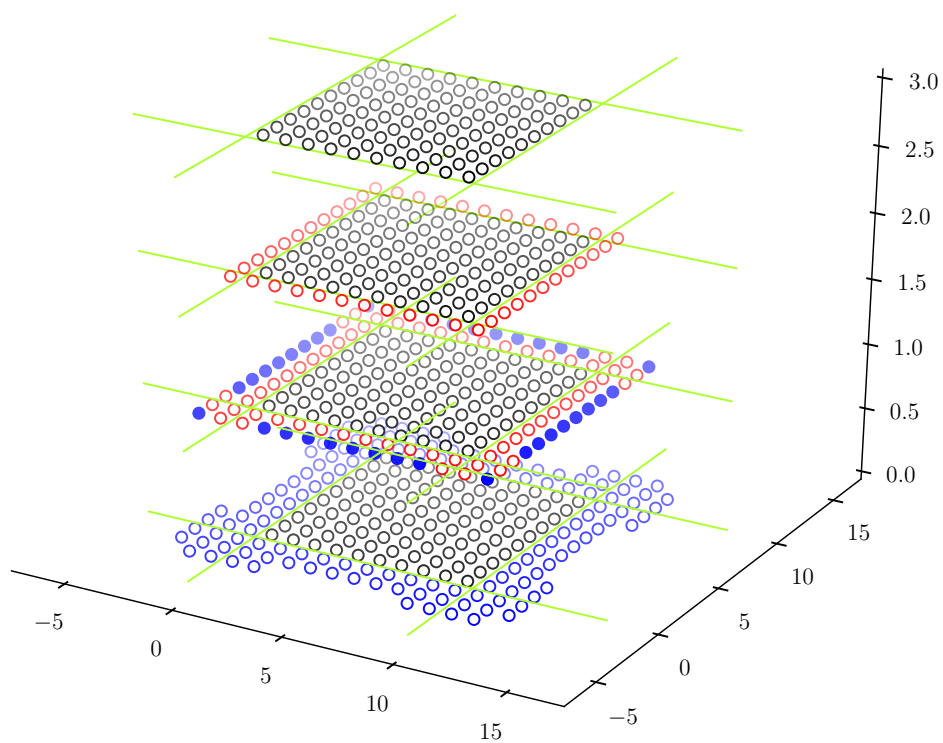Figure 2.8: Remote dependencies in PA1 for $[Ax, ..., A^3x]$ for 2D mesh with 9 point stencil, 3D view

Figure 2.9: Remote dependencies in PA2 for $[Ax, ..., A^3x]$ for 2D mesh with 9 point stencil, 3D view

2.4.4   General Graphs

Here we show how to extend the approaches PA1 and PA2 to general sparse matrices. To do so we need some graph theoretic notation. It is natural to associate a directed graph with a square sparse matrix A, with one vertex for every row/column, and an edge from vertex i to vertex j if $A_{ij} \neq 0$, meaning that component i of $y = Ax$ depends on component j of x. We will build an analogous graph, essentially consisting of k copies of this basic graph: Let $x_j^{(i)}$ be the j-th component of $x^{(i)} = A^i \cdot x^{(0)}$. We associate a vertex with each $x_j^{(i)}$ for $i = 0, ..., k$ and $j = 1, ..., n$ (and use the same notation to name the vertex), and an edge from $x_j^{(i+1)}$ to $x_m^{(i)}$ when $A_{jm} \neq 0$, and call this graph of $n(k+1)$ vertices G. (We will not need to construct all of G in practice, but using G makes it easy to describe our algorithms, in a fashion analogous to Figures 1 through 3.) We say that i is the *level* of vertex $x_j^{(i)}$. Each vertex will also have an *affinity* q, corresponding to the processor number where it is stored; we assume all vertices $x_j^{(0)}, x_j^{(1)}, ..., x_j^{(k)}$ have the same affinity, depending only on j.

We write G_q to mean the subset of vertices of G with affinity q, $G^{(i)}$ to mean the subset of vertices of G with level i, and $G_q^{(i)}$ to mean the subset with affinity q and level i.

Let S be any subset of vertices of G. We let $R(S)$ denote the set of vertices reachable by directed paths starting at vertices in S (so $S \subset R(S)$). We need $R(S)$ to identify dependencies of sets of vertices on other vertices. We let $R(S, m)$ denote vertices reachable by paths of length at most m starting at vertices in S. We write $R_q(S)$, $R^{(i)}(S)$ and $R_q^{(i)}(S)$ as before to mean the subsets of $R(S)$ with affinity q, level i, and both affinity q and level i, respectively.

Next we need to identify the locally computable components, that processor q can compute given only the values in $G_q^{(0)}$. We denote the set of locally computable components by $L_q \equiv \{x \in G_q : R(x) \subset G_q\}$. As before $L_q^{(i)}$ will denote the vertices in L_q at level i.

Finally, for PA2 we need to identify the minimal subset $B_{q,r}$ of vertices (i.e. their values) that processor r needs to send processor q so that processor q can finish computing all its vertices G_q (e.g., the 8 solid blue circles in Figure 2.3): We say that $x \in B_{q,r}$ if and only if $x \in L_r$, and there is a path from some $y \in G_q$ to x such that x is the first vertex of the path in L_r.

Given all this notation, we can finally state versions of PA0 (Algorithm 2.4), PA1 (Algorithm 2.5) and PA2 (Algorithm 2.6) for general graphs and partitions among processors.

We illustrate the algorithm PA1 on the matrix A whose graph is in Figure 2.10. The vertices represent rows and columns of A and the edges represent nonzeros; for simplicity we use a symmetric matrix so the edges can be undirected. The dotted gray lines separate vertices owned by different processors, and we will let q denote the processor owning the black vertices in the center partition of the figure. In other words the black vertices are $G_q^{(0)}$. For all the neighboring processors $r \neq q$, the red vertices are $R_r^{(0)}(G_q)$ for $k = 1$, the red and green vertices together are $R_r^{(0)}(G_q)$ for $k = 2$, and the red, green and blue vertices together are $R_r^{(0)}(G_q)$ for $k = 3$.

The Phases in PA2 will be referred to in Section 2.7.

Algorithm 2.4 PA0: Conventional parallel algorithm (code for processor q)

$\quad$ **for** $i = 1, \ldots, k$ **do**

$\quad\quad$ **for all** processors $r \neq q$ **do**

$\quad\quad\quad$ send all $x_j^{(i-1)} \in R_q^{(i-1)}(G_r^{(i)})$ to processor r

$\quad\quad$ **for all** processors $r \neq q$ **do**

$\quad\quad\quad$ receive all $x_j^{(i-1)} \in R_r^{(i-1)}(G_q^{(i)})$ from processor r

$\quad\quad$ compute all $x_j^{(i)} \in L_q^{(i)}$

$\quad\quad$ wait for receives to finish

$\quad\quad$ compute remaining $x_j^{(i)} \in G_q^{(i)} - L_q^{(i)}$

Algorithm 2.5 PA1 for general graph (for processor q)

$\quad$ **for all** processors $r \neq q$ **do**

$\quad\quad$ send all $x_j^{(0)} \in R_q^{(0)}(G_r)$ to processor r

$\quad$ **for all** processors $r \neq q$ **do**

$\quad\quad$ receive all $x_j^{(0)} \in R_r^{(0)}(G_q)$ from processor r

$\quad$ **for** $i = 1, \ldots, k$ **do**

$\quad\quad$ compute all $x_j^{(i)} \in L_q$

$\quad$ wait for receives to finish

$\quad$ **for** $i = 1, \ldots, k$ **do**

$\quad\quad$ compute $x_j^{(i)} \in (R(G_q) - L_q)$
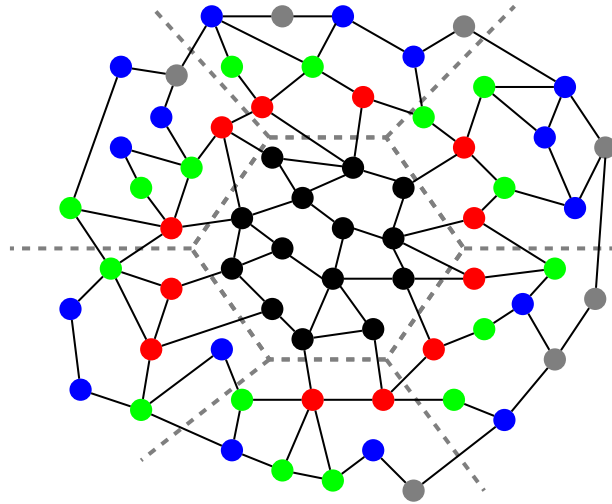

Figure 2.10: Example general graph.

Algorithm 2.6 PA2 for general graph (code for processor q)

{Phase I}
for $i = 1, \ldots, k$ **do**
    compute $x_j^{(i)} \in \cup_{r \neq q}(R(G_r) \cap L_q)$ {blue circled vertices in Figure}
for all processors $r \neq q$ **do**
  send $x_j^{(i)} \in B_{r,q}$ to processor r
for all processors $r \neq q$ **do**
  receive $x_j^{(i)} \in B_{q,r}$ from processor r
{Phase II}
for $i = 1, \ldots, k$ **do**
  compute $x_j^{(i)} \in L_q - \cup_{r \neq q}(R(G_r) \cap L_q)$
wait for receives to finish
{Phase III}
for $i = 1, \ldots, k$ **do**
  compute remaining $x_j^{(i)} \in (R(G_q) - L_q - \cup_{r \neq q}(R(G_q) \cap L_r))$

Figure 2.11 shows the entries (colored red) needed to be fetched from neighboring processors for $k = 3$. Figure 2.11(a) shows the entries needed from $x^{(0)}$ (level 0) for PA1. For PA2, however, red entries from $x^{(0)}$ (Figure 2.11(b)) and $x^{(1)}$ (Figure 2.11(c)) are needed. Figure 2.12 shows a side-by-side illustration of the entries computed in PA1 and PA2. Note that both the black and the blue entries are computed at a given level—the black entries are redundant computation.
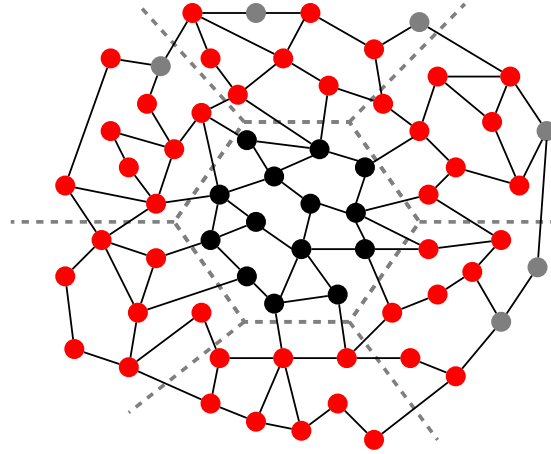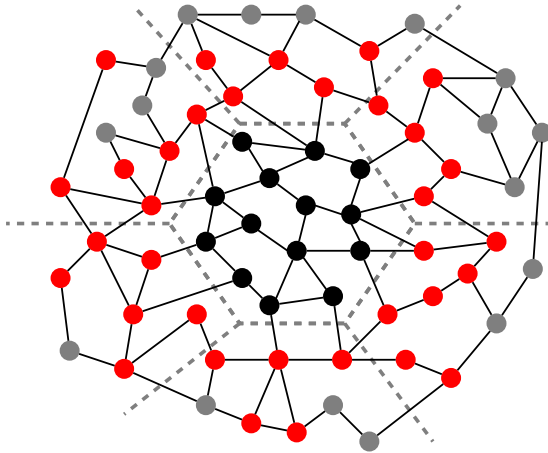
(a) PA1: entries to fetch from $x^{(0)}$

(b) PA2: entries to fetch from $x^{(0)}$
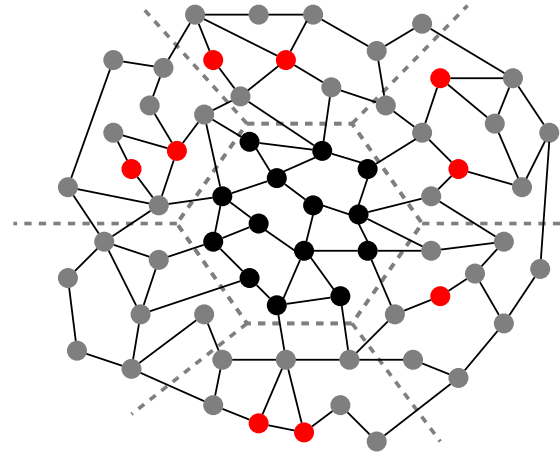
(c) PA2: entries to fetch from $x^{(1)}$

Figure 2.11: PA1 vs PA2: entries fetched

(a) PA1: computed entries of $x^{(1)}$.

(b) PA2: computed entries of $x^{(1)}$.

(c) PA1: computed entries of $x^{(2)}$.

(d) PA2: computed entries of $x^{(2)}$.

(e) PA1: computed entries of $x^{(3)}$.
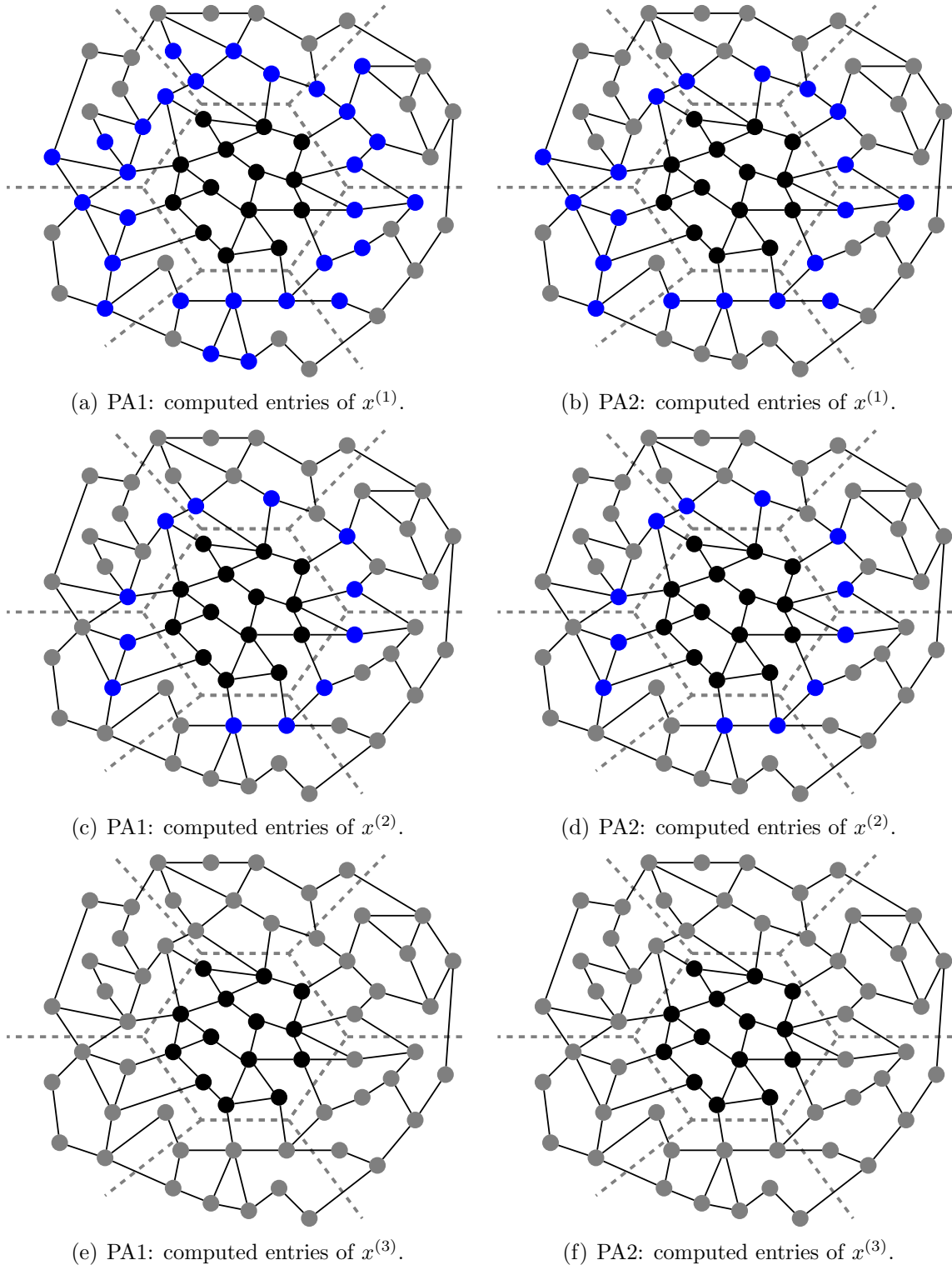
(f) PA2: computed entries of $x^{(3)}$.

Figure 2.12: PA1 vs. PA2 for a general graph for $k = 3$. Since the total number of blue circles is less for PA2, it performs fewer redundant computations.

2.5　Sequential Algorithms

Now consider the sequential algorithm with fast and slow memories. Let us motivate the two situations we are trying to optimize. It is typical for a large N-by-N sparse matrix to take many times as much memory as vector of length N. For example, a typical density of .1% means that an N-by-N matrix takes roughly $.001N^2$ 8-byte floats and 4-byte indices to store (assuming CSR[7] format), so roughly $.012N^2$ bytes, whereas a vector takes $8N$ bytes, which is smaller for $N > 666$, and typically many times smaller. So the two cases of most interest are:

1. the matrix does not fit in fast memory but the $k+1$ vectors $[x, Ax, \ldots, A^k x]$ do, and

2. neither the matrix nor the vectors fit in fast memory.

We note that for our model problems, the relative sizes of the vectors and matrix depends on k and the bandwidth b, and our algorithms are of most interest when the matrix is larger than the vectors. While this is not the case for tridiagonal matrices and $k > 2$, we will still use this simple case to illustrate how the algorithms work.

Conventional Sequential Approach (SA0). We assume the matrix does not fit in fast memory but the vectors do. This algorithm will keep all the components of $[x, Ax, \ldots, A^k x]$ in fast memory, and read all the entries of A from slow to fast memory to compute each vector $A^j x$, thereby reading A k times in all.

New Sequential Approach 1 (SA1). We again assume that the matrix does not fit in fast memory but the vectors do. SA1 will emulate PA1 by partitioning the matrix into p block rows, and looping from $i = 1$ to $i = p$, reading from slow memory those parts of the matrix needed to perform the same computations performed by processor i in PA1, and updating the appropriate components of $[Ax, \ldots, A^k x]$ in fast memory. Since all components of $[Ax, \ldots, A^k x]$ are in fast memory, no redundant computation is necessary. We choose p as small as possible, to minimize the number of slow memory accesses, as described below.

New Sequential Approach 2 (SA2). Now we assume that neither the matrix nor the vectors fit in memory. SA2 will still emulate PA1 by looping from $i = 1$ to $i = p$, but read from slow memory not just parts of the matrix but also those parts x needed to perform the same computations performed by processor i in PA1, and finally writing back to slow memory the corresponding components of $[Ax, ..., A^k x]$. Depending on the structure of A, redundant computation may or may not be necessary. We again choose p as small as possible.

For SA1, the total number of slow memory accesses is roughly the minimum number needed to read in the whole matrix once from slow memory, while also keeping $[x, Ax, ..., A^k x]$

---

[7]Compressed Sparse Row (CSR) format stores the nonzeros of the matrix as a row-oriented adjacency list.

in fast memory. For a sparse matrix with nnz nonzeros and a fast memory of size M bytes, the number of slow memory accesses is therefore roughly $(12nnz)/(M - 8(k+1)n)$. As long as $M \gg n$ and $nnz \gg n$, this number grows very slowly with k, justifying our claims of the latency cost being independent of k.

For SA2, we also need to read and write $k + 1$ vectors to and from slow memory, so the number of slow memory accesses is roughly $(8(k+1)n + 12nnz)/M$. As long as $8(k+1)n \overset{<}{\approx} 12nnz$, or

$$k \overset{<}{\approx} \frac{3}{2} \cdot \frac{nnz}{n} = \frac{3}{2} \cdot \text{ the average number of nonzeros per row,}$$

the number of slow memory accesses will again be roughly independent of k as claimed.

Our sequential approach is broadly similar to that of Strout [95] and Vuduc [104] but differs in that we assume a cost of "latency $+ n$/bandwidth" to read or write any contiguous set of n bytes from slow memory, where latency may be dominant. Therefore it is critical for us to organize our data structures so that data to be read from slow memory is entirely contiguous. We will see that this leads to new data layouts where, for example, we interleave matrix and vector entries. (In Section 2.5.5 we show that finding the optimal way to reorganize the data may be formulated via the Traveling Salesman Problem (TSP), but we only need to solve it approximately to get a reasonable solution.) This reorganization is not necessary in the parallel case, because the sending and receiving processors can pack and unpack data structures into contiguous memory segments, something a disk or memory prefetch unit cannot do (yet). This packing/unpacking has the effect of decreasing the effective bandwidth, but not the number of messages.

The rest of this section is organized as follows:

- Subsection 2.5.1 describes SA1 and SA2 in more detail for 1D meshes (band matrices).

- Subsection 2.5.2 describes SA1 and SA2 in more details for 2D and 3D meshes.

- Subsection 2.5.3 presents a tabular summary of all the operation counts for meshes.

- Subsection 2.5.4 describes SA1 and SA2 on general sparse matrices.

- Subsection 2.5.5 describes how to find the optimal ordering of unknowns, as well as good approximations to this ordering.

2.5.1   1D Meshes

We will explain both SA1 and SA2 for tridiagonal matrices, even though (as stated above) only SA2 makes sense in practice for such matrices, since SA1 uses too much fast memory. Algorithm 2.7 presents the conventional algorithm, for contrast.

The cost of this algorithm is $5kn$ flops, $3kn$ words read from slow memory, and kp accesses to slow memory. The memory required is $(k + 1)n + 3\frac{n}{p}$. Since the tridiagonal matrix has so few nonzeros, this conventional approach has no memory advantages over computing (and using and then overwriting!) the powers $A^j x$ one at a time.

Algorithm 2.7 SA0: Conventional sequential approach with fast/slow memory for 1D mesh with $b = 1$

{assume matrix stored in slow memory in p equal sized chunks, where chunk q consists of row s_q through e_q, assume $k + 1$ vectors $[x, Ax, ..., A^k x]$ all fit in fast memory}

{let $x^{(0)} = x$}

for $j = 1, \ldots, k$ **do**

   for $q = 1, \ldots, p$ **do**

      read rows s_q through e_q of matrix from slow memory

      {this assumes that the matrix is stored by rows, so that rows s_q through e_q are located contiguously}

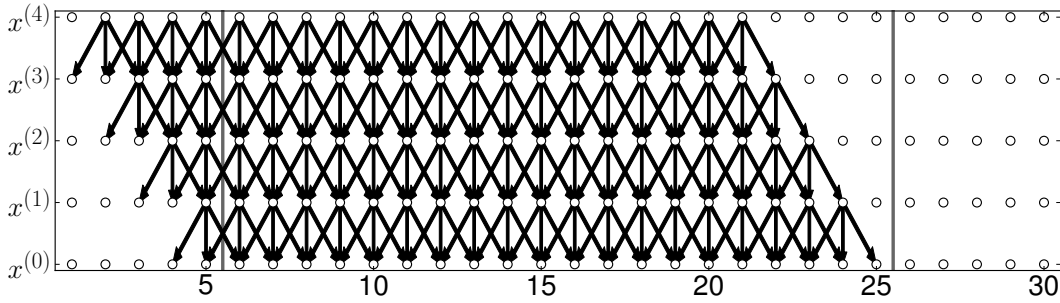   compute $x_{s_q}^{(j)} = (Ax^{(j-1)})_{s_q}, \ \ldots \ , x_{e_q}^{(j)} = (Ax^{(j-1)})_{e_q}$

Figure 2.13: Local dependencies in SA1 for $[Ax, ..., A^4 x]$ for tridiagonal matrix

As stated in the introduction, the presence of all components of $[Ax, ..., A^k x]$ in the same memory makes it unnecessary to perform any redundant computation in the new approach (Algorithm 2.8).

Algorithm 2.8 SA1 with fast/slow memory for 1D mesh with $b = 1$.

Require: matrix stored in slow memory in p equal sized chunks where chunk q consists of rows $s_q, s_q + 1, \ldots, e_q$

{assume $k + 1$ vectors $[x, Ax, ..., A^k x]$ all fit in fast memory}

{ignore boundaries $q = 1$ and $q = p$}

for $q = 1$ to p **do**

   read rows s_q through e_q of A from slow memory

   {note: for $q > 1$, rows $s_q - k$ through $s_q - 1$ were already read last time}

   {and must be kept in memory}

   {this also assumes that the matrix is stored by rows, so that rows s_q through e_q are located contiguously}

   compute locally dependent components of $A^j x$ as shown in Figure 2.13

The cost of this algorithm is $5kn$ flops (no more than the conventional algorithm), $3n$ words read from slow memory (the matrix is read just once), and p accesses to slow memory

(in contrast to kp for the conventional algorithm). The memory required is $(k+1)n + 3\frac{n}{p}$ as for SA0, plus $3k$ more for keeping rows $s_q - k$ through $s_q - 1$ of A in fast memory.

By unrolling the loop in SA1 (but at the cost of more fast memory), the latency of the read from slow memory could be overlapped with computation.

Algorithm 2.9 SA2 (1D mesh with $b = 1$)

Require: matrix A and vectors $[x, Ax, ..., A^k x]$ stored in slow memory in p equal sized chunks, where chunk q consists of rows $s_q, s_q + 1, \ldots, e_q$
  {ignore boundaries $q = 1$ and $q = p$}
  for $q = 1$ to p **do**
    read rows $s_q, s_q + 1, \ldots, e_q$ of A and of $[x, Ax, ..., A^k x]$ from slow memory
    {for $q > 1$, rows $s_q - k$ through $s_q - 1$ of A and of $[x, Ax, ..., A^k x]$ were already read last time, and must be kept in memory}
    compute locally dependent components of $A^j x$ as shown in Figure 2.13
    write rows $s_q - k, s_q - k + 1, \ldots, e_q - k$ of $[Ax, ..., A^k x]$ back to slow memory

---

In order to perform the read in SA2 (Algorithm 2.9) in exactly one slow memory access, we would need to interleave the data structures of A and of $[x, Ax, ..., A^k x]$ so that the first n/p consecutive rows of A and of $[x, Ax, ..., A^k x]$ were stored contiguously together, then the second n/p rows of both, and so on. In order to also perform the write in SA2 in exactly one slow memory access, the n/p rows of x would have to come at the end of the corresponding rows of A and of $[Ax, ..., A^k x]$. We can simplify these data structures at the cost of increasing the number of slow memory accesses from 2 to at most 5, by storing A, x and $[Ax, ..., A^k x]$ separately (but still by rows).

The cost of this algorithm is $5kn$ flops (no more than the conventional algorithm), $3n + (k+1)n$ words read from slow memory (the matrix and all the vectors are read just once), and p accesses to slow memory, assuming the best possible interleaved layout of A and $[x, Ax, ..., A^k x]$ just described (in contrast to kp for the conventional algorithm).

The memory required is $3\frac{n}{p} + 3k$ for the matrix and $(k+1)\frac{n}{p} + k(k+1)$ for the vectors, roughly a factor of p less than the conventional algorithm. If the main memory size is M words, we get the inequality $(k+4)(\frac{n}{p} + k) \leq M$, or $p \geq \frac{n(k+4)}{M - k(k+4)} \approx \frac{n(k+4)}{M}$ as the minimum number of slow memory accesses.

We can avoid all redundant flops since we can "leave behind" the components of $[x, Ax, ..., A^k x]$ in memory. This phenomenon is unfortunately limited to a 1D mesh, and higher dimensional meshes will again have some redundant work, as they did in the parallel case.

By unrolling the loop in SA2 (but at the cost of more fast memory), the latency of the read from slow memory can again be overlapped with computation.

Now we consider matrices with bandwidth $b > 1$, i.e., band matrices. The conventional algorithm for $b > 1$ differs very little from the $b = 1$ base described above. The costs are $(4b+1)kn$ flops, $(2b+1)kn$ words read from slow memory, and kp accesses to slow memory. The fast memory required is $(k+1)n + (2b+1)\frac{n}{p}$.

SA1 for bandwidth $b > 1$ differs from $b = 1$ as follows: Instead of leaving rows $s_q - k$ through $s_q - 1$ of A in memory, we must leave rows $s_q - kb$ through $s_q - 1$. Altogether, the

costs are $(4b+1)kn$ flops as before, $(2b+1)n$ words read from slow memory (the matrix is read once), and p accesses to slow memory. The fast memory required is $(k+1)n$ words for the vector entries plus $(2b+1)(\frac{n}{p}+kb)$ words for the matrix entries.

SA2 for bandwidth $b>1$ also differs from $b=1$ by needing to keep rows $s_q - kb$ through $s_q - 1$ of A and $[x, Ax, ..., A^k x]$ in memory. The number of flops and slow memory accesses are the same as for SA1. The fast memory required goes down to $(2b+1)(\frac{n}{p}+kb)$ words for the matrix entries and $(k+1)(\frac{n}{p}+kb)$ for the vector entries, or $(2b+k+2)(\frac{n}{p}+kb)$ in all. The number of words read from slow memory increases to $(k+1)n+(2b+1)n$, since the vectors and the matrix need to be read in once.

2.5.2   2D and 3D Meshes

With the 1D mesh, the natural ordering of the unknowns had several attractive properties: the matrix and vector entries were ordered to minimize the number of accesses to slow memory (i.e., the "boundary vertices" were always contiguous to each partition), and all redundant flops could be avoided because the required data was already in memory, without requiring extra fast memory. Neither of these is possible for 2D or 3D meshes (or for a general graph), but we will nevertheless minimize the amount of redundant work and number of slow memory accesses, as we did in the parallel case. This is more difficult in the sequential than the parallel case, because in the parallel case we could have the sending processor pack up all the desired vectors entries (matrix entries were not communicated) into a single contiguous message to be sent. This could be modeled by using a slightly lower communication bandwidth to account for the copying (which may be done anyway by the communication layer). In contrast, in the sequential case, there is generally no opportunity to reorder data in the slow memory: If data is not contiguous, either more accesses are needed (and so more latency costs are incurred), or else more data than needed is fetched (and so more bandwidth costs are incurred).

To illustrate, consider the grid points associated with a single partition, as shown in left of Figure 2.14. The unknowns within each numbered region are ordered contiguously, and the regions are ordered as shown. Thus, when the North region requires data, it can read regions 1 through 5 in one access. Similarly, the NE region can read region 5, the E region can read regions 4 through 8, the SE region can read region 8, the S region can read regions 7 through 11, and the SW region can read region 11. However, the W region needs 10, 11, 12, 1 and 2, and so requires 2 slow memory accesses (or else fetching regions 1 through 12 in one access and throwing away the unneeded data). Since the adjacency graph of the regions 1 through 12 has a cycle, one can easily see that no linear order exists requiring only 1 slow memory access for all the required data.

Unlike the 1D case, we will store x and the other vectors $[Ax, ..., A^k x]$ separately, using the order shown in Figure 2.14(a). Figure 2.14(b) shows a similar, simpler ordering with the same number of slow memory accesses, and with simpler indexing, but which accesses slightly more words than necessary from slow memory (the triangles 2, 4, 7 and 10 are unnecessarily sent to the corner processors). The simpler indexing could result in a faster algorithm.

(a) Ordering for 2D 5-pt stencil mesh   (b) Ordering for 2D 9-pt stencil mesh
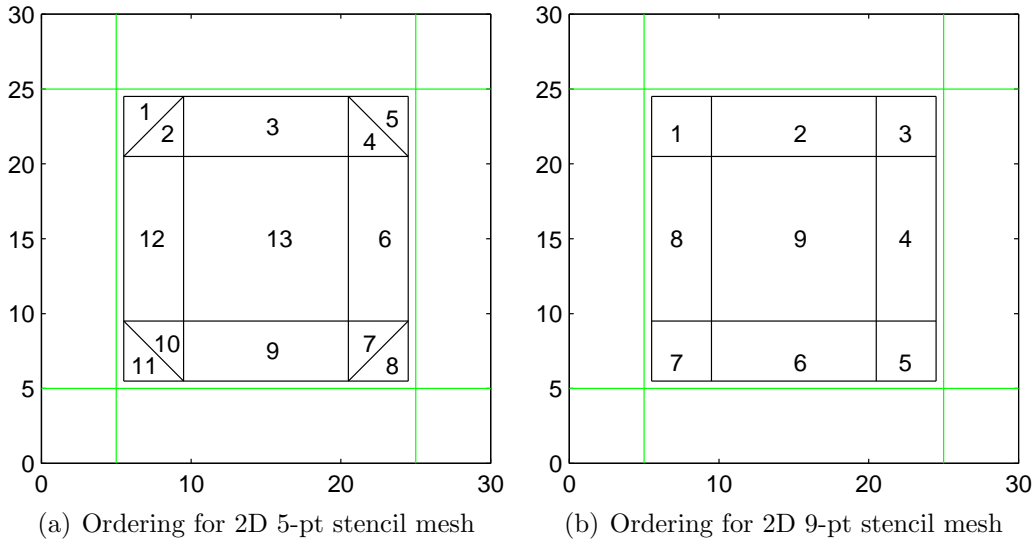
Figure 2.14: Ordering the unknowns in a 2D Mesh for contiguity

Figuring out orderings of grid points analogous to these, but for general graphs, that minimize the number of slow memory accesses, may be reduced to an instance of the Traveling Salesman Problem (see Section 2.5.5).

Since the matrix entries are read-only, we can minimize the number of slow memory accesses to the matrix by using extra slow memory (which is cheap) to store some rows of A redundantly, so that the needed rows of A for any region are always stored contiguously. We will henceforth assume this has been done as a preprocessing step, and not count its costs in the summary table. This extra slow memory, like other extra costs, is proportional to the size of the boundary, and so is asymptotically small.

Using the above assumptions and data layouts, the conventional sequential algorithm works analogously to SA0 in Section 2.5.1: The cost is $9kn^2$ flops, $5kn^2$ words read from slow memory in kp slow memory accesses, and $(k+1)n^2$ fast memory words for the vectors and $5\frac{n^2}{p}$ fast memory words for the matrix.

The costs for SA2 are $12p$ slow memory accesses, $(6+k)n^2 + 24knp^{1/2} + 12pk^2$ words read from or written to slow memory, and $9kn^2 + 18k^2np^{1/2} + 6k^3p$ flops, p times as many as PA1.

A similar counting exercise leads to the other entries in the summary table in the next section.

2.5.3   Summary of Sequential Complexity on Meshes

In the summary table for sequential algorithms (Table 2.4), "Acc" is the number of accesses of slow memory (reads or writes), "MWords" is the total number of matrix entries accessed, "VWords" is the total number of vector entries accessed, "Flops" is the number of floating point operations, "MMem" is the fast memory required for matrix entries, and

---

Algorithm 2.10 SA1: New sequential approach 1 with fast/slow memory for 2D mesh with 5-point stencil. It costs $9kn^2$ flops, accesses slow memory p times, and reads $5n^2 + 20kn^{1/2} + 10pk^2$ words from slow memory.

{assume matrix stored in slow memory in p equal sized chunks, where chunk q consists of rows s_q through e_q along with, B "boundary rows" needed for redundant computation of border regions, assume $k + 1$ vectors $[x, Ax, ..., A^k x]$ all fit in fast memory;}

{ignore boundaries $q = 1$ and $q = p$}

for $q = 1, \ldots, p$ **do**

    read rows s_q through $e_q + B$ of A from slow memory

    {rows $e_q + 1$ through $e_q + B$ are redundant copies of "boundary rows"}

    compute locally dependent components of $A^j x$ as shown in Figure 2.5 reusing previously computed red circles

---

Algorithm 2.11 SA2: New sequential approach 2 with fast/slow memory for 2D mesh with 5 point stencil

---

{assume matrix stored in slow memory in p equal sized chunks, where chunk q consists of rows s_q through e_q along with B "boundary rows" needed for redundant computation of border regions, assume x stored in slow memory in corresponding p chunks, where each chunk internally ordered as in the left of Figure 2.14; assume k vectors $[Ax, ..., A^k x]$ stored analogously to x}

{ignore boundaries $q = 1$ and $q = p$}

for $q = 1, \ldots, p$ **do**

    read rows $s_q, \ldots, e_q + B$ of A from slow memory {rows $e_q + 1, \ldots, e_q + B$ are redundant copies of "boundary rows"}

    read rows $s_q, \ldots, e_q$ of x from slow memory {this costs one slow memory access}

    read needed rows of x from N, NE, E, SE, S, SW, W and NW regions {costs 9 slow memory accesses}

    compute locally dependent components of $A^j x$ as shown in Figure 2.5

    write rows $s_q, \ldots, e_q$ of $[Ax, ..., A^k x]$ to slow memory

---

"VMem" is the fast memory required for vector entries. Lower order terms are sometimes omitted for clarity.

We consider the special case of stencil matrices, where in addition to the graph being a stencil, each row of the matrix has the same nonzero entries (modulo boundaries). In this case no matrix entries need to be fetched from slow memory (MWords=0) and at most MMem $\leq (2b + 1)^d$ words are needed to store the matrix entries for a d-dimensional mesh, not $(2b + 1)^d \frac{n^d}{p}$. Otherwise the table entries do not change.

2.5.4   General Graphs

We use notation defined in section 2.4.4. In the pseudocode "read" means from slow to fast memory, and "write" means from fast to slow. Algorithms 2.12, 2.13 and 2.14 describe

Problem	Costs	Conventional Approach	Sequential Approach 1	Sequential Approach 2
1D mesh $b \geq 1$	Acc	kp	p	p
	MWords	$(2b+1)kn$	$(2b+1)n$	$(2b+1)n$
	VWords	0	0	$(k+1)n$
	Flops	$(4b+1)kn$	$(4b+1)kn$	$(4b+1)kn$
	MMem	$(2b+1)\frac{n}{p}$	$(2b+1)(\frac{n}{p}+bk)$	$(2b+1)(\frac{n}{p}+bk)$
	VMem	$(k+1)n$	$(k+1)n$	$(k+1)(\frac{n}{p}+bk)$
2D mesh $(2b+1)^2$ pt stencil	Acc	kp	p	$12p$
	MWords	$(2b+1)^2kn^2$	$(2b+1)^2(n^2+4bknp^{1/2}+4pb^2k^2)$	$(2b+1)^2(n^2+4bknp^{1/2}+4pb^2k^2)$
	VWords	0	0	$(k+1)n^2+4bknp^{1/2}+4pb^2k^2$
	Flops	$(8b^2+8b+1)kn^2$	$(8b^2+8b+1)kn^2$	$(8b^2+8b+1)\cdot$ $(kn^2+2bk^2np^{1/2}+\frac{4}{3}b^2k^3p)$
	MMem	$(2b+1)^2\frac{n^2}{p}$	$(2b+1)^2(\frac{n^2}{p}+4bk\frac{n}{p^{1/2}}+4b^2k^2)$	$(2b+1)^2(\frac{n^2}{p}+4bk\frac{n}{p^{1/2}}+4b^2k^2)$
	VMem	$(k+1)n^2$	$(k+1)n^2$	$(k+1)\frac{n^2}{p}+4bk\frac{n}{p^{1/2}}+4b^2k^2$
3D mesh $(2b+1)^3$ pt stencil	Acc	kp	p	$O(p)$
	MWords	$(2b+1)^3kn^3$	$(2b+1)^3\cdot$ $(n^3+6bkn^2p^{1/3}+O(b^2k^2np^{2/3}))$	$(2b+1)^3\cdot$ $(n^3+6bkn^2p^{1/3}+O(b^2k^2np^{2/3}))$
	VWords	0	0	$(k+1)n^3+6bkn^2p^{1/3}+O(b^2k^2np^{2/3})$
	Flops	$(2(2b+1)^3-1)kn^3$	$(2(2b+1)^3-1)kn^3$	$(2(2b+1)^3-1)\cdot$ $(kn^3+3bk^2n^2p^{1/3}+O(b^2k^3np^{2/3}))$
	MMem	$(2b+1)^3\frac{n^3}{p}$	$(2b+1)^3\cdot$ $(\frac{n^3}{p}+6bk\frac{n^2}{p^{2/3}}+O(b^2k^2\frac{n}{p^{1/3}}))$	$(2b+1)^3\cdot$ $(\frac{n^3}{p}+6bk\frac{n^2}{p^{2/3}}+O(b^2k^2\frac{n}{p^{1/3}}))$
	VMem	$(k+1)n^3$	$(k+1)n^3$	$(k+1)\frac{n^3}{p}+6bk\frac{n^2}{p^{2/3}}+O(b^2k^2\frac{n}{p^{1/3}})$

Table 2.4: Summary for sequential algorithms (lower order terms omitted).

SA0, SA1 and SA2 respectively for general sparse matrices.

Algorithm 2.12 SA0: conventional sequential algorithm for a general graph

---

{vectors fit in fast memory but the matrix doesn't}

for $i = 1$ to k **do**

    for $q = 1$ to p **do**

        read all rows j of A such that some $x_j^{(i)} \in G_q^{(i)}$

        compute all $x_j^{(i)}$ in $G_q^{(i)}$

2.5.5 The Ordering Problem in Sequential Algorithms

As illustrated in Figure 2.14, the order in which vector components are stored within each block influences the number of slow memory accesses needed to read the data needed from neighboring blocks, namely the data $R_r^{(0)}(G_q)$ that block q needs from block r, for all $r \neq q$. The left part of Figure 2.14 shows the best order for a 2D mesh with a 5 point stencil. (The components within each block can be numbered in any order, but all the components in block i must be numbered before those in block $i+1$, and so on.) In this case, where each block has 8 neighboring blocks, 8 (simultaneous) accesses is clearly a lower bound. If we insist on reading only the needed data, then the best we can do is 9 accesses, as discussed in Section 2.5.2.

Alternatively, when regions 1, 2, 10, 11 and 12 are needed, all regions 1 through 12 could be fetched and regions 3 through 9 discarded. But in this section we consider solutions that

---

Algorithm 2.13 SA1 for general graph

> {emulate PA1, assuming the vectors fit in fast memory but the matrix does not}
> {no redundant arithmetic required}
> $S \leftarrow \phi$ {S is set of $x_j^{(i)}$ computed so far}
> **for** $q = 1$ to p **do**
> $S' \leftarrow \{x_j^{(i)} : R^{(0)}(x_j^{(i)}) \subset G_q^{(0)} \cup S^{(0)}\} - S$
>   {S' = set of $x_j^{(i)}$ that depend only on current and previous $x_m^{(0)}$}
>   read all rows j of A such that some $x_j^{(i)} \in S'$
>   {may store some rows of A redundantly to reduce # reads to 1}
>   compute all $x_j^{(i)}$ in S' (in order of increasing i)
> $S \leftarrow S \cup S'$

Algorithm 2.14 SA2 for general graph

> {emulate PA, assuming neither vectors nor matrix fits in fast memory}
> {will perform redundant arithmetic as in PA1}
> $S \leftarrow \phi$ {S is set of $x_j^{(i)}$ computed so far}
> **for** $q = 1$ to p **do**
>   read $G_q^{(0)}$
>   read all rows j of A such that some $x_j^{(i)} \in R(G_q)$
>   {may store some rows of A redundantly to reduce # reads to 1}
>   compute all $x_i^{(j)}$ in L_q
>   **for all** $r \neq q$ **do**
>   read $R_r^{(0)}(G_q)$
>   {possible to optimize order in which $x_j^{(0)}$ stored to minimize # slow memory accesses}
>   compute remaining $x_j^{(i)}$ in $R(G_q) - L_q$
>   write $G_q^{(1:k)}$

---

fetch only the required data, and so always minimize the bandwidth costs.

Furthermore, the order in which we access blocks is important. For example, for the 2D mesh, if we access blocks from left to right in each processor row, then the needed data from the previous block is still in fast memory and does not need to be accessed. So we also need to choose the order in which to access blocks. We call this *block ordering*, as opposed to the *component ordering* within an individual block, as discussed in the first paragraph.

In this section we ask how to determine the optimal block ordering and component ordering for SA2 for a general graph, by reducing the question to an instance of the Traveling Salesman Problem (TSP).

For the rest of this subsection, we let n denote the dimension of the matrix A. We will start by assuming a block ordering is given, so that the blocks of the vector x are $B_1, \ldots, B_p$, where each B_i is a disjoint subset of $\{1, ..., n\}$, and show how to choose the optimal component ordering within each block. Later we will show how to choose the optimal block ordering. Let $|B_i|$ denote the number of elements in block B_i. Let $N_{i,j}$ denote

the set of elements of block B_i needed when computing block B_j. Let $E_{i,j}$ denote the set of elements of block B_i needed to be fetched from slow memory when computing for block B_j. The set $E_{i,j}$ is fixed by the ordering of the blocks—if block B_j comes immediately after block B_l, then $E_{i,j} = N_{i,j} - N_{i,l}$, i.e., we only fetch elements which are not already in fast memory. So, there will always be an implicit ordering of the blocks when we talk about $E_{i,j}$. We call block B_i a neighbor of block B_j if $E_{i,j} \neq \emptyset$.

Component ordering

Let $A(E_{i,j})$ denote the number of accesses required to fetch the set of elements $E_{i,j}$ from slow memory, assuming blocks are processed in increasing order from B_1 to B_p. Therefore, total number of slow memory accesses required is $\mathcal{A} = \sum_{j=1}^{p}(\sum_{i=1}^{p} A(E_{i,j})) = \sum_{i=1}^{p}(\sum_{j=1}^{p} A(E_{i,j}))$. Since the ordering of elements inside block B_i only affects the sum $\mathcal{A}_i = \sum_{j=1}^{p} A(E_{i,j})$, we simply need to optimize $\mathcal{A}_i$ independently for block B_i. Given this observation, we now formalize the block level ordering problem for an individual block.

Let B_i be the block under consideration. Without loss of generality, let $1, 2, \ldots, m$ be the elements of block B_i. Also, assume that none of $E_{i,j}$ ($j \neq i$) are empty (if there is an empty $E_{i,j}$, then we simply remove it from consideration resulting in a smaller p). Similarly, assume that all the elements of B_i are in some $E_{i,j}$ (if not, such element(s) can be placed in a contiguous segment at the end without affecting the optimality of an ordering; for example these would be the components in region 13 on the left of Figure 2.14). Let $I_{i,j}$ be the indicator function for whether $a \in E_{i,j}$ ($I_{i,j}(a) = 1$ iff $a \in E_{i,j}$). We now have the following theorem:

Theorem 1 *Let $1, 2, \ldots, m$ be the elements of block B_i. Add a dummy element $m + 1$ to the block. Let $m + 1 \notin E_{i,j}$ ($1 \leq j \leq p$). Given an ordering ρ of these $m + 1$ elements such that $\rho(m + 1) = m + 1$, we have $A(E_{i,j}) = \sum_{k=1}^{m} I_{i,j}(\rho(k)) \cdot (1 - I_{i,j}(\rho(k + 1)))$.*

Proof: $A(E_{i,j})$ is the same as the number of contiguous segments of the set $E_{i,j}$ under the ordering ρ. One way of counting this is to look at the elements of B_i in the order specified by ρ and add 1 to the count whenever we encounter a boundary, i.e., when $\rho(k) \in E_{i,j}$ and $\rho(k + 1) \notin E_{i,j}$. Equivalently, for the k-th element in the ordering, we add $I_{i,j}(\rho(k)) \cdot (1 - I_{i,j}(\rho(k + 1)))$ to the count. The reason we added a dummy element is to account for the case when the last contiguous segment ends at element m. So, we get $A(E_{i,j}) = \sum_{k=1}^{m} I_{i,j}(\rho(k)) \cdot (1 - I_{i,j}(\rho(k + 1)))$. $\square$

Using Theorem 1, we get

$$
\begin{aligned}
\mathcal{A}_i &= \sum_{j=1}^{p} \sum_{k=1}^{m} I_{i,j}(\rho(k)) \cdot (1 - I_{i,j}(\rho(k + 1))) \\
&= \sum_{k=1}^{m} \left(\sum_{j=1}^{p} I_{i,j}(\rho(k)) \cdot (1 - I_{i,j}(\rho(k + 1))) \right).
\end{aligned}
$$

Now, consider the weighted directed complete graph $\mathcal{G}_i = (V_i, E_i)$, $V_i = \{v_1, \ldots, v_{m+1}\}$ (one node for each element of block B_i, v_{m+1} for the dummy element in Theorem 1). Let $wt(v_a, v_b)$

(weight of edge from node v_a to node v_b) be $\sum_{j=1}^{p} I_{i,j}(a) \cdot (1 - I_{i,j}(b))$—the contribution to the total count of the number of disk accesses if element b is placed immediately after element a. Consider an ordering ρ_i of the elements of block B_i (such that $\rho_i(m+1) = m+1$). The total number of disk accesses due to this ordering is

$$\mathcal{A}_i = \sum_{k=1}^{m} \left(\sum_{j=1}^{p} I_{i,j}(\rho_i(k)) \cdot (1 - I_{i,j}(\rho_i(k+1))) \right).$$

The total weight of the path $v_{\rho_i(1)}, v_{\rho_i(2)}, \ldots, v_{\rho_i(m+1)}$ is also $\mathcal{A}_i$. Equivalently, we can say that the Hamiltonian path ending at node v_{m+1} has the same weight as the number of disk accesses for the corresponding ordering. Thus, an optimal ordering (with the constraint of $m+1$ being the last element) corresponds to the lowest weight Hamiltonian path (with the constraint of v_{m+1} being the last node). By setting $wt(v_{m+1}, v_a)$ $(1 \le a \le m)$ large enough, v_{m+1} will be the last node in any lowest weight Hamiltonian path. In fact, $wt(v_{m+1}, v_a) = 1 + \max_{k=1}^{m} \sum_{j=1}^{p} I_{i,j}(a)$ does the trick. Since $wt(v_{m+1}, v_a)$ is independent of a (for $1 \le a \le m$), the lowest weight Hamiltonian path corresponds to the lowest weight Traveling Salesman tour by using the edge between v_{m+1} (since it lies at the end in any optimal Hamiltonian path) and the first node in the Hamiltonian path. So, the problem can also be formulated as a Traveling Salesman problem.

In summary, to find the optimal ordering of the components of the block B_i, construct the graph $\mathcal{G}_i$ (discussed in the previous paragraph) and find the lowest weight Hamiltonian path. The ordering defined by the Hamiltonian path is the optimal ordering for the components of the block.

Reducing the problem size for component ordering

It appears that the size of the component ordering problem is $O(m^2)$, where m is the number of components of each block needed by other blocks. In general m will grow with problem dimension n and be quite large. Here we show how to reduce the TSP problem size for each block to at most its number of neighboring blocks, which is usually quite small, for example $3^d - 1$ in the case of a d-dimensional mesh, independent of the number of mesh points. The intuition, again from Figure 2.14, is that the components can be put into equivalence classes (numbered there on the left from 1 through 12) each of which is needed by the same set of neighboring blocks, and then the equivalence classes ordered. So here we will formally construct the equivalence relation on components, and show that in any optimal ordering, equivalent components are numbered consecutively.

For the above graph $\mathcal{G}_i$, we say that node v_a is related to node v_b ($v_a R_i v_b$) iff $wt(v_a, v_b) = wt(v_b, v_a) = 0$. Clearly, the relation R_i is an equivalence relation. The next 3 theorems make several observations about the relation R_i.

Theorem 2 *If $v_a R_i v_b$ then, $a \in E_{i,j}$ iff $b \in E_{i,j}$ and vice versa. In other words, the equivalence relation R_i is also defined as being contained in the same set of sets $E_{i,j}$.*

Proof: First of all, we note that v_{m+1} is not related to any other node, since $wt(v_{m+1}, v_a) \ne 0$ for any $1 \le a \le m$. Also, there is no $1 \le a \le m$ such that $a \in E_{i,j}$ iff $m+1 \in E_{i,j}$ since each a

is in some $E_{i,j}$ but $m+1$ is not in any $E_{i,j}$. So, we only need to consider other elements/nodes. $wt(v_a, v_b) = \sum_{j=1}^{p} I_{i,j}(a) \cdot (1 - I_{i,j}(b)) = 0$ implies that for all $1 \le j \le p$, $I_{i,j}(a) \cdot (1 - I_{i,j}(b)) = 0$ (all these terms are non-negative). Similarly, $wt(v_b, v_a) = \sum_{j=1}^{p} I_{i,j}(b) \cdot (1 - I_{i,j}(a)) = 0$ implies that for all $1 \le j \le p$, $I_{i,j}(b) \cdot (1 - I_{i,j}(a)) = 0$. This implies that, for all $1 \le j \le p$, $I_{i,j}(a) = I_{i,j}(b)$ which means that $a \in E_{i,j}$ iff $b \in E_{i,j}$. Similarly, if $a \in E_{i,j}$ iff $b \in E_{i,j}$, then for all $1 \le j \le p$, $I_{i,j}(a) = I_{i,j}(b)$ which implies that $I_{i,j}(b) = I_{i,j}(b) \cdot I_{i,j}(a) = I_{i,j}(a)$ for all $1 \le j \le p$, which implies that $wt(v_a, v_b) = wt(v_b, v_a) = 0$. This completes the proof. $\square$

From Theorem 2, we get the following—if $v_a R_i v_b$, then for any v_c, $wt(v_a, v_c) = wt(v_b, v_c)$ and $wt(v_c, v_a) = wt(v_c, v_b)$. Now, consider an optimal ordering ρ_i of the elements of B_i. Theorem 2 implies that there are optimal orderings in which all elements which are equivalent are placed contiguously.

Theorem 3 *For any 3 distinct nodes v_a, v_b and v_c, $wt(v_a, v_b) + wt(v_b, v_c) \ge wt(v_a, v_c)$.*

Proof: We consider 4 possible cases:

1. $a = m + 1$: $wt(v_{m+1}, v_b) + wt(v_b, v_c) - wt(v_{m+1}, v_c) = wt(v_b, v_c) \ge 0$ ($wt(v_{m+1}, v_b)$ is the same for all $b \ne m + 1$).

2. $b = m + 1$: $wt(v_a, v_{m+1}) + wt(v_{m+1}, v_c) - wt(v_a, v_c) \ge wt(v_a, v_{m+1}) - wt(v_a, v_c)$. But, $I_{i,j}(a) - I_{i,j}(a) \cdot (1 - I_{i,j}(c)) \ge 0$ for all a, c. Since $wt(v_a, v_{m+1}) = \sum_{j=1}^{p} I_{i,j}(a)$ and $wt(v_a, v_c) = \sum_{j=1}^{p} I_{i,j}(a) \cdot (1 - I_{i,j}(b))$, the inequality holds true.

3. $c = m + 1$: Consider

$$wt(v_a, v_b) + wt(v_b, v_{m+1}) - wt(v_a, v_{m+1}) = \sum_{j=1}^{p} \left(I_{i,j}(a) \cdot (1 - I_{i,j}(b)) + I_{i,j}(b) - I_{i,j}(a) \right).$$

Since $I_{i,j}(a) \cdot (1 - I_{i,j}(b)) + I_{i,j}(b) - I_{i,j}(a) \ge 0$ for all a, b, the inequality holds true here too.

4. $a, b, c \ne m + 1$: Consider

$$
\begin{aligned}
I_{i,j}(a)(1 - I_{i,j}(b)) + I_{i,j}(b)(1 - I_{i,j}(c)) - I_{i,j}(a)(1 - I_{i,j}(c)) &= \\
I_{i,j}(b)(1 - I_{i,j}(a) - I_{i,j}(c)) + I_{i,j}(a)I_{i,j}(c) &\ge \\
1 - I_{i,j}(a) - I_{i,j}(c) + I_{i,j}(a)I_{i,j}(c) &\ge \quad 0.
\end{aligned}
$$

Since

$$
\begin{aligned}
wt(v_a, v_b) + wt(v_b, v_c) - wt(v_a, v_c) &= \sum_{j=1}^{p} (I_{i,j}(a)(1 - I_{i,j}(b)) + I_{i,j}(b)(1 - I_{i,j}(c)) - \\
&\quad I_{i,j}(a)(1 - I_{i,j}(c))),
\end{aligned}
$$

the inequality holds.

$\square$

Theorem 4 *There exists an optimal Hamiltonian path such that if $v_a R_i v_b$, and v_a comes before v_b in the path, then there is no v_c between v_a and v_b in the path such that $\neg v_a R_i v_c$.*

Proof: Consider an optimal Hamiltonian path such that the condition in the theorem does not hold true. By our choice of weights, v_{m+1} would be the last node in the path. Let v_a and v_b be nodes such that the following holds: v_a comes before v_b in the path, $v_a R_i v_b$ and $\neg v_a R_i v_c$ for any v_c which lies between v_a and v_b in the path. This must be possible simply because we assumed the condition in the theorem to be false. Let v_d be the node immediately after v_a, v_e be the node immediately after v_b and v_f be the node immediately before v_b. If we move node v_b to between v_a and v_d, then the change in the weight of the path is

$$
\begin{aligned}
wt(v_a, v_b) + wt(v_b, v_d) + wt(v_f, v_e) - wt(v_a, v_d) - wt(v_f, v_b) - wt(v_b, v_e) &= \\
0 + wt(v_a, v_d) + wt(v_f, v_e) - wt(v_a, v_d) - wt(v_f, v_b) - wt(v_b, v_e) &= \quad \text{(Theorem 2)} \\
wt(v_f, v_e) - wt(v_f, v_b) - wt(v_b, v_e) &\leq \quad 0 \ \text{(Theorem 3)}
\end{aligned}
$$

Since the original path was optimal, the cost must not decrease, hence the change must be 0. Applying this procedure of putting together related nodes eventually terminates in a Hamiltonian path such that the condition in the theorem is true. Furthermore, after each application of this procedure the cost did not change, which implies that the final path is also optimal. $\square$

Theorem 4 tells us that we can indeed reduce the problem size to ordering only equivalence classes of components, where two components are equivalent if and only if they are needed to compute the same set of blocks.

Block ordering

We now construct a weighted directed graph where there is one vertex per block B_i and one edge from every vertex to every other vertex. The weight of the edge e pointing from B_i to B_j will be the memory cost of processing B_j immediately after B_i. Clearly, given this graph, the goal is to find the lowest weight Hamiltonian path. As before, we can add a "dummy node" and appropriately weighted edges to convert to an instance of TSP.

Now we discuss the edge weights. If e points from B_i to B_j, so that B_j is processed immediately after B_i, the discussion in previous subsections tells us how to compute the minimum number of slow memory accesses needed to process B_j. This could be used as a weight by itself, if latency were dominant. But we can easily take the full cost into account, including bandwidth and latency, by setting the edge weight to the sum of $\alpha \cdot$ (# slow memory accesses) and $\beta \cdot$ (# words fetched from slow memory), where α is latency and β is reciprocal bandwidth. The number of words fetched from slow memory does not depend on the ordering, and is a by-product of the graph traversals needed to compute all the regions $R_r^{(0)}(G_q)$ needed by SA2.

Other problem formulations

In the previous section, we only considered exact solutions, i.e., we fetch only those elements which are required. However, if we sometimes fetch more elements than needed

when working on a block, we might be able to further lower the latency cost, at the price of higher bandwidth cost. For example, suppose we merged $E_{i,j}$ and $E_{i,j+1}$ ($E_{i,j} \neq E_{i,j+1}$), so that while computing block B_j we fetched the components in $E_{i,j} \cup E_{i,j+1}$, and similarly for block B_{j+1}. However, by merging the sets, we have reduced the number of equivalence classes in the component ordering problem, which might allow for fewer slow memory accesses.

If latency is the dominant cost we can go further and discuss solutions limited to a single slow memory access per E_{ij}, and choose the component ordering to minimize the bandwidth cost. In this case, the cost of accessing E_{ij} given a component ordering will be $h - l + 1$, where x_h is the highest numbered component in E_{ij} and x_l is the lowest numbered component (because all of $\{x_l, x_{l+1}, ..., x_h\}$ will needed to be fetched).

The best formulation may depend on other details of the performance model. Here we have been assuming a simple latency + bandwidth model, but depending on opportunities for overlap, parallelism, prefetching, etc. a different model may be appropriate. There are also many options for approximate solutions to the resulting combinatorial optimization problems, like TSP.

2.6 Asymptotic Performance Models

We consider how to asymptotically minimize the time to compute $[Ax, ..., A^{\bar{k}}x]$ for matrices with stencil graphs. In other words, the graph of the matrix is assumed to be a d-dimensional mesh with a $(2b+1)^d$ point stencil. We will treat all dimensions d simultaneously by using the notation $\gamma = \frac{n}{p^{1/d}}$. We can think of γ as a measure of problem size per processor, since it is the d-th root of the number of components per vector per processor. We also note that $2d/\gamma$ is the surface-to-volume ratio of a d-dimensional cube of side length γ.

We will compare the conventional parallel method (run $k = \bar{k}$ times) with the new method, where $\bar{k}/k$ groups of k matrix-vector-products are computed using $O(\bar{k}/k)$ messages. We will choose k to minimize the time of the new algorithm. We also compare the new method with the conventional method using overlap of communication and computation, and ask when this is sufficient to hide communication costs.

We let α be the message latency, β be the reciprocal bandwidth (so it takes $\alpha + n\beta$ seconds to send a message of length n), and f be the time per flop. Sample values are $f = 1$ ns, $\alpha = 190$ μs and $\beta = 10^{-4}$ μs/byte (for 10 Gigabit Ethernet running 802.3ae), and $\alpha = 5700$ μs and $\beta = .016$ μs/byte (for a 15000 RPM Seagate ST373307 disk). In both cases α exceeds β by at least five orders of magnitude, and f is much smaller again. Patterson [78] suggests that this gap between latency and bandwidth will continue to increase exponentially for a variety of technologies (memory, network and disk).

2.6.1 Parallel Algorithms

Combining this notation with the analysis leading to Table 2.3, we get that the running time for the conventional algorithm to compute $[Ax, ..., A^k x]$ is

$$T_{PA0}(k) = O(\alpha k + \beta b k \gamma^{d-1} + f b^d k \gamma^d) \tag{2.1}$$

and that the running time for either new algorithm is

$$T_{PA1,2}(k) = O(\alpha + \beta bk(\gamma^{d-1} + \delta_d bk\gamma^{d-2}) + fb^d k(\gamma^d + bk\gamma^{d-1})) \tag{2.2}$$

where $\delta_d = 0$ if $d = 1$ and $\delta_d = 1$ if $d > 1$. We use this notation in order to analyze all values of d at once.

The ultimate goal is to compute $[Ax, ..., A^{\bar{k}}x]$. The conventional algorithm will take time $T_{PA0}(\bar{k})$. We will use the new algorithm $\frac{\bar{k}}{k}$ times on chunks of size k, taking time $\frac{\bar{k}}{k}T_{PA1,2}(k)$. The optimization problem is to choose k to minimize this quantity:

$$\frac{\bar{k}}{k}T_{PA1,2}(k) = O(\alpha\frac{\bar{k}}{k} + \beta b\bar{k}(\gamma^{d-1} + \delta_d bk\gamma^{d-2}) + fb^d\bar{k}(\gamma^d + bk\gamma^{d-1})) \tag{2.3}$$

When α is sufficiently large (suppose we are doing message passing by the post office), and so latency dominates all the bandwidth and flop terms, it is clearly best to minimize the number of messages in the new algorithm, i.e., to set $k = \bar{k}$, leading to a speedup of $O(\bar{k})$. In other words, $[Ax, ..., A^{\bar{k}}x]$ can be computed in approximately the same time as Ax (or within a constant factor of this time, since constants are hidden by our use of $O()$).

If α is not this large, then choosing the best k is more interesting. The dominant bandwidth and flop terms in (2.3) (i.e., those proportional to β and f and with the highest powers of γ) are identical to those in $T_{PA0}(\bar{k})$, and dependent only on $\bar{k}$. The latency term in (2.3) decreases proportionally to k, and the smaller bandwidth and flop terms increase proportionally to k. The minimizing value of k is easily found to be

$$k_{\min} = \min\left(k, \max\left(1, \left(\frac{f}{\alpha}b^{d+1}\gamma^{d-1} + \delta_d\frac{\beta}{\alpha}b^2\gamma^{d-2}\right)^{-1/2}\right)\right) \tag{2.4}$$

Notice that $k_{\min}$ increases with increasing latency α, and decreases with increasing problem size per processor γ. For $k_{\min}$ to exceed 1 requires both that $\alpha > fb^{d+1}\gamma^{d-1}$, roughly that α exceeds the floating point work on the boundary, and that $\alpha > \delta_d\beta b^2\gamma^{d-2}$, which is likely.

The minimum running time with the new algorithm (assuming $1 < k_{\min} < k$) is therefore

$$
\begin{aligned}
T_{PA1,2,min}(\bar{k}) &\equiv \frac{\bar{k}}{k_{\min}}T_{PA1,2}(k_{\min}) \\
&= O(\bar{k}[(\alpha(fb^{d+1}\gamma^{d-1} + \delta_d\beta b^2\gamma^{d-2}))^{1/2} + fb^d\gamma^d + \beta b\gamma^{d-1}])
\end{aligned} \tag{2.5}
$$

When $k_{\min} = 1$, the new algorithm and conventional algorithm are equivalent; when $k_{\min} = k$, the time is given by (2.2).

When α is very large, the speedup is close to k as expected. When α is not that large, The best that we could hope for is for the new running time to be fast independent of the latency α. More precisely, we ask whether $T_{PA1,2,min}(\bar{k})$ is within a constant factor of the time it would take the conventional algorithm with $\alpha = 0$. In fact, when $\alpha = 0$ both $T_{PA0}(\bar{k})$ and $T_{PA1,2,min}(\bar{k})$ are $O(\bar{k}(fb^d\gamma^d + \beta b\gamma^{d-1}))$, so we need to ask whether the first term in $T_{PA1,2,min}(\bar{k})$ is smaller than this:

$$\text{when is} \quad (\alpha(fb^{d+1}\gamma^{d-1} + \delta_d\beta b^2\gamma^{d-2}))^{1/2} \le fb^d\gamma^d + \beta b\gamma^{d-1} \quad ?$$

We consider the bandwidth and flop terms separately.

For bandwidth (which is only relevant when $d > 1$), the question is when $(\alpha\beta b^2\gamma^{d-2})^{1/2} \leq \beta b\gamma^{d-1}$, or when $\alpha \leq \beta\gamma^d$. This is easy to interpret: it holds when the time is takes to send the entire local content of a processor $\gamma^d = \frac{n^d}{p}$ is dominated by the bandwidth $\beta\frac{n^d}{p}$, not the latency. Once problem sizes are reasonably large, this is sure to be the case.

For the flop time, the question is when $(\alpha f b^{d+1}\gamma^{d-1})^{1/2} \leq f b^d\gamma^d$, or when $\alpha \leq f b^{d-1}\gamma^{d+1} = f b^{d-1}\left(\frac{n^d}{p}\right)^{1+\frac{1}{d}}$. This is also easy to interpret: it holds when the time it takes to run an $O(N^{1+\frac{1}{d}})$ algorithm on the entire local content of a processor (i.e., $N = \frac{n^d}{p}$) exceeds the latency of one message. When $d = 1$, this means an $O(N^2)$ algorithm, and is very likely to hold for large problem sizes.

In summary, the new algorithm can be used to make the cost of computing any number of matrix-vector products run at a speed that is roughly independent of the communication latency, provided $\alpha \stackrel{<}{\approx} \min(\beta\gamma^{d-1}, f b^{d-1}\gamma^{d+1})$ when $d > 1$, or $\alpha \stackrel{<}{\approx} f\gamma^2$ when $d = 1$. The limiting factor in achieving this will be the need for enough memory to store $k_{\min}$ vectors locally, since $k_{\min}$ grows as latency increases.

It is worth asking when just overlapping communication and computation in the conventional algorithm is good enough to hide all the latency, making our techniques unnecessary. This happens roughly when $\alpha < f b^d\gamma^d$, which is more restrictive than our condition $\alpha < f b^{d-1}\gamma^{d+1}$.

2.6.2   Sequential Algorithms

The corresponding asymptotic performance models for sequential algorithms are

$$
\begin{aligned}
T_{SA0}(k) &= O(\alpha kp + \beta b^d kn^d + f b^d kn^d) \\
T_{SA1}(k) &= O(\alpha p + \beta(b^d n^d + b^{d+1} kn^{d-1}p^{1/d}) + f b^d kn^d) \\
T_{SA2}(k) &= O(\alpha p + \beta((b^d + k)n^d + b^{d+1}kn^{d-1}p^{1/d}) + f(b^d kn^d + b^{d+1}k^2 n^{d-1}p^{1/d}))
\end{aligned}
$$

Since SA0 and SA1 use roughly the same amount of memory, we can compare their running times directly, and see that SA1 sends k-times fewer messages, sends roughly k-times fewer words, and does only slightly more floating point operations than SA0. So we expect SA1 to be uniformly superior to SA0.

SA2 is designed to use much less memory than either SA0 or SA1, and so a fair comparison is between SA2 and the conventional algorithm consisting of applying SA2(1) k times, where SA2(1) just computes Ax. The running time for this algorithm is easily seen to be $O(k\alpha p + k\beta(b^d n^d + b^{d+1}n^{d-1}p^{1/d}) + kf(b^d n^d + b^{d+1}n^{d-1}p^{1/d}))$, which has a k times larger latency term, up to k times larger bandwidth term (when $b^d \gg k$), and almost the same floating point term. Clearly overlapping communication and computation will benefit SA2(k) at least as much as SA2(1).

Another natural question is to ask under what circumstances SA2 is about as fast as a conventional algorithm with an infinite amount of fast memory available, but where the

matrix and $x^{(0)}$ initially reside in slow memory, and the result is eventually supposed to reside in slow memory, an algorithm we call SA3:

$$T_{SA3}(k) = O(\alpha + \beta(b^d + k)n^d + fb^d k n^d)$$

Comparing $T_{SA2}(k)$ to $T_{SA3}(k)$ we see that the bandwidth and floating point costs of SA2 are only slightly larger than for SA3, so the only issue is latency, which is p-times lower for SA3. So a natural question is when SA2's latency cost is less than or equal to its bandwidth and floating point cost. This will be true when the cost of filling up all of fast memory with one fast memory access, and then performing the algorithm on the subset of matrix and vectors filling all of fast memory, is dominated by bandwidth and floating point, which is very likely to be true.

We now turn to the question of optimal speedup for SA2. It can be seen that the optimal speedup for SA2 for general sparse matrices (when $\beta/t_f = \infty$) is upper bounded by $2 + 1.5(nnz/n)$ (nnz/n denotes then number of nonzeros per row of the matrix A). The 1.5 term is due to the 1.5 words per entry of the A. A simple argument for this is that the speedup is roughly

$$\frac{\text{Time to read/write vector and matrix } k \text{ times}}{\text{Time to read 1 vector / write } k \text{ vectors and read matrix once}} = \frac{2k + 1.5(nnz/n)k}{k + 1 + 1.5(nnz/n)}$$
$$< 2 + 1.5(nnz/n)$$

A similar argument gives us the following upper bound for optimal speedup in the case when β/t_f is finite:

$$\frac{(2(nnz/n) - 1)\, k + (2k + 1.5(nnz/n)k)\frac{\beta}{t_f}}{(2(nnz/n) - 1)\, k + (k + 1 + 1.5(nnz/n))\frac{\beta}{t_f}} < \frac{2(nnz/n) - 1 + (2 + 1.5(nnz/n))\frac{\beta}{t_f}}{2(nnz/n) - 1 + \frac{\beta}{t_f}.}$$

Note that $2(nnz/n) - 1$ is the number of flops performed per entry of the vector x. These upper bounds are tight for large memory size. However, they might not be close to the optimal speedup for small memory size. To analyze this, we now state some bounds for the optimal speedup of SA2 for d-dimensional stencils with bandwidth b. The following are some scenarios for a d-dimensional stencil with bandwidth b.

1. $m = \infty$, $\frac{\beta}{t_f} = \infty$: In this case, the optimal speedup is the same as the upper bound, i.e., $2 + 1.5(2b + 1)^d$.

2. $m = \infty$, $\frac{\beta}{t_f}$ finite: The optimal speedup again equals its upper bound

$$\frac{2(2b+1)^d - 1 + \left(2 + 1.5(2b+1)^d\right)\frac{\beta}{t_f}}{2(2b+1)^d - 1 + \frac{\beta}{t_f}}$$

3. m finite, $\frac{\beta}{t_f} = \infty$: In this case, the optimal speedup is lower bounded by

$$\max\left(\left(2 + 1.5(2b+1)^d\right)\left(1 + \frac{A}{m^{\frac{1}{d+1}}}\right)^{-\left(1 + \frac{1}{d}\right)}, 1\right),$$

where A is a constant[8] which depends on b and d. As can be seen, the lower bound approaches the upper bound as m is increased. For this case, the optimal speedup is obtained by choosing k such that $2bkd = n$.

4. m finite, $\frac{\beta}{t_f}$ finite: The optimal speedup is lower bounded by

$$\max\left(\frac{2(2b+1)^d - 1 + \left(2 + 1.5(2b+1)^d\right)\frac{\beta}{t_f}}{(2(2b+1)^d - 1)B + \left(1 + \frac{A}{m^{\frac{1}{d+1}}}\right)^{\left(1+\frac{1}{d}\right)}\frac{\beta}{t_f}}, 1\right),$$

where A is the same constant as in the previous case and B is another constant[9] dependent on d. Figures 2.15 and 2.16 show the optimal speedup (obtained from the analytical model) and the lower bounds for the speedup as function of the memory size. Figures 2.15(a) and 2.15(b) show the speedups if the β/t_f ratio is 64 (the OOC machine model in Section 2.7.2). Figures 2.15(a) and 2.15(b) show the speedups if the β/t_f ratio is 3.2 (the CacheBlocked machine model in Section 2.7.2). As can be seen in these figures, if the memory is not sufficiently large, the optimal speedup drops as the bandwidth b is increased. However, for large enough memory, we observe the speedup increase as the bandwidth b is increased. Another observation is that the lower bound is within a factor of 2 of the optimal speedup for the β/t_f ratios and the memory sizes considered. Furthermore, for the optimal speedup, the k is such that $2bkd \leq n$ (the computational cost of SA2 always increases with k, so it can only decrease the optimal k predicted by the previous case).

The above model makes some approximations which are good if the optimal k and n are large enough. Given this, the optimal speedup predicted is still close to (but lower than) the speedup in the more detailed performance model discussed in Section 2.7.2. Except for the CacheBlocked model ($\beta/t_f = 3.2$, $mem = 10^6$, has an optimal $k = 3$ which is small) in Section 2.7.2, the optimal speedups in the analytical model (the plots in Figure 2.15) match the speedups in the detailed model very well. As for the measured speedup in Section 6, the analytical performance model overestimates the optimal speedup. The main reason for this is that the read and write bandwidths differ significantly, which is not handled by the analytical model.

2.7 Detailed Performance Modeling

In this section we present detailed performance models of matrices with 2D and 2D stencil graphs for PA2 and SA2 using realistic machine parameters, in order to identify situations where significant speedups are likely. The two parallel machines for which we model PA2 are called Peta (which is a model of a nominal 8100 processor petascale machine) and Grid
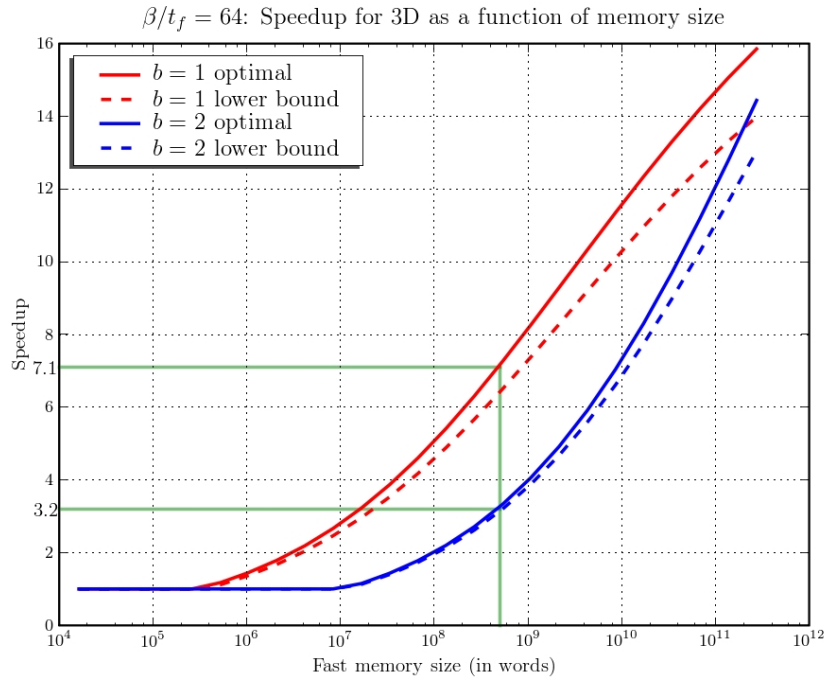
---

[8] $A = \frac{1.5(d+1)^d(2b)^{d/(d+1)}(1+(2b+1)^d)}{d^{\frac{d2}{d+1}}}$.

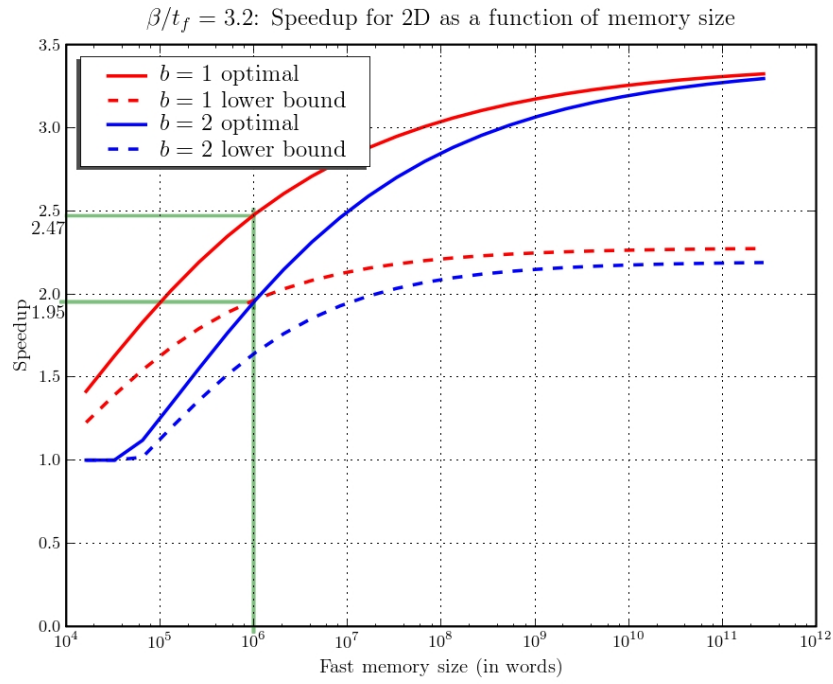[9] $B = \frac{(d+1)^d}{d^d} - \frac{d}{d+1}$.
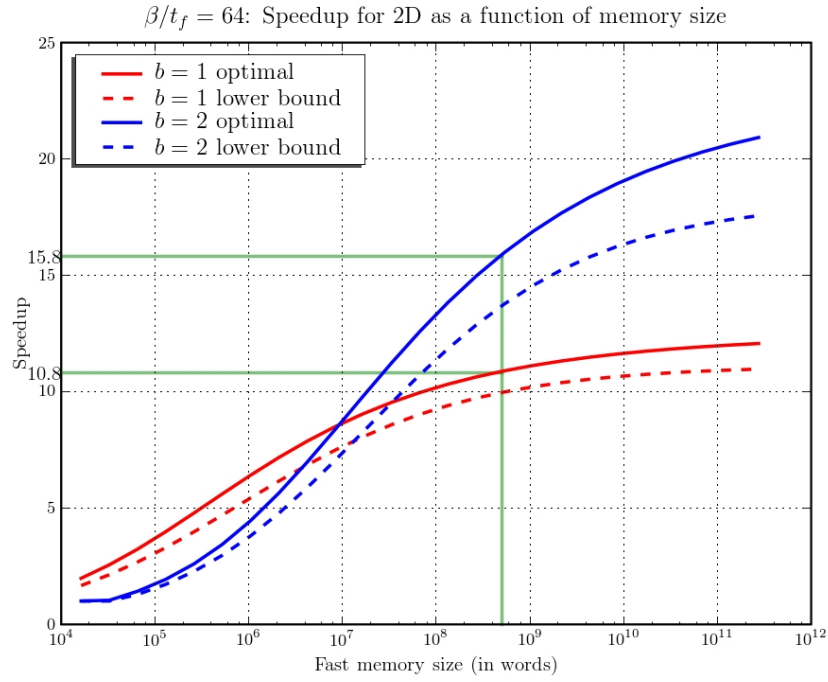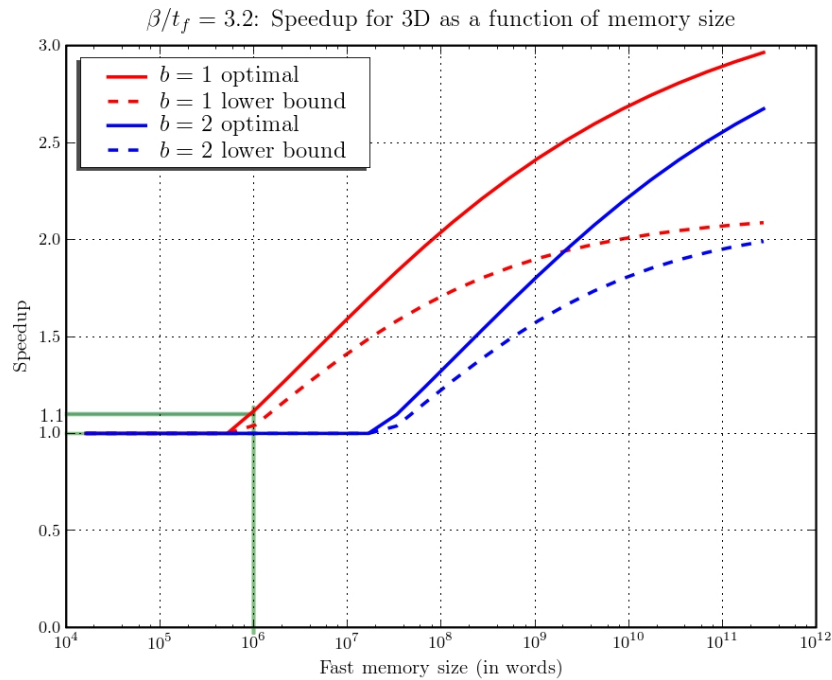
(a) Optimal speedups and lower bounds for 2D stencil

(b) Optimal speedups and lower bounds for 3D stencil

Figure 2.15: Optimal speedup vs. DRAM size for the OOC model ($\beta/t_f = 64$). The vertical green lines indicate the memory size for the architectures modeled in Section 2.7.2.

(a) Optimal speedups and lower bounds for 2D stencil.

(b) Optimal speedups and lower bounds for 3D stencil.

Figure 2.16: Optimal speedup vs. cache size for the CacheBlocked model ($\beta/t_f = 3.2$). The vertical green lines indicate the memory size for the architectures modeled in Section 2.7.2.

(which is a model of 125 terascale machines connected over the internet). We consider both overlapping (non-blocking) and non-overlapping (blocking) communication models; only the former can overlap communication and computation. The two sequential machines for which we model SA2 are OOC (which models an out-of-core implementation, where fast memory is DRAM and slow memory is disk) and CacheBlocked, the Intel multi-core processor (where fast memory is cache and slow memory is DRAM). This variety of models of course suggests that our techniques can be applied more than once, if there are several levels of memory hierarchy and possibly also parallelism.

Specifically, we consider matrices whose graphs are 2D $(2b+1)^2$ point and 3D $(2b+1)^3$ point stencils. As before, we assume that quantities like $p^{1/2}$ and $\frac{n}{p^{1/3}}$ are integers.

2.7.1 Performance Modeling of PA2

We consider parallel machines with the following parameters:

p_{max}: The maximum number of processors available. The actual number of processors used is $p \le p_{max}$. We may choose $p < p_{max}$ if that is faster, or if $p_{max}^{1/2}$ is not an integer, etc.

t_f: The time per floating-point operation (in units of seconds), modeled as 10% of machine peak value, a typical value attainable for SpMV.

mem: The memory available per processor (in units of 8-byte words).

α: The network processor latency (in units of seconds).

β: The inverse network bandwidth (in units of seconds/8-byte word).

Thus the time to send m words between any pair of processors is modeled as $\alpha + \beta m$ seconds.

We modeled machines with the following parameter values:

Peta: $p_{max} = 8100$, $t_f = 2 \cdot 10^{-11}$ secs ($1/t_f = 50$ GFlops/s), $mem = 62.5 \cdot 10^9$ words, $\alpha = 10^{-5}$ secs, $\beta = 2 \cdot 10^{-9}$ secs ($1/\beta = 500$ MWords/s = 4 GByte/s)

Grid: $p_{max} = 125$, $t_f = 10^{-12}$ secs ($1/t_f = 1 TFlop/s$), $mem = 1.2 \cdot 10^{12}$ words, $\alpha = 10^{-1}$ secs, $\beta = 25 \cdot 10^{-9}$ secs ($1/\beta = 40$ MWords/s = .32 GBytes/s) (estimated by dividing the Teragrid backbone bandwidth of 40 GBytes/s by p_{max})

Note that each processor in Peta and Grid is assumed to be a significant parallel computer itself, but we are only modeling the parallelism between these processors, not within them. Again, one could potentially apply our techniques for each level of parallelism, but we have not modeled this here.

In section 2.4.4 we described the three computational phases of PA2: Phase I must be done before any communication can be initiated, Phase II can be fully overlapped with communication, and Phase III can only begin after communication is complete. This justifies the performance model for the case of overlapping communication below.

Let N_I, N_{II}, and N_{III} respectively denote the flop counts for Phases I, II, and III of PA2. Let N_w denote the total number of words sent by a processor. Let $T_{n,k,p}^{overlap}$ denote the time taken for PA2 when overlapping communication is used; in this case we assume all messages can be in-flight simultaneously while computation is occurring. Let $T_{n,k,p}^{nonoverlap}$ denote the time taken for PA2 when non-overlapping communication is used; in this case we assume only one message can be in flight at a time and not overlapped with computation. Our latency-avoiding algorithm may have more opportunity to demonstrate speedups in the non-overlapping case. In an actual machine the degree of overlap may lie somewhere between these two extremes. Let $M_{n,k,p}$ denote the memory required per processor when p processors are used. We let $T_{n,k,p}$ denote the time taken for the algorithm. So, if non-overlapping communication is used, then $T_{n,k,p} = T_{n,k,p}^{nonoverlap}$, else $T_{n,k,p} = T_{n,k,p}^{overlap}$.

We use the following formulas for these quantities (which are slightly more detailed than the entries in Table 2.3):

$$N_I = (8b^2 + 8b + 1) \cdot \left(\frac{n}{p^{1/2}} - bk \right) \cdot (bk^2 - 2bk)$$

$$N_{II} = (8b^2 + 8b + 1) \cdot \left(\frac{3n^2}{p} - \frac{9bkn}{p^{1/2}} + 7b^2k^2 + 2b^2 \right) \cdot k/3$$

$$N_{III} = (8b^2 + 8b + 1) \cdot bk \cdot \left(\frac{9nk}{p^{1/2}} + \frac{6n}{p^{1/2}} - bk^2 - 6bk - 8b \right) /3$$

$$N_w = 2bk \cdot \left(\frac{2n}{p^{1/2}} + 3kb - 2b \right)$$

$$T_{n,k,p}^{overlap} = (N_I + N_{III}) \cdot t_f + \max \left(N_{II} \cdot t_f, \alpha + \beta \cdot N_w \right)$$

$$T_{n,k,p}^{nonoverlap} = (N_I + N_{II} + N_{III}) \cdot t_f + 8 \cdot \alpha + \beta \cdot N_w$$

$$M_{n,k,p} = (k+1)\frac{n^2}{p} + 1.5(2b+1)^2 \left(\frac{n}{p^{1/2}} + bk \right)^2 + 1.5N_w$$

Note that the coefficient for the α term is 8 in $T_{n,k,p}^{nonoverlap}$ because each processor needs to communicate with 8 other processors. However, the coefficient for the α term is 1 in $T_{n,k,p}^{overlap}$ because all 8 sends to other processors can be overlapped.

Similarly, for the 3D stencils, we have the following formulas:

$$N_I = (2(2b+1)^3 - 1) \cdot \frac{(bk^2 - 2bk)}{4} \cdot \left(\frac{6n^2}{p^{2/3}} - \frac{12bkn}{p^{1/3}} + 7b^2k^2 - 2b^2k \right)$$

$$N_{II} = (2(2b+1)^3 - 1) \cdot \frac{k}{4} \cdot \left(\frac{4n^3}{p} - \frac{18bkn^2}{p^{2/3}} + \frac{(28b^2k^2 + 8b^2)n}{p^{1/3}} + O(b^3k^3) \right)$$

$$N_{III} = (2(2b+1)^3 - 1) \cdot \frac{bk}{4} \cdot \left(\frac{n^2}{p^{2/3}}(18k + 12) - \frac{nb}{p^{1/3}}(4k^2 + 24k + 32) + O(b^2k^3) \right)$$

$$N_w = 2bk \cdot \left(\frac{3n^2}{p^{2/3}} + \frac{9bkn}{p^{1/3}} - \frac{6bn}{p^{1/3}} + O(b^2k^2) \right)$$

$$T_{n,k,p}^{overlap} = (N_I + N_{III}) \cdot t_f + \max\left(N_{II} \cdot t_f, \alpha + \beta \cdot N_w \right)$$

$$T_{n,k,p}^{nonoverlap} = (N_I + N_{II} + N_{III}) \cdot t_f + 26 \cdot \alpha + \beta \cdot N_w$$

$$M_{n,k,p} = (k+1)\frac{n^3}{p} + 1.5(2b+1)^3 \left(\frac{n}{p^{1/3}} + bk \right)^3 + 1.5N_w$$

For performance modeling, we also vary the parameter p within the range allowed to find the optimal value of p for specific n, k, b values. This range is limited by two parameters—p_{max} and mem. If p is made small, then the memory required per processor $M_{n,k,p}$ might exceed the memory available per processor mem. Furthermore, p can be at most p_{max}. Another limit we impose on p is the number of entries per dimension of the stencil should be be at least $2bk$ since our operation counts assume this condition. We may also round p down to the nearest perfect square or cube. The optimal p is strongly problem dependent, e.g., for small problem sizes, $p = 1$ might be sufficient and better since it avoids the overhead of communication. Therefore, a good measure of how well PA2 performs with respect to the conventional algorithm is the speedup with respect to the conventional algorithm assuming optimal p values were used for each algorithm:

$$speedup = \frac{\min_{1 \leq p \leq p_{max}} T_{n,1,p} \cdot k}{\min_{1 \leq p \leq p_{max}} T_{n,k,p}}$$

Note that we used $T_{n,1,p} \cdot k$ for the time taken for the conventional algorithm as the conventional algorithm turns out to be k invocations of PA2 with $k = 1$.

Another interesting metric for evaluating the algorithm is how well it can hide the communication cost. Specifically, we compare the time due to PA2 on a machine with the time if the same machine had zero communication overhead. If both the times are close, then our algorithm is latency and bandwidth insensitive, i.e., it can make the cost of communication disappear. So, we also plot this ratio in our plots. In addition, we also look at the additional floating-point operations performed by PA2 to hide latency.

We now discuss the performance modeling results for each combination of machine (Peta or Grid), communication style (overlapping or non-overlapping) and stencil (2D or 3D). There are 8 plots shown for each of these 8 combinations:

The first 4 plots of each group of 8 assume a bandwidth of $b = 1$, and respectively show for each combination of n and k (a) the best speedup attainable over all choices of

$p \leq p_{max}$, (b) the corresponding optimal choice of p, (c) the corresponding fraction of time spent in computation, and (d) the ratio of floating point operations done by the optimized algorithm to the number done by the conventional algorithm. The conventional algorithm corresponds to $k = 1$, the bottom row of each plot. Plots (c) and (d) show how successful our new algorithm is at reducing the fraction of time spent communicating, and the price paid in extra computation.

Now we describe the next 4 plots of each group of 8. For each combination of n and k, and for bandwidth $b = 1$, we show (a) the ratio of time taken by the new algorithm to the time that would be taken on the same machine but with zero latency, and (b) the ratio of time taken by the new algorithm to the time that would be taken on the same machine but with zero latency and infinite bandwidth. We also show (c) for a fixed value of n, and each combination of k and bandwidth b, the best speedup attainable over all choices of $p \leq p_{max}$, and (d) the corresponding optimal choice of p. In plot (a) (or (b)) the time that would be taken with zero latency (or zero latency and infinite bandwidth, resp.) is measured for the same k but a possibly different optimal value of p. Plots (a) and (b) provide another metric of how well our algorithm does at reducing the cost of communication (the closer the ratios are to 1, the better).

2D Stencil on Peta Using Overlapping Communication

As can be seen in Figure 2.17(a), for smaller n and k (the bottom left corner), the speedup is close to linear in k, which is the best possible speedup. This is explained by Figure 2.17(c) which shows that almost all the time is spent in communication for these values of n and k.

The best speedup is $6.9\times$, which occurs when $n = 2^{11}$, $k = 12$, with an optimal $p = 7225 = 85^2$, a bit less than $p_{max} = 8100 = 90^2$; for these values of n, k and p the fraction of time spent in computation is 23% (versus 2% for the conventional $k = 1$ algorithm) and the number of floating point computations is $1.74\times$ larger than the conventional algorithm (Figure 2.17(d)).

Indeed, for any problem size n, we can choose k to make the fraction of time spent doing arithmetic exceed 20% (up from under 1% for $k = 1$). And this never increases the number of floating point operations by more than $1.74\times$.

On the other hand, the algorithm has no benefit for large values of n, because computation totally dominates communication, as again shown by the bottom row of Figure 2.17(c); in this case no optimization is necessary either. We also note that for smaller n the speedup decreases as k is increased beyond a certain point, because the overhead of extra floating point operations exceeds the gains from reducing latency. The optimal p also decreases as k increases for some values of n because of the constraint $n/\sqrt{p} \geq 2bk$ we impose to guarantee that boundary regions only extend into the nearest neighboring processor.

As can be seen in Figure 2.18(a), the latency has an enormous effect for small n and k, with the conventional algorithm ($k = 1$) running up to $89.39\times$ slower than the 0-latency machine. But the algorithm can lower the latency to equal the floating point time even with small values of k (e.g., for $n = 2^{11}$, $k = 1$ we see that the conventional algorithm is $46.53\times$ slower than the case if latency were 0, but raising k to 12 makes PA2 only $6.78\times$ slower

than the 0 latency case). For large n reducing latency to zero yields no speedups because computation dominates. Figure 2.18(b) tells a similar story as Figure 2.18(a), except that additionally increasing bandwidth to infinity would speed up the conventional code even more. In Figure 2.18(c) we see that for $n = 2^{12}$ the speedup due to PA2 decreases as b increases: this is because computation scales as b^2 which rapidly dominates communication.

2D Stencil on Peta Using Non-Overlapping Communication

As can be seen in Figure 2.19(a), the algorithm is expected to obtain high speedups of up to 15.1× for smaller matrices. In fact, we get good speedups even for $n = 2^{14}$ in contrast to the case when overlapping communication was used. This is because non-overlapping communication has 8× higher latency than overlapping communication, making our latency-avoiding algorithm even more valuable. As in the overlapping case, for sufficiently large n computation dominates communication and there is no benefit from our algorithm. Figure 2.19(c) shows lower values when compared to Figure 2.17(c) because of the 8× larger latency. Figure 2.19(d) shows the same ratios of extra arithmetic as for the overlapping communication case, because the same optimal values of p are chosen (Figure 2.19(b)).

The comparisons to zero latency (Figure 2.20(a)) and zero latency/infinite bandwidth (Figure 2.20(b)) machines are even more extreme than in the overlapping case, because of the 8× higher latency assumed here. This also causes our algorithm to yield at least some speedup (1.28×) all the up to bandwidth $b = 10$ (Figure 2.20(c)).

2D Stencil on Grid Using Overlapping Communication

The white region in all the figures for $n = 2^{21}$ and $n = 2^{22}$ indicates that the problem needed too much memory to be solved by the machine.

As can be seen in Figure 2.21(a), the algorithm is expected to obtain an impressive speedup of up to 22.22× for large matrices ($n = 2^{17}$). Indeed, speedup is still increasing for the maximum value of k shown ($k = 30$), and larger k might show further improvements. The algorithm does not show any speedups for small values of n because the problem can be solved using only 1 processor and latency is too high to benefit from using more processors for $k \leq 30$. As before we can see that for very large problem sizes ($n \geq 2^{20}$), we also see no gains from our algorithm because computation dominates communication. In Figure 2.21(b) the optimal p takes on two values—either 1 or $p_{max} = 121$. Figure 2.21(c) shows the fraction of time in computation increasing from 2% at $k = 1$ to 53% at $k = 30$ for the value of $n = 2^{17}$ where speedup is best, but for smaller n and $k = 30$ the fraction of computation is still quite small; larger k might help. Figure 2.21(d) shows that in no case does the algorithm do more than 1.02× as many flops as the conventional algorithm.

As can be seen in Figure 2.22(a), the ratio of time taken compared to the zero latency machine roughly doubles for each value of $\log_2 n$ from 10 to 16. The reason is that in this range, the optimal p for the Grid is 1, so all time is spent in computation, and for the 0 latency machine the optimal p is 121 and the largest part of the time is the bandwidth term. Thus doubling n quadruples the time on the Grid, but only doubles the time on the 0 latency machine, causing the ratio of these times to double. For larger n, computation

begins to dominate both machines. It is at $n = 2^{16}$ and $n = 2^{17}$ that increasing k yields the largest speedup. We note that many of the ratios in Figure 2.22(b) equal 121, because they correspond to cases where the optimal $p = 121$ for the 0 latency / infinite bandwidth machine, and $p = 1$ with nonzero latency and finite bandwidth.

Figure 2.22(c) shows that there is at least some speedup for $n = 2^{17}$ and all values of bandwidth b up to 10, although speedup decreases as b increases.

2D Stencil on Grid Using Non-overlapping Communication

As can be seen in Figure 2.23(a), the algorithm is expected to obtain speedups of up to 15.63× for large matrices, with speedup still increasing at the largest k shown. The speedups are sometimes larger and sometimes smaller than the overlapping case, depending on dimension. The extra expense of non-overlapping communication means that $p = 1$ is optimal for larger values of n than in the overlapping case (Figure 2.23(b)). The number of extra floating point operations never reaches 1% (Figure 2.23(d)).

In Figure 2.24(b), many of the runtime ratios equal 121, because they correspond to cases where the optimal p was 1 for the actual Grid, whereas the optimal $p = 121$ for the 0 latency / infinite bandwidth Grid.

Figure 2.24(c) shows that there is speedup for all values of bandwidth b, but it is not a monotonic decreasing function of b, rather peaking (at least for $n = 2^{17}$) at 16.67× for $b = 3$ and $k = 30$.

3D Stencil on Peta Using Overlapping Communication

In contrast to the 2D case, in the 3D case no speedup is possible using our new algorithm (with the exception of a 2% speedup for $n = 2^9$ and $k = 2$). The reason is that the conventional $k = 1$ algorithm is already completely dominated by computation (Figure 2.25(c)), and indeed running nearly as fast as a zero latency machine (Figure 2.26(a)) or even a zero latency / infinite bandwidth machine (Figure 2.26(b)). Increasing the bandwidth b (Figure 2.26(c)) only makes it more computation dominated. We also note that the for larger b, the problem quickly becomes too large to be solved by the machine—this is evident by the large "whitespaces" in Figures 2.26(c) and 2.26(d).

3D Stencil on Peta Using Non-overlapping Communication

In contrast to the last case with overlapping communication, PA2 achieves a speedup of up to 3.58×, for $n = 2^9$ and $k = 8$, running only 3.04× slower than a zero latency machine (down from 10.89×) and only 4.50× slower than a zero latency / infinite bandwidth machine (down from 16.10×).

3D Stencil on Grid Using Overlapping Communication

In this case we get speedups of up to 4.41× for $n = 2^{10}$ and $k = 30$, doing only 1.29× as much arithmetic as the conventional algorithm, and running only 2.03× slower than a zero latency machine (down from 8.94×). However, it is 28.35× slower than a zero latency

/ infinite bandwidth machine, showing that bandwidth is the bottleneck. Some speedups are possible up to bandwidth $b = 5$.

3D Stencil on Grid Using Non-overlapping Communication

Not overlapping communication on the Grid yields a higher speedup of $7.79\times$ for $n = 2^{12}$ and $k = 30$, running only $1.76\times$ slower than a 0 latency machine (down from $13.73\times$), and only $7.87\times$ slower than a 0 latency / infinite bandwidth machine (down from $61.26\times$). Speedups up to $7.04\times$ are possible for higher bandwidth b.

(a) Speedup

(b) Optimal p

(c) Fraction of time in computation

(d) Ratio of additional flops
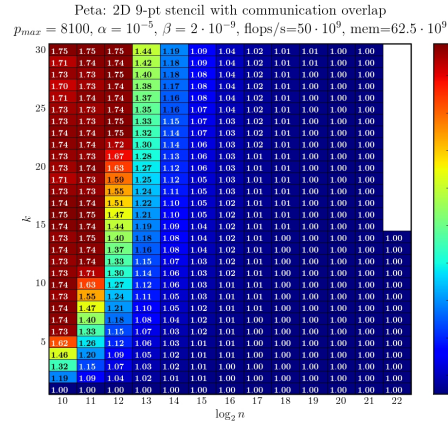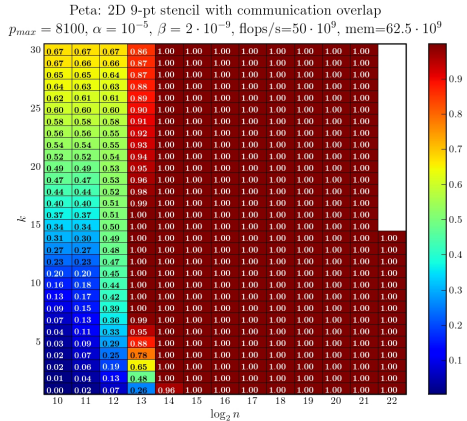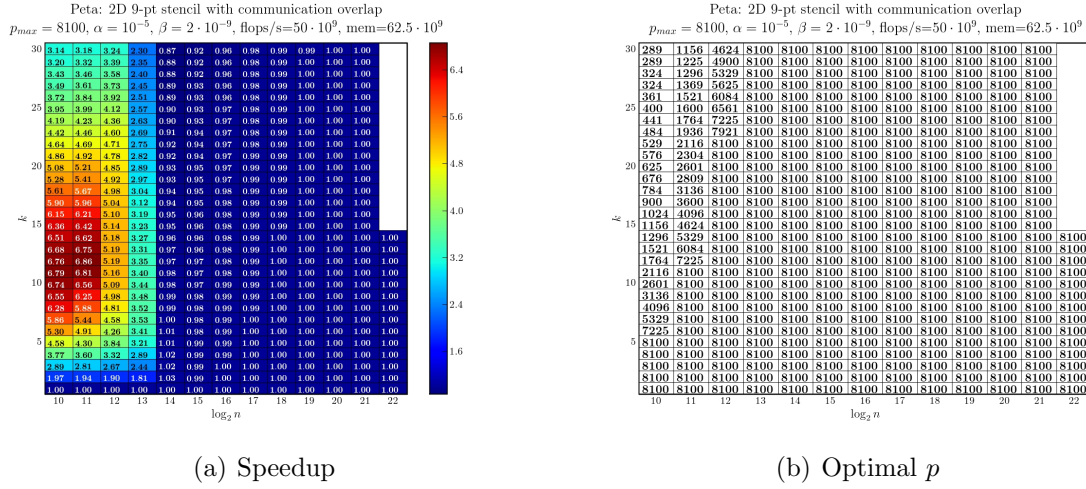
Figure 2.17: Plots for 2D stencil on Peta using overlapping communication. Best speedup of $6.9\times$ attained at $p = 7225 = 85^2, k = 12, n = 2^{11}$ For each n, the best k makes the fraction of time in computation $\geq 20\%$, up from $< 1\%$, and increases the number of flops by $\leq 1.74\times$.

Peta: 2D 9-pt stencil with communication overlap
$p_{max} = 8100$, $\alpha = 10^{-5}$, $\beta = 2 \cdot 10^{-9}$, flops/s=$50 \cdot 10^9$, mem=$62.5 \cdot 10^9$

(a) Ratio of time w.r.t. 0 latency machine

Peta: 2D 9-pt stencil with communication overlap
$p_{max} = 8100$, $\alpha = 10^{-5}$, $\beta = 2 \cdot 10^{-9}$, flops/s=$50 \cdot 10^9$, mem=$62.5 \cdot 10^9$

(b) Ratio of time w.r.t. 0 latency, ∞ bandwidth machine

Peta: 2D $(2b+1)^2$-pt stencil with communication overlap
$p_{max} = 8100$, $\alpha = 10^{-5}$, $\beta = 2 \cdot 10^{-9}$, flops/s=$50 \cdot 10^9$, mem=$62.5 \cdot 10^9$, $n = 2^{12}$

(c) Speedup as a function of matrix bandwidth ($n = 2^{12}$)

Peta: 2D $(2b+1)^2$-pt stencil with communication overlap
$p_{max} = 8100$, $\alpha = 10^{-5}$, $\beta = 2 \cdot 10^{-9}$, flops/s=$50 \cdot 10^9$, mem=$62.5 \cdot 10^9$, $n = 2^{12}$
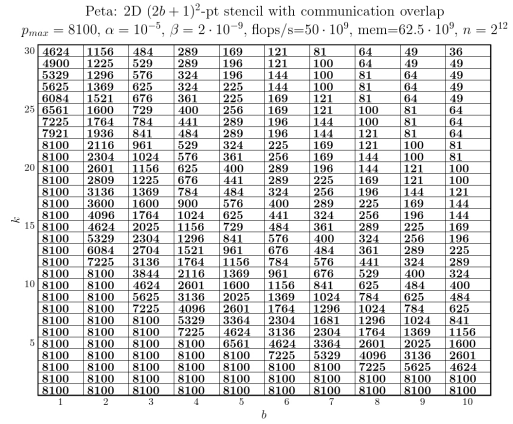
(d) Optimal p for (c)

Figure 2.18: Plots for 2D stencil on Peta using overlapping communication. For each n, the best k makes the runtime ratio w.r.t. 0 latency ≤ 13.16, down from 89.39 and, makes the runtime ratio w.r.t. 0 latency/∞ BW ≤ 33.82, down from 229.77.
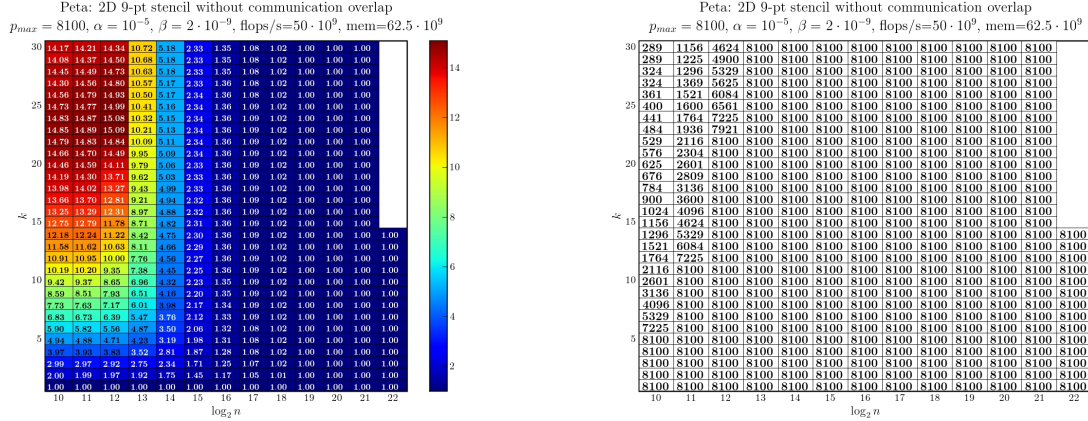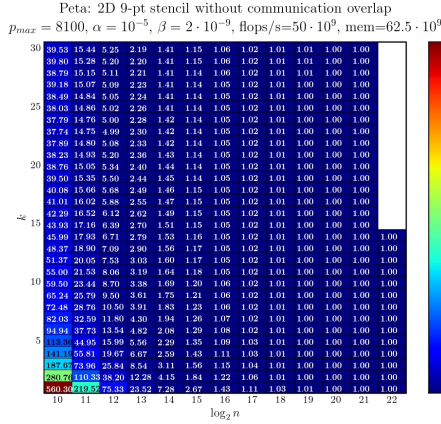
(a) Speedup

(b) Optimal p
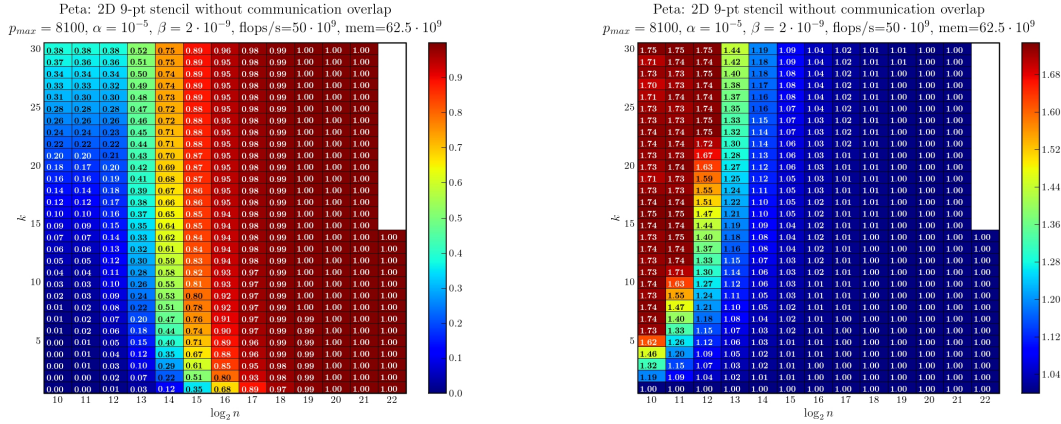
(c) Fraction of time in computation

(d) Ratio of additional flops

Figure 2.19: Plots for 2D stencil on Peta using non-overlapping communication. Best speedup of $15.09\times$ attained at $p = 7921 = 89^2, k = 23, n = 2^{12}$ For each n, the best k makes the fraction of time in computation $\geq 23\%$, up from $< 1\%$, and increases the number of flops by $\leq 1.75\times$
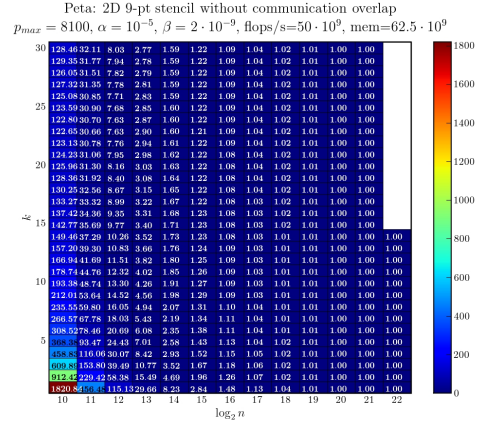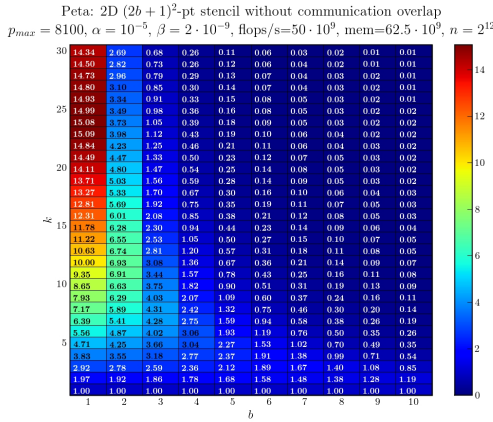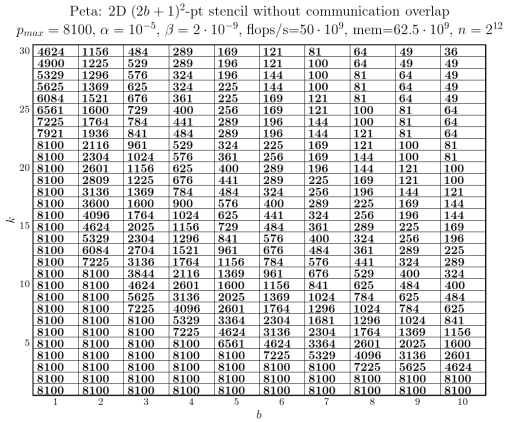
(a) Ratio of time w.r.t. 0 latency machine

(b) Ratio of time w.r.t. 0 latency, ∞ bandwidth machine

(c) Speedup as a function of matrix bandwidth ($n = 2^{12}$)

(d) Optimal p for (c)

Figure 2.20: Plots for 2D stencil on Peta using non-overlapping communication. For each n, the best k makes the runtime ratio w.r.t. 0 latency ≤ 37.71, down from 560.30 and, makes the runtime ratio w.r.t. 0 latency/∞ BW ≤ 122.65, down from 1829.84.

65

(a) Speedup

(b) Optimal p

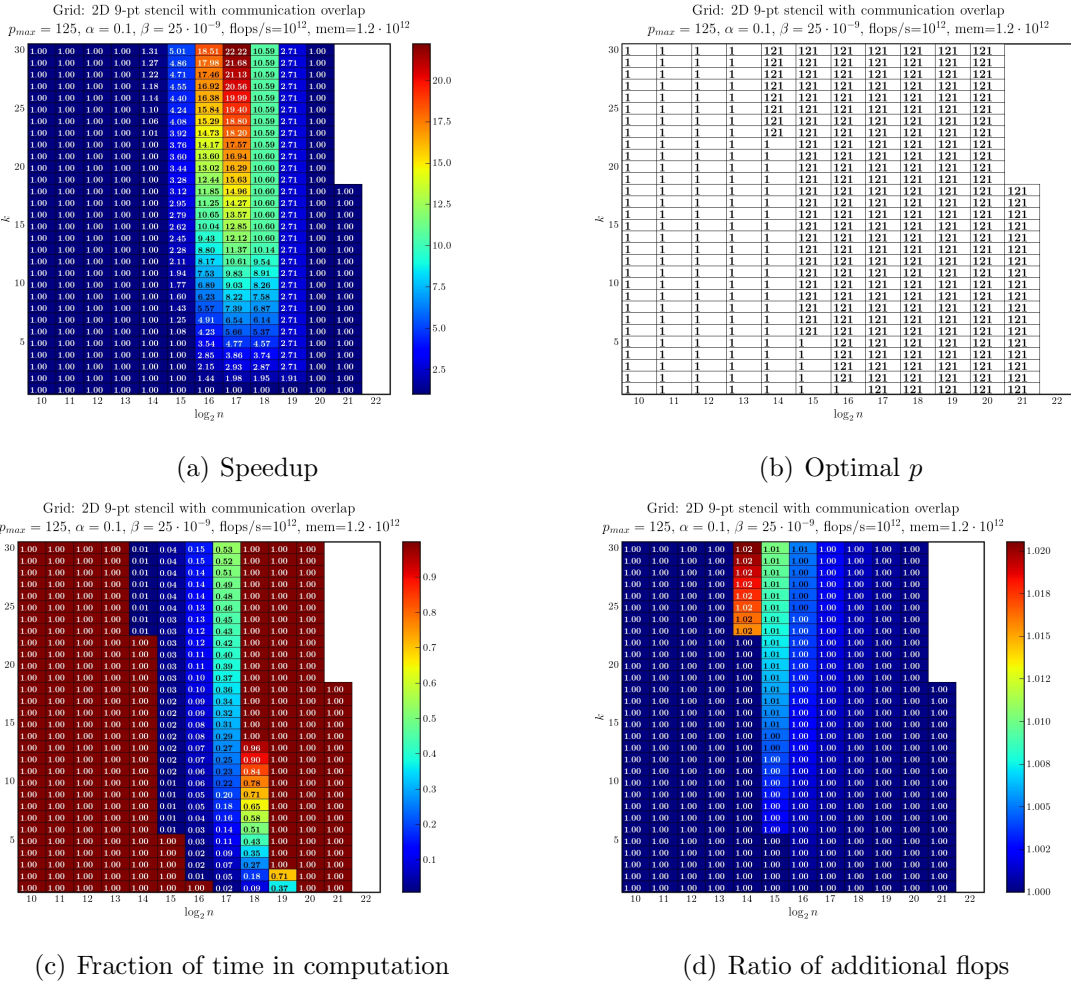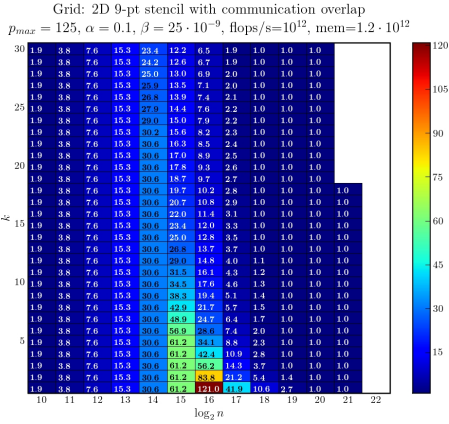(c) Fraction of time in computation

(d) Ratio of additional flops

Figure 2.21: Plots for 2D stencil on Grid using overlapping communication. Best speedup of $22.22\times$ attained at $p = 121, k = 30, n = 2^{17}$. For each n, the best k makes the fraction of time in computation $\geq 1\%$ and increases the number of flops by $\leq 1.02\times$
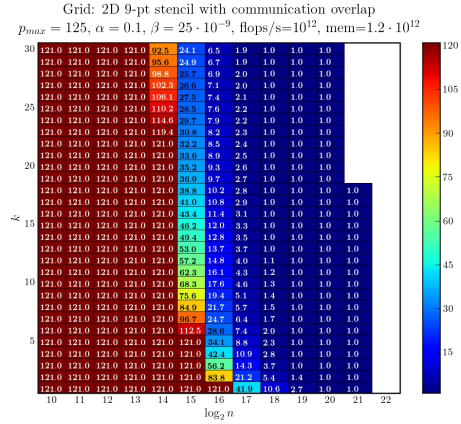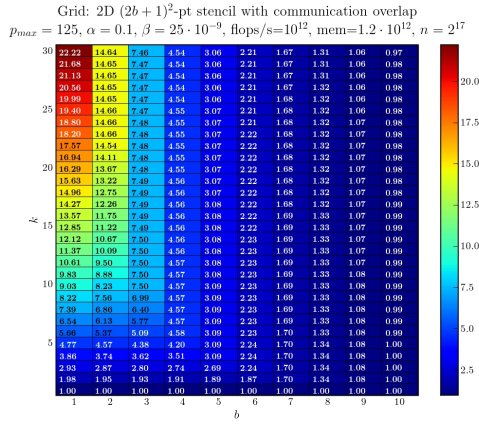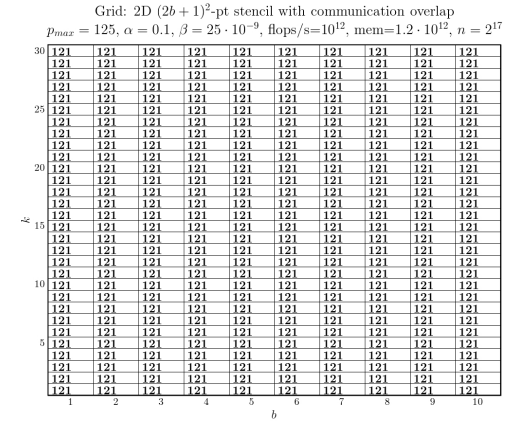
(a) Ratio of time w.r.t. 0 latency machine

(b) Ratio of time w.r.t. 0 latency, ∞ bandwidth machine

(c) Speedup as a function of matrix bandwidth ($n = 2^{17}$)

(d) Optimal p for (c)

Figure 2.22: Plots for 2D stencil on Grid using overlapping communication. For each n, the best k makes the runtime ratio w.r.t. 0 latency ≤ 23.4, down from 121, and makes the runtime ratio w.r.t. 0 latency/∞ BW ≤ 92.5, down from 121.

(a) Speedup

(b) Optimal p

(c) Fraction of time in computation
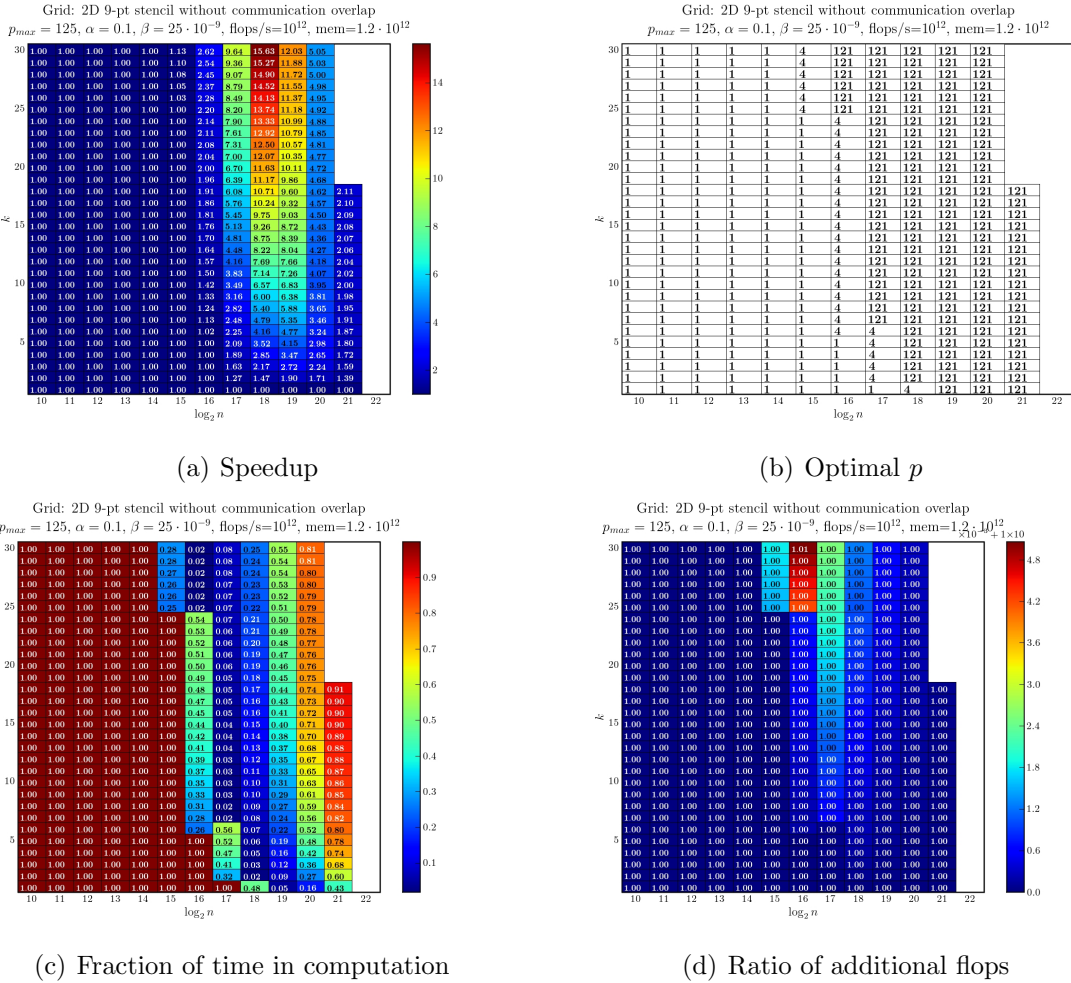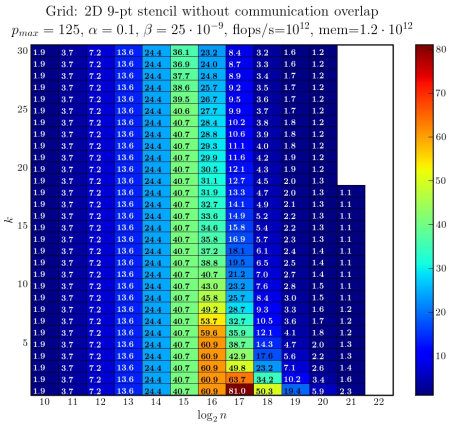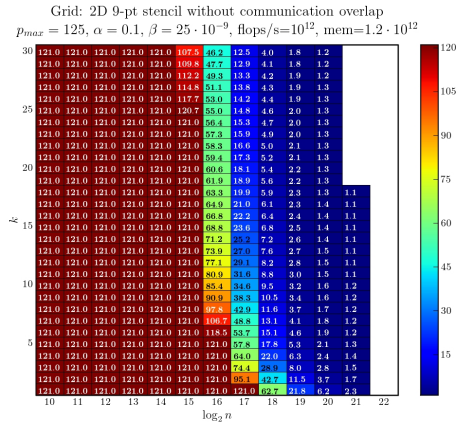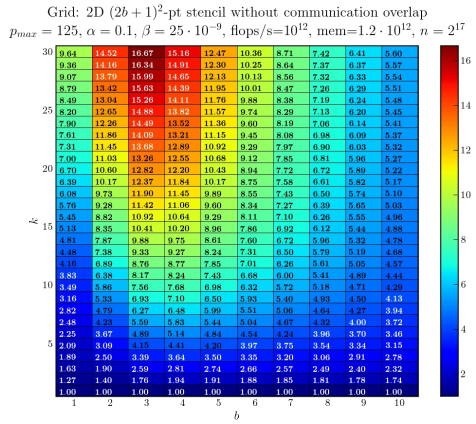
(d) Ratio of additional flops

Figure 2.23: Plots for 2D stencil on Grid using non-overlapping communication. Best speedup of $15.63\times$ attained at $p = 121, k = 30, n = 2^{18}$. For each n, the best k makes the fraction of time in computation $\geq 2\%$, and increases the number of flops by $\leq 1.01\times$
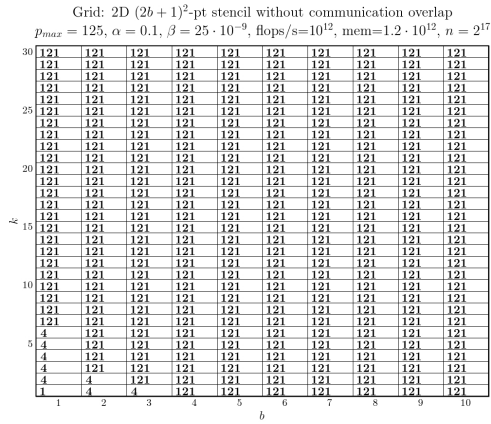
Grid: 2D 9-pt stencil without communication overlap
$p_{max} = 125$, $\alpha = 0.1$, $\beta = 25 \cdot 10^{-9}$, flops/s$=10^{12}$, mem$=1.2 \cdot 10^{12}$

(a) Ratio of time w.r.t. 0 latency machine

Grid: 2D 9-pt stencil without communication overlap
$p_{max} = 125$, $\alpha = 0.1$, $\beta = 25 \cdot 10^{-9}$, flops/s$=10^{12}$, mem$=1.2 \cdot 10^{12}$

(b) Ratio of time w.r.t. 0 latency, ∞ bandwidth machine

Grid: 2D $(2b+1)^2$-pt stencil without communication overlap
$p_{max} = 125$, $\alpha = 0.1$, $\beta = 25 \cdot 10^{-9}$, flops/s$=10^{12}$, mem$=1.2 \cdot 10^{12}$, $n = 2^{17}$

(c) Speedup as a function of matrix bandwidth ($n = 2^{17}$)

Grid: 2D $(2b+1)^2$-pt stencil without communication overlap
$p_{max} = 125$, $\alpha = 0.1$, $\beta = 25 \cdot 10^{-9}$, flops/s$=10^{12}$, mem$=1.2 \cdot 10^{12}$, $n = 2^{17}$

(d) Optimal p for (c)

Figure 2.24: Plots for 2D stencil on Grid using non-overlapping communication. For each n, the best k makes the runtime ratio w.r.t. 0 latency ≤ 36.1, down from 81.0.

Peta: 3D 27-pt stencil with communication overlap
$p_{max} = 8100,\ \alpha = 10^{-5},\ \beta = 2 \cdot 10^{-9},\ \text{flops/s} = 50 \cdot 10^9,\ \text{mem} = 62.5 \cdot 10^9$

(a) Speedup

Peta: 3D 27-pt stencil with communication overlap
$p_{max} = 8100,\ \alpha = 10^{-5},\ \beta = 2 \cdot 10^{-9},\ \text{flops/s} = 50 \cdot 10^9,\ \text{mem} = 62.5 \cdot 10^9$

(b) Optimal p

Peta: 3D 27-pt stencil with communication overlap
$p_{max} = 8100,\ \alpha = 10^{-5},\ \beta = 2 \cdot 10^{-9},\ \text{flops/s} = 50 \cdot 10^9,\ \text{mem} = 62.5 \cdot 10^9$

(c) Fraction of time in computation

Peta: 3D 27-pt stencil with communication overlap
$p_{max} = 8100,\ \alpha = 10^{-5},\ \beta = 2 \cdot 10^{-9},\ \text{flops/s} = 50 \cdot 10^9,\ \text{mem} = 62.5 \cdot 10^9$
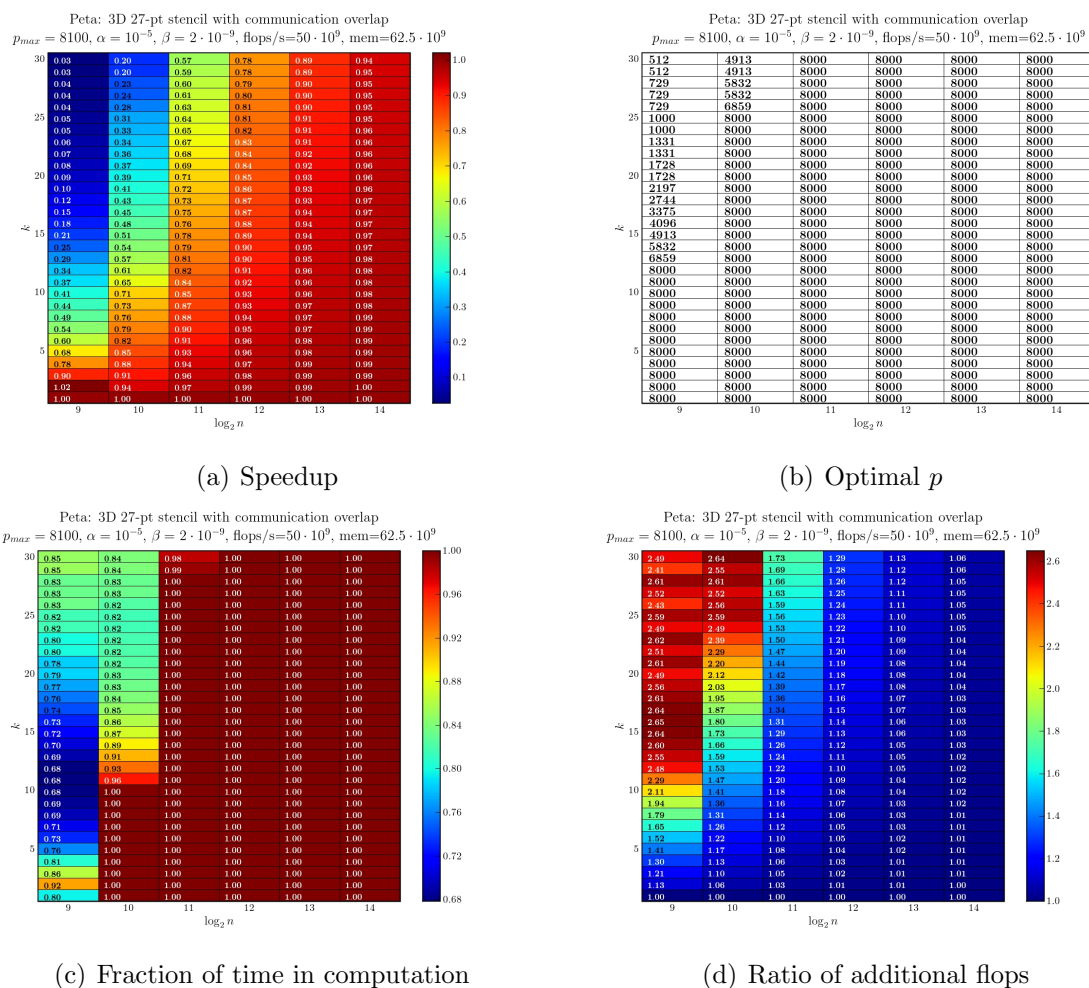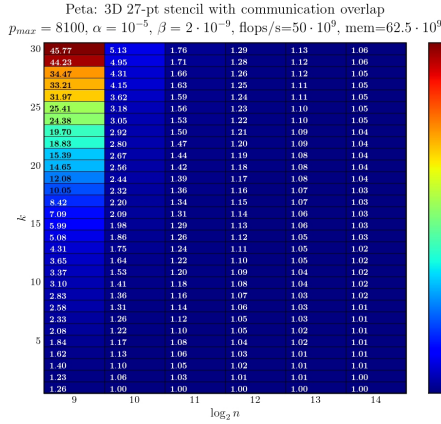
(d) Ratio of additional flops

Figure 2.25: Plots for 3D stencil on Peta using overlapping communication. Best speedup of $1.02\times$ attained at $p = 8000, k = 2, n = 2^9$. For each n, $k = 1$ makes the fraction of time in computation $\geq 80\%$

Peta: 3D 27-pt stencil with communication overlap
$p_{max} = 8100$, $\alpha = 10^{-5}$, $\beta = 2 \cdot 10^{-9}$, flops/s=$50 \cdot 10^9$, mem=$62.5 \cdot 10^9$

$\log_2 n$ = 9	10	11	12	13	14
45.77	5.13	1.76	1.29	1.13	1.06
44.23	4.95	1.71	1.28	1.12	1.06
34.47	4.31	1.66	1.26	1.12	1.05
33.21	4.15	1.63	1.25	1.11	1.05
31.97	3.62	1.59	1.24	1.11	1.05
25.41	3.18	1.56	1.23	1.10	1.05
24.38	3.05	1.53	1.22	1.10	1.05
19.70	2.92	1.50	1.21	1.09	1.04
18.83	2.80	1.47	1.20	1.09	1.04
15.39	2.67	1.44	1.19	1.08	1.04
14.65	2.56	1.42	1.18	1.08	1.04
12.08	2.44	1.39	1.17	1.08	1.04
10.05	2.32	1.36	1.16	1.07	1.03
8.42	2.20	1.34	1.15	1.07	1.03
7.09	2.09	1.31	1.14	1.06	1.03
5.99	1.98	1.29	1.13	1.06	1.03
5.08	1.86	1.26	1.12	1.05	1.03
4.31	1.75	1.24	1.11	1.05	1.02
3.65	1.64	1.22	1.10	1.05	1.02
3.37	1.53	1.20	1.09	1.04	1.02
3.10	1.41	1.18	1.08	1.04	1.02
2.83	1.36	1.16	1.07	1.03	1.02
2.58	1.31	1.14	1.06	1.03	1.01
2.33	1.26	1.12	1.05	1.03	1.01
2.08	1.22	1.10	1.05	1.02	1.01
1.84	1.17	1.08	1.04	1.02	1.01
1.62	1.13	1.06	1.03	1.01	1.01
1.40	1.10	1.05	1.02	1.01	1.01
1.23	1.06	1.03	1.01	1.01	1.00
1.26	1.00	1.00	1.00	1.00	1.00

(vertical axis k, from 5 to 30; color scale 5 to 45)

(a) Ratio of time w.r.t. 0 latency machine

Peta: 3D 27-pt stencil with communication overlap
$p_{max} = 8100$, $\alpha = 10^{-5}$, $\beta = 2 \cdot 10^{-9}$, flops/s=$50 \cdot 10^9$, mem=$62.5 \cdot 10^9$

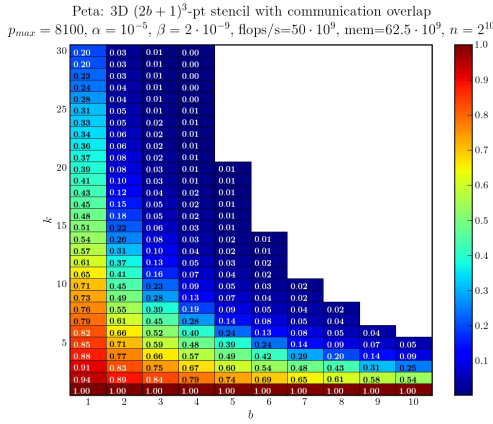| $\log_2 n$ = 9	10	11	12	13	14
45.77	5.13	1.76	1.29	1.13	1.06
44.23	4.95	1.71	1.28	1.12	1.06
34.47	4.31	1.66	1.26	1.12	1.05
33.21	4.15	1.63	1.25	1.11	1.05
31.97	3.62	1.59	1.24	1.11	1.05
25.41	3.18	1.56	1.23	1.10	1.05
24.38	3.05	1.53	1.22	1.10	1.05
19.70	2.92	1.50	1.21	1.09	1.04
18.83	2.80	1.47	1.20	1.09	1.04
15.39	2.67	1.44	1.19	1.08	1.04
14.65	2.56	1.42	1.18	1.08	1.04
12.08	2.44	1.39	1.17	1.08	1.04
10.05	2.32	1.36	1.16	1.07	1.03
8.42	2.20	1.34	1.15	1.07	1.03
7.09	2.09	1.31	1.14	1.06	1.03
5.99	1.98	1.29	1.13	1.06	1.03
5.08	1.86	1.26	1.12	1.05	1.03
4.31	1.75	1.24	1.11	1.05	1.02
3.65	1.64	1.22	1.10	1.05	1.02
3.37	1.53	1.20	1.09	1.04	1.02
3.10	1.41	1.18	1.08	1.04	1.02
2.83	1.36	1.16	1.07	1.03	1.02
2.58	1.31	1.14	1.06	1.03	1.01
2.33	1.26	1.12	1.05	1.03	1.01
2.08	1.22	1.10	1.05	1.02	1.01
1.84	1.17	1.08	1.04	1.02	1.01
1.62	1.13	1.06	1.03	1.01	1.01
1.40	1.10	1.05	1.02	1.01	1.01
1.23	1.06	1.03	1.01	1.01	1.00
1.26	1.00	1.00	1.00	1.00	1.00

(b) Ratio of time w.r.t. 0 latency, ∞ bandwidth machine

Peta: 3D $(2b+1)^3$-pt stencil with communication overlap
$p_{max} = 8100$, $\alpha = 10^{-5}$, $\beta = 2 \cdot 10^{-9}$, flops/s=$50 \cdot 10^9$, mem=$62.5 \cdot 10^9$, $n = 2^{10}$

b=1	2	3	4	5	6	7	8	9	10
0.20	0.03	0.01	0.00						
0.20	0.03	0.01	0.00						
0.29	0.03	0.01	0.00						
0.24	0.04	0.01	0.00						
0.28	0.04	0.01	0.00						
0.31	0.05	0.01	0.01						
0.33	0.05	0.02	0.01						
0.34	0.06	0.02	0.01						
0.36	0.06	0.02	0.01						
0.37	0.08	0.02	0.01						
0.39	0.08	0.03	0.01	0.01					
0.41	0.10	0.03	0.01	0.01					
0.43	0.12	0.04	0.02	0.01					
0.45	0.15	0.05	0.02	0.01					
0.48	0.18	0.05	0.02	0.01					
0.51	0.22	0.06	0.03	0.01					
0.54	0.26	0.08	0.03	0.02	0.01				
0.57	0.31	0.10	0.04	0.02	0.01				
0.61	0.37	0.13	0.05	0.03	0.02				
0.65	0.41	0.16	0.07	0.04	0.02				
0.71	0.45	0.23	0.09	0.05	0.03	0.02			
0.73	0.49	0.28	0.13	0.07	0.04	0.02			
0.76	0.55	0.39	0.19	0.09	0.05	0.04	0.02		
0.79	0.61	0.45	0.28	0.14	0.08	0.05	0.04		
0.82	0.66	0.52	0.40	0.24	0.13	0.08	0.05	0.04	
0.85	0.71	0.59	0.48	0.39	0.24	0.14	0.09	0.07	0.05
0.88	0.77	0.66	0.57	0.49	0.42	0.29	0.20	0.14	0.09
0.91	0.83	0.75	0.67	0.60	0.54	0.48	0.43	0.51	0.25
0.94	0.89	0.84	0.79	0.74	0.69	0.65	0.61	0.58	0.54
1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00

(c) Speedup as a function of matrix bandwidth ($n = 2^{10}$)

Peta: 3D $(2b+1)^3$-pt stencil with communication overlap
$p_{max} = 8100$, $\alpha = 10^{-5}$, $\beta = 2 \cdot 10^{-9}$, flops/s=$50 \cdot 10^9$, mem=$62.5 \cdot 10^9$, $n = 2^{10}$

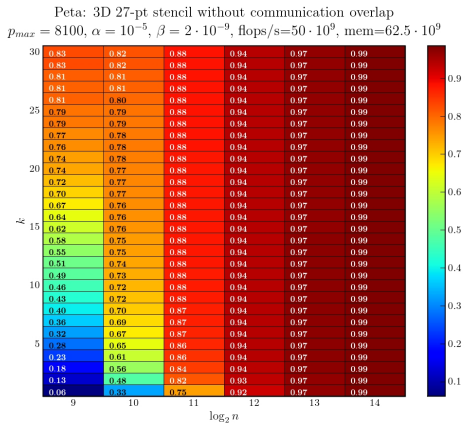| b=1	2	3	4	5	6	7	8	9	10
4913	512	125	64						
4913	512	125	64						
5832	729	216	64						
5832	729	216	64						
6859	729	216	64						
8000	1000	216	125						
8000	1000	343	125						
8000	1331	343	125						
8000	1331	343	125						
8000	1728	512	216						
8000	1728	512	216	125					
8000	2197	512	216	125					
8000	2744	729	343	125					
8000	3375	1000	512	216					
8000	4096	1000	512	216					
8000	4913	1331	512	216					
8000	5832	1728	729	343	216				
8000	6859	2197	729	343	216				
8000	8000	2744	1000	512	343				
8000	8000	3375	1331	729	343				
8000	8000	4913	1728	1000	512	343			
8000	8000	5832	2744	1331	729	512			
8000	8000	8000	4096	1728	1000	729	512		
8000	8000	8000	5832	2744	1728	1000	729		
8000	8000	8000	8000	4913	2744	1728	1000	729	
8000	8000	8000	8000	8000	4913	2744	1728	1331	1000
8000	8000	8000	8000	8000	8000	5832	4096	2744	1728
8000	8000	8000	8000	8000	8000	8000	8000	5832	4913
8000	8000	8000	8000	8000	8000	8000	8000	8000	8000
8000	8000	8000	8000	8000	8000	8000	8000	8000	8000

(d) Optimal p for (c)

Figure 2.26: Plots for 3D stencil on Peta using overlapping communication. For each n, $k = 1$ makes the runtime ratio w.r.t. 0 latency ≤ 1.26. For each n, $k = 1$ makes the runtime ratio w.r.t. 0 latency/∞ BW ≤ 1.26.
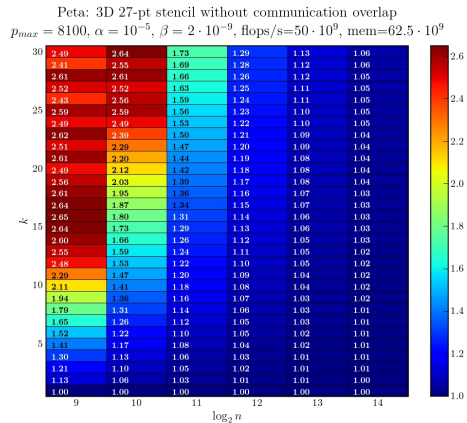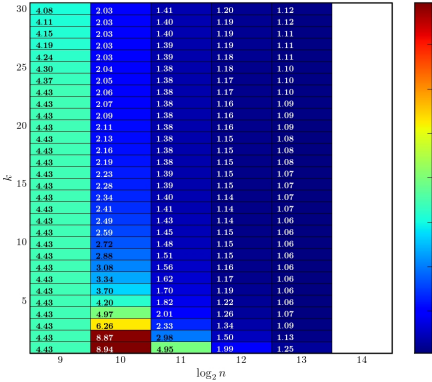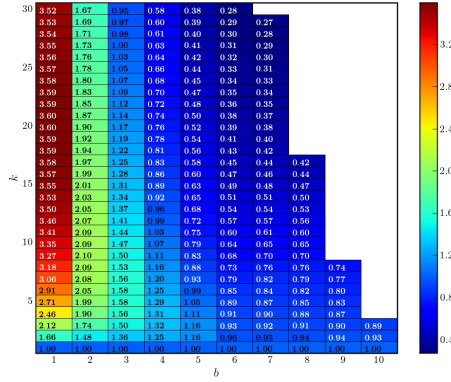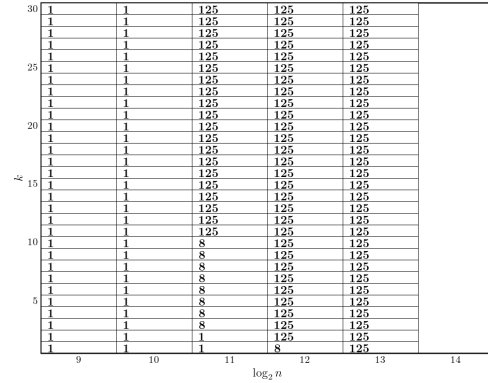
Peta: 3D 27-pt stencil without communication overlap
$p_{max} = 8100$, $\alpha = 10^{-5}$, $\beta = 2 \cdot 10^{-9}$, flops/s=$50 \cdot 10^9$, mem=$62.5 \cdot 10^9$

(a) Speedup

Peta: 3D 27-pt stencil without communication overlap
$p_{max} = 8100$, $\alpha = 10^{-5}$, $\beta = 2 \cdot 10^{-9}$, flops/s=$50 \cdot 10^9$, mem=$62.5 \cdot 10^9$

(b) Optimal p

Peta: 3D 27-pt stencil without communication overlap
$p_{max} = 8100$, $\alpha = 10^{-5}$, $\beta = 2 \cdot 10^{-9}$, flops/s=$50 \cdot 10^9$, mem=$62.5 \cdot 10^9$

(c) Fraction of time in computation

Peta: 3D 27-pt stencil without communication overlap
$p_{max} = 8100$, $\alpha = 10^{-5}$, $\beta = 2 \cdot 10^{-9}$, flops/s=$50 \cdot 10^9$, mem=$62.5 \cdot 10^9$

(d) Ratio of additional flops

Figure 2.27: Plots for 3D stencil on Peta using non-overlapping communication. Best speedup of $3.56\times$ attained at $p = 8000, k = 8, n = 2^9$. For each n, the best k makes the fraction of time in computation $\geq 40\%$, up from 6%, and increases the number of flops by $\leq 1.79\times$
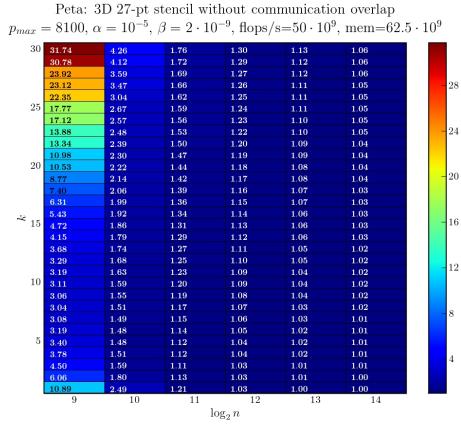
(a) Ratio of time w.r.t. 0 latency machine
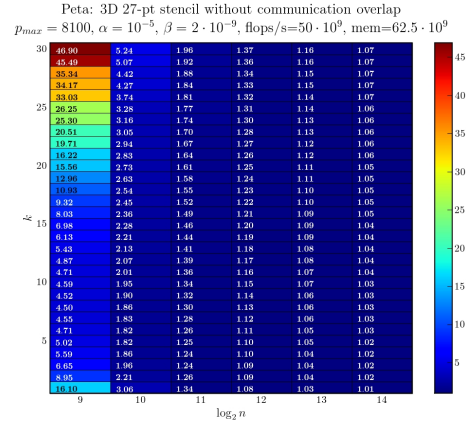
(b) Ratio of time w.r.t. 0 latency, ∞ bandwidth machine
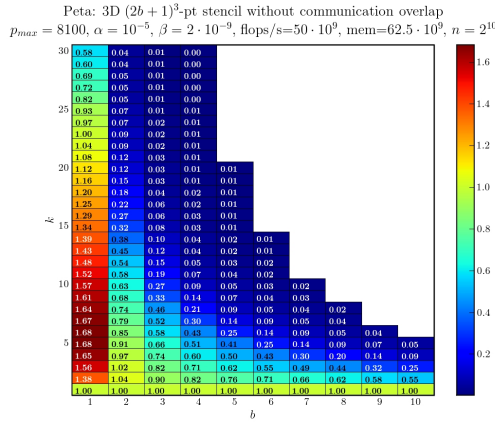
(c) Speedup as a function of matrix bandwidth ($n = 2^{10}$)
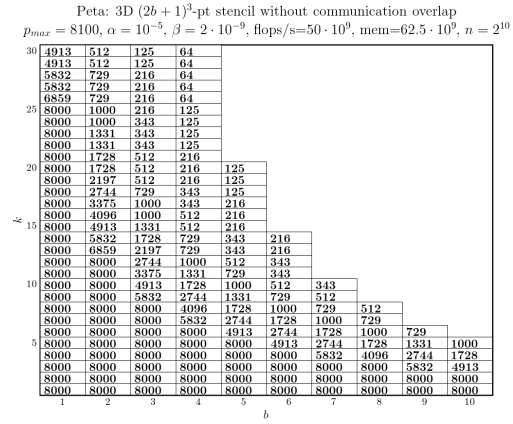
(d) Optimal p for (c)

Figure 2.28: Plots for 3D stencil on Peta using non-overlapping communication. For each n, the best k makes the runtime ratio w.r.t. 0 latency ≤ 3.04, down from 10.89. For each n, the best k makes the runtime ratio w.r.t. 0 latency/∞ BW ≤ 4.50, down from 16.10.
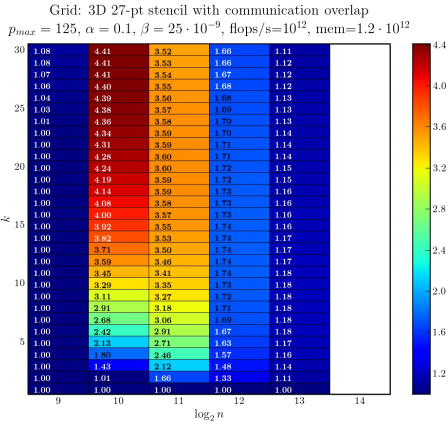
(a) Speedup
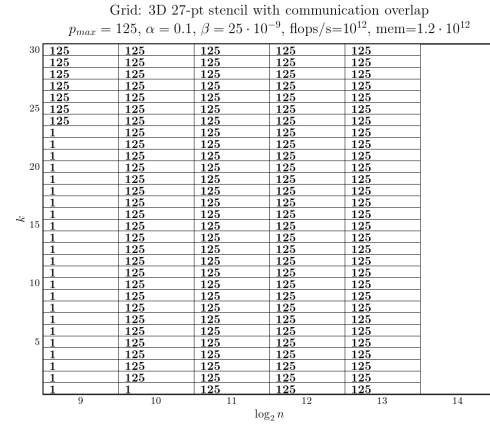
(b) Optimal p
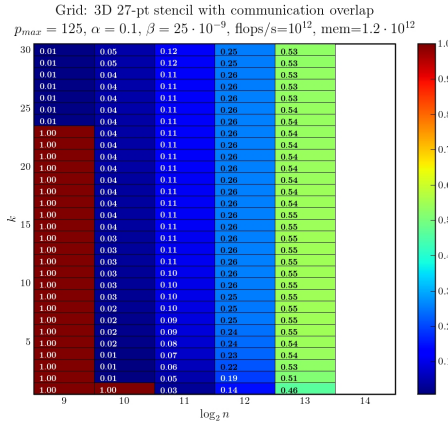
(c) Fraction of time in computation
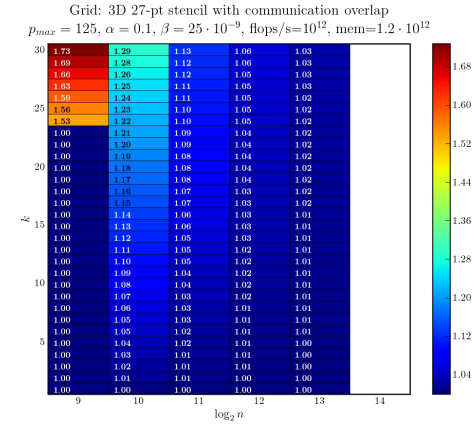
(d) Ratio of additional flops

Figure 2.29: Plots for 3D stencil on Grid using overlapping communication. Best speedup of $4.41\times$ attained at $p = 125, k = 30, n = 2^{10}$. For each n, the best k makes the fraction of time in computation $\geq 1\%$ and, increases the number of flops by $\leq 1.73\times$

Grid: 3D 27-pt stencil with communication overlap
$p_{max} = 125$, $\alpha = 0.1$, $\beta = 25 \cdot 10^{-9}$, flops/s=10^{12}, mem=$1.2 \cdot 10^{12}$

(a) Ratio of time w.r.t. 0 latency machine

Grid: 3D 27-pt stencil with communication overlap
$p_{max} = 125$, $\alpha = 0.1$, $\beta = 25 \cdot 10^{-9}$, flops/s=10^{12}, mem=$1.2 \cdot 10^{12}$
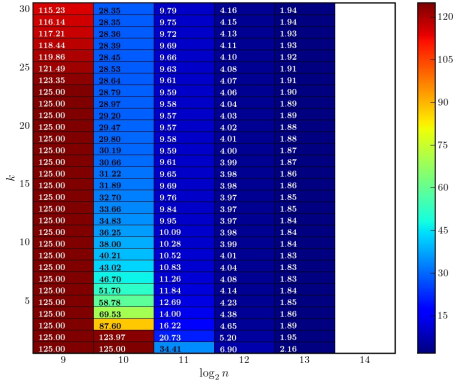
(b) Ratio of time w.r.t. 0 latency, ∞ bandwidth machine

Grid: 3D $(2b+1)^3$-pt stencil with communication overlap
$p_{max} = 125$, $\alpha = 0.1$, $\beta = 25 \cdot 10^{-9}$, flops/s=10^{12}, mem=$1.2 \cdot 10^{12}$, $n = 2^{11}$

(c) Speedup as a function of matrix bandwidth ($n = 2^{11}$)

Grid: 3D $(2b+1)^3$-pt stencil with communication overlap
$p_{max} = 125$, $\alpha = 0.1$, $\beta = 25 \cdot 10^{-9}$, flops/s=10^{12}, mem=$1.2 \cdot 10^{12}$, $n = 2^{11}$
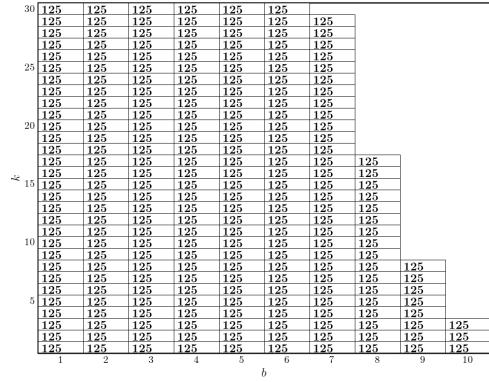
(d) Optimal p for (c)

Figure 2.30: Plots for 3D stencil on Grid using overlapping communication. For each n, the best k makes the runtime ratio w.r.t. 0 latency ≤ 4.08, down from 8.94. For each n, the best k makes the runtime ratio w.r.t. 0 latency/∞ BW ≤ 115.23, down from 125.
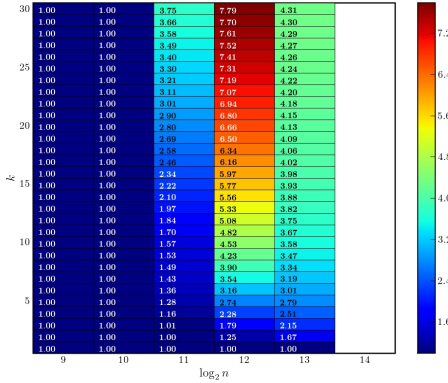
(a) Speedup

(b) Optimal p
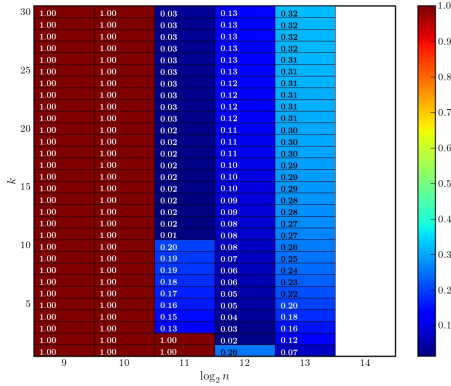
(c) Fraction of time in computation

(d) Ratio of additional flops

Figure 2.31: Plots for 3D stencil on Grid using non-overlapping communication. Best speedup of $7.79\times$ attained at $p = 125, k = 30, n = 2^{12}$. For each n, the best k makes the fraction of time in computation $\geq 3\%$, and increases the number of flops by $\leq 1.13\times$
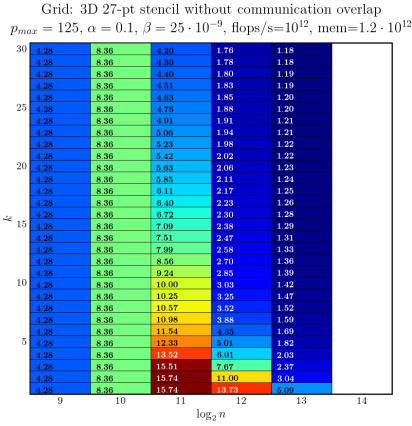
(a) Ratio of time w.r.t. 0 latency machine
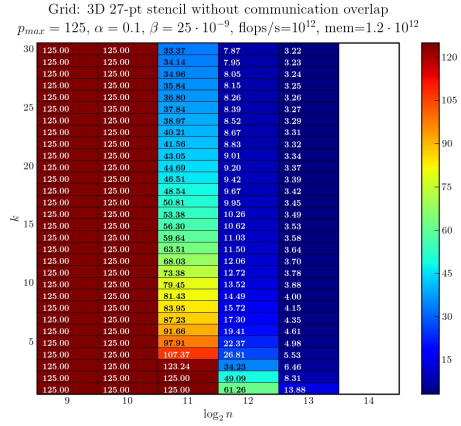
(b) Ratio of time w.r.t. 0 latency, ∞ bandwidth machine
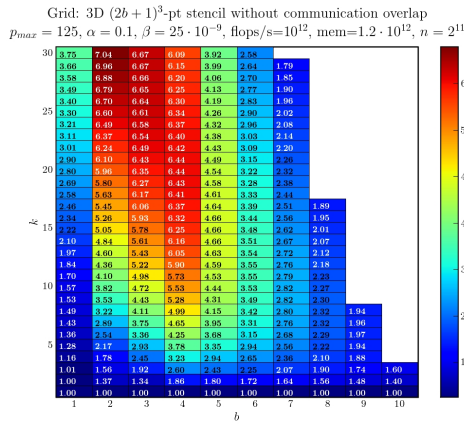
(c) Speedup as a function of matrix bandwidth ($n = 2^{11}$)
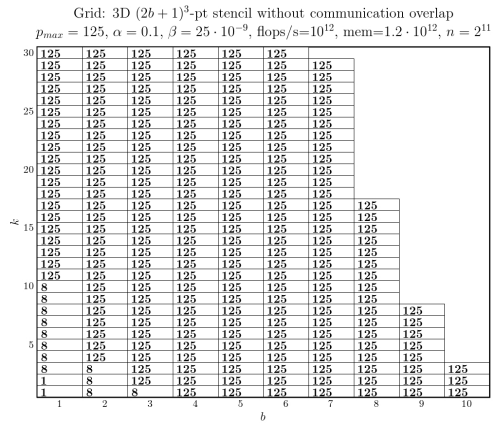
(d) Optimal p for (c)

Figure 2.32: Plots for 3D stencil on Grid using non-overlapping communication. For each n, the best k makes the runtime ratio w.r.t. 0 latency ≤ 8.36, down from 15.74.

2.7.2 Performance Modeling of SA2

We consider uniprocessor machines with the following parameters:

p_{max}: The maximum number of number of blocks to be used. The actual number of blocks used is $p \leq p_{max}$. We may choose $p < p_{max}$ if that is faster, or if $p_{max}^{1/2}$ is not an integer, etc. In our simulations we choose p_{max} extremely large, so that the optimal p is sure to satisfy $p < p_{max}$.

t_f: The time per floating-point operation (in units of seconds), modeled either as 10% of machine peak value (a typical value attainable for SpMV), or a median of measured values.

mem: The size of fast memory (in units of 8-byte words).

α: The slow memory latency (in units of seconds).

β: The inverse bandwidth available between slow and fast memory (in units of seconds/8-byte word).

Specifically, we model two machines with the following parameters:

OOC: This out-of-core implementation models a 500 MFlop/s uniprocessor with DRAM as fast memory and a 15000 RPM Seagate ST373307 disk as slow memory, with $p_{max} = 10^7$, $t_f = 2 \cdot 10^{-9}$ secs (500 MFlops/s), $mem = 5 \cdot 10^8$ words, $\alpha = 5.7 \cdot 10^{-3}$ secs, $\beta = 1.28 \cdot 10^{-7}$ secs ($1/\beta = 7.8$ MWords/s = 62.5 MB/s).

CacheBlocked: This multi-core implementation models a quad-core Intel Clovertown chip with on-chip cache as fast memory and DRAM as slow memory, with $p_{max} = 10^7$, $t_f = 5 \cdot 10^{-10}$ secs (2 GFlops/s) (based on measurements in [113]), $mem = 10^6$ words, $\alpha = 2 \cdot 10^{-7}$ secs, $\beta = 1.6 \cdot 10^{-9}$ secs ($1/\beta = 625$ MWords/s = 5 GB/s).

In our performance models below, we assume the entire matrix and vector x are initially stored in slow memory, and that $[Ax, A^2x, ..., A^kx]$ is eventually stored in slow memory at the end of the computation. Also, we only model non-overlapping communication and computation.

Let N denote the number of floating-point operations performed by SA2 (we sometimes write $N_{b,n,k,p}$ to indicate functional dependencies). Let N_a denote the number of slow memory accesses. Let N_w denote the number of words transferred between fast and slow memory. Let $T_{b,n,k,p,\alpha,\beta}$ denote the time taken by SA2. Let $M_{b,n,k,p}$ denote the main memory required. Formulas for these are given below.

Given the machine and problem parameters for 2D stencil matrices, we state the following formulas (which are slightly more detailed than the formulas in Table 2.4):

$$
\begin{aligned}
N_{b,n,k,p} &= (8b^2 + 8b + 1) \cdot \frac{k}{3} \cdot \\
&\quad \left(3n^2 + 6b(k-1)(p^{1/2}-1)n + 2b^2(2k-1)(k-1)(p^{1/2}-1)^2\right) \\
N_a &= 11p \\
N_w &= (k+1)n^2 + 1.5(2b+1)^2\left(n + 2b(k-1)(p^{1/2}-1)\right)^2 + \\
&\quad 6bk(p^{1/2}-1)(n + bk(p^{1/2}-1)) \\
M_{b,n,k,p} &= 1.5\left(\left(\frac{n}{p^{1/2}} + 2bk\right)^2 - \frac{n^2}{p}\right) + (k+1)\frac{n^2}{p} + 1.5(2b+1)^2\left(\frac{n}{p^{1/2}} + 2b(k-1)\right)^2 \\
T_{b,n,k,p,\alpha,\beta} &= N \cdot t_f + N_a \cdot \alpha + \beta \cdot N_w
\end{aligned}
$$

Similarly, we state the following formulas for 3D stencil (which are also slightly more detailed than in Table 2.4):

$$
\begin{aligned}
N_{b,n,k,p} &= (2(2b+1)^3 - 1) \cdot k\left(n^3 + 3b(k-1)(p^{1/3}-1)n^2 + O(b^2k^2np^{2/3})\right) \\
N_a &= 44p \\
N_w &= (k+1)n^3 + 1.5(2b+1)^3\left(n + 2b(k-1)(p^{1/3}-1)\right)^3 + \\
&\quad 1.5\left(\left(n + 2bk(p^{1/3}-1)\right)^3 - n^3\right) \\
M_{b,n,k,p} &= 1.5\left(\left(\frac{n}{p^{1/3}} + 2bk\right)^3 - \frac{n^3}{p}\right) + (k+1)\frac{n^3}{p} + 1.5(2b+1)^3\left(\frac{n}{p^{1/3}} + 2b(k-1)\right)^3 \\
T_{b,n,k,p,\alpha,\beta} &= N \cdot t_f + N_a \cdot \alpha + \beta \cdot N_w
\end{aligned}
$$

Given b, n, k, α and β, we want to choose p to minimize the run time $T_{b,n,k,p,\alpha,\beta}$, subject to the memory constraint $M_{b,n,k,p} < mem$; write this optimal runtime as

$$
T^{SA2,opt}_{b,n,k,\alpha,\beta,mem} = \min_{p:M_{b,n,k,p}<mem} T_{b,n,k,p,\alpha,\beta}
$$

with the p achieving the minimum written $p^{opt}_{b,n,k,\alpha,\beta,mem}$, and the corresponding number of arithmetic operations written $N^{SA2,opt}_{b,n,k,\alpha,\beta,mem} = N_{b,n,k,p^{opt}_{b,n,k,\alpha,\beta,mem}}$.

We present performance modeling data of SA2 for each combination of machine (OOC and CacheBlocked) and matrix (2D and 3D). We present 6 plots for each of these 4 combinations. These plots are slightly different from the ones shown for PA2, since we want to evaluate the savings in both latency and bandwidth costs. The first 5 plots are all for stencil bandwidth $b = 1$, and the last plot is for other bandwidths $b > 1$. Note that along vertical axis in each plot data may only be shown for odd values of k.

1. The speedup $k \cdot T^{SA2,opt}_{1,n,1,\alpha,\beta,mem} / T^{SA2,opt}_{1,n,k,\alpha,\beta,mem}$ of the new algorithm versus the conventional algorithm run k times (with $b = 1$). The conventional algorithm corresponds to $k = 1$, and so reads the matrix and a vector from slow to fast memory (at least) k times, and writes a vector from fast to slow memory (at least) k times.

2. The minimizing $p^{opt}_{1,n,k,\alpha,\beta,mem}$ for the new algorithm (with $b = 1$), in the denominator in the fraction in the bullet (1).

3. The fraction of time spent by the new algorithm in arithmetic (with $b = 1$); when this ratio is close to 1, it tells us that we are running close to the peak floating point speed.

4. The ratio of floating point operations done by the new algorithm to the number done by the conventional algorithm (with $b = 1$): $\frac{N^{SA2,opt}_{1,n,k,\alpha,\beta,mem}}{k \cdot N^{SA2,opt}_{1,n,1,\alpha,\beta,mem}}$. Note that the optimizing p is chosen independently for new algorithm and the conventional algorithm; this is true in later formulas as well. This ratio tells us how much redundant work is done by the new algorithm in order to achieve the best possible speedup.

5. The ratio $k \cdot T^{SA2,opt}_{1,n,1,0,0,mem}/T^{SA2,opt}_{1,n,k,\alpha,\beta,mem}$ of the time of the conventional algorithm run on a machine with zero latency and infinite bandwidth (to "slow" memory) to the time of the new algorithm. The time on a zero latency / infinite bandwidth machine is a lower bound on what the new algorithm can achieve, so this ratio tells us well our new algorithm has succeeded in avoiding most cost of accessing slow memory (the ratio is less than 1, and the closer it is to 1, the better). It can also be interpreted as the fraction of peak performance attained by the new algorithm.

6. The speedup $k \cdot T^{SA2,opt}_{b,n,1,\alpha,\beta,mem}/T^{SA2,opt}_{b,n,k,\alpha,\beta,mem}$ of the new versus conventional algorithm for a fixed n and varying k and stencil size b.

We now present the performance modeling results for SA2.

2D Stencil on OOC

Figure 2.33(a) shows that for every value of n modeled, a maximum speedup of 10.2 is attained by choosing $k = 59$. Indeed, the speedup is still increasing slowly at $k = 59$, the largest value of k modeled, and so further speedups may be possible. Figure 2.33(c) shows that for each n, increasing k from 1 to 59 raised the fraction of time spent in computation from 2% to 18%. Since Figure 2.33(d) shows that this is accomplished without ever increasing the number of flops by more than $1.05\times$, we know further speedup would be limited to $1.05/.18 \approx 5.8\times$. This same potential further speedup (actually the reciprocal, .17) is also expressed in Figure 2.33(e), which shows the time of the conventional algorithm running on a 0 latency / infinite bandwidth machine divided by the new algorithm's running time. Finally, Figure 2.33(f) shows that even higher speedups are possible for larger matrix bandwidths b, up to $13.8\times$ speedup for $b = 3$.
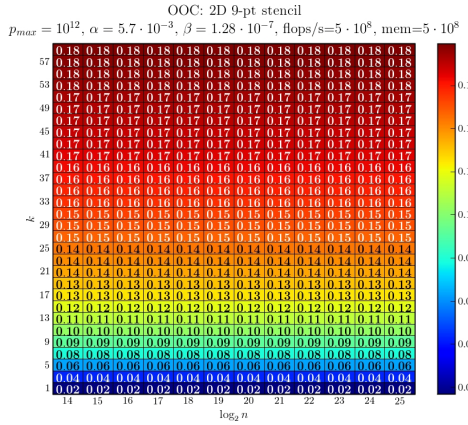
We note that our new sequential algorithm provides speedups that are more uniform across values of n and b than our new parallel algorithm. The reason is that both the number of arithmetic operations and the number of words transferred grow proportionally to $b^2 n^2$, making it always advantageous to avoid bandwidth costs.
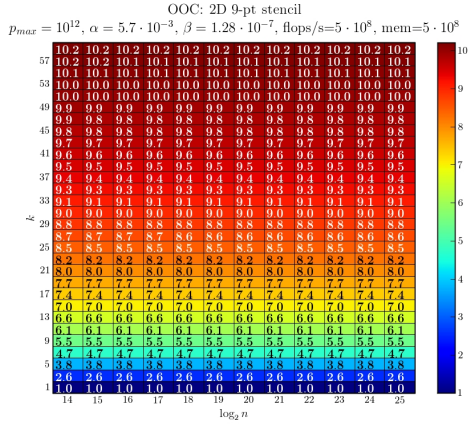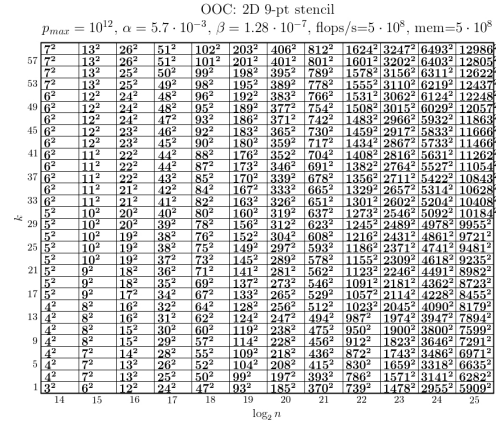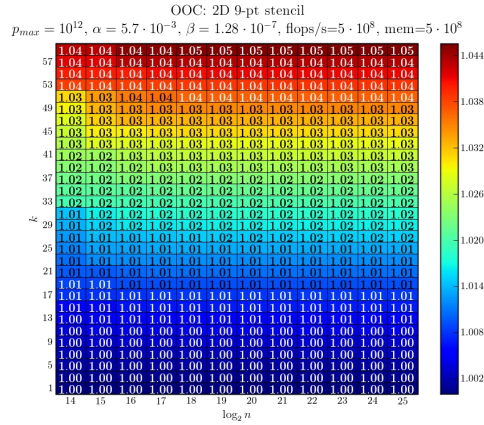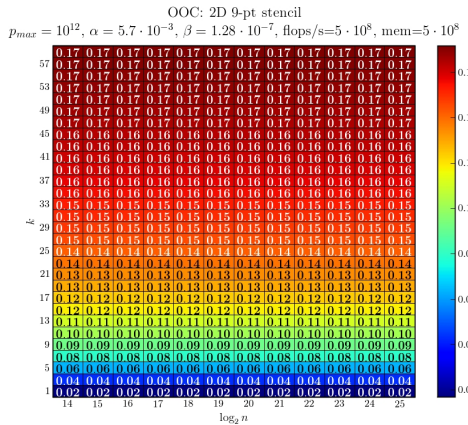
OOC: 2D 9-pt stencil
$p_{max} = 10^{12}$, $\alpha = 5.7 \cdot 10^{-3}$, $\beta = 1.28 \cdot 10^{-7}$, flops/s=$5 \cdot 10^8$, mem=$5 \cdot 10^8$

(a) Speedup

OOC: 2D 9-pt stencil
$p_{max} = 10^{12}$, $\alpha = 5.7 \cdot 10^{-3}$, $\beta = 1.28 \cdot 10^{-7}$, flops/s=$5 \cdot 10^8$, mem=$5 \cdot 10^8$

(b) Optimal p

OOC: 2D 9-pt stencil
$p_{max} = 10^{12}$, $\alpha = 5.7 \cdot 10^{-3}$, $\beta = 1.28 \cdot 10^{-7}$, flops/s=$5 \cdot 10^8$, mem=$5 \cdot 10^8$

(c) Fraction of time in computation

OOC: 2D 9-pt stencil
$p_{max} = 10^{12}$, $\alpha = 5.7 \cdot 10^{-3}$, $\beta = 1.28 \cdot 10^{-7}$, flops/s=$5 \cdot 10^8$, mem=$5 \cdot 10^8$

(d) Ratio of additional flops

OOC: 2D 9-pt stencil
$p_{max} = 10^{12}$, $\alpha = 5.7 \cdot 10^{-3}$, $\beta = 1.28 \cdot 10^{-7}$, flops/s=$5 \cdot 10^8$, mem=$5 \cdot 10^8$

(e) Slowdown vs conventional alg. with $\alpha = \beta = 0$ (fraction of peak)

OOC: 2D $(2b+1)^2$-pt stencil
$p_{max} = 10^{12}$, $\alpha = 5.7 \cdot 10^{-3}$, $\beta = 1.28 \cdot 10^{-7}$, flops/s=$5 \cdot 10^8$, mem=$5 \cdot 10^8$, $n = 2^{20}$

(f) Speedup as a function of matrix bandwidth ($n = 2^{20}$)
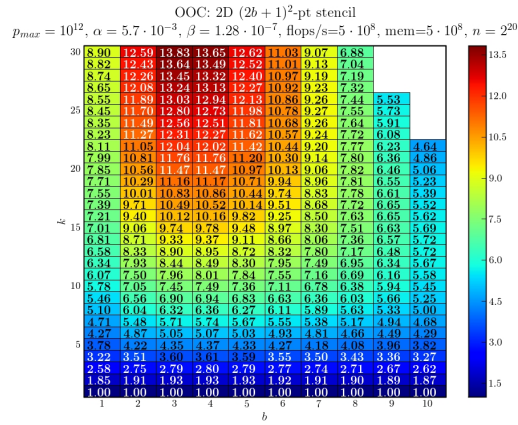
Figure 2.33: Plots for 2D stencil on OOC. Only odd k shown. For all n, choosing $k = 59$ yields a speedup of $10.2\times$ and yields a fraction of peak of 17%, up from 2%

3D Stencil on OOC

The 3D case is broadly similar to the 2D case. Figure 2.34(a) shows that for every value of n modeled, a maximum speedup in the range $[7.39, 9.51]$ is attainable, where the best k depends on n and varies in the range $[23,43]$. Figure 2.34(c) shows that for each n, choosing the best k raised the fraction of time spent in computation from 2% to at least 21%. Figure 2.34(d) shows that this is accomplished without ever increasing the number of flops by more than $1.57\times$. This potential fraction of peak is shown in Figure 2.34(e), which lies in the range $[14\%,18\%]$, up from 2%. Figure 2.33(f) shows that good speedups are possible for larger matrix bandwidths b, but not as high as in the 2D case.

2D Stencil on CacheBlocked

Figure 2.35(a) shows that for every value of n modeled, a maximum speedup in the range $[2.45, 2.58]$ is attainable. Figure 2.35(c) shows that for each n, choosing the best k raised the fraction of time spent in computation from 25% up to at least 71%. Figure 2.35(d) shows that this is accomplished without ever increasing the number of flops by more than $1.14\times$. The fraction of peak is expressed in Figure 2.35(e), and is in the range $[.62,.65]$ when choosing the best value of k. Finally, Figure 2.35(f) shows that some speedups are possible for larger matrix bandwidths b.

3D Stencil on CacheBlocked

Figure 2.36(a) shows that for every value of n modeled, a maximum speedup of $1.34\times$ to $1.36\times$ is attainable, by choosing $k = 3$, Figure 2.36(c) shows that for each n, choosing $k = 3$ raised the fraction of time spent in computation from 28% to 48%. Figure 2.36(d) shows that this is accomplished by increasing the number of flops by $1.27\times$. This potential fraction of peak is shown in Figure 2.36(e), and is 38%, up from 28%. Figure 2.36(f) shows that speedups are not possible for larger matrix bandwidths b. The speedups in this case are not as impressive as other cases because of the fast memory being too small (this is also confirmed by the analytical model in Figure 2.16(b)). Figure 2.16(b) also tells us that even doubling the fast memory size is not expected to give good gains.
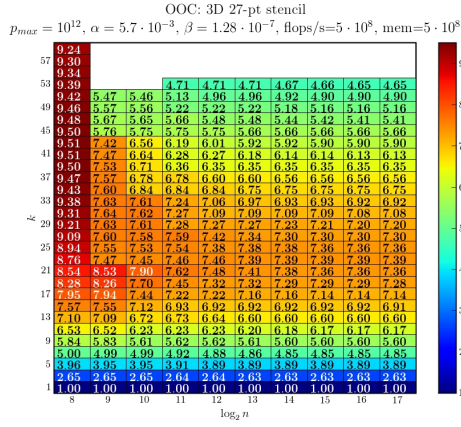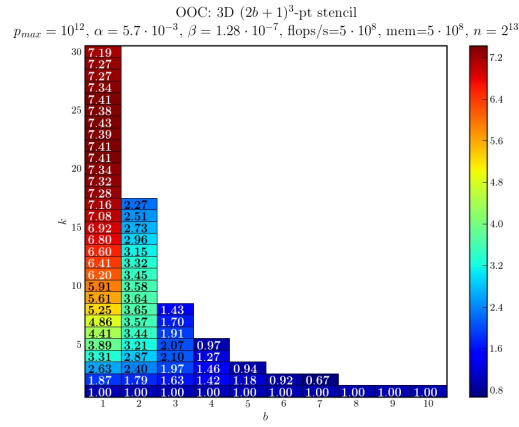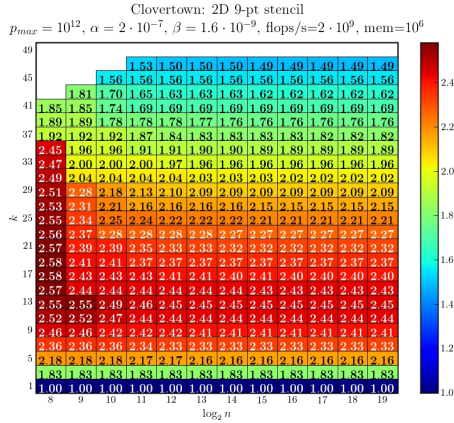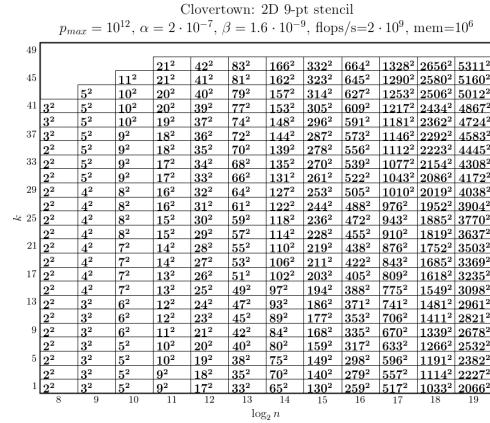
(a) Speedup
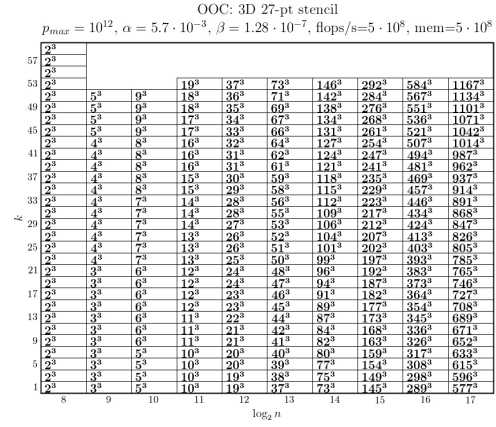
(b) Optimal p
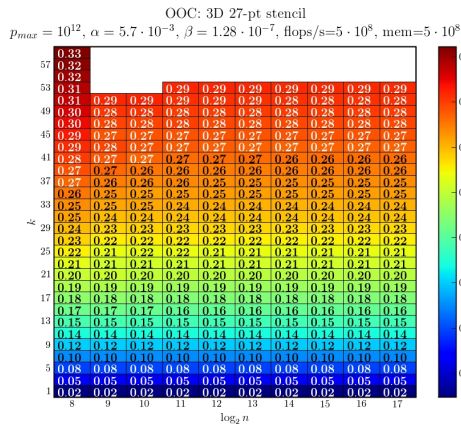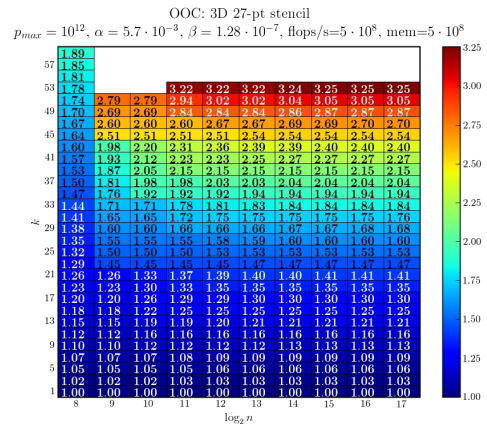
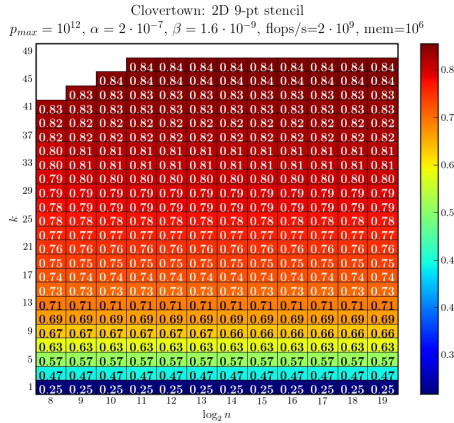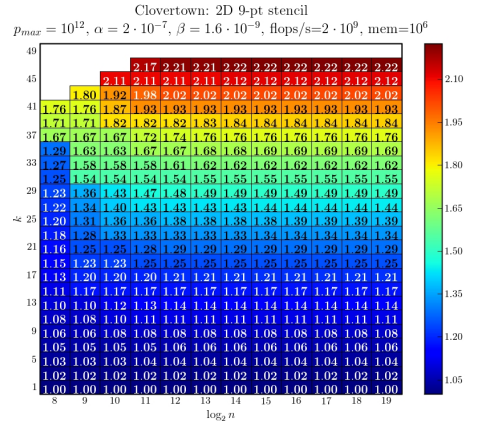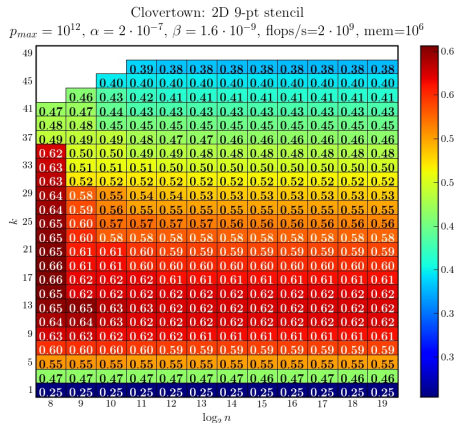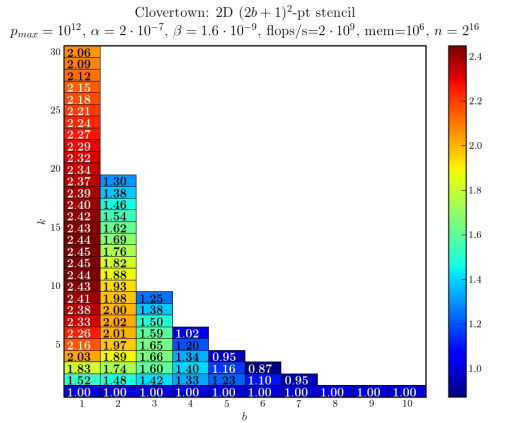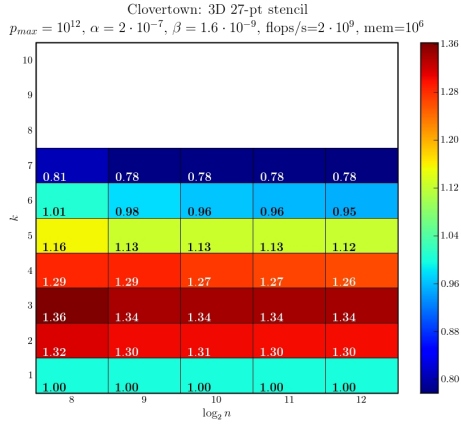(c) Fraction of time in computation

(d) Ratio of additional flops
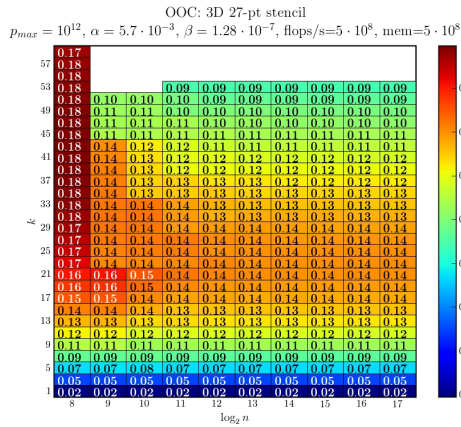
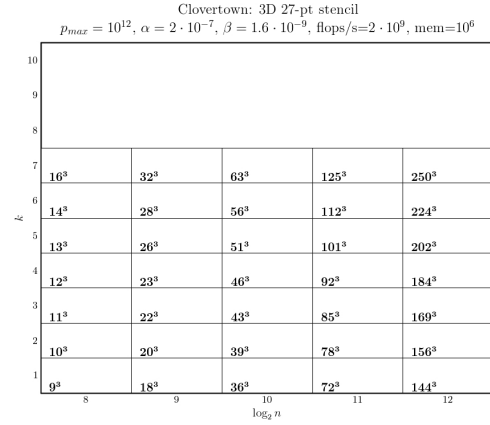(e) Slowdown vs conventional alg. with $\alpha = \beta = 0$

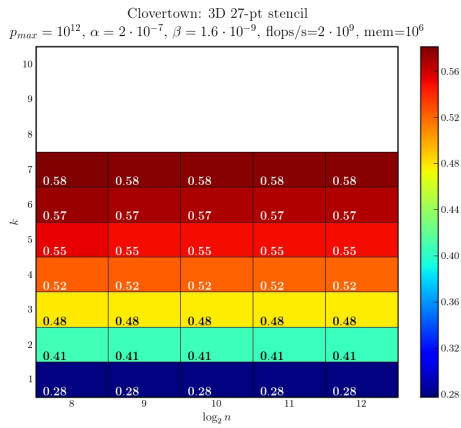(f) Speedup as a function of matrix bandwidth ($n = 2^{13}$)

Figure 2.34: Plots for 3D stencil on OOC. Only odd k shown. For all n the best k yields a speedup in the range [7.39,9.51] and a fraction of peak in the range [14%,18%]
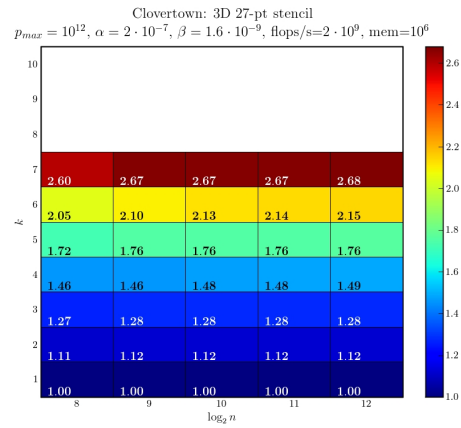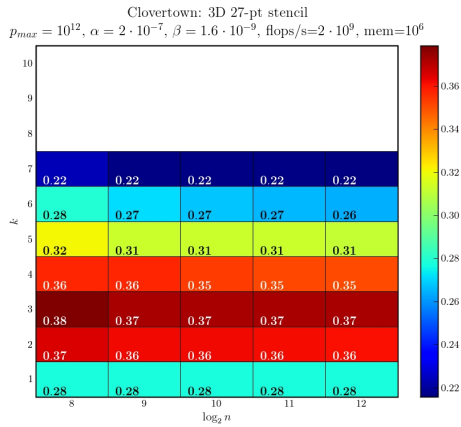
(a) Speedup

(b) Optimal p

(c) Fraction of time in computation

(d) Ratio of additional flops

(e) Slowdown vs conventional alg. with $\alpha = \beta = 0$

(f) Speedup as a function of matrix bandwidth ($n = 2^{16}$)

Figure 2.35: Plots for 2D stencil on CacheBlocked. For all n the best k yields a speedup in the range [2.45,2.58] For all n the best k yields a fraction of peak in the range [.62,.65]
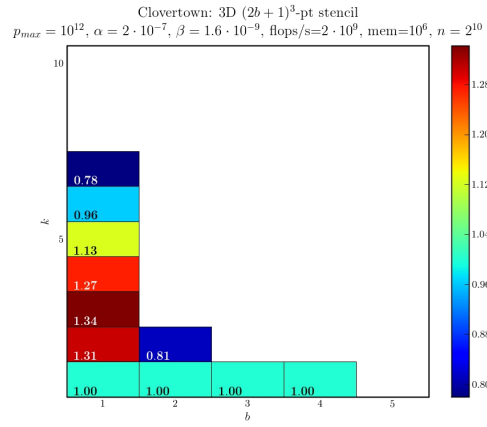
(a) Speedup

(b) Optimal p

(c) Fraction of time in computation

(d) Ratio of additional flops

(e) Slowdown vs conventional alg. with $\alpha = \beta = 0$

(f) Speedup as a function of matrix bandwidth ($n = 2^{10}$)

Figure 2.36: Plots for 3D stencil on CacheBlocked. For all n the best k yields a speedup in the range $[1.34, 1.36]$ For all n the best k yields a fraction of peak of .38.

2.8 Implementation of PA1, P2 and Out-Of-Core SA2

We implemented PA1 and PA2 for general sparse matrices in UPC [37]. We tested our implementation on the UC Berkeley CITRIS cluster (Intel Itanium 2-based) and the NERSC Jacquard cluster (AMD Opteron-based). However, since the network for these machines has low latency, there were no speedups from our parallel algorithms. Our sequential implementation, however, shows good speedups. So, we report the performance results for our implementation of SA2.

For the implementation of SA2, we needed to solve an ordering problem for the rows of x and A in order to minimize the communication cost (Section 2.5.5). This minimization problem corresponds to minimizing the number of words fetched from disk during the course of the algorithm. We used a simple random sampling strategy to choose the best ordering from a sequence of random orderings. This worked out well as the actual number of words transferred between disk and main memory was close to the lower bound for SA2 (within 2%). Another level of reordering was done on a per-block basis in our implementation. This allowed the computations in SA2 to be done as a sequence of k calls to separate, tuned sparse matrix multiplication (SpMV) routines. In our implementation we used the OSKI library [105].

We tested our implementation on the UC Berkeley CITRIS cluster—a cluster of Itanium 2 nodes each with a theoretical peak performance of 5.2 GFlops/s. Each node has 2 Itanium processors with 4 gigabytes of memory per processor.

Our test problem was a matrix with a 27-point stencil on a 3D mesh (stored as a general sparse matrix) with $n = 368$ and $p = 64$ (the choice of n was limited by the available disk space). Thus the matrix had dimension $368^3 = 49,836,032$ with 27 nonzeros in most rows, broken into $4^3 = 64$ blocks of $(\frac{368}{4})^3 = 92^3 = 778,688$ rows each. The value of p was chosen to optimize performance.

For accurate performance modeling, we used measured values for all important machine parameters: time per floating point operation and disk bandwidth. The disk bandwidth differs significantly for reads and writes, so we augmented our model to distinguish reads and writes. Disk latency turned out to play a negligible role.

- $t_f = 3.12$ ns ($1/t_f = 321$ Mflops/s): This is the measured inverse flop rate for SA2. This was taken as the median of the flop rates observed for the computational phases in SA2.
- $\beta_r = 56$ ns ($1/\beta_r = 143$ MBytes/s): This is the measured inverse read bandwidth.
- $\beta_w = 240$ ns ($1/\beta_w = 33$ MBytes/s): This is the measured inverse write bandwidth.

Figures 2.37 and 2.38 show the results, both modeled and measured, which closely match. Figure 2.37 breaks the total runtime down into computation and communication, and Figure 2.38 shows the speedup, which reaches 3.2× at $k = 15$, and is at least 3× for $k \geq 8$.

We also compare the results to those on a hypothetical machine with infinite DRAM, so that the entire computation can proceed in main memory. Such an algorithm obviously provides an upper bound on our speed. We go from running 20× slower than this algorithm at $k = 1$ to just 6× slower at $k = 15$ (these are measured values).
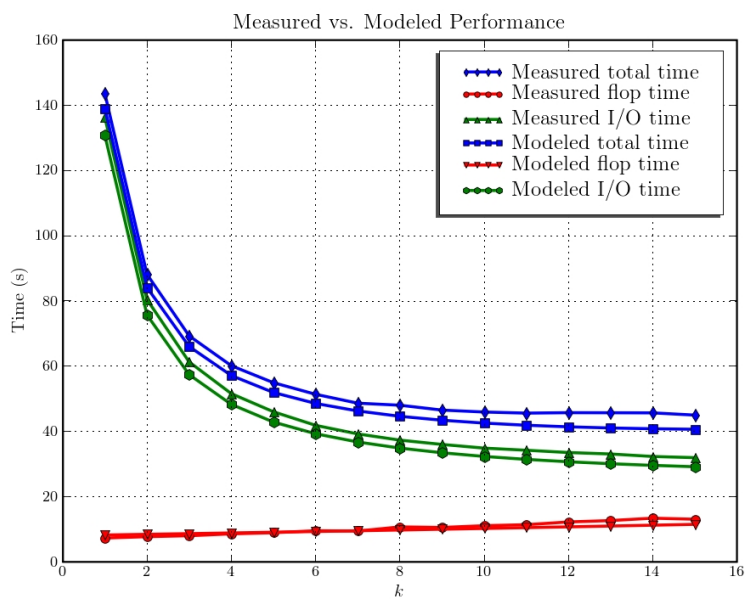
Figure 2.37: Measured vs. modeled performance for SA2 on an Itanium2 machine.
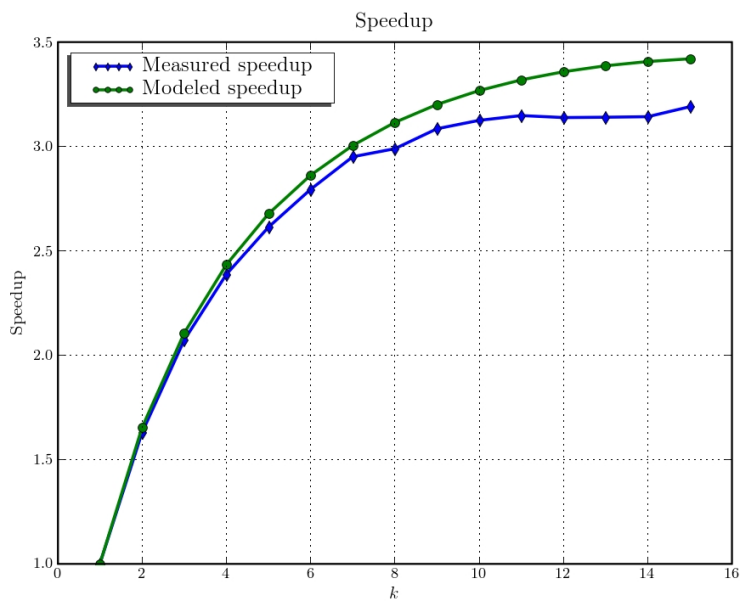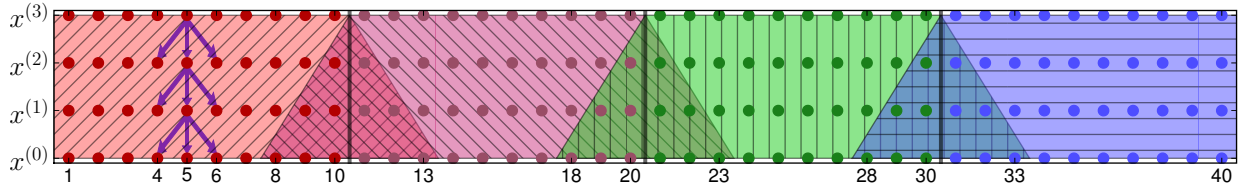

Figure 2.38: Measured vs. modeled speedup for SA2 on an Itanium2 machine.

(a) Parallel algorithm/explicit sequential algorithm for a tridiagonal matrix with $n = 40$, $k = 3$, $p = 4$. The purple arrows indicate the dependence pattern for the tridiagonal matrix for some of the entries. The vertices are colored by their affinities. The overlapping regions (the three triangles) indicate redundant computation. Note that each trapezoid can be computed independently of each other, which means they can be computed in any order sequentially or in parallel.
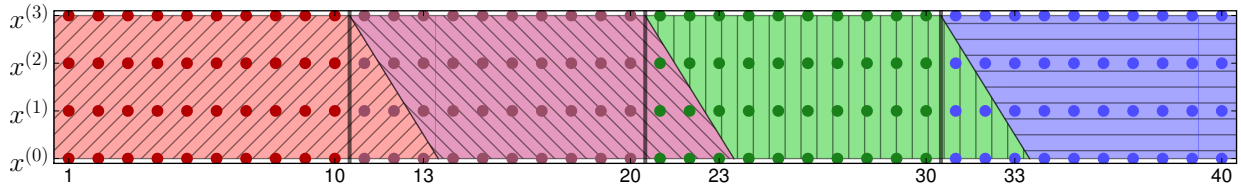
(b) Implicit sequential algorithm for a tridiagonal matrix with $n = 40$, $k = 3$, $p = 4$. The vertices are colored by their affinities. The blocks have to be computed in a specific order—in this case, red first, followed by magenta, green and then blue.

Figure 2.39: Our shared-memory algorithms illustrated on a tridiagonal matrix.

2.9 Shared Memory Algorithms for Multi-Cores

We now describe algorithms for matrix powers on multi-core platforms. This implementation was part of an effort to implement a communication-avoiding version of the GMRES [86], a widely used iterative solver for sparse systems of equations $Ax = b$. In contrast to previous sections, we consider the Newton basis version of matrix powers. Nonetheless, we still use the monomial basis version for the purpose of examples.

To motivate our shared-memory algorithms, we reconsider the simple case when the matrix A is tridiagonal (the dependency graph of the vectors is shown in Figure 2.39). Let $x_j^{(i)}$ be the j-th component of $x^{(i)} = (A - \lambda_i I) \cdots (A - \lambda_1 I)x^{(0)}$. Although not shown throughout the figure, each entry depends on the one below it and its two neighbors, e.g., $x_5^{(1)}$ depends on $x_4^{(0)}$, $x_5^{(0)}$ and $x_6^{(0)}$ as shown by purple arrows in Figure 2.39. The vectors and the rows of the matrix are partitioned into $p = 4$ blocks. In the parallel algorithm, each block resides on a different processor and in case of a sequential algorithm, the blocks are computed one at a time. Consider the green colored block in Figure 2.39(a). For the *parallel algorithm*, if one processor has the entries of $x^{(0)}$ numbered from 18 to 33 (the base of the third trapezoid), then we can compute all the green entries—the non-green entries will need to be explicitly fetched from other blocks. Thus, instead of fetching non-green entries for every $x^{(i)}$, we fetch them only once, which improves performance by reducing the number of inter-processor messages. However, we will be computing extra entries, which do not reside on the green partition, e.g., entries 19, 20, 31, 32 of $x^{(1)}$—this constitutes redundant computation. The *explicit sequential algorithm* emulates the parallel algorithm by iterating over blocks (each block fits in fast memory) and computing on a block in the

Parallel Algorithm (code for proc. q)	Explicit Sequential Algorithm	Implicit Sequential Algorithm

copy entries in $R(V_q,k)^{(0)} - V_q^{(0)}$ from shared memory to ghost zone
for $i = 1$ to k **do**
    compute all $x_j^{(i)} \in R(V_q,k)^{(i)}$
	for $q = 1$ to p **do**
	    copy entries in $R(V_q,k)^{(0)} - V_q^{(0)}$ from slow memory to ghost zone
	    for $i = 1$ to k **do**
	        compute all $x_j^{(i)} \in R(V_q,k)^{(i)}$
		$C = \emptyset$ {C = set of computed entries}
		for $q = 1$ to p **do**
		    for $i = 1$ to k **do**
		        compute all $x_j^{(i)} \in R(V_q,k)^{(i)} - C$
		        $C \leftarrow C \cup R(V_q,k)^{(i)}$

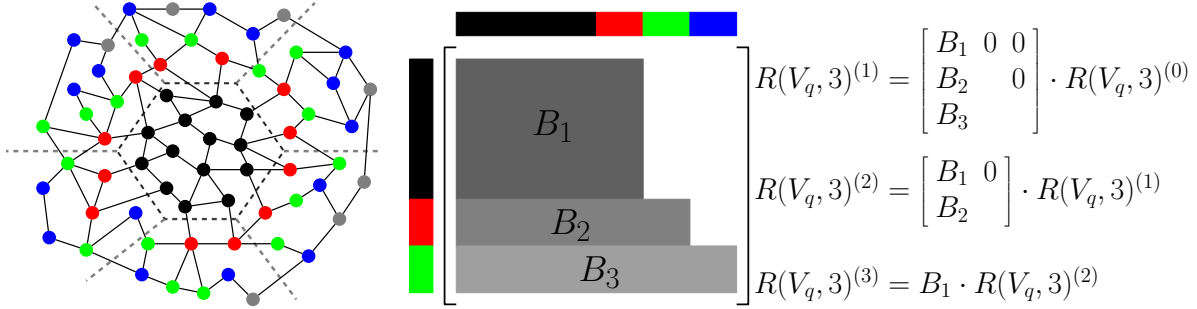Figure 2.40: Shared memory algorithms for the matrix powers kernel.

same manner as the parallel algorithm. In the explicit sequential algorithm, the benefits are even more significant, because we are reading the matrix A from slow memory just a little more than once. However, because we perform redundant flops, some of the entries of A are fetched more than once from slow memory. If the number of redundant flops is small, we are effectively reading the matrix only once, whereas the naïve strategy would have read the matrix k times, which translates to a potential speedup of k if the naïve algorithm is memory bound.

Next we consider the *implicit sequential algorithm* (illustrated in Figure 2.39(b)), which has no redundant flops. We improve upon the explicit sequential algorithm by only computing entries which have yet not been computed. In contrast to the explicit sequential algorithm, which used explicit copies of the entries on other blocks, the implicit algorithm maintains no such copies, which is why we do not see overlapping trapezoids in Figure 2.39(b).

2.9.1 Parallel Algorithm

Our parallel algorithm (Figure 2.40) for shared-memory multicore machines is a simplification of PA1 discussed in Section 2.4. Due to the shared memory model, processors do not need to use explicit sends; the required data can be simply pulled from the shared memory. Each block is assumed to reside on a different processor. We illustrate the parallel algorithm on the symmetric matrix A whose graph is described in Figure 2.41(a). Letting q denote the central block, the red and black vertices constitute $R(V_q, 1)^{(0)}$, whereas the blue, green, red and black constitute $R(V_q, 3)^{(0)}$. These vertices, which are not local to the block, constitute the ghost vertices for the block.

Figure 2.41(b) shows that ordering the local and ghost entries in increasing order of distance from the local vertices enables the use of highly optimized SpMV routines. Local vertices (colored black) go first, followed by the ghost entries within distance 1 (colored red), followed by ghost entries within distance 2 (colored green), and so on. Given this ordering, one can use SpMV to compute the entries at level i. Computation of a higher level $i + 1$ requires SpMV involving a smaller set of contiguous block of rows of a matrix and a vector. This is aptly shown by the set of equations in Figure 2.41(b).

(a) Example general graph. For computing the central block the red vertices from $x^{(0)}$ are needed when $k = 1$, red and green, when $k = 2$ and red, green and blue when $k = 3$.

(b) Ordering the vertices for the parallel and explicit sequential algorithm when $k = 3$. At any level, the vertices are ordered in increasing order of distance from the local vertices (colored black). B_1, B_2 and B_3 are sparse matrices derived from the original matrix A by restricting it to the corresponding set of local and ghost rows, e.g., B_2 says how to compute the red vertices of $x^{(i)}$ given the black, red and green vertices of $x^{(i-1)}$. When the matrix A is symmetric, the columns of B_3 corresponding to the black nodes would be all zeros.

Figure 2.41: An example sparse matrix and an illustration of how its rows may be ordered for the parallel and the explicit sequential algorithms.

2.9.2 Sequential Algorithms

Figure 2.40 shows both sequential algorithms: explicit and implicitly cache-blocked. One contrast to the explicit sequential algorithm SA2 in Section 2.5 is that we do not need to explicitly fetch or store the entries in the current cache block. Since we are targeting cache-based architectures, this is done implicitly by the hardware. Furthermore, this also implies that we only need to keep two vectors in cache (and the matrix rows) instead of all $k + 1$ of them: once a vector has been used to compute the entries at the next level, it is no longer needed. Thus, when $R(V_q, k)^{(i-1)}$ has been used to compute $R(V_q, k)^{(i)}$, it is no longer needed for computations in block q for higher levels. Thus, not only can the cache block partitions be larger, the amount of redundant computation is also reduced, since fewer blocks means less redundant computation. The example in Figure 2.39(a) illustrates the explicit sequential algorithm the same way as the parallel algorithm.

As evident in Figure 2.40, the implicitly cache-blocked algorithm performs no extra flops. This has the potential advantage when k is large enough to result in significantly fewer flops than the explicit sequential algorithm, and can provide speedups over the naïve algorithm even when the explicit sequential algorithm does not. However, this improvement comes at the cost of bookkeeping to keep track of which entries need to be computed when looking at a given level of block q, i.e., the computation schedule. The computation schedule also includes the order in which the blocks are traversed, thus making the implicit algorithm more sensitive to the block ordering when compared to the explicit sequential algorithm. Finally, we note that a good ordering of entries within each block as well as an ordering of the blocks are useful in improving locality of accesses when looking at entries in other blocks as well as improving reuse of already computed entries. Both these ordering problems can

Explicit Cache-Blocked Parallel Algorithm (Code for proc. q)	Implicit Cache-Blocked Parallel Algorithm (Code for proc. q)

for $m = 1$ to b_q **do**
    fetch entries in $R(V_{q,m}, k)^{(0)} - V_{q,m}^{(0)}$ to ghost zone
    for $i = 1$ to k **do**
      compute all $x_j^{(i)} \in R(V_{q,m}, k)^{(i)}$
	$C = \emptyset$ $\{C = \text{set of computed entries}\}$
	fetch entries in $R(V_q^{(0)}, k) - V_q^{(0)}$ to ghost zone
	for $m = 1$ to b_q **do**
	    for $i = 1$ to k **do**
	      compute all $x_j^{(i)} \in R(V_{q,m}, k)^{(i)} - C$
	    $C \leftarrow C \cup R(V_{q,m}, k)^{(i)}$

Figure 2.42: Implemented hybrid algorithms.

be formulated as instances of the Traveling Salesman problem [31]. However, we leave the incorporation of solutions of these Traveling Salesman problems as future work.

2.10 Multi-Core Implementation

In Section 2.8, we focused on a sequential out-of-core implementation, where the gap between bandwidth and computational capability is especially large. This section demonstrates that significant improvements are possible even for a multi-core out-of-cache implementation. Performance data on an 8-core 2.33 GHz Intel Clovertown shows speedups for computing W of up to $2.7\times$ over k calls to the best optimized algorithm just for a single sparse matrix-vector multiplication (SpMV, or $A \cdot x$).

Because of the hierarchical memory structure of multi-cores, our matrix powers implementation is hierarchical: it uses the parallel algorithm on the outer level, i.e., for multiple cores, and a sequential algorithm at the inner level, i.e., for off-chip data movement. Thus, the matrix is thread-blocked first for the parallel algorithm and then cache-blocked within each thread. To this end, it is useful to introduce additional notation. We now define the affinity of a vertex as a pair (q, b), in which q is the thread number and b is the cache block number on thread q. Given this definition, V_q means the set of vertices on thread q and $V_{q,b}$ would be the set of vertices in cache block b on thread q. For thread q, we let b_q denote the number of cache blocks for that thread.

Figure 2.42 describes both cases of whether the inner sequential algorithm is explicit or implicit. We distribute the cache blocks to different threads in a balanced manner. For the implicit implementation, we note that it performs redundant flops when compared to the implicit sequential algorithm in Figure 2.40—these extra flops are due to parallelization at the outer level. Therefore, the computation schedule for the implicit algorithm must also account for computing the ghost entries.

One reason we describe both the explicit and implicit algorithms is because there is no clear winner between the two—the choice depends on the nonzero pattern of the matrix. The explicit algorithm has a memory access pattern which does not go beyond the current block after the data has been fetched in to the ghost zones and also admits the use of the cache bypass optimization. Because of the explicit copy of the entries on neighboring cache blocks, contiguous blocks of data are computed at each level. However, the number of redundant

---

Explicit Parallel Algorithm Using Cache Bypass (Code for proc. q)

{Vectors z_0, z_1 have size $\max\limits_{m=1}^{b_q} |R(V_{q,m}, k)^{(1)}|$ to store any $R(V_{q,m}, k)^{(i)}$ $(1 \leq i \leq k)$}

for $m = 1$ to b_q **do**

    fetch entries in $R(V_{q,m}, k)^{(0)} - V_{q,m}^{(0)}$ to ghost zone

    compute all rows in $R(V_{q,m}, k)^{(1)}$ using $R(V_{q,m}, k)^{(0)}$ and store in z_0

    copy from z_0 (in cache) to $x_{q,m}^{(1)}$ (in slow memory) using cache bypass

    for $i = 2$ to k **do**

        compute all rows in $R(V_{q,m}, k)^{(1)}$ using $z_{i \bmod 2}$ and store in $z_{(i+1) \bmod 2}$

        copy $z_{(i+1) \bmod 2}$ (in cache) to $x_{q,m}^{(i)}$ (in slow memory) using cache bypass

---

Figure 2.43: Explicit implementation with cache bypass optimization.

flops for the explicit algorithm can increase dramatically for matrices A whose powers A^i grow quickly in density, resulting in no performance gains. The implicit algorithm, on the other hand, only has redundant flops due to parallelization at the outer level. Since the number of threads is typically small, the number of redundant flops is expected to grow slowly for the implicit algorithm. However, the memory accesses for the implicit algorithm can span multiple cache blocks, making it more sensitive to the ordering in which the cache blocks are computed/stored. Furthermore, the implicit algorithm has a higher overhead resulting in performance degradation if the kernel is memory bound. In general, if the matrix density grows slowly with the power, then the explicit algorithm is better, otherwise the implicit algorithm is expected to win.

2.10.1 Optimizations

In addition to the algorithmic optimizations discussed earlier in Section 2.9, we also implement additional low-level optimizations to get good performance. Some of these optimizations are borrowed from a highly optimized SpMV implementation [113], which also serves as the baseline for our performance comparisons. We implement the usual optimizations for sparse matrix computations and storage [113]: branch elimination, SIMD intrinsics, register tiling, and shorter integer types for the indices, as well as additional optimizations. Since hand-coding the computational kernels can be tedious and time consuming, we use code generators in the same manner as in [113]. Given the difficulty of deciding the right optimizations and parameters, our implementation has an auto-tuning phase where it benchmarks the input matrix to figure out the right data structures and optimization parameters. These are described below:

Partitioning strategy: We use a recursive algorithm which first creates partitions the vectors among the threads, and then recursively creates cache blocks for each thread. The recursion stops when the current cache block is small enough to fit in the thread cache. For creating the partitions at each recursion level, we either use METIS [58] or sub-partition the current partition into contiguous blocks, with equal work in each sub-partition. When METIS is used, the matrix and the vectors need to be permuted to make each thread block and cache block contiguous. Using METIS can result in lower inter-block communication and lower flops/block. However, this benefit due to METIS can be offset by a more irregular
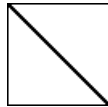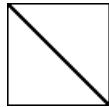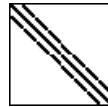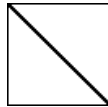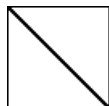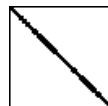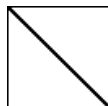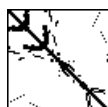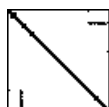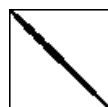
1d 3-pt	**1d 5-pt**	**2d 9-pt**	**marcat**
Tridiagonal matrix	Pentadiagonal matrix	9-pt operator on 2D mesh	Impatient customers
			on telephone exchange
(1M, 3M, 3)	(1M, 5M, 5)	(1M, 9M, 9)	(547K, 2.7M, 5)
bmw	**cant**	**cfd**	**mc2depi**
Stiffness matrix	FEM cantilever	Pressure matrix	2D Markov model
			of epidemic
(141K, 7.3M, 51)	(62K, 4M, 65)	(123K, 3.1M, 25)	(525K, 2.1M, 4)
gearbox	**pwtk**	**shipsec**	**xenon**
Aircraft flap actuator	Pressurized wind tunnel	FEM ship section/detail	Complex zeolite,
	stiffness matrix		sodalite crystals
(153K, 9.1M, 59)	(218K, 12M, 55)	(141K, 7.8M, 55)	(157K, 3.9M, 25)

Table 2.5: Each matrix is described by its spyplot, name, description and the triple showing (#rows, #nonzeros, #nonzeros/#rows).

memory access pattern due to the permutation, particularly for the implicit algorithm, where memory accesses can be spread over multiple cache blocks. Thus, the decision of whether to use METIS or not is made during the auto-tuning phase by actually timing both partitioning strategies.

Inner sequential algorithm: Since the choice of whether to use the explicit or the implicit implementation depends on the matrix nonzero pattern, we make the decision by timing during the auto-tuning phase.

Register tile size: Register tiling a sparse matrix can reduce the required slow memory bandwidth [113] or improve instruction throughput [104]. Since SpMV is typically memory bound on modern multi-core platforms, heuristics which try to minimize memory footprint of the matrix are sufficient [113]. However, the arithmetic intensity (i.e., the flops to DRAM byte ratio) of the matrix powers kernel can increase with k, making it computation bound. Since use of a larger register tile can mean extra flops, performance can degrade when the kernel is computation bound. Therefore, we auto-tune to decide whether a larger register tile should be used or not. Note that for the implicit implementation, the register tiles must

be square because we need to track the dependencies between same-sized groups of vertices.

Cache bypass optimization: For the case of the explicit implementation, we note that although we compute extra entries in the ghost zones, they do not need to be written back to slow memory. Furthermore, we do not need to keep all the $k+1$ vectors in a given cache block in cache; only the vectors at the current level being computed and the level below need to be in cache. Thus, we compute by cycling over two buffers, which are always in cache: one buffer is used to compute the current level, and the other holds the previous level. Once the current level has been computed, we copy the buffer to the vector in slow memory by bypassing the cache (using SIMD intrinsics, e.g., the `mm_stream_pd` intrinsic on Intel machines). This optimization is particularly useful in reducing memory traffic on write-allocate architectures, like the one in this work. Without cache bypass, due to write-allocate behavior, each output vector will contribute twice the bandwidth: once for allocating cache lines when being written, and again when it is evicted from cache while computing the next cache block. Furthermore, since all the rows in $V_{q,m}^{(i)}$ for level i on cache block m of thread q are stored contiguously, this copy to slow memory is cheap. Figure 2.43 shows the explicit cache-blocked parallel algorithm with cache bypass optimization. Note that this optimization cannot be applied to the implicit algorithm because memory accesses can span multiple blocks.

Stanza encoding: This is a memory optimization to minimize the bookkeeping overhead for the implicit implementation. Note that we need to iterate through the computation sequence for each level on each block. We encode the computation sequence as a sequence of contiguous blocks (*stanzas*)—each contiguous block is encoded by its starting and ending entries. Since we stream through the computation sequence, this optimization can improve performance by reducing the memory bandwidth. We also try to use fewer bits for encoding these stanzas when possible to further reduce the overhead.

Software prefetching: Software prefetching can be particularly useful for a memory-bound kernel like SpMV. However, the prefetch distance and strategy depends both on the algorithm (implicit and explicit algorithms have different data structures implying different software prefetch strategies) and the matrix. Thus, the right software prefetch distances are chosen during the auto-tuning phase. Since software-prefetching is a streaming optimization, it was found to be useful only for $k=1$, where there is no reuse of the matrix entries.

2.10.2   Auto-tuning Matrix Powers

Algorithm 2.15 shows the auto-tuner at a high-level. Since we perform an exhaustive exploration of the search space, the auto-tuner loop is deeply nested. One simple pruning strategy which is implemented in the auto-tuner is that we avoid benchmarking a matrix powers configuration if it performs too many flops or has a large memory footprint—we do this by counting the flops and the minimum memory footprint before actually benchmarking a configuration. Not only does this pruning strategy avoid benchmarking poorly performing configurations, it also avoids crashing the machine when the memory footprint is too large—this can happen if k is large for a poorly partitioned matrix.

Algorithm 2.15 Auto-tuner for the matrix powers kernel.

    for $k = 1, 2, \ldots$ **do**
        for all *partitioner* in possible partitioners **do**
            for all possible values of *nthreads*, *cachesize* **do**
                generate the partition data (map from rows to partition id)
        for all *partition* in generated partitions **do**
            for sequential algorithm = implicit, explicit **do**
                count flops, bytes
                if flops or bytes large **then**
                    continue
                else
                    for each cache block (if explicit) / thread block (if implicit) **do**
                        choose best register tile using SpMV heuristic
                        for cyclic buf. opt. (exp.)/stanza enc. (imp.) enabled = 0,1 **do**
                            encode computation schedule accordingly
                            for all possible software prefetch strategies **do**
                                benchmark matrix powers with the right parameters
  use the best performing parameter values for the input matrix

---

2.10.3 Results

We now describe the performance results of our matrix powers kernel on our target platform—an Intel Clovertown. We consider two cases—one in which all the λ_i's are zero and the other in which all of them are non-zero. As baselines, we shall consider the performance of the matrix powers kernel for $k = 1$, which constitutes the naïve algorithm. We emphasize that the 'naïve' algorithm uses a highly optimized SpMV implementation [113] for the $\lambda = 0$ case (and appropriately modified when $\lambda \neq 0$). For speedup calculation, the time taken for the matrix powers kernel is normalized by dividing by k and compared with the time taken for a single SpMV, i.e., the naïve algorithm. Thus, the speedup is defined as

$$\frac{\text{time(matrix powers kernel for } [p_1(A)x, \ldots, p_k(A)x])/k}{\text{time(SpMV)}}.$$

Platform Summary

Our target machine for this work was an 8-core 2.33 GHz Intel Clovertown. It has a total of 16 MB of on-chip L2 caches, with a 4 MB L2 cache shared by every 2 cores. Each core is capable of executing 4 double-precision flops every cycle, which implies a peak performance of 75 double-precision GFlop/s. However, because of the overheads associated with sparse matrix storage, performance of an in-cache SpMV computation (for a dense matrix in CSR format) is small: 10 GFlop/s. Note that this in-cache performance number incurs no DRAM (which is the slow memory) bandwidth cost, since all the data fits in cache. Thus, 10 GFlop/s is an upper bound on the raw flop rate achievable (i.e., including redundant computation) when the register tile size is 1×1. However, this upper bound only
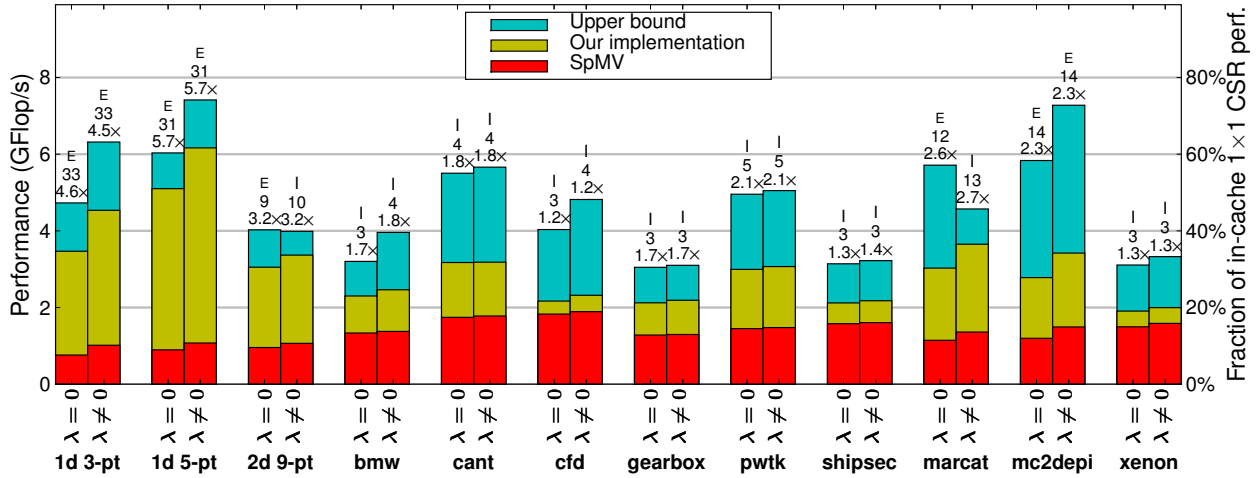
Figure 2.44: Performance of the matrix powers kernel on Intel Clovertown. The yellow bars indicate the best performance over all possible k. Each bar is annotated with some of the parameters for the best performance: whether implicit/explicit (indicated by 'I' or 'E' at the top), the corresponding value of k (just below the 'I' or 'E') and the speedup with respect to the naïve implementation (just below this), which is the highly optimized SpMV code. The label '$\lambda = 0$' indicates the case when all λ_is are 0, whereas the label '$\lambda \neq 0$' indicates the case when all λ_is are nonzero. The label 'upperbound' indicates the performance predicted by scaling the naïve performance by the change in arithmetic intensity (Equation 2.6). Note that the runtimes for $\lambda = 0$ and $\lambda \neq 0$ cases are the same since they transfer the same number of bytes. We can see a big variation in performance as well as speedups over different matrices.

applies to the $\lambda = 0$ case. For $\lambda \neq 0$, we must scale the upper bound on a per-matrix basis in proportion to the increase in arithmetic intensity, i.e., the upper bound for a matrix with m rows and nnz nonzeros is $10 \cdot \left(1 + \dfrac{m}{nnz}\right)$ GFlop/s.

Matrices

In addition to matrices whose graphs are meshes, which are expected to perform well [31], we selected sparse matrices from a variety of real applications [48] (see Table 2.5). Since cache-blocking is only going to benefit when the matrix or vectors do not fit in cache, we deliberately chose matrices with enough nonzeros. Since METIS only works on symmetric matrices, we use METIS to partition $A + A^T$ when the matrix A has an asymmetric nonzero pattern (matrices `marcat`, `mc2depi` and `xenon`).

Performance Results

Figure 2.44 shows the performance of the matrix powers kernel for different matrices. As expected, the speedups for the mesh matrices `1d 3-pt`, `1d 5-pt` and `2d 9-pt` are quite good due to the regularity of memory accesses, even though their naïve performances are

among the lowest. We note that the performance of the $\lambda \neq 0$ kernel was better than that for $\lambda = 0$ kernel simply because it performed additional flops at no extra bandwidth, i.e., it had a higher arithmetic intensity. This difference is marginal when the average number of nonzeros per row of the matrix is large, e.g., the `pwtk` matrix, but is significant when it is small, e.g., the `1d 3-pt` matrix. Since the matrix powers kernel was bandwidth-limited even for the best performance, both the $\lambda = 0$ kernel and $\lambda \neq 0$ kernel had almost the same runtime because they had the same bandwidth.

Note that SpMV performance and the best matrix powers kernel performance across the different matrices is quite different. Therefore, even though `1d 5-pt` had the second lowest SpMV performance, it was able to achieve the best matrix powers kernel performance. In fact, `cfd`, which had the best SpMV performance, gains the least from our implementation. We also note that although we get more than $2\times$ speedups for some of the matrices, we are still far below the upper bound of 10 GFlop/s. As a special case, we note that the optimal performance of `1d 5-pt` required 2×2 register tiling, which has an upper bound of 16 GFlops/s on raw performance. Figure 2.44 also shows another per-matrix upper bound on performance corresponding to the optimal k, which was calculated as

$$\frac{\text{arithmetic\_intensity}(\text{matrix powers})}{\text{arithmetic\_intensity}(\text{SpMV})} \cdot \text{performance}(\text{SpMV}), \qquad (2.6)$$

i.e., by scaling the naïve performance by the factor of increase in arithmetic intensity. Note that the other upper bound of 10 GFlop/s on raw flop rate did not kick in for any of the matrices, i.e., the upper bound by scaling with arithmetic intensity was lower than 10 GFlop/s. We observe that the performance is within 75% of this bound for nicely structured matrices like `1d 3-pt`, `1d 5-pt` and `2d 9-pt`. However, for the rest of the matrices the gap between the upper bound and the measured performance can be quite large—sometimes as much as 60%. For some of the matrices like `marcat` and `mc2depi` part of this difference comes from the reduced instruction throughput due to the explicit copy for the cache bypass optimization. Since the ratio nonzeros/row for these matrices is small, the cost of copying is significant. It is also interesting to note that the implicit algorithm provided the best speedups for most of the matrices. In fact, the explicit algorithm failed to obtain any speedups at all for matrices like `bmw` and `xenon` because the increase in redundant flops was significant.

As we stated earlier, our implementation has an auto-tuning phase where it figures out the right inner algorithms and other parameters. Figure 2.45 serves to illustrate why we need to auto-tune. We show performance for three of the matrices—`cant`, `mc2depi` and `pwtk`. We can see that the use of METIS to partition, which resulted in the rows of the matrix and the vectors being reordered, did not always improve performance. For example, for `mc2depi` reordering improved performance for the explicit algorithm whereas it decreased performance for the implicit algorithm. In fact, for `cant`, reordering degrades performance significantly. We also note that the implicit algorithm provided good speedups for `cant` and `pwtk`, whereas the explicit algorithm was actually worse off for $k > 1$. The fact that the density of `cant` and `pwtk` grew quickly with k is also demonstrated by their relatively small optimal $k = 4$. In contrast, performance of `mc2depi` improved for a larger range of k.

Figures 2.46-2.55 show the performance for different matrices. For each figure, the (a) subplot shows performance and the (b) subplot shows the computed arithmetic intensity as k and other parameters are varied. Note that the computed arithmetic intensity is an upper bound on the achieved arithmetic intensity as the achieved memory traffic may be more than the computed minimum memory traffic. We see that different matrices show different trends in performance as k is varied. This reaffirms the need for run-time tuning for matrix powers.

(a) Performance vs. k for the matrix `cant` (b) Performance vs. k for the matrix `mc2depi`

(c) Performance vs. k for the matrix `pwtk` (d) Performance vs. k for the matrix `2d 9-pt`
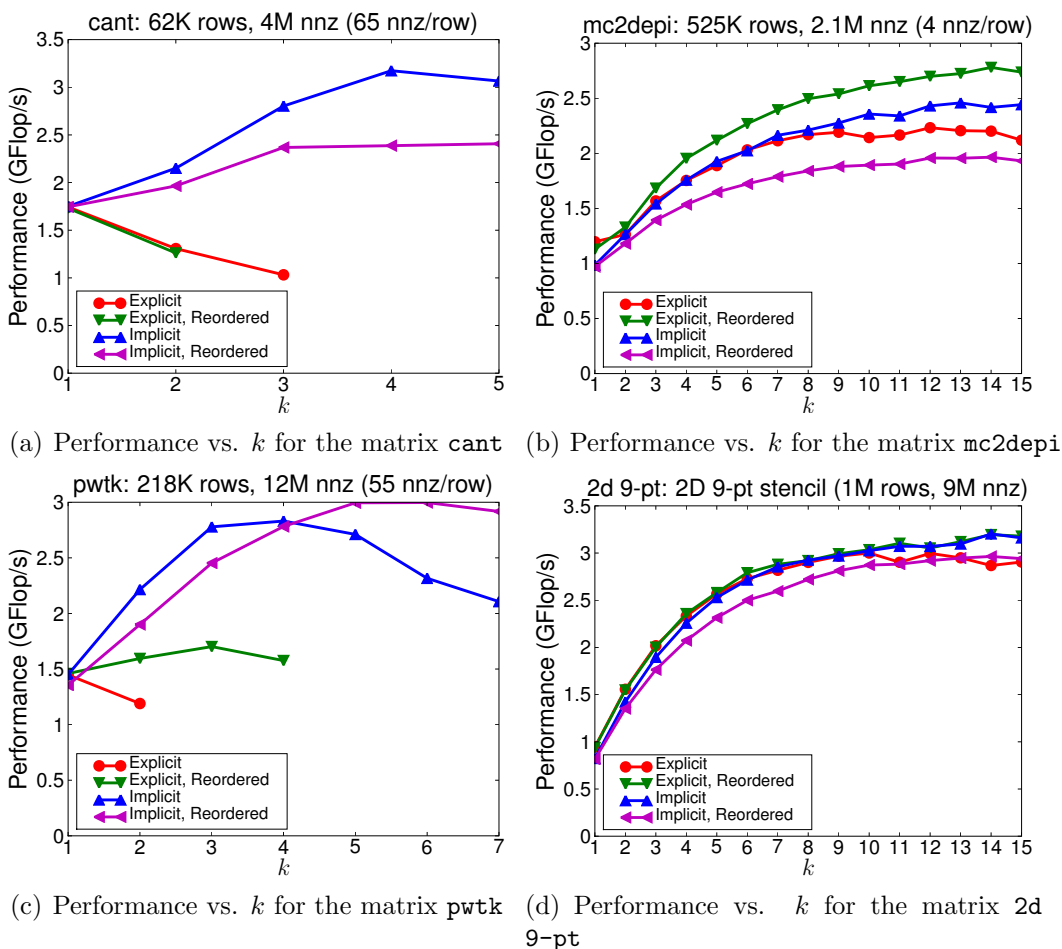
Figure 2.45: Variation of performance with k and tuning parameters for some matrices. 'Explicit' and 'Implicit' indicate whether the cache-blocking was explicit or implicit respectively. 'Reordered' indicates that METIS was used to partition the rows of the matrix and the vectors, resulting in them being reordered. The missing points for some of the k values correspond to when the number of redundant flops was so large that no performance gains were possible, so those cases were not timed at all.

(a) 2D 9-pt: performance

(b) 2D 9-pt: arithmetic intensity
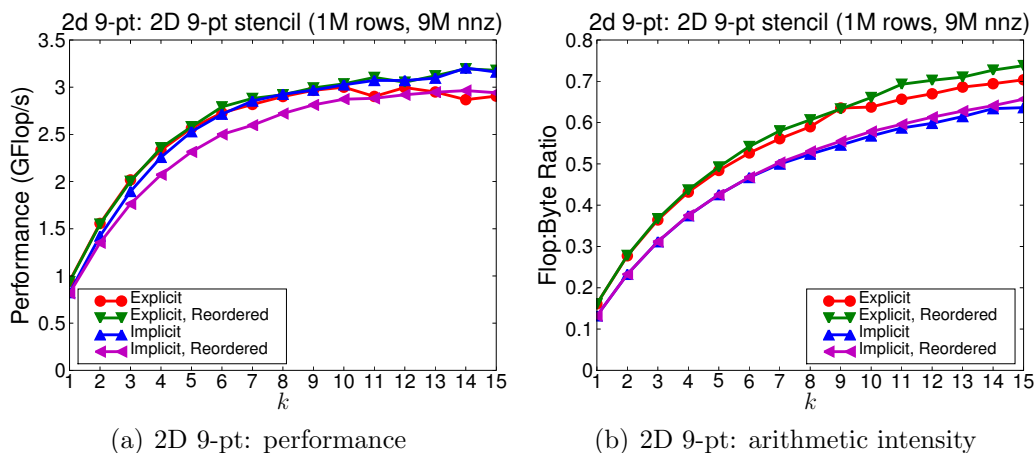
Figure 2.46: Matrix powers performance and achieved arithmetic intensity for 2D 9-pt. Since 2D mesh partitions well, we see performance improvements even for k as large as 10. Also, since the surface-to-volume ratio is small even for large k, explicit cache blocking performs better than implicit cache blocking due to lower instruction and memory overheads.

(a) bmw: performance

(b) bmw: arithmetic intensity
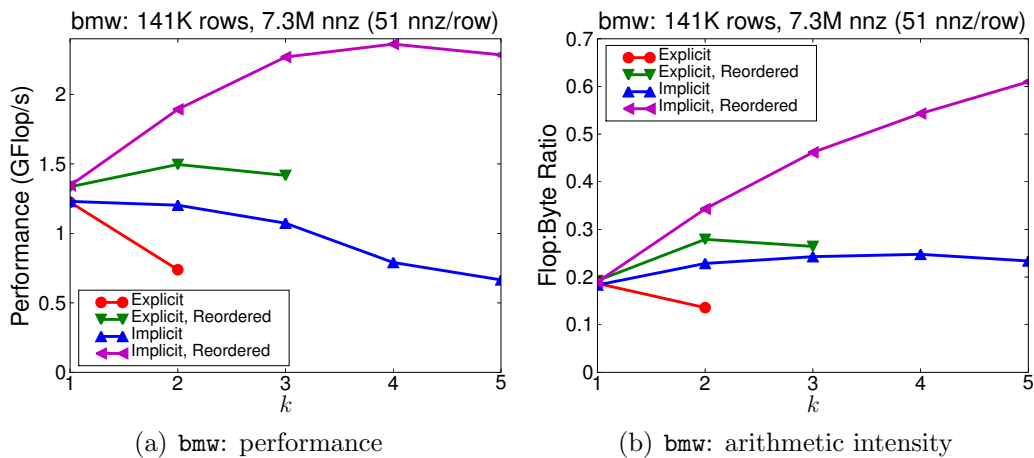
Figure 2.47: Matrix powers performance and achieved arithmetic intensity for bmw. Since the matrix doesn't partition well and its powers grow rapidly with k, implicit cache blocking was required to get performance improvements. Note that METIS was required to get a good partitioning—the plots labeled "reordered" had better performance.

(a) `cant`: performance

(b) `cant`: arithmetic intensity

Figure 2.48: Matrix powers performance and achieved arithmetic intensity for `cant`. Interestingly, METIS found a worse partition as compared to the simple strategy of scanning row and row and creating partitions.

(a) `cfd`: performance

(b) `cfd`: arithmetic intensity
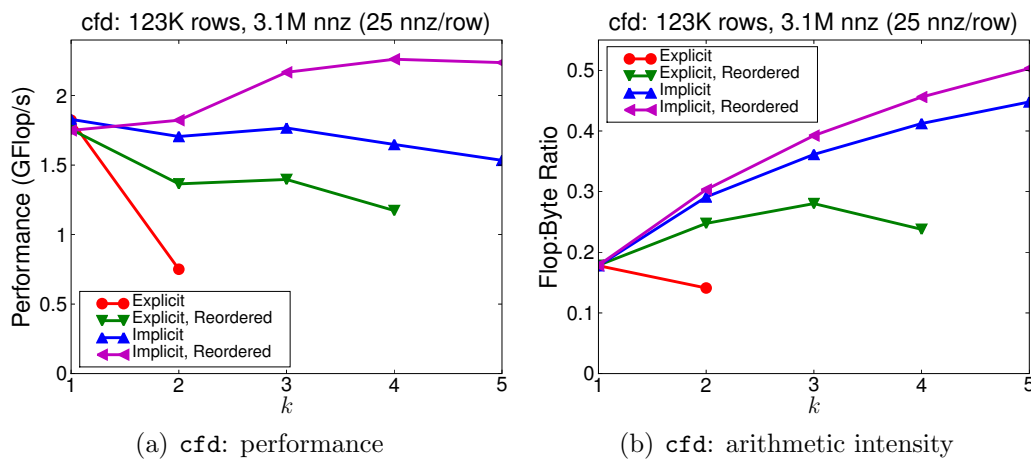
Figure 2.49: Matrix powers performance and achieved arithmetic intensity for `cfd`. The performance here degrades for all $k > 1$ and METIS along with implicit algorithm was needed to get the benefit of matrix powers.

(a) `gearbox`: performance

(b) `gearbox`: arithmetic intensity
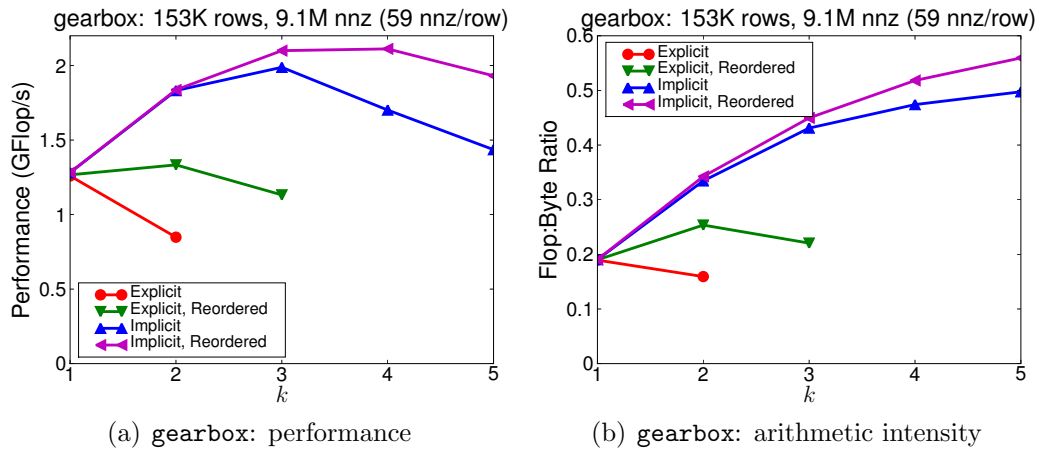
Figure 2.50: Matrix powers performance and achieved arithmetic intensity for `gearbox`. Due to the high surface-to-volume ratio for smaller partitions, the explicit algorithm performs poorly.

(a) `marcat`: performance

(b) `marcat`: arithmetic intensity
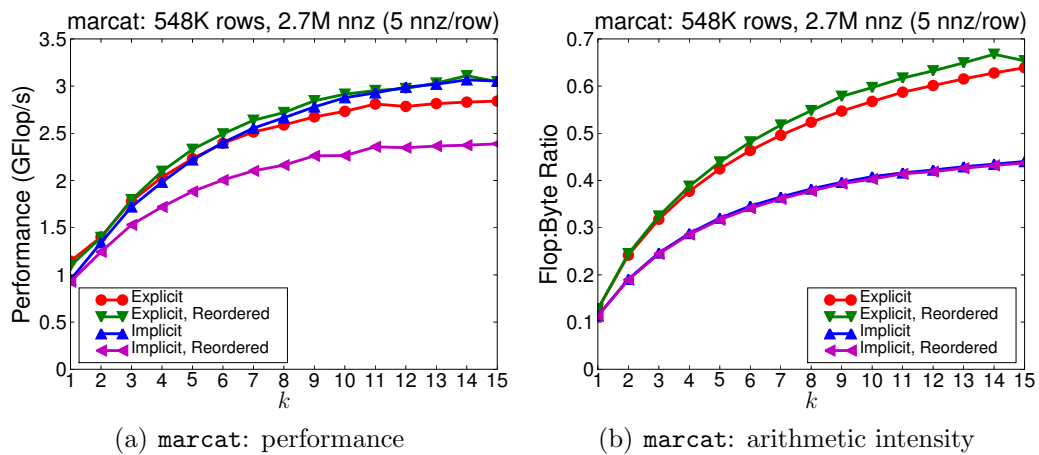
Figure 2.51: Matrix powers performance and achieved arithmetic intensity for `marcat`. This matrix partitioned well which is why the performance kept on increasing for relatively large k and the explicit algorithm worked well.

(a) `mc2depi`: performance

(b) `mc2depi`: arithmetic intensity
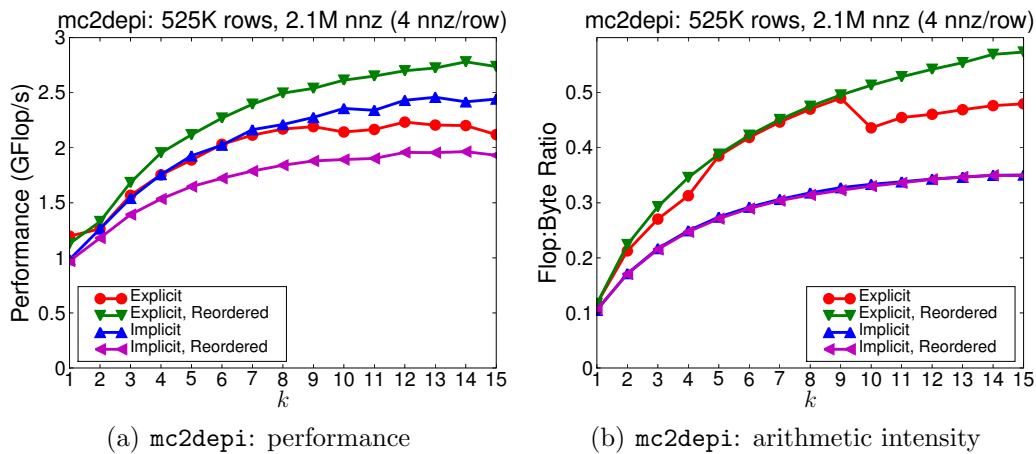
Figure 2.52: Matrix powers performance and achieved arithmetic intensity for `mc2depi`. This matrix also partitioned well with METIS which is why the performance kept on increasing until $k = 15$. The sudden drop in arithmetic intensity for $k = 10$ is due to significant increase in the number of cache blocks.

(a) `pwtk`: performance

(b) `pwtk`: arithmetic intensity
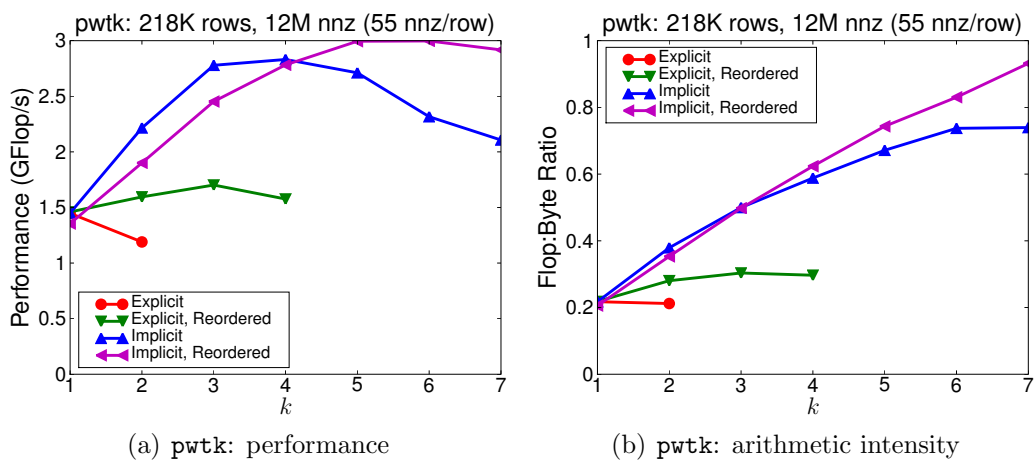
Figure 2.53: Matrix powers performance and achieved arithmetic intensity for `pwtk`. This matrix has a high surface-to-volume ratio which is why the explicit algorithm performs poorly.

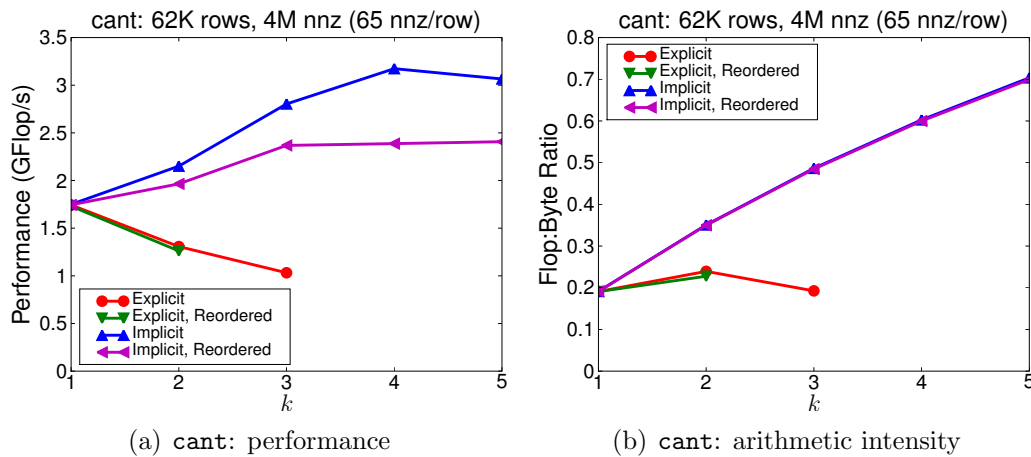(a) `shipsec`: performance

(b) `shipsec`: arithmetic intensity
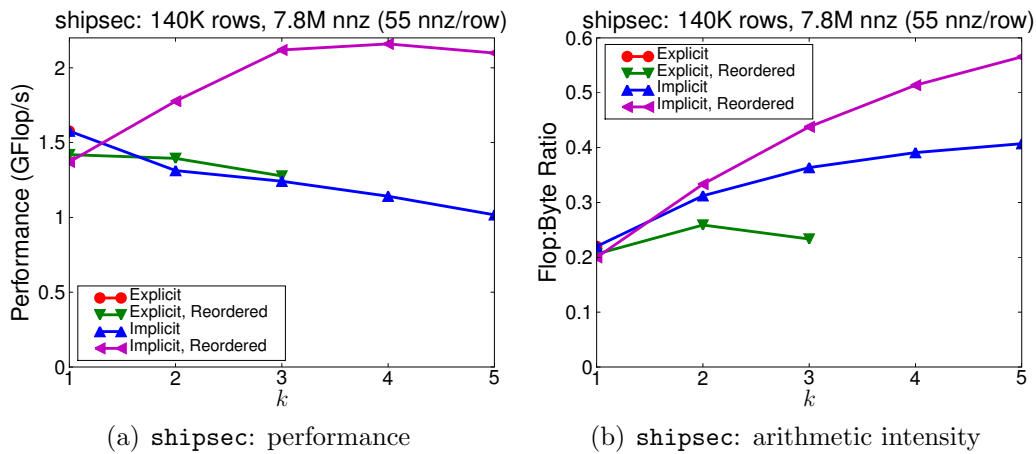
Figure 2.54: Matrix powers performance and achieved arithmetic intensity for `shipsec`. Interestingly, METIS computed a poorer partition for SpMV ($k = 1$) which resulted in decrease in performance. However, for $k > 1$, the use of METIS provided a good partitioning which improved the performance for the implicit algorithm.

(a) `xenon`: performance

(b) `xenon`: arithmetic intensity

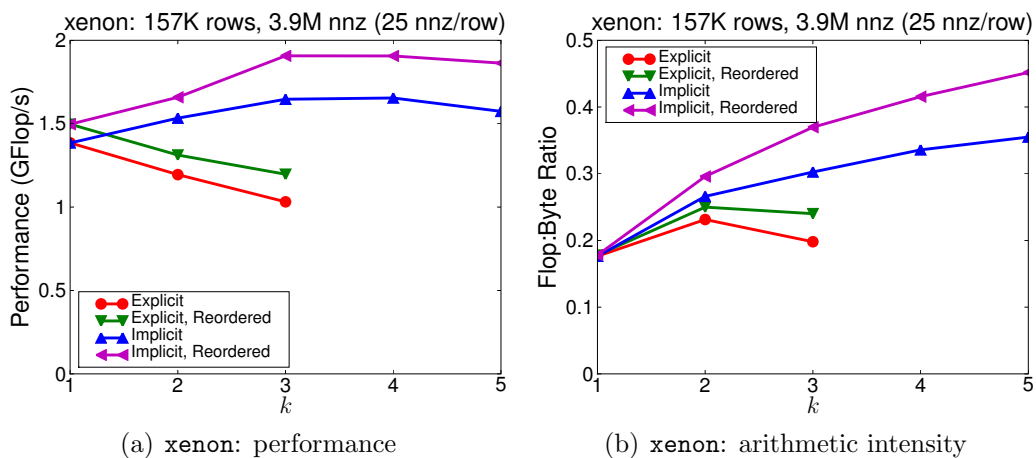Figure 2.55: Matrix powers performance and achieved arithmetic intensity for `xenon`. Just like `shipsec`, METIS reordering decreased the partition quality for $k = 1$ but provided performance improvements for $k > 1$.

2.11   Integration of Matrix Powers in GMRES

This section briefly describes our Communication-Avoiding GMRES (CA-GMRES) algorithm for iterative solution of a nonsymmetric system of linear equations $Ax = b$. It produces results that are mathematically equivalent to GMRES described in [86], but computes them differently. Nevertheless, it was designed with numerical stability in mind and converges in the same number of iterations as standard GMRES for a large suite of test problems.

As Section 2.10.3 showed, the sparse matrix structure often limits the best choice of k. Some numerically challenging problems also limit k for stability reasons. Restarting GMRES with a small k can slow or even stagnate convergence. Our CA-GMRES solves this problem by being able to continue the iteration without restarting, for multiple groups of k steps. If we perform t groups of k steps each, the resulting "CA-GMRES(k,t)" algorithm is mathematically equivalent to standard GMRES with a restart length of $m = k \cdot t$. In general, the algorithm does not require that the restart length m be a multiple of k, but we chose it this way for simplicity of explanation and implementation.

Before we describe CA-GMRES, we briefly describe the other key kernels mentioned in Section 2.1.

2.11.1   Orthogonalization

GMRES often takes as long or longer to orthogonalize the basis vectors as it does to perform the sparse matrix-vector products. Common implementations of standard GMRES orthogonalize using Modified Gram-Schmidt (MGS). However, MGS communicates (reads and writes the vectors, and passes messages between processors) a factor of k times more than the lower bound, over k iterations of GMRES [33]. Furthermore, the data dependencies in MGS GMRES force it to use the least optimal BLAS 1 version of MGS. Walker's version of GMRES orthogonalizes using Householder QR[10] [108], but this communicates about as much as MGS does. Optimization matters because for the matrices in our test suite, the runtime of LAPACK's QR factorization for matrices with as many rows as the sparse matrix and $\approx k$ columns was comparable to the runtime of our matrix powers kernel. One expects this for most sparse matrices being solved by GMRES.

Communication-Avoiding GMRES replaces traditional orthogonalization procedures with two kernels. The first is called *Block Gram-Schmidt* (BGS), and it orthogonalizes the $k+1$ basis vectors generated by the matrix powers kernel, against all previous basis vectors. The second kernel, called *Tall Skinny QR* (TSQR), makes those $k+1$ basis vectors orthogonal with respect to each other. Combined, these two kernels do the work of updating a QR factorization with new columns.

When using TSQR and BGS in CA-GMRES with a restart length of 60, TSQR and BGS treated together as an orthogonalization kernel achieved a speedup of nearly $4\times$ over

---

[10]Householder QR is an algorithm for computing the QR factorization of a matrix. It involves processing the matrix column by column, computing an orthogonal transform called a Householder reflector for each column.

the MGS orthogonalization used by standard GMRES. TSQR is also faster than LAPACK's QR factorization and better able to exploit parallelism. Both TSQR and BGS move asymptotically less data between levels of the memory hierarchy than MGS. Also, BGS consists almost entirely of DGEMM operations, unlike MGS. A further advantage is that TSQR and BGS can work with the cache-blocked format of the basis vectors in CA-GMRES without needing to copy the blocks into contiguous vectors, as would be required if using LAPACK or ScaLAPACK QR. In this regime, copying has a significant overhead.

Tall Skinny QR (TSQR) Factorization

The TSQR factorization described in this work is a hybrid of the sequential and parallel TSQR algorithms described in Demmel et al. [33]. It begins with an $m \times n$ matrix with $m \gg n$, divided into blocks of rows. In our case, each block consists of those components of a cache block from the matrix powers kernel that do not overlap with another cache block. TSQR distributes the blocks so the P processors get disjoint sets of blocks. (If running on a NUMA[11] system, this distribution can be arranged to respect memory locality.) Then, each processor performs sequential TSQR on its set of blocks in a sequence of steps, one per block. Each intermediate step requires combining a small $n \times n$ R factor from the previous step with the current block, by factoring the two matrices "stacked" on top of each other. We improve the performance of this factorization by a factor of about two by performing it in place, rather than copying the R factor and the current cache block into a working block and running LAPACK's standard QR factorization on it. The sequential TSQR algorithms running on the P processors require no synchronization, because the cores operate on disjoint sets of data. Once all P processors are done with their sets of blocks, P small R factors are left. The processors first synchronize, and then one processor stacks these into a single $nP \times n$ matrix and invokes LAPACK's QR factorization on it. As this matrix is small for P and n of interest, parallelizing this step is not worth the synchronization overhead. The result of this whole process is a single $n \times n$ R factor, and a Q factor which is implicitly represented as a collection of orthogonal operators. Assembling the Q factor in explicit form uses almost the same algorithm, but in reverse order.

Our implementation of TSQR spends most of its time in a custom Householder QR factorization that exploits the structure of matrices in intermediate steps. However, it does call LAPACK's QR factorization at least once per processor. We implemented TSQR using the POSIX Threads (Pthreads) API with a SPMD-style algorithm. We used Pthreads rather than OpenMP in order to avoid harmful interactions between our parallelism and the OpenMP parallelism found in many BLAS implementations (such as Intel's MKL and Goto's BLAS). The computational routines were written in Fortran 2003, and drivers were written in C. The TSQR factorization and applying TSQR's Q factor to a matrix each only require two barriers, and for the problem sizes of interest, the barrier overhead did not contribute significantly to the runtime. Therefore, we used the Pthread barriers implementation, although it is known to be slow [77].

---

[11]Non-Uniform Memory Access

Block Gram-Schmidt

Unlike usual Gram-Schmidt implementations, our Block Gram-Schmidt (BGS) kernel orthogonalizes the current group of $k+1$ basis vectors $\underline{V}_j$ against all the previously orthogonalized basis vectors Q at one time. It does so by computing

$$\underline{V}_j := (I - QQ^T)\underline{V}_j = \underline{V}_j - Q(Q^T\underline{V}_j)$$

Here, both $\underline{V}_j$ and Q have the same cache block layout that TSQR uses. The above operation requires two matrix-matrix multiplications: $C_j := Q^T\underline{V}_j$ involves a parallel reduction over the cache blocks, and $\underline{V}_j - QC_j$ happens in parallel with no communication. The use of BLAS 3 operations (matrix-matrix multiply) and the block structure means that BGS communicates asymptotically less than either Householder QR or Modified Gram-Schmidt. Furthermore, TSQR ensures that previous basis vectors are locally and unconditionally orthogonal to machine precision [33] within consecutive groups of $k+1$ vectors (that overlap by one vector). Even though BGS in general is numerically equivalent to the less stable Classical Gram-Schmidt orthogonalization method, using it in combination with TSQR, and using it for only a small number of outer iterations, ameliorate the potential loss of orthogonality in finite-precision arithmetic.

In this work, we only show the performance of BGS as part of the CA-GMRES solver. For the performance results, see Figure 2.56 and Table 2.6.

2.11.2 CA-GMRES

CA-GMRES replaces the sparse matrix-vector products and BLAS 1 - based Modified Gram-Schmidt orthogonalization of standard GMRES with the matrix powers kernel, and a combination of a QR factorization (either TSQR (see Section 2.11.1) or LAPACK QR) and dense matrix products (see Section 2.11.1), respectively. This means that CA-GMRES moves asymptotically less data and synchronizes asymptotically fewer times than standard GMRES; in fact, it nearly minimizes the amount of data movement. On the practical matrices we tested, CA-GMRES achieved speedups of up to $4.3\times$ over standard GMRES.

Algorithm 2.16 shows the complete CA-GMRES algorithm. For details, many other algorithms, and a mathematical analysis, see [46]. In order to make the algorithm numerically stable for larger k, one must choose the basis carefully. The obvious *monomial basis* v_1, Av_1, A^2v_1, ..., A^kv_1 used by Walker [108] becomes numerically rank deficient once k exceeds a certain threshold (see e.g., [21]). For many problems, this threshold may be small enough that it prevents us from choosing the optimal k for performance. This is because the monomial basis corresponds to the so-called "power method": the basis vectors converge to the principal eigenvector, so they get closer and closer together as k increases. Other authors suggested using a different basis to reduce the rate of increase of the basis' condition number as k increases [28, 53, 4]. When the matrix is symmetric positive definite[12], picking a good basis requires only some information about the distribution of eigenvalues. That information comes "for free," as the Krylov method itself computes estimates of the

---

[12]A matrix A is positive definite if $x^T A x > 0$ for all vectors $x \neq 0$

Algorithm 2.16 CA-GMRES algorithm

1: Begin with an $n \times n$ linear system $Ax = b$ and $n \times 1$ initial residual $r_0 = b - Ax_0$

2: $\beta := \|r_0\|_2$, $v_1 := r_0/\beta$, $q_1 := v$

3: **for** $j = 1$ to t **do**

4:      Use *matrix powers* kernel on A and $v_{k(j-1)+1}$ to compute k more basis vectors $v_{k(j-1)+2}, \ldots,$
     v_{jk+1}

5:      Let $V_j = \left[v_{k(j-1)+1}, \ldots, v_{jk}\right]$ and $\underline{V}_j = [V_j, v_{jk+1}]$

6:      Let $\underline{B}_j$ be the $k+1$ by k basis conversion matrix $\underline{B}_j$ such that $AV_j = \underline{V}_j\underline{B}_j$

7:      **if** $j = 1$ **then**

8:          $\underline{\mathfrak{B}}_j := \underline{B}_j$

9:      **else**

10:          $\underline{\mathfrak{B}}_j := \begin{pmatrix} \mathfrak{H}_{j-1} & 0 \cdot e_1 e_k^T \\ h_{j-1} e_1 e_{k(j-1)}^T & \underline{B}_j \end{pmatrix} \{h_{j-1}$ is lower right entry of $\underline{\mathfrak{H}}_{j-1}$, and $\mathfrak{H}_{j-1}$ is upper jk
         by jk submatrix of $\underline{\mathfrak{H}}_{j-1}\}$

11:          Compute $R_{1:j-1,j} := [Q_1, \ldots, Q_{j-1}]^*\underline{V}_j$ using a matrix-matrix multiplication

12:          Compute $\underline{V}_j := \underline{V}_j - [Q_1, \ldots, Q_{j-1}]R_{1:j-1,j}$ using a matrix-matrix multiplication

13:      Compute the QR factorization $\underline{Q}_j\underline{R}_j = \underline{V}_j$ using *TSQR*. Let Q_j be the first k columns of
     $\underline{Q}_j$, and let q_{jk+1} be the last column of $\underline{Q}_j$.

14:      **if** $j = 1$ **then**

15:          $\underline{\mathfrak{R}}_1 := \underline{R}_1$, and $\underline{\mathfrak{H}}_1 := \underline{R}_1\underline{B}_1 R_1^{-1}$

16:      **else**

17:          $\underline{\mathfrak{R}}_j := \begin{pmatrix} \underline{\mathfrak{R}}_{j-1} & R_{1:j-1,j} \\ 0_{k(j-1),k+1} & \underline{R}_j \end{pmatrix} \{$The new R factor of all the basis vectors$\}$

18:          $\underline{\mathfrak{H}}_j := \underline{\mathfrak{R}}_j\underline{\mathfrak{B}}_j\mathfrak{R}_j^{-1}\{$The $jk+1$ by jk upper Hessenberg matrix from jk iterations of standard
         GMRES. Here, we can exploit structure to compute this for about the same amount of
         work as if all the matrices were upper triangular.$\}$

19:      Solve the least squares problem $y_j = \text{argmin}_y \|\underline{\mathfrak{H}}_j y - \beta e_1\|_2$. The residual error $\|\underline{\mathfrak{H}}_j y_j - \beta e_1\|_2$
     is the residual error of the current GMRES approximate solution.

20:      Optionally, use y (of length jk) to compute the current approximate solution $x_j = x_0 + [Q_1, \ldots, Q_j]y_j$

---

eigenvalues that improve with the number of iterations. For nonsymmetric and particularly for nonnormal[13] matrices, the eigenvalues may not give all the information needed to pick a good basis, but in practice one can use adaptive methods that gradually increase the basis length. Furthermore, we found that the best k for performance is often much smaller than the threshold for poor numerical behavior of the basis.

2.11.3   Performance Results

Figure 2.56 shows performance results for both standard GMRES and CA-GMRES on 8 cores of the Intel Clovertown test machine. We see speedups of up to $4.3\times$ on a 1-D, three-point mesh, and of up to $2.2\times$ on more general sparse matrices from real-life problems. Table 2.6 gives the performance of each kernel in Gflop/s. For the matrix powers kernel,

---

[13]A square matrix A is normal if $A^*A = AA^*$, where A^* is the conjugate transpose of A

Matrix	SpMV	Matrix powers			MGS	TSQR	BGS
		Useful	Actual	k			
pwtk	0.66	1.35	1.58	5	1.48	6.96	6.61
bmw	0.57	1.02	1.20	5	1.44	7.74	6.52
xenon	1.15	1.59	1.87	5	2.10	7.63	6.84
cant	1.26	2.64	3.10	5	2.11	8.13	7.26
1d3pt	0.68	3.13	3.68	15	1.32	12.38	13.42
cfd	0.62	0.94	1.10	5	2.14	7.80	6.72
shipsec	0.69	1.01	1.19	4	2.07	7.38	5.86

Table 2.6: Performance in Gflop/s per kernel, for all test matrices, using 8 threads and restart length 60. Kernels SpMV and MGS belong to standard GMRES, and the matrix powers kernel as well as the TSQR and BGS kernels belong to CA-GMRES. CA-GMRES performance shown is for the best (k, t) allowed by the matrix structure such that $\lfloor \text{restart length}/k \rfloor = t$. Also shown is the corresponding k value.

we show this Gflop/s rate both for the actual floating-point operations done (including redundant computations) and for the useful operations (minus redundant computations). Thus, the ratio of "Actual" to "Useful" gives the ratio of redundant floating-point arithmetic in the matrix powers kernel.

Figure 2.57 shows performance results similar to that in Figure 2.56 but on the 8 core 2.66 GHz Intel Nehalem test machine, which has 2 hardware threads per core (16 threads in total), has NUMA, higher bandwidth than Intel Clovertown. Since Nehalem has NUMA, memory allocation was pinned to threads in order to avoid allocating to the wrong socket. Since Nehalem has a higher bandwidth, speedups are expected to be low—this effect is more pronounced on matrix powers. Overall, the speedups are less impressive, ranging from 1.3× to 4.1×.

2.12   Summary

In this chapter, we covered in detail communication-avoiding algorithms, performance models, asymptotic results, highly tuned implementations targeting distributed memory machines, out-of-core uniprocessor and shared memory multi-cores for the matrix powers kernel. In addition, the kernel was integrated in a communication-avoiding sparse solver called CA-GMRES which achieved significant performance improvement (up to 2.2× as shown in Table 2.6) over the state-of-the-art in iterative solvers for sparse linear systems.
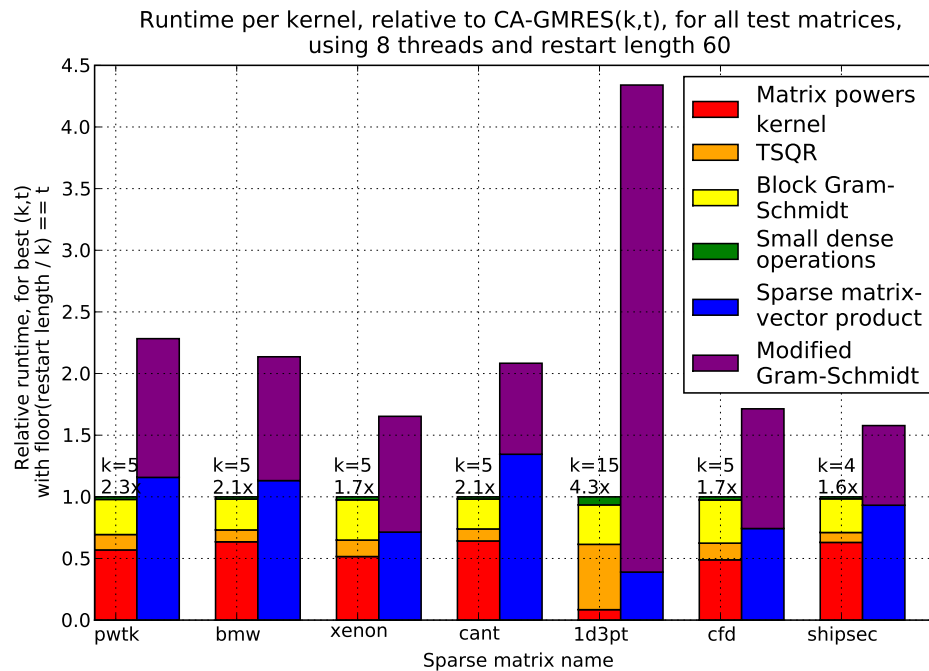
Figure 2.56: Runtime of standard GMRES and CA-GMRES on 8 cores of the Intel Clovertown test machine, on a large subset of the test problems, using the monomial basis. Both CA-GMRES and standard GMRES here use restart length 60 (so CA-GMRES uses values of k and t with $k \cdot t = 60$). Each pair of bars shows for a particular matrix, the runtime scaled by CA-GMRES runtime for that matrix: so the top of the left, CA-GMRES, bar is always one, and the top of the right, standard GMRES, bar is equal to the speedup of CA-GMRES over standard GMRES. The CA-GMRES runtime shown is for the best choice of k. The colors show runtime for the individual kernels: the "matrix powers kernel", "TSQR," "Block Gram-Schmidt" (BGS), and "small dense operations" are the parts of CA-GMRES, and "sparse matrix-vector product" (SpMV) and "Modified Gram-Schmidt" (MGS) are part of standard GMRES. TSQR runtime includes both factorization and computing the explicit representation of the Q factor. The $k = 5$ or like notation atop each CA-GMRES bar gives the choice of k achieving that runtime, and the $2.1\times$ notation below it gives the speedup of CA-GMRES over standard GMRES on that matrix.
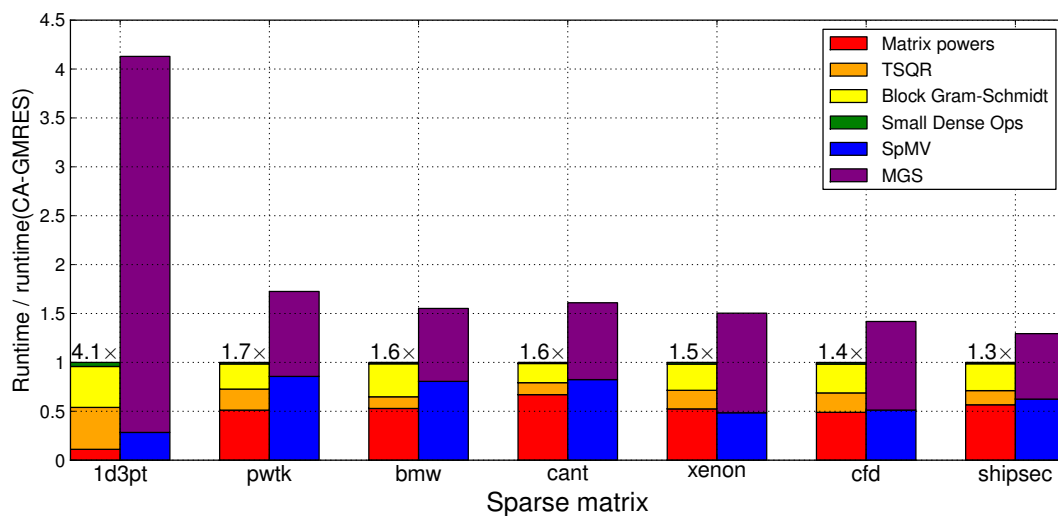
Figure 2.57: Relative runtime of standard GMRES and CA-GMRES on 16 threads of the Intel Nehalem test machine. Restart length is 60 as in Figure 2.56.

Chapter 3

Hardware/Software Co-Tuning

3.1 Introduction

Energy efficiency is rapidly becoming the primary concern of all large-scale scientific computing facilities. According to power consumption data collected by the Top500 list [101], high-performance computing (HPC) systems draw on the order of 2–5 Megawatts of power to reach a petaflop of peak performance. Furthermore, current projections suggest that emerging multi-petaflop systems are expected to draw as much as 15 MW of power including cooling. Extrapolating the current trends, the Department of Energy (DOE) E3 [91] report predicts an exascale system would require 130 MW. At these levels, the cost of electricity will dwarf the procurement cost of the hardware systems; unless the energy efficiency of future large-scale systems increases dramatically, HPC will face a crisis in which the cost of running large scale systems is impractically high. In another study on exascale feasibility study by Kogge et al. [59], the authors conclude that radical approaches for improving energy efficiency are needed to avert a power crisis or a stagnation in computing performance. Thus, an energy-efficiency crisis is already upon us. According to an August 6, 2006 article in the Baltimore Sun, the NSA has an HPC center at an undisclosed location near Baltimore that is consuming 75 Megawatts of power and growing at a substantial rate per year—causing concerns that there would not be sufficient power available for the city of Baltimore. More recently (July 6, 2009), NSA was forced to build a new facility in Utah to avert an overload of the Baltimore metro area energy grid.

Recently, the nature of silicon technology trends has changed. Device dimensions are decreasing at historical rates, but dynamic and static power dissipation have flat-lined [14, 40]. To increase circuit density, designers must also reduce clock frequency and circuit activity, or the power density of integrated circuits (ICs) will exceed practical limits. Due to power limitations, sequential performance of the last several generations of high-end processors has improved very little, and clock frequencies have remained the same, if not decreased. Moreover, commodity processors remain grossly inefficient for many scientific codes, achieving only a small fraction of their advertised peak performance [113].

Despite egregious inefficiency, commodity processors are usually the most cost-effective alternative for HPC system design. Economies of scale have enabled enormous investment

into optimization for the binary-compatible sequential codes driving the market for server and desktop processors. It is rare that a software vendor can afford to tune their software for individual microarchitectures. Viability of server and desktop software products depends primarily on on feature availability, and software development frequently takes the form of feature accretion enabled by sequential performance improvements. Thus, the reigning methodology for HPC system design begins with processor cores optimized for desktop and server applications, rather than the scientific codes which need large-scale compute resources.

Our approach in this work is inspired by embedded system design methodologies, which routinely employ specialized processors to meet demanding cost and power efficiency requirements. Leveraging design tools from embedded systems can dramatically reduce time-to-solution as well as non-recurring engineering (NRE) design and implementation cost of architecturally specialized systems.

Building a System-on-Chip (SoC) from pre-verified parameterized core designs in the embedded space, such as the Tensilica approach, enables fully programmable solutions that offer more tractable design and verification costs compared to a full-custom logic design. For this reason, we use the Stanford Smart Memories [70], which is based on Tensilica cores, as the target architecture in this work.

Given that the cost of powering HPC systems will soon dwarf design and procurement costs, energy efficiency will justify a larger investment in the original system design—thus necessitating approaches that can significantly decrease energy consumption.

General-purpose commodity microprocessors, which form the building blocks of most massively parallel systems, are grossly energy inefficient because they have been optimized for serial performance. This energy inefficiency has not been a concern for small-scale systems where the power budget is typically sufficient. However, energy efficiency becomes a concern for large-scale HPC systems, where even a few megawatts of power savings can make a dramatic difference in operating costs or even feasibility. From the perspective of an application, energy efficiency is obtained by tailoring the code to the target machine, whereas from the perspective of a machine, energy efficiency comes by tailoring the machine to the target applications. Naturally, tailoring both the hardware and software to each other is expected to achieve better energy efficiency—this is the approach taken in this work.

The novelty of our proposed methodology, illustrated in Figure 3.1, is to incorporate extensive software tuning into an iterative process for system design. Due to the increasing diversity of target architectures, software auto-tuning is becoming the de-facto optimization technique to tailor applications to target machines. As discussed in Section 1.1.3, hardware design space exploration is routinely performed to determine the right hardware design parameters for the target applications and perform sensitivity studies [75, 66, 64, 76, 60, 89, 90, 27, 49, 36, 3, 62, 52]. However, hardware DSE studies almost always use untuned benchmark codes [23, 6, 117]. In contrast, our co-tuning strategy integrates the two paradigms of hardware and software design exploration; we employ automatically tuned software to maximize the performance of each potential architectural design point. The auto-tuning methodology (discussed in Section 1.1.2) achieves performance by searching over a large space of software implementations of an algorithm to find the best
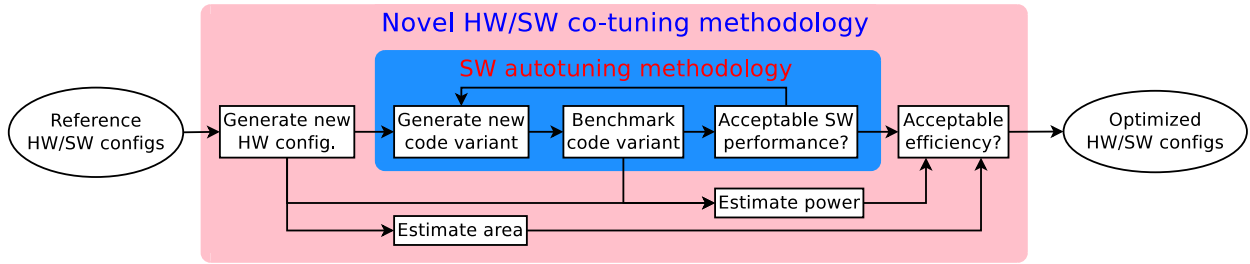
Figure 3.1: Our proposed approach for hardware/software co-tuning. In essence we have embedded a conventional auto-tuning framework within our novel-cotuning framework. As efficiency rather than peak performance is our metric of interest, we use models in conjunction with performance counters to to estimate area and power efficiency. The result is both a hardware configuration and a software implementation.

mapping to a microarchitecture [29]. Though our proposed approach may seem intuitive, this work is the first to quantify the potential benefits of co-tuning.

In this work, we demonstrate the effectiveness of our co-tuning methodology by exploring the design space of the Stanford Smart Memories [70] architecture using three of the most heavily used kernels in scientific computing: sparse matrix vector multiplication (SpMV), stencil-based computation, and general matrix-matrix multiplication (SGEMM). Our experiments examine co-tuning advantages on isolated kernels, as well as multi-kernel application experiments.

3.1.1 Motivating Examples

As a simple example, we show the effect of co-tuning the SpMV kernel[1] on a set of Smart Memories configurations. Figure 3.2 shows how performance varies with the number of cores for a specific sparse matrix. As we can see, performance scales very well for untuned SpMV but almost saturates at 2 cores for tuned SpMV since tuning makes an effective use of the available system bandwidth. Thus, if performance is the metric under consideration, it makes sense to choose the 2-core configuration rather than the 4-core configuration which would have been chosen if the untuned code was used for performance evaluation.

As another example, we consider the buoyancy loop code from a global climate modeling application [111]. The buoyancy loop code is a stencil computation[2] and, therefore, has a low computational intensity. Figures 3.3 and 3.4 show the result of co-tuning a single processor design involving one Tensilica XTensa core and the auto-tuned buoyancy loop code. We considered different hardware configurations by varying the total cache size, the associativity of the cache and the cache line size. For each hardware configuration, we auto-tuned the buoyancy loop and report the best performance. We report the impact of the cache parameters on performance and energy consumption of the buoyancy loop

---

[1]Our auto-tuner for SpMV was an adaptation of the multi-core-based auto-tuner in [113].
[2]A stencil computation operates on a regular grid and each computation on a grid point is a weighted sum of the neighboring grid points
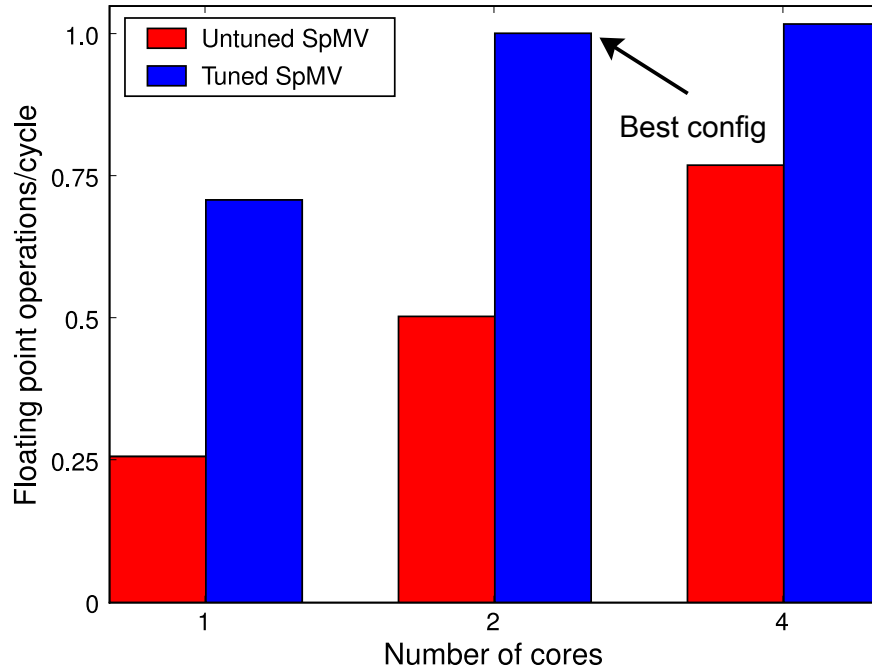
Figure 3.2: Motivating example for co-tuning. SpMV refers to the sparse matrix vector multiplication kernel from sparse linear algebra. The DRAM bandwidth is 1.6 GBytes/s and the cache sizes are 32 KBytes. The performance is normalized to flops/cycle.

computation. Figure 3.3 shows that auto-tuning can significantly improve performance for small cache sizes. The impact of tuning is small for large cache sizes as the problem can completely fit in the cache. Furthermore, Figure 3.4 shows that although larger caches have the best performance, they have a higher energy consumption as each cache access consumes more energy.

As illustrated in Figure 3.1 our co-tuning methodology is fairly simple: jointly explore the hardware and software design space by incorporating auto-tuned codes instead of benchmark codes, which perform little code adaptation. Note that, by jointly exploring the design spaces, we only increase the size of the search space, thus making the idea appear somewhat impractical due to the sheer size of the search space involved. Investigating approaches to efficiently perform these joint DSE studies is, however, beyond the scope of this work. We only demonstrate the effective of our co-tuning methodology by proof of concept studies. We also note that the joint search space exploration can be accelerated by accelerating the hardware simulation either using FPGAs (Field Programmable Gate Arrays) or using a large computing facility. We discuss FPGA-based acceleration of hardware simulation in Section 3.6.

Having stated our co-tuning methodology, the rest of the chapter discusses applications of this methodology using different examples. We describe the hardware and software setup for our co-tuning studies in Section 3.2. Section 3.3 describes the models we used for modeling the performance and energy for the different hardware components of our target system. Section 3.4 covers the metrics of importance in our co-tuning experiments. Section 3.5
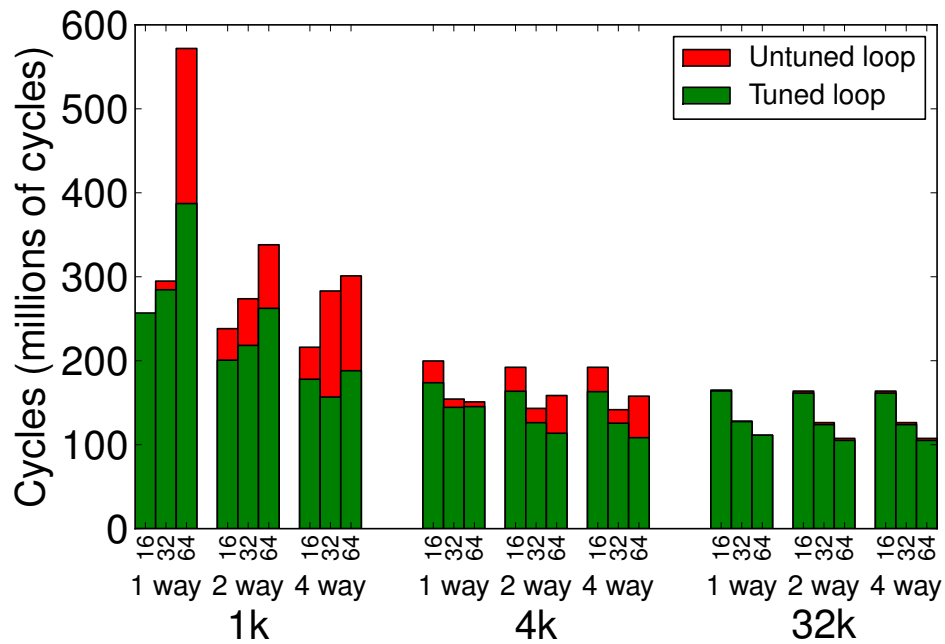
Figure 3.3: Effect of auto-tuning the buoyancy loop on performance as a function of the cache parameters. The cache line size was varied as 16, 32, 64 bytes, associativity was varied as 1, 2, 4 and total size was varied as 1K, 4K, 32K. The impact of tuning is dramatic on small caches but performance saturates when the cache is large enough.

Figure 3.4: Impact of auto-tuning the buoyancy loop on energy consumed for the processor and cache. Because performance saturates when the cache is large, energy consumption increases for larger cache because they consume more energy per access.

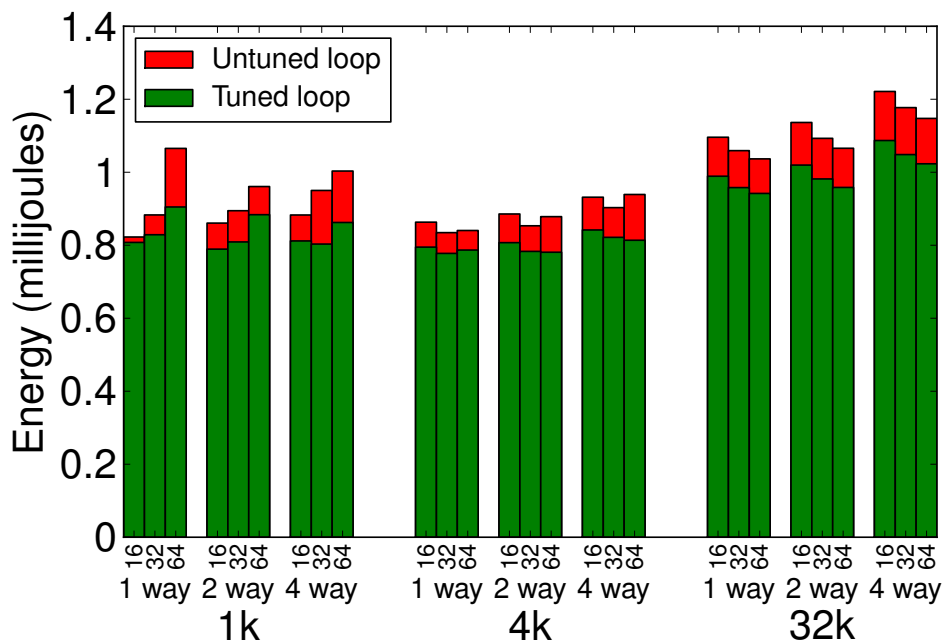| Kernel	Compulsory FLOPS	Arithmetic Intensity (Flop:Byte)	Requisite Cache/LS (Bytes)
SpMV	$2 \cdot NNZ$	< 0.5	$< 4 \cdot N$
7pt-Stencil	$8 \cdot N^3$	< 1.0	$8 \cdot XY$ (CC) $24 \cdot XY$ (LS)
SGEMM	$2 \cdot N^3$	$< \frac{B}{6}$	$12 \cdot B^2$ (CC) $20 \cdot B^2$ (LS)

Table 3.1: Numerical kernel arithmetic intensity and required cache/local-store size to attain that intensity: N is the matrix/grid dimension, B is the cache-block dimension, NNZ is the number of non-zeros, and XY is the cache-blocked plane size.

covers the results of co-tuning using the software-based simulation of the Smart Memories architecture. Section 3.6 covers the setup for FPGA-based hardware simulation, while Section 3.7 discusses the results of FPGA-based simulation.

3.2 Experimental Setup

In this section, we describe the software and the hardware setup for demonstrating our co-tuning methodology.

3.2.1 Software Setup

We describe the three numerical kernels evaluated in our study—SpMV, stencil, and GEMM—as well as the auto-tuning methodologies used to optimize their performance. These methods are at the heart of numerous important algorithms both within the scientific- and commercial-computing arena. Table 3.1 presents an overview of the kernels' arithmetic intensities (AI)—the ratio of compulsory floating point operations to compulsory DRAM traffic—and the requisite cache/local-store size required to attain that intensity. Observe that GEMM has the highest AI and is almost exclusively computationally-bound, while SpMV has the lowest AI and is primarily memory-bound. Stencil is characterized by an intermediate AI, although it is generally displays memory-bound characteristics on modern architectures [24]. Overall, these three kernels have substantially different computational requirements and memory access patterns, and would thus be best served by vastly different micro-architectural designs—making them particularly interesting for this study. Finally, note that all reported computations are in single-precision due to the constraints of the Tensilica's Xtensa processor [98] (described in Section 3.2.2).

Dense Matrix Matrix Multiplication

GEMM (General Matrix-Matrix Multiplication) is a critical dense linear algebra kernel. As a fundamental BLAS 3 routine, an efficient GEMM implementation is crucial for efficient implementation of many linear algebra codes. GEMM has a high computational intensity, and highly-tuned implementations usually achieve close to peak machine performance. Note that extensive optimization is necessary, as a naïve version is incapable of exploiting cache and register resources. Given dense matrices A, B, C and scalars α, β, the GEMM operation is $C \leftarrow \alpha \cdot op(A) \cdot op(B) + \beta \cdot C$, where $op(A)$ is either A or A^T (the transpose of the matrix A). Since all of these can be restructured as $C \leftarrow \alpha \cdot A \cdot B + C$ with little overhead, we examine only $C \leftarrow A \cdot B + C$ in this work.

The GEMM optimizations [112, 13, 44] we have implemented are blocking for cache and register file utilization, and reducing loop overhead via unrolling of the innermost loop. Register blocking operates on an in-register block of C, and streams though panels of A, and B. Cache blocking tiles all three loop nests to exploit locality. For cache-based architectures, we store matrices A and B in block-major format, and transpose A to enable unit-stride access.

Our auto-tuner operates in two phases, first determining the register blocking and loop unrolling to optimize single-core performance and then determining the cache-blocking parameters for best multi-core performance. To maximize performance we utilize the well-known ATLAS [112] code generator for the innermost kernel codes. Due to the time constraints of the software-based simulation framework, we limit our optimizations to matrices of dimension 512×512.

7-point Stencil from the Heat Equation PDE

A frequent approach to solving partial differential equations (PDE) is the iterative, explicit finite-difference method. Typically, it sweeps through the discretized space, usually out-of-cache, performing a linear combination of each point's nearest-neighbors—a *stencil*. Stencils can be used to implement a variety of PDE solvers, ranging from simple Jacobi iterations, to complex multi-grid and adaptive mesh refinement methods [10]. In this work, we examine performance of Jacobi's method to the single-precision 7-point 3D heat equation on a N^3 grid, naïvely expressed as triply nested loops ijk over:

$B[i, j, k] = C_0 \cdot A[i, j, k] + C_1 \cdot (A[i+1, j, k] + A[i-1, j, k] + A[i, j+1, k] + A[i, j-1, k] + A[i, j, k+1] + A[i, j, k-1])$.

The auto-tuner used in this work implements a subset of those described in previous investigations [24], which had proven to be extremely effective over a wide range of multi-core architectures. The work here focuses exclusively on optimizations that are relevant to the architectures within our design space: register blocking, array padding, and cache/local store blocking, including an implementation of the circular queue DMA blocking algorithm. We now briefly describe the implemented optimizations. Interested readers should refer to the prior work for more details [24].

Stencil register blocking consists of an unroll-and-jam in the X (unit-stride) and Y dimensions. This enables re-use of data in the register file, and decreases loop overheads. The
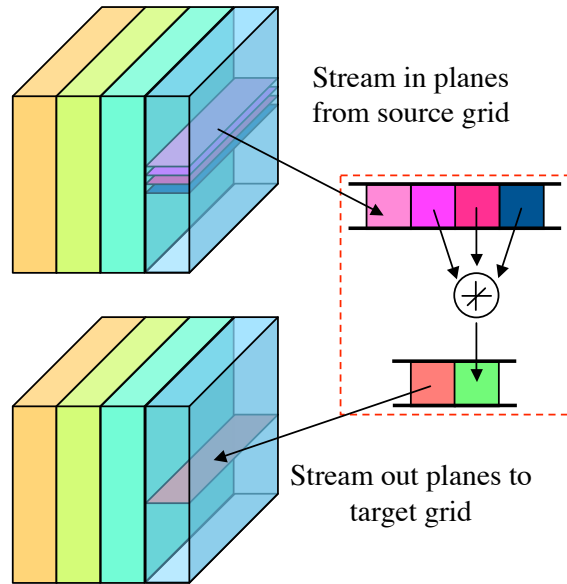
Figure 3.5: Visualization of stencil circular queue optimization for local store systems [24].

best unrolling factors balance an increase in register-pressure against decreased L1 data cache bandwidth. Array padding consists of adding a small number of dummy cells at the end of each pencil (1D row of the 3D stencil arrays), and perturbs the aliasing pattern of the arrays in set-associative caches to decrease cache conflict misses. Such an optimization avoids the need for designs with highly associative caches. The cache-blocking optimization is an implementation of the Rivera-Tseng blocking algorithm [83]. We tile in the X (unit stride) and Y dimensions, and perform a loop-interchange to bring the Z-dimension (least unit stride) loop inside of the tiled loop nests, exploiting re-use between vertically adjacent planes. For cacheless, local store-based targets with software-managed DMA, the circular-queue technique of local store management [24] is used to implement the Rivera Tiling and schedule the DMAs to overlap memory accesses with computation (see Figure 3.5).

Our approach to auto-tuning the stencil code is designed to balance coverage of the search space against the amount of simulation time required. To that end, we implement a greedy algorithm that starts with the "innermost" optimizations and works its way outward. Thus, it begins by tuning the register-block size, then tunes for the optimal array padding, and tunes for the optimal cache-block size last.

Sparse Matrix Vector Multiplication

SpMV dominates the performance of diverse applications in scientific and engineering computing, economic modeling, information retrieval, among others; yet, conventional implementations of SpMV have historically been relatively poor, running at 10% or less of machine peak on single-core cache-based microprocessor-based systems [104]. Compared to dense linear algebra kernels, sparse kernels suffer from higher instruction and storage

spyplot	Name	Dimensions	Nonzeros (nnz/row)	Description
	Dense	2K x 2K	4.0M (2K)	Dense matrix in sparse format
	FEM / Spheres	83K x 83K	6.0M (72)	FEM concentric spheres
	FEM / Cantilever	62K x 62K	4.0M (65)	FEM cantilever
	Wind Tunnel	218K x 218K	11.6M (53)	Pressurized wind tunnel
	QCD	49K x 49K	1.90M (39)	Quark propagators (QCD/LGT)
	FEM/Ship	141K x 141K	3.98M (28)	FEM Ship section/detail
	Epidemiology	526K x 526K	2.1M (4)	2D Markov model of epidemic
	Circuit	171K x 171K	959K (6)	Motorola circuit simulation

Figure 3.6: Overview of matrices used for SpMV evaluation, representing a variety of computational structures.

overheads per flop, as well as indirect and irregular memory access. Achieving higher performance on these platforms requires choosing a compact data structure and code transformations that best exploit properties of both the sparse matrix—which may be known only at run-time—*and* the underlying machine architecture. This need for optimization and tuning at run-time is a major distinction from the dense case.

Given the SpMV operation $y \leftarrow Ax$, where A is a sparse matrix, and x, y are dense vectors, we refer to x as the *source vector* and y as the *destination vector*. In a sparse matrix-vector multiplication $(y \leftarrow Ax)$ the zeros of the matrix do not contribute to the result. As such there is no value in either storing or computing on the zeros. Thus, one may remove all the zeros from a matrix. The resulting sparse matrix consists of only non-zeros.

The typical storage format associated with SpMV is compressed sparse row (CSR) (illustrated in Figure 3.7). In this format, the non-zeros of a matrix are sorted by rows, and
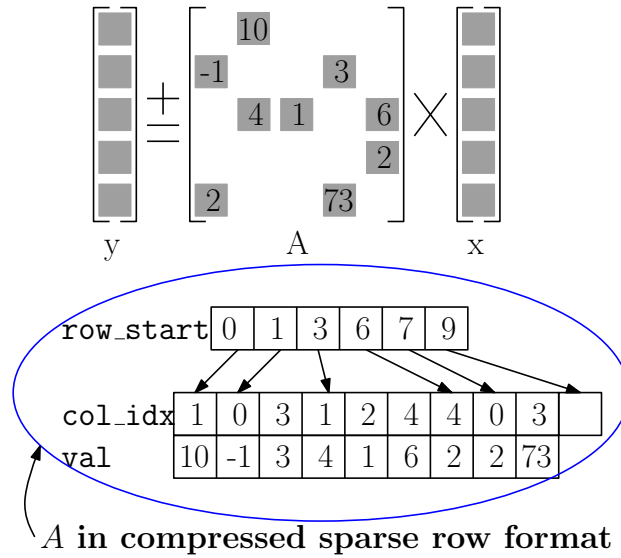
Figure 3.7: Sparse matrix data structure for CSR format

then within each row, by columns. Thus, it is likely that adjacent non-zeros are on the same or adjacent rows. A matrix stored in CSR is represented by three arrays: the non-zero values, the non-zero column indices, and a row start pointer array. The latter marks the beginning of each row within the non-zero arrays. For our purposes, we examine large single precision matrices. Thus the value array is composed of 32-bit floats, and the indices are 32-bit integers.

SpMV has a low *arithmetic intensity*, as the ratio of compulsory floating point operations to compulsory DRAM traffic. Since $A_{i,j}$ is touched exactly once per SpMV to perform a multiply-accumulate operation, and assuming the elements of x and y see good cache reuse, one expects SpMV to have an arithmetic intensity of about 0.5 flops/byte.

Our SpMV optimization approach utilizes previously established techniques [113, 104], which we describe only briefly; interested readers should refer to prior work for detailed descriptions. SpMV performance suffers primarily from large instruction and storage overheads for sparse matrix data structures and from irregular memory access patterns. Optimizations focus on selecting a compact data structure to represent the matrix and code transformations that exploit both the structure of the sparse matrix and the underlying machine architecture.

This work considers thread, cache, and register blocking, and software prefetching. On local-store based architectures, cache blocking is called local-store blocking, and prefetching becomes DMA (Direct Memory Access[3]). Thread blocking is slightly distinguished from parallelization in that the matrix is partitioned into equal-sized sub-matrices that can be individually allocated, padded, and optimized. Cache blocking exploits re-use of the source vector by tiling accesses to columns of the matrix. The tiling transformation is equivalent on local-store architectures with software-managed DMA, but a necessity rather than an op-

---

[3]DMA is a feature in modern processors which allows memory transfers to happen independently of CPU intervention. DMA enables computation to be overlapped with computation, thus improving performance.

timization. Register blocking hierarchically restructures the matrix into small dense matrices. This data-structure transformation inserts explicit zeroes into the matrix, regularizing memory access patterns, decreasing data-structure overhead, and improving computational intensity at the expense of explicit zero-entries in the matrix and potentially higher memory traffic. Our software prefetching and DMA optimizations load only the non-zero values and column index arrays. For local-store based architectures, we must explicitly load all the referenced source-vector elements, as well as the matrix data structure.

3.2.2 Hardware Setup

Our study is heavily geared towards producing area- and power-efficient designs. As such, we embrace SiCortex's and IBM's decision to utilize embedded processors for HPC systems. In this work, we used the Tensilica XTensa core primarily due to its flexibility, ease of system integration, configurability of micro-architectural resources, and of course, XTensa's target market is energy-efficient embedded systems. Secondarily, the sparing nature of an embedded processor's architecture is a much more power-efficient point of departure than other processor designs. It is clear that embedded processors are significantly more power efficient than high-performance processors. Processors like the Tensilica XTensa draw several milliwatts, and SoCs based on them typically draw at most a few Watts. Each node of a massively parallel machine based on AMD Opteron processors consumes several hundred watts. This disparity in efficiency is a part of the motivation for IBM Blue Gene's utilization of the embedded PowerPC 440 processor [107], which was originally designed for embedded use.

As a testbed for the evaluation of the myriad of different hardware configurations, our work utilizes the Smart Memories [70] (SM) reconfigurable simulator developed at Stanford for general-purpose multi-core research.

The Smart Memories Architecture

The SM architecture employs a three-level hierarchy for the on-chip network and the processor cores (illustrated in Figure 3.8(a)). The configurability of the system arises from parameters specified at each level of the hierarchy. The lowest level, referred to as a "tile", consists of processor cores connected via a crossbar to a set of SRAM blocks as well as to a "quad interface". The SRAM blocks are either caches or software-managed local stores.

The next level of the SM hierarchy groups multiple tiles into a "quad". Tiles within a quad share the logic which handles cache miss, eviction and coherence traffic, as well as DMA channels. This level of sharing reduces the number of connections to the chip-wide global network, simplifying the task of global routing in an actual VLSI implementation of an SM processor. In a cache-coherent system, the quad controller tries to satisfy a coherence request from the local tiles' caches before creating additional chip-level network traffic. Similarly, for any other intra-quad inter-tile data movement operation; for our work, this primarily means local-store DMA operations.

The highest level of the SM hierarchy is the chip-wide network. Each node in this network is either a quad, as described above, a memory controller for interfacing with off-
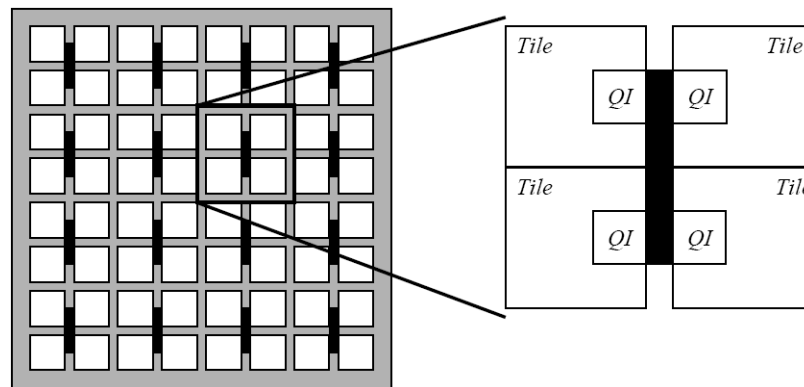
(a) Smart Memories global architecture
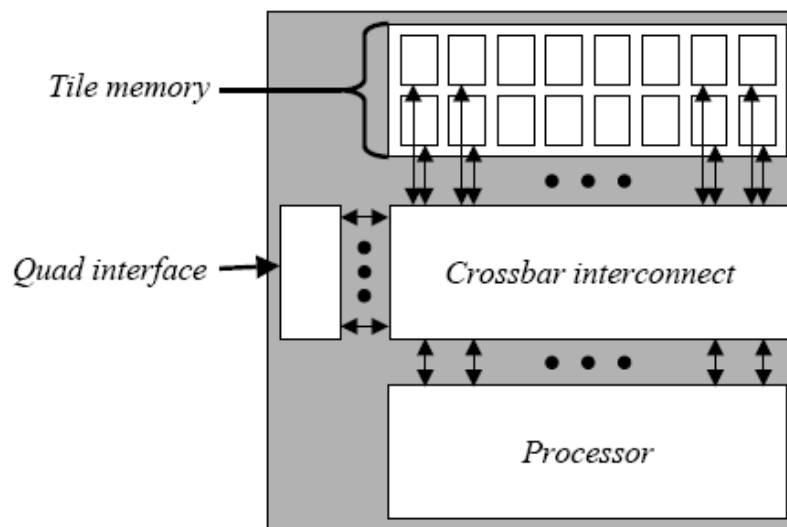
(b) Smart Memories tile architecture

Figure 3.8: Smart Memories hierarchical architecture (figures taken from [70]). A chip is composed of 'quads' in a mesh network. A quad is composed of 'tiles' sharing a 'quad interface'. A tile is composed of processors cores connected to SRAM blocks and the quad interface through a crossbar network. The SRAM blocks can be configured as local memories or caches.

chip DRAMs, or a block of configurable memory. Memory controllers each provide an independent DRAM channel. The blocks of configurable memory can either be used as L2 caches or as L2 software-managed memories. The SM architecture does not prescribe any architectural details beyond the single-chip network. In particular, it does not attempt to deal with multi-socket architectures, or system-level architectural issues.

The SM simulator was designed to simulate a wide variety of on-chip memory hierarchies for multi-core processor, and utilizes cycle-accurate simulator of the Tensilica XTensa processor for performance modeling of the individual cores. The goal of SM is functional emulation and accurate performance estimation, subject to the throughput and latency
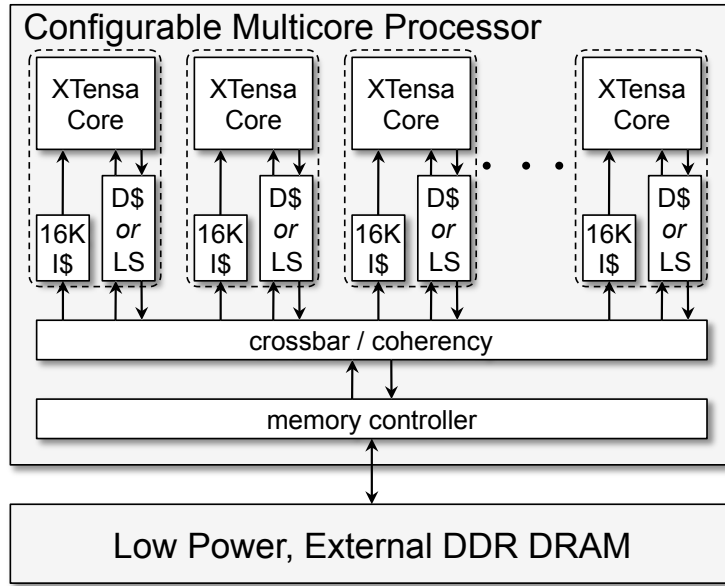
Figure 3.9: Restricted SM architecture for some number of cores. Each core has a private instruction cache and either a private data cache or a private local store.

specifications of the system configuration. Our power and area estimation methodology is presented in Section 3.4. In this work, we use the configurability of the simulator to explore an enumerated design space. Since the experiments are conducted in a software simulation environment, we have pruned this design space to reduce the amount of compute time needed. Section 3.6 discusses exploring the design space faster by using FPGA-based hardware emulation [110].

Previous studies of numerical algorithms have shown that cache hierarchy, memory system, and on-chip computational resources are crucial system design parameters for HPC architectures. Table 3.2 enumerates our hardware design space. The core architecture is a fixed 500MHz single-issue, in-order Tensilica XTensa core with a private 16KB instruction cache. The 500MHz rate is useful, as it allows the Tensilica toolchain to provide us with accurate power and area projections. We only explored 1, 4, and 16-core designs. The memory hierarchy is divided into two parts: on-chip memories, and off-chip memory. On-chip memories are either a private coherent caches (CC) per core or a private disjoint local stores (LS) per core. We fixed cache associativity as 4-way and line size is 64 bytes. For caches and local stores we explore four different capacities. All cores use the same design—there is no heterogeneity. Off-chip memory is abstracted as a uniform memory access DRAM running at one of three different possible bandwidths. Figure 3.9 shows a simpler restricted view SM architecture from the viewpoint of our co-tuning study. We limit the SM designs to one quad per chip.

Component		Parameters	Explored Configs
cores	**in-order XTensa core**	Issue width	**single-issue**
		Frequency	**500 MHz**
		Number of Cores	1, 4, **16**
		Inst. Cache (per core)	**16 KB**
Memory Hierarchy	**Coherent Data Caches**	Capacity (per core)	16, 32, 64, **128 KB**
		Associativity	**4 way**
		Line size	**64 Bytes**
	Local Store	Capacity (per core)	16, 32, 64, 128 KB
	External DRAM	Bandwidth	0.8, 1.6, **3.2 GB/s**
		Latency	**100 core cycles**

Table 3.2: Hardware parameters explored in co-tuning architectural-space exploration. The parameters corresponding to baseline (untuned) hardware configuration are in boldface. Note that data cache and local store designs are mutually exclusive.

3.3   Modeling Performance and Energy

We now describe the power and performance models for the different hardware components.

3.3.1   Modeling Chip Power

The power estimation is based on the model used in [64] by weighting a number of key architectural events counted by the software simulator with appropriately modeled or derived energies weighted by the total execution time. Energy for events originating in the cores are derived using the energy estimates from the Tensilica tools. The effect of clock gating is taken into account by a reduced power consumption when the core is stalled (assumed to be 10% of peak power). The dynamic energy for the caches and local stores is modeled on a per transaction basis using a CACTI 5 [99] model. On-chip network energy is calculated based on the total on-chip network traffic and using the scaled energy numbers from [64]. Finally, leakage power is assumed to be 15% of peak power for any configuration. Although, every software implementation for a given hardware design will yield different power estimates, this model allows us to explore variations in the constants without having to resimulate the design.

3.3.2   Modeling Chip Area

The area of a given processor configuration is an important metric due to its effect on the end cost of fabricating and packaging the circuit. To this end, we model the hardware configuration area within the design space, assuming 65nm node technology for the core chip. Core area estimates are provided by the Tensilica toolchain, while CACTI 5 [99] was used to model cache or local store area. To mitigate the effect of area on yield, we assume

sparing is used for increasing yield—one spare core is assumed for chips with up to eight cores, and two spare cores are assumed for chips with 16 cores. Each Tensilica cores is extremely small when compared with modern high-performance microprocessors—less than 0.5 mm^2. As such, we expect such a sparing strategy to have very high die yields and a yield percentage that is effectively independent of the chip area. In essence, the resultant yield-adjusted chip costs should be roughly linear with core chip area. We assume that the on-chip network and clocking add another 20% to the core chip area.

3.3.3 Modeling DRAM

We modeled DRAM energy using the current profiles from the Micron datasheets [72] for a 256-MB-DDR2-400 memory module. Given the low power nature of the Tensilica cores, DRAM power is a substantial component of total power. For modeling DRAM area, we assume that DRAM area is 3× more cost efficient than chip area. For this reason we assume that the memory interface and DIMMs contribute a constant 35 mm^2 to the total area. Since this is substantial compared to the area of a Tensilica core, which is less than 0.5 mm^2, and caches (each 128 KB cache is less than 1 mm^2), we include this area in our calculations. As such, there is a clear economy of scale by incorporating many cores.

We note that vector systems, such as the Cray X2 and NEC SX-9 are able to achieve substantially better computing efficiency by organizing DRAM to support word-granularity data accesses. However, because of the internal page-oriented organization of internal DRAM architecture, support for such access patterns requires over-provisioning of DRAM bandwidth, with commensurate increase in power loads. So although we find massively bank-switched memories of these systems enable much higher computational efficiency, the power consumed by DRAM makes this approach uncompetitive in energy efficiency when we tried to include this approach in our study. Therefore, we have focused on primarily conventional DRAM organization that supports cache-line granularity accesses and did not include word-granularity access in this study.

Table 3.3 lists the formulas for computing the area and energy numbers for the different hardware components.

3.4 Evaluation Metrics

Our area of focus is parallelized scientific applications running on large-scale, energy-efficient, *throughput*-oriented high-performance systems consisting of tens of thousands, if not millions, of individual processing elements. Obtaining enough power for such systems can obviously be an impediment to their adoption. Thus, achieving high performance when designing such machines is less dependent on maximizing each node's performance, but rather on maximizing each node's power efficiency. Moreover, large silicon surface area can be expensive both from a fabrication cost, and also in its impact on mean time between failures (MTBF). Thus, our design methodology focuses on two key optimization metrics: *power efficiency*—the ratio of attained MFlop/s per chip to chip power, and *area efficiency*—the ratio of attained MFlop/s per chip to chip area. In essence, power efficiency is a measure

	Formula
Core energy (E_{core})	$\sum_{i=1}^{N_{cores}} \left(P_{core}^{exec} \cdot Cyc_{exec\_core\_i} + P_{core}^{stall} \cdot Cyc_{stall\_core\_i} + P_{core}^{leak} \cdot Cyc_{total} \right)$
FPU energy (E_{fpu})	$\sum_{i=1}^{N_{fpus}} \left(P_{fpu}^{exec} \cdot Cyc_{fpu\_i\_busy} + P_{fpu}^{leak} \cdot Cyc_{total} \right)$
Cache energy (E_{cc})	$\sum_{i=1}^{N_{caches}} \left(E_{cc}^{read} \cdot N_{reads\_cache\_i} + E_{cc}^{write} \cdot N_{writes\_cache\_i} + P_{cc}^{leak} \cdot Cyc_{total} \right)$
Local store energy (E_{ls})	$\sum_{i=1}^{N_{caches}} \left(E_{ls}^{read} \cdot N_{reads\_ls\_i} + E_{ls}^{write} \cdot N_{writes\_ls\_i} + P_{ls}^{leak} \cdot Cyc_{total} \right)$
Network energy (E_{nw})	$E_{byte} \cdot N_{bytes\_transferred}$
DRAM energy (E_{dram})	$E_{read} \cdot N_{read\_accesses} + E_{write} \cdot N_{write\_accesses} + P_{dram}^{leak} \cdot Cyc_{total}$
Total energy (E)	$E_{core} + E_{fpu} + E_{cc} + E_{ls} + E_{nw} + E_{dram}$
Total power (P)	$E \cdot frequency/Cyc_{total}$
Total chip area (A)	$(A_{core+fpu} + A_{D\$} + A_{I\$} + A_{LS}) \cdot (N_{cores} + N_{spares}) \cdot (1 + ovrhd) + A_{dram}$
Power efficiency (PE)	$flops/s/P$
Area efficiency (AE)	$flops/s/A$

Table 3.3: Expressions for power efficiency and area efficiency calculations. Cache/local store energy numbers like $E_{cc/ls}^{read/write}$ and $P_{cc/ls}^{leak}$ depend on the cache/local store parameters like size, linesize, etc., and are computed using CACTI [99]. We scale the DRAM energy components when bandwidth is scaled—this is assumed to be achieved through a combination of frequency scaling and bank interleaving.

of number of tasks completed per unit of energy be it Joules or kWh.

Given these metrics, one can impose either a per-node or per-supercomputer chip power (or area) budget and estimate the resultant attainable performance: minimum of area efficiency × chip area budget and power efficiency × chip power budget. Systems with limited power budgets should be selected based on power efficiency, whereas systems with limited silicon budgets should be selected based on area efficiency. Identifying the trade-off between the two allows a designer to balance the system acquisition costs, system power requirements, and system up time. It is important to note that further gains in power efficiency can be realized by optimizing all the system components (in addition to chip power)—this will be focus of future investigations.

We anticipate most of the growth in parallelism for future systems will be primarily within a computational "node". For a fixed aggregate MPI performance level, the range of practical core counts differs by only one order of magnitude whereas the number of nodes in a given system is likely to differ by a far smaller amount. The challenge of our decade is to make more efficient use of explicit parallelism within the node to extract a strong scaling speed-up in lieu of continued clock-frequency scaling. This will require better abstractions for fine grained parallelism within the node as well as more efficient computational elements within the node. As such, we may focus only on single node performance.

3.5 Software-Based Simulation Results

This section describes the results of co-tuning using the Smart Memories multiprocessor. A software-based simulator for the Smart Memories multiprocessor was used. Before delving into the experimental results, we briefly reiterate our novel co-tuning approach shown in Figure 3.1: for each of the three kernels, and for each processor configuration in our search space, the kernel is auto-tuned on that hardware configuration to find the software implementation that maximizes performance. Therefore, given a hardware configuration, we always report data corresponding to the best performance achieved by the auto-tuner on it. Note that while our auto-tuners heuristically prune their own search spaces of tuned software implementations, we explore the hardware configuration space exhaustively by running the auto-tuners on all the configurations within our design space. An application of co-tuning for a real system, however, would use a more efficient search strategy to explore a much larger hardware design space. For the purpose of this work, though, exhaustive search suffices as the hardware design space is small and serves well to illustrate the effectiveness of co-tuning.

We now quantify the effectiveness of our co-tuning methodology for a variety of hardware and software optimization strategies on each of the three numerical kernels. We commence with a study of the relationship between architectural configuration and attained performance. Next, we measure the potential improvements in kernel power and area efficiency using our co-tuned design space exploration. Finally, we analyze the benefit of co-tuning for applications dominated by a varying mix of these three kernels.

3.5.1 Performance of Design Parameters

We now explore the per kernel performance response to changes in both software optimization and processor configuration. Doing so provides insights into the inherent hardware requirements and attainable efficiencies for each kernel as well as quantifying the importance of the instantiation of the auto-tuning component within the co-tuner. Not only do these simulation results, shown in Figure 3.10, follow our intuitions regarding the expected performance of our kernels under varying architectural conditions, they also serve to validate the underlying simulation environment.

Auto-tuning Figures 3.10(a–c) show the performance benefit of auto-tuning for SpMV, Stencil, and SGEMM using a fixed memory bandwidth of 1.6 GB/s and either a 64 KB data cache or a 64 KB local store for 1, 4, or 16 cores. The stacked bars indicate the improvement of our software optimizations.

Observe that due to SpMV's constant and low arithmetic intensity, with enough cores, SpMV performance plateaus for both the tuned and untuned code versions using either caches or local stores. In effect, the changes in processor configuration transitioned its behavior from compute-bound to memory-bound. Note that for smaller concurrencies, the untuned DMA-based implementations outperform the cache coherent versions by a factor of $2\times$. Such a situation arises because the DMA version utilizes block transfers, which represent
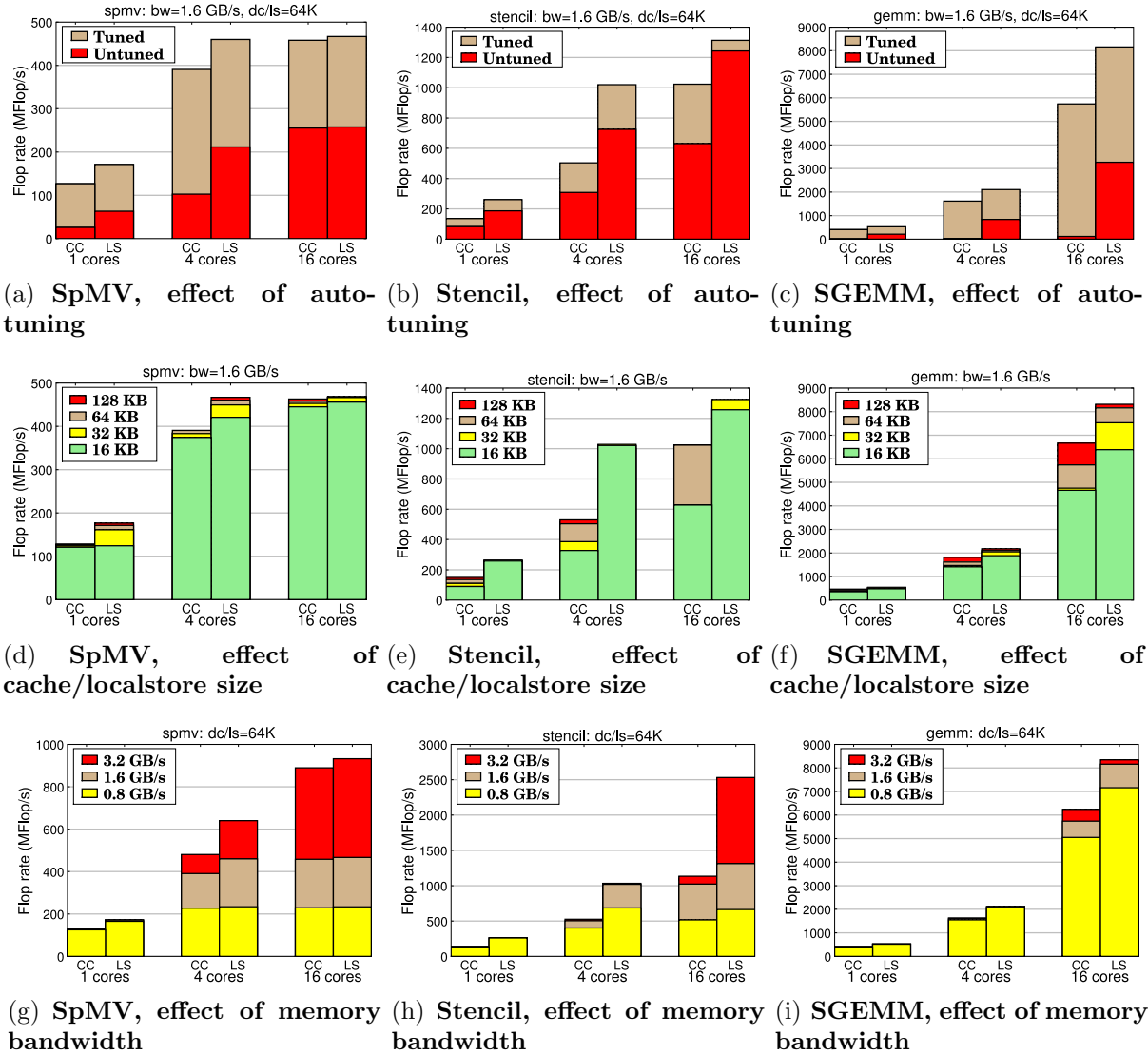
Figure 3.10: The interplay between processor configuration and auto-tuning for the SpMV, Stencil, and SGEMM kernels. Note: 'LS' indicates DMA-managed local store architectures, and 'CC' indicates coherent-cache systems.

a means of more easily satisfying Little's Law [5] and mandates a reuse-friendly blocked implementation for correctness. Nevertheless, the SpMV auto-tuner provides significant benefit even on bandwidth-limited configurations. This class of SpMV auto-tuner attempts to both minimize memory traffic and express more instruction-level-parallelism. The results reaffirm the significant impact of auto-tuning shown previously shown on numerous multi-core architectures [113, 29, 24].

The stencil code is both moderately more arithmetically intense than SpMV, and also contains more regularity in its memory access pattern. Figure 3.10(b) demonstrates that, relative to SpMV, the higher arithmetic intensity forestalls the advent of a memory-bound

processor configuration. Thus, as applications shift their focus from SpMV to stencil-like kernels, they may readily exploit more cores. Most interesting, the quad-core local store version attains the performance comparable to the 16-core cache-based implementation. In effect, DMA transfers eliminate superfluous write allocate traffic and express more memory-level parallelism. The incorporation of effective prefetching into the cache-based stencil auto-tuner might mitigate the latter. Finally, the tuned local store stencil code can utilize a large portion ($\sim$75%) of memory system with four cores; hence the performance improvement is limited to about 30% when quadrupling the number of cores to 16. With 16-cores, even the untuned DMA-based code is nearly able to saturate the memory system.

SGEMM has a high arithmetic intensity arithmetic hierarchically limited by the register and cache capacities. Thus, it alone among our kernels is capable of exploiting increasing numbers of cores (and cache capacities). Figure 3.10(c) shows the performance of SGEMM scales linearly with the number of cores, for any processor configuration both with and without auto-tuning. This does not imply all configurations delivered the same performance or benefits. To the contrary, auto-tuning was essential on cache-based architectures; improving performance by 64$\times$, but only provided a moderate speedup on the already well-blocked local store implementations. Moreover, the local store configuration consistently outperformed the cache configurations. The naïve code incurs significant cache conflict misses for large matrices, especially when the matrix dimensions are powers of 2— the common case in our experiments. Furthermore, the latency penalty of a cache miss is high due to the absence of an L2 cache in our configurations. Due to SGEMM's hierarchal arithmetic intensity, the effects of inner-loop code generation and blocking for data reuse are extremely important. In contrast to the stencil and SpMV codes, even our most highly optimized SGEMM implementations are not significantly limited by memory bandwidth.

On-chip Memory Capacity   Figures 3.10(d–f) quantify performance response to changes in core count and per-core memory capacity for the auto-tuned codes. We show both the cache-based and DMA-based codes for each of 1, 4, and 16 cores. Although the cache-based configurations can be more sensitive to cache size compared with the local store versions— since it is harder to control blocking and data movement via scalar loads and stores— performance is relatively insensitive to cache and local-memory sizes. SpMV performance hardly changes at all, as the smallest cache size is enough to exploit re-use of the two vectors. The cache-based stencil code sees about 60% performance improvement as the cache size increase from 16 KB to 64 KB. However, the explicitly-blocked, DMA-based stencil code can exploit nearly all temporal locality using the smallest local memory. SGEMM on 16-core systems benefits from increased cache and local memory sizes due to memory bandwidth contention; the larger caches enable larger block sizes and reduce pressure on memory bandwidth—i.e. higher arithmetic intensity.

Memory Bandwidth   Figures 3.10(g–i) show performance as the processors' memory bandwidth is changed. Clearly, for SpMV and stencil, increasing the number of cores is only viable when the memory bandwidth is similarly increased since they are ultimately memory limited. This effect is less pronounced for Stencil due to its higher arithmetic intensity.

SGEMM, on the other hand, only begins to show the limitations of memory bandwidth with 16 cores.

3.5.2 Tuning for Power and Area Efficiency

Having established raw performance characteristics, we now examine the power and area efficiency of our methodology. Figure 3.11 plots these efficiency metrics (as defined in Section 3.4) for our three test kernels. Each point in the scatter plot represents a unique processor configuration, with yellow circles, green triangles, and red triangles corresponding to auto-tuned cache, untuned cache, and (either auto-tuned or untuned) local store versions respectively. Additionally, a circle is drawn to highlight the configurations with the best power or area efficiencies.

These figures serve to demonstrate the extreme variation in both efficiency metrics spanned by the design points within our configuration search space. Figure 3.11(a) shows that a poor choice of hardware can result in as much as a $3\times$ degradation in power efficiency for SpMV (MFlops/s per Watt), whether software tuning is employed or not. Figure 3.11(b) shows that for stencil, the difference is nearly $8\times$. For SGEMM in Figure 3.11(c), this difference is nearly two orders of magnitude! Since the operational cost and performance ceiling of future HPC facilities are limited by the power consumption of compute resources, these results quantify the potential impact of an energy-efficient design and hold the promise of reducing petascale system power by several megawatts.

We now measure the potential effectiveness of our combined hardware/software tuning methodology. Performance is explored in the context of four configurations: untuned software on the fastest processor configuration, auto-tuned software on the fastest processor configuration, tuned hardware running untuned (fixed) software, and co-tuned hardware/software. This serves to differentiate the efficiency gains from tuning software and hardware individually from the efficiency gains of co-tuning.

Out-of-the box: Untuned Software on the Fastest Processor Configuration   Our lowest baseline comparison is the conventional wisdom strategy of choosing a system design by using the most powerful hardware configuration. We do not tune the software running on these processors. The most powerful hardware configuration within our search space is a coherent-cache chip multi-processor with 16 cores, 128 KB of L1 data cache per core, and 3.2 GB/s of main memory bandwidth. While local store architectures generally provide better performance, it is impossible to produce untuned codes to utilize them. As this comparison represents putting essentially no effort into the system design, it is highly unlikely to be a viable power- or area-efficient solution. Rather, we present it as a point of comparison to illustrate how much efficiency our coupled hardware/software design space exploration provides. Table 3.4 presents an overview of the optimal power and area efficiency data for each optimization strategy (including the improvement impact of co-tuning), starting with this baseline configuration, shown in the fourth column. Observe that our co-tuning methodology would deliver 3.2–$80\times$ better power and area efficiencies for our evaluated kernels.

(a) **SpMV area vs. power efficiency**

(b) **Stencil area vs. power efficiency**
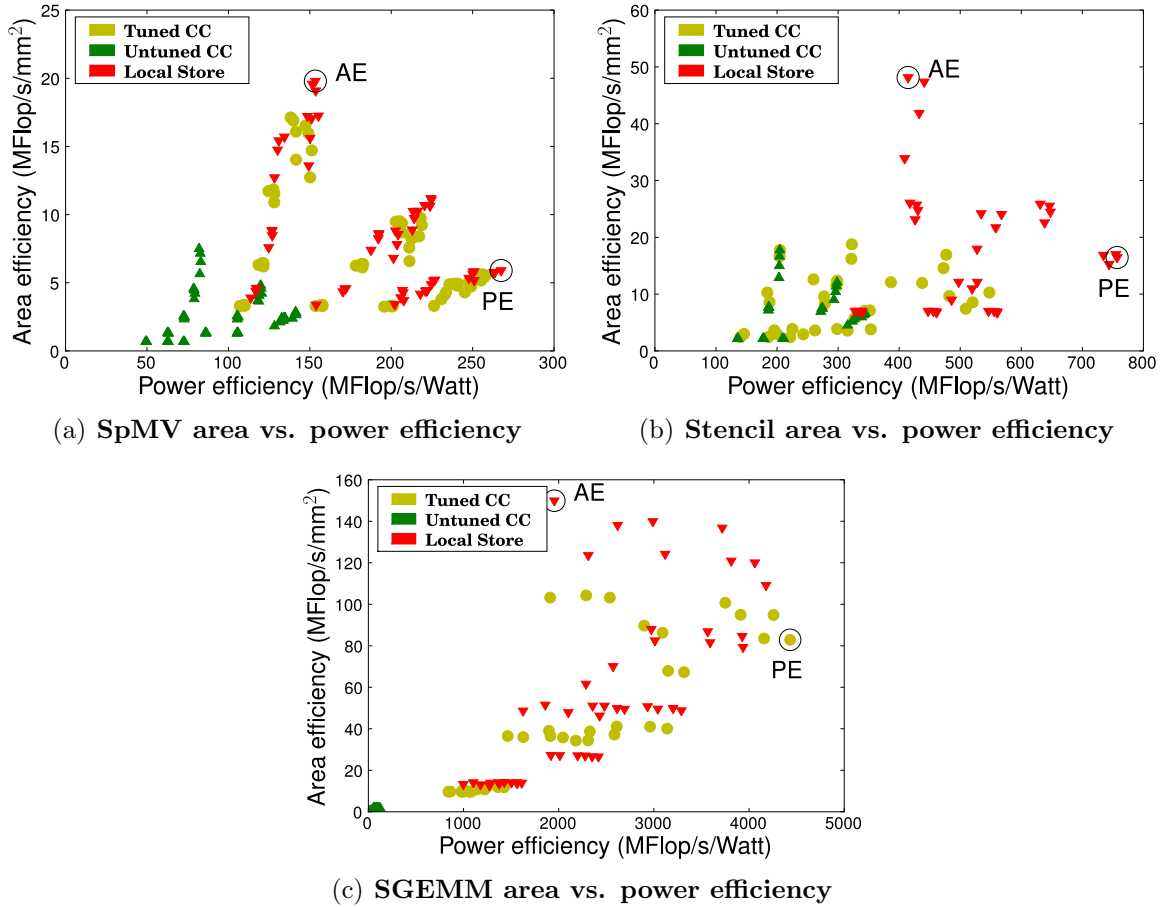
(c) **SGEMM area vs. power efficiency**

Figure 3.11: Area efficiency vs. power efficiency for each of the three Kernels. 'AE' and 'PE' denote the most area- and power-efficient configurations respectively.

Auto-Tuned Software on the Fastest Processor Configuration In order to differentiate the efficiency that the current state-of-the-art provides, we present the result of auto-tuned software on the fastest hardware. This combination is analogous to building a system from high-performance commodity cores and utilizing auto-tuning for software optimization—an increasingly common solution. The hardware provides as much of each architectural resource as our design space allows, but without having been specifically tailored to any specific kernel. The auto-tuned kernels exploit those resource to maximize performance. The fifth column (auto-tuned SW, fastest HW) of Table 3.4 shows the optimal power and area efficiency produced by this approach.

Since the hardware configuration is the same as the untuned SW on fastest HW, the efficiency gains correspond roughly to improvements in attained floating-point performance through auto-tuning. These ratios are different for power and area efficiency, since power depends on the activities of the various architectural resources, while area depends only on their physical quantity. Comparing the fourth and fifth columns of Table 3.4 shows that SGEMM's auto-tuning achieves an impressive 54× and 53× improvement in power and

area efficiency (respectively), due to the enormous performance impact of auto-tuning on this kernel. For Stencil and SpMV, the improvement is not as spectacular, but nonetheless substantial: Auto-tuning improves Stencil's power efficiency by 1.5× and its area efficiency by 1.2×, while SpMV benefits by 1.8× in power efficiency, and 2.2× in area efficiency. These results reiterate the conclusions of prior works [113, 29, 24] that auto-tuned codes can outperform compiler-only optimizations by a wide margin.

Design Objective	Co-Tuned Kernel	Metric	Untuned SW Fastest HW	Auto-Tuned SW Fastest HW	Untuned SW Tuned HW*	Co-Tuned SW/HW
Power Eff.	SpMV	MFlop/s	397	895	127	229
		Power (W)	4.8	6.0	0.9	0.9
		Power Eff.	82.4	150.2	141.6	267.5
		Co-Tuning Adv.	**3.2x**	**1.7x**	**1.9x**	—
	Stencil	MFlop/s	906	1139	262	686
		Power (W)	4.5	3.5	0.8	0.9
		Power Eff.	203.2	321.9	344.5	756.9
		Co-Tuning Adv.	**3.7x**	**2.4x**	**2.2x**	—
	SGEMM	MFlop/s	132	7079	122	5823
		Power (W)	1.9	1.9	1.0	1.3
		Power Eff.	68.7	3750.5	124.7	4431.4
		Co-Tuning Adv.	**65x**	**1.2x**	**36x**	—

| Design Objective	Co-Tuned Kernel	Metric	Untuned SW Fastest HW	Auto-Tuned SW Fastest HW	Untuned SW Tuned HW*	Co-Tuned SW/HW
Area Eff.	SpMV	MFlop/s	397	895	390	897
		Area (mm^2)	70.3	70.3	52.0	45.3
		Area Eff.	5.8	12.7	7.5	19.8
		Co-Tuning Adv.	**3.5x**	**1.6x**	**2.6x**	—
	Stencil	MFlop/s	906	1139	923	2502
		Area (mm^2)	70.3	70.3	52.0	52.0
		Area Eff.	12.9	16.2	17.9	48.1
		Co-Tuning Adv.	**3.7x**	**3x**	**2.7x**	—
	SGEMM	MFlop/s	132	7079	110	8173
		Area (mm^2)	70.3	70.3	52.0	55.0
		Area Eff.	1.9	100.7	2.1	149.9
		Co-Tuning Adv.	**80x**	**1.5x**	**70x**	—

Table 3.4: Summary of optimal power-efficiency (in MFlops/s/Watt) and area-efficiency (in MFlops/s/mm^2) data (and relative improvement of co-tuning) for each optimization configuration.*The hardware configuration space for 'Untuned SW, Tuned HW' only includes coherent-cache based configurations.

Untuned software, Tuned Hardware We now examine the effect of hardware design space exploration alone, without the benefit of software auto-tuning. Note that we omit local store-based configurations in the hardware tuning design space for this case. This is because our so-called "untuned" kernels on local store-based configurations are cognizant of both the local store capacity, as well as the locality inherent in the algorithms. A truly untuned (architecturally and algorithmically agnostic) local store code would doubtlessly achieve significantly lower performance. The green triangles in the scatter plots in Figure 3.11 represent the range of efficiencies that hardware-only tuning achieves, while the sixth column (untuned SW, tuned HW) in Table 3.4 shows the efficiencies of the Pareto-optimal hardware configurations.

Looking at the sixth column of Table 3.4 shows that simply tuning over the hardware space improves both power and area efficiency for all our kernels. SpMV, stencil and SGEMM achieve power efficiency improvements of approximately $1.7\times$, and area efficiency gains of $1.3\times$, $1.4\times$, and $1.1\times$ (respectively), when compared the untuned SW/HW. Note that since we are optimizing for power and area efficiency, the attained floating-point performance is lower compared with the untuned SW/HW case. Examining the fifth and sixth columns of Table 3.4 shows that even after searching over the hardware design space, we can still under-perform even on the most powerful hardware configuration, if auto-tuning software is not employed. This difference is quite dramatic for SGEMM ($30\times$ and $47\times$ lower power and area efficiency) because it benefits tremendously from tuning, especially for the specific matrix dimensions used in our experiments.

Hardware/Software Co-Tuning Our hardware/software co-tuning methodology performs software auto-tuning for each potential point in the hardware design space, and thus represents a more complete coverage of the overall system design space than the other examined approaches. Each hardware design point is evaluated with a more complete picture of its potential for performance and efficiency. Please note, for each kernel, the software auto-tuning loop is embedded within the hardware loop, and thus optimizes performance for each individual hardware configuration. We then record the power and area efficiencies for the fastest auto-tuned software configuration for each individual hardware configuration. The last column in Table 3.4 shows the power and area efficiency of the overall Pareto-optimal configurations using the co-tuning methodology. Results show that this approach yields significant improvements in power and area efficiency when compared to the three previously discussed configurations (as shown in the parenthesized values of each column).

It is interesting to compare co-tuning (seventh column) with only software-based (fifth column) or hardware-based (sixth column) tuning. Given the tremendous benefits of software tuning, it is not surprising to see that co-tuning outperforms the hardware-only tuning approach. This difference is particularly dramatic for SGEMM—$36\times$ and $70\times$ in power and area efficiency respectively—where the untuned code performs quite poorly. Additionally, co-tuning gains for stencil and SpMV range from $1.9\times$–$2.7\times$. Comparison of the co-tuning versus the software-only tuning approach shows that even after fully optimizing the code on the fastest hardware, there is still significant room for efficiency improvements. Notably, the power efficiency gains of co-tuning for SpMV, SGEMM, and stencil are $1.7\times$, $2.4\times$,
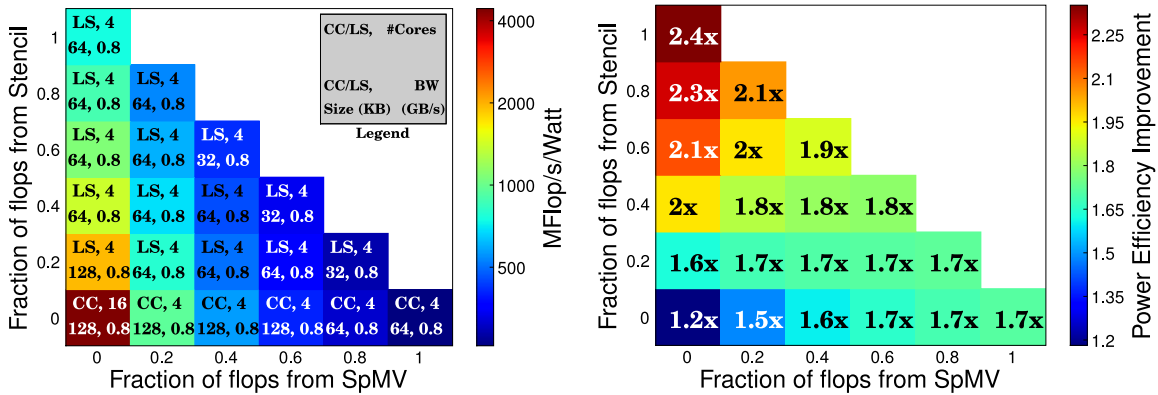
(a) Co-tuned weighted power efficiency showing optimal configuration: CC/LS, core count, CC/LS size (KB), memory BW (GB/s).

(b) Weighted improvements in power efficiency of co-tuning versus untuned-hardware with tuned-software approach.

Figure 3.12: Co-tuning for multiple kernels, using a 3D graph with the fractional contribution of SPMV on the x-axis, stencil on the y-axis, and SGEMM on the implicit z-axis. Each square in (a) depicts the HW parameters of the corresponding square in (b). The sum of the three kernels' flops contributions (x-,y-, and z-axis) always adds up to one.

and $1.2\times$ respectively, whereas, the area efficiency improvements are $1.6\times$, $3\times$, and $1.5\times$ respectively. These are the first results to quantify the intuitive notion that ignoring either hardware or software optimization while performing system design will naturally lead to suboptimal solutions. Finally, we also note that each individual kernel can have different optimal hardware configurations which is evident by the different areas for the best area efficiency configurations.

3.5.3 Co-Tuning for Multi-Kernel Applications

No one individual kernel can give a complete picture of an entire application's performance on a given system. Realistic large-scale scientific applications consist of multiple subsystems that solve different parts of the overall problem. We therefore approximate the effect of co-tuning on a multi-kernel application by combining the results from Section 3.5.2.

We thus construct co-tuning results for the set of kernels by taking a weighted mean of their tuned performance data on each hardware configuration. Given that each individual kernel contributes some fraction of the floating-point operations for an entire application, we sum these kernel contributions via a weighted harmonic mean—the weights are the relative fractions of the individual kernels. This basic strategy assumes that interaction between kernels does not have significant impact on whole-application performance. Although this is clearly a simplifying assumption, it is nonetheless an important first step toward quantifying the potential impact of the co-tuning methodology to full-scale applications. Expanding this approach will be the focus of future work.

Figure 3.12(a) plots the power efficiency of the best co-tuned hardware configuration relative to the fractions of floating-point operations contributed by the three kernels, where

the individual contributions always sum to one. We present these data by defining the x-axis and y-axis of Figure 3.12(a) as the fractional contribution of Stencil and SpMV to an application's computation, while the remaining fractional portion represents the contribution of SGEMM (on the implicit z-axis). Therefore, the lower-left corner of the plot represents optimized hardware configurations for applications consisting entirely of SGEMM-like dense linear algebra algorithms. Similarly, the lower-right corner represents applications consisting entirely of SpMV-like sparse linear algebra, while and the upper left corner is stencil-based grid computations. As the most power efficient configuration differs for each mix of kernels, we annotate Figure 3.12(a) with the parameters of the best configuration: CC/LS, core-count (1-16), CC/LS size (16K-128K), and memory bandwidth (0.8-3.2 GB/s). Results show the variety of co-tuning architectural solutions and corresponding power efficiencies based on a given application's underlying characteristics.

Figure 3.12(b) plots the power efficiency improvements of the co-tuned systems for each kernel mix compared with the untuned-software tuned-hardware approach (described in Section 3.5.2). This approach most closely resembles prior work in automated system design, which have hitherto not included extensive software optimization. Recall, that we only consider coherent-cache based configurations for untuned-software base case (see Section 3.5.2). Results show that co-tuning results in power efficiency gains ranging between $1.2\times$–$2.4\times$ depending on each kernels contributions. A similar analysis for area efficiency (not shown), demonstrates improvements varying from $1.6\times$ to $3\times$ (as seen in Table 3.4). Overall this approach points to the potential of applying our co-tuning methodology to more complex, multi-algorithmic applications.

3.6 FPGA-Based Simulation

We note that, although our study with the Smart Memories architecture produced promising results, the searched space was small—only 72 hardware configurations were explored. Due to the small hardware design space, all hardware configurations were explored. Despite the small hardware design space, our co-tuning experiments in Section 3.5 took almost a week on a cluster of 6 quad core machines. To mitigate the slowness of software-based simulation of hardware, we use FPGAs (Field Programmable Gate Arrays) to accelerate the simulation. FPGAs have been traditionally used for prototyping of hardware designs [45, 120] but have also found use as accelerators for various kinds of applications; examples include bioinformatics [122, 65], machine learning [68, 8], finance [118], molecular dynamics [2], and cryptanalysis [18]. In this work, we use FPGAs to accelerate the simulation of hardware. This FPGA-based simulation will also be referred to as "emulation".

3.6.1 Approaches for Emulation

A straightforward way of emulation would be to directly implement the hardware design on the target FPGA. Since our objective is to investigate a space of hardware configurations, this necessitates the hardware design to be parameterized—every time a new hardware configuration is to be explored, we rebuild the design for the FPGA with the current hardware parameters. However, this simple approach suffers from some major drawbacks:

- The design may be too big to fit on the FPGA. Although modern FPGAs have gotten large enough to fit multiple embedded cores, they use chip area inefficiently due to reconfigurability. In fact, the on-chip memory (most of it available as Block RAMs) may be too small to even fit the large on-chip caches found in modern processor designs. Schelle et al. [87] implemented a modified version of the Intel Nehalem processor using multiple Xilinx Virtex-5 FPGAs. Furthermore, due to the need for partitioning the design over multiple FPGAs, the emulated design ran at 500 KHz. In contrast, Intel's embedded processor Atom could fit in a single Virtex-5 LX330 FPGA and ran at 50 MHz [109].

- FPGA synthesis times may dominate. When a new hardware configuration needs to be explored, the corresponding hardware design must be synthesized for the FPGA. Given that a reasonable design may use the FPGA resources heavily, the synthesis, place and route times can easily be a couple of hours. Therefore, unless the performance benchmarking on the emulated design dominates the time to build the design for the FPGA, it is a significant overhead in simulation.

Another way of emulation would be to implement the simulator rather than the target design. Since our objective is to simulate a hardware design, this approach employs FPGAs to accelerate the hardware simulator. Note that FPGAs may be used to accelerate both functional simulation as well as cycle-accurate simulation of hardware. Software-based cycle-accurate simulation can be particularly slow even on modern multiprocessors—the Smart Memories simulator achieved an *aggregate* throughput of 100 KHz!

The RAMP (Research Accelerator for Multiprocessors) project [110] aims to develop systems for FPGA-based simulation of hardware. One of the key ideas in the RAMP project is that although the emulator may run at 100 MHz (which is slow compared to the gigahertz clocks on modern processors), it still represents a speedup of $1000\times$ over a software-based simulator which may run in the 100,000 cycles/second regime. Furthermore, when compared to the single-threaded software-simulator of the Smart Memories architecture, the aggregate emulation throughput can potentially scale with the number of cores instead of being roughly constant which is the case with software-based simulation.

More recently, Tan et al. [97] built RAMP Gold which is a cycle-accurate emulator for multiprocessors. Due to the overheads involved in emulation, multithreading in the FPGA is used to improve the aggregate emulation throughput. RAMP Gold can model up to 64 SPARC V8 cores with a shared memory hierarchy on an Xilinx XUP FPGA board. The memory subsystem parameters as well the number of cores were runtime configurable, i.e., the emulated system parameters could be changed without the need to synthesize the emulated design on the FPGA. Emulation was able to speed up the simulation of a 64-core system by around $250\times$.

Chung et al. [22] developed ProtoFlex to accelerate functional simulation of hardware. Only the performance critical components (e.g., the ALU) of the hardware simulator were ported to FPGAs. The authors report a speedup of $38\times$ over software-based simulation.

Pellauer et al. [80] describe HAsim, which is performs a detailed simulation of multi-core based on the 64-bit Alpha ISA. The authors used a Xilinx Virtex-5 LX330T FPGA for accelerating hardware simulation. Although the techniques employed were similar to the ones in [97], the authors extended them to also model on-chip interconnects. A single core pipeline was time-multiplexed to simulate up to 16 core designs.

3.6.2 Emulation Details

In this work, we use a combination of direct implementation and emulation to build a simulator for a Tensilica XTensa-based multiprocessor. Our target design is a distributed memory multiprocessor and can be viewed a simplification of the Smart Memories multiprocessor. We now describe how the different components of the target design were emulated. For the FPGA platform, we use the BEE3 FPGA board [26].

Core Emulation

We use a direct implementation of Tensilica's XTensa cores for FPGAs. Since we only had access to the netlist, clock gating was required to pause the cores when multiple FPGA clock cycles were needed to simulate a single target machine cycle. Specifically, this scenario would only occur when we needed to emulate an L2 hit—since only the tags were stored on-chip, if the "hit penalty" to L2 was small, we would end up having more host cycles than the "hit penalty". We also note that the L1 cache was included in the core—so it would also be paused when clock-gated. The L1 cache is also a direct implementation so no emulation techniques are needed there.

The core communicates to the memory subsystem through a PIF interface which is a point-to-point connection protocol from Tensilica and supports both burst and block modes of data transfer. It also provides a atomic conditional write primitive for inter-processor synchronization.

Note that in contrast to multiprocessors, the gap between core performance and the DRAM performance is almost non-existent since the core clocks are of the order of 25 MHz. Thus, when emulating long DRAM latencies, it is possible that the whole physical system is doing nothing except wait for the clock cycles to elapse because a certain DRAM latency is to be emulated—the memory request has already been served by the memory controller on the FPGA! In this specific case, we can advance the global clock by the required number of cycles, i.e., one clock cycle on the FPGA emulates multiple clock cycles on the target design—we call this *clock skipping*. This idea can improve the emulation throughput significantly when the emulated system is almost always stalled due to being DRAM latency bound. When emulating a multi-core over multiple FPGAs, however, detecting when to *globally* skip clock cycles is more complicated and its impact on emulation performance is expected to be lower as the instants when clock skipping can be done would be lesser.

Cache Emulation

We use L2 caches in our design, which can be too large to fit on modern FPGAs if implemented directly. So, the L2 caches are emulated by only storing the cache tags on chip and the timing of the system is controlled to mimic a real cache. The idea of only storing cache tags on-chip has also been implemented in other FPGA-based emulators [97, 80]. The hit/miss information from cache tag access is used to drive the timing of the response from the cache.

We make the following cache parameters configurable: line size, associativity, set size, and replacement policy. Given that we make it runtime configurable, we just need to synthesize the largest possible instance and then turn off portions depending on the current parameters. Let L denote the line size, A denote the associativity, S the set size. Note that we only store the tags on chip and the data is actually fetched from DRAM. Let T denote the tag data width. Let A_{max} denote the maximum associativity, S_{max} denote the maximum set size, L_{min}, L_{max} denote the minimum and maximum line sizes in bytes respectively. Assuming 32-bit addressing, $T_{min} = 32 - \log_2(S_{max})$, $T_{max} = 32 - \log_2(S_{min})$. Thus, the size of on-chip memory needed would be $T_{max} \cdot A_{max} \cdot S_{max}/L_{min}$.

The latency and bandwidth of the cache are also configurable but instead of using arbitrary latencies and bandwidths, these numbers are determined by the cache modeling tool CACTI once other cache parameters are fixed.

Interconnection Emulation

We simulate a ring network. The simulated ring network is directly mapped onto the physical ring network on the BEE3 board [26]. The building blocks for the FPGA implementation are the interconnect switches, which route packets based on destination id. Although
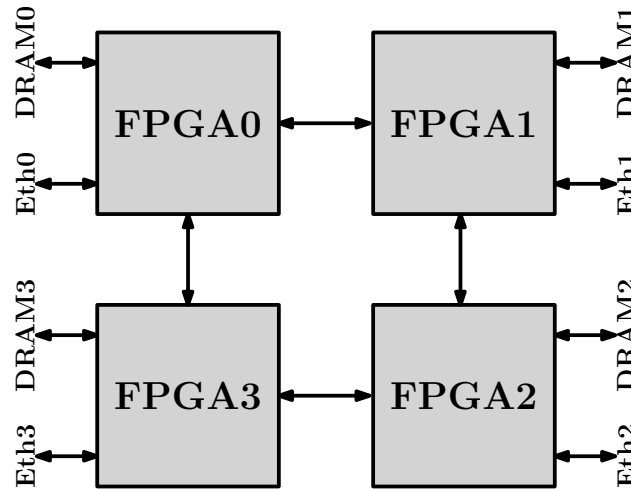
Figure 3.13: Physical ring network on the BEE3 board

we directly implement the NoC switches, the links between the switches can be configured to have different latencies and bandwidths.

Figure 3.13 shows the physical ring network on the BEE3 FPGA board. We use the Ethernet ports for setting the configuration (including resetting), reading the performance counters, and loading the DRAMs with code and data. All the memories (DRAM0, DRAM1, DRAM2, DRAM3) can be accessed through the control software on the host machine serving up the FPGA board.

DRAM Emulation

Similar to the Smart Memories simulator, we model DRAM as a flat memory, with each memory access having fixed access time and response time. This is the same model we had in the Smart Memories simulator. DRAM is modeled by the following parameters, which are runtime configurable:

- Latency: The response from the DRAM controller is delayed by an appropriate number of cycles to emulate the latency.

- Bandwidth: This is emulated by adding delays between consecutive words being returned by the memory.

Emulating Latency and Bandwidth

We defined building blocks to emulate latency and bandwidth parameters which can be a part of different system components, e.g., DRAM, interconnect, caches. For emulating latency, we defined a "configurable delay" block and for emulating bandwidth we defined a "configurable bandwidth" block:

- Configurable delay: We use this block to emulate multi-cycle delay wires, i.e., to emulate latency. The latency is a runtime configurable parameter to the block—the
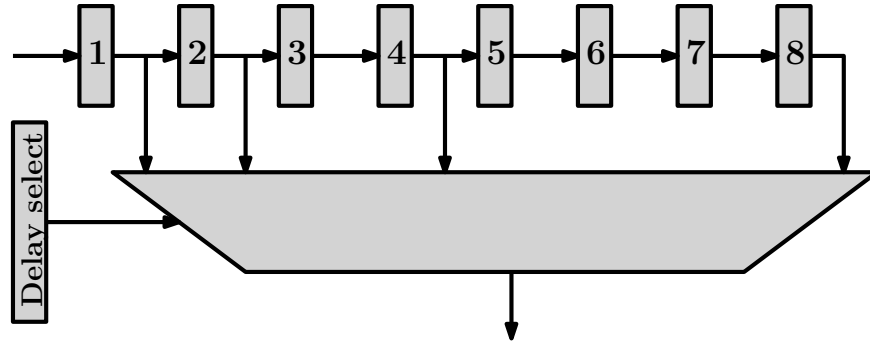
Figure 3.14: Latency emulation pipeline. For emulating L max-latency, we only allow $\log_2(L)$ possible delays: 1, 2, 4, $\ldots$, L to reduce the multiplexor size.

latency parameter is used to select from the different stages of the delay pipeline. Since the cost of the select circuit can be expensive if the latency is large, we only emulate some fixed number of latencies, i.e., select from only a certain small number of stages of the delay pipeline. All the stages of the pipeline have the same pipeline interface to ensure flow control along the pipeline. Figure 3.14 illustrates this block.

• Configurable bandwidth: We use this block to emulate bandwidth for certain wires. This has a single pipeline stage and the pipeline progresses every given number of cycles determined by the inverse of the bandwidth.

3.6.3 Physical Implementation on a Single FPGA

Figure 3.15 shows the implemented design for a single FPGA. The Xtensa cores from Tensilica are supplemented with L2 caches (only the tags are stored) and are interconnected to other "devices" in the system through an interconnect. These "devices" could be other cores, memory, (DRAM, configuration registers, performance counters), serial port (for I/O), Ethernet port (for I/O, control). Since two Xtensa cores can be accommodated on a BEE3 FPGA, that makes it 6 devices in total per FPGA and 24 for a BEE3 board. Each "device" is assign a unique id so that the switch is able to route the NoC packets properly. The core clock frequency of the synthesized design was 25 MHz.

We also note that the Xtensa core uses the PIF protocol for memory requests, so the PIF requests and responses are packed into the NoC packet format before sending them over the NoC. On the other end of the switch, the NoC packets are unpacked into PIF requests for the PIF slaves to handle them. The serial port device packets are routed through the Ethernet port along with the packets destined for the Ethernet device. The Ethernet device is used to access the memories, the configuration registers and the performance counters.

3.6.4 Performance Counters and Configuration Registers

For computing energy, we need to get event counts in order to use the formulas in Table 3.3. For this reason, we supplanted our design to have performance counters for
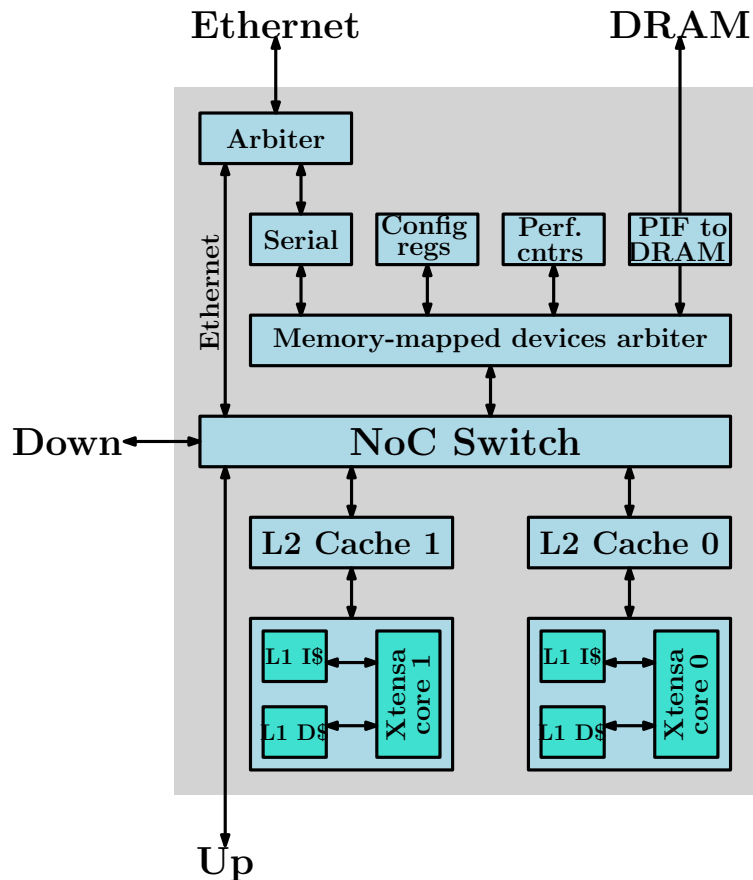
Figure 3.15: Block diagram of single FPGA design. The "Up" and "Down" links refer to the two bi-directional links to the "next" and the "previous" FPGA in the FPGA ring network on the BEE3 board.

counting cache access events (hits, and accesses), interconnect events (bytes transferred) and DRAM events (bytes transferred and number of accesses). Furthermore, the emulated target is specified by the values in the configuration registers—the bits in the configuration registers define the different system parameters for the current design being emulated. Note that our FPGA-based emulator allows the hardware configuration to be changed on the fly.

Both the performance counters and the configuration registers are memory-mapped and can be accessed using the PIF protocol. Since accesses to these special devices need not be fast, we avoid using a register-file and instead use a ring-network of registers for them. Since a ring network is employed, the cost of access to a register can vary from 1 to 10s of cycles depending on the register being accessed. As long as the code being timed is large enough, this overhead is small enough to be ignored. We also provide a mechanism to turn on/off all performance counters at once in order to turn the profiling on/off. To keep the interface simple, performance counters are read-only devices and configuration registers are write-only devices. Therefore, the software has to maintain state to remember the last read/written data to these devices.

Although dynamic reconfiguration of hardware parameters is as easy as modifying a

configuration register, care needs to be taken to make sure the effect of the parameter change propagates to the right hardware modules properly. For example, when reconfiguring the L2 cache, we need to make sure that there are no pending requests and also need to clear out all the cache tags. Similarly, for the DRAM, the request delay pipeline needs to be emptied before the new configuration can be considered valid. We finally note that we only one physical design to emulate our entire hardware design space—all the hardware parameters in the experiments are run-time configurable.

3.6.5 Software Infrastructure

Our initial design required the use of a USB wiggler to be able to load/debug/perform IO for the software on the hardware for the Xilinx XUP board. However, the lack of drivers for the BEE3 board meant that we had to write our own custom program loader for the BEE3 board. The loader program is very limited in its abilities—it can only initialize the board DRAMs with data/code, reset the design and perform I/O through Ethernet.

3.7 FPGA-based Simulation Results

We now report the performance results for FPGA emulation. We only report single core emulation due to being unable to complete the multi-core emulation—the physical multi-core worked but the emulated multi-core (which is the result of adding timing details to the physical multi-core) did not.

3.7.1 Single Core Emulation

This section describes the results of emulating a single-core system. We explore similar parameters as described in Table 3.2, except that the number of cores is set to one. The following describes the hardware parameters which are new here or had a different explored space in Table 3.2:

- Separate instruction and data caches of 16 KB each.

- L2 cache: The cache size is configurable from 32 KB to 1 MB (in powers of 2), associativity can be varied from 1 to 4, linesize could be 32, 64 or 128 bytes, line replacement policy could be LRU (Least Recently Used) or round-robin, and the latency was determined based on the CACTI model for caches as usual.

- DRAM: employed a flat-memory model with configurable latency and bandwidth. As with the Smart Memories study, we vary the bandwidth as .8/1.6/3.2 GBytes/s.

The synthesized design ran at 25 MHz, which is a speedup of $250\times$ when compared to the throughput of the software simulator for single-core. Since this is a single-core system, clock skipping was enabled to speed up emulation time when emulating large DRAM latencies. Clock skipping improved emulation throughput when running untuned codes as the target

system would be stalled for memory most of the time. For estimating energy, we used the same scripts and numbers we used for Smart Memories experiments.

We now describe co-tuning results for GEMM and SpMV kernels.

SGEMM

As with the Smart Memories experiments, we implement the optimizations as mentioned in Section 3.2.1 and use the GEMM code generator from ATLAS [112]. As earlier, we performed the auto-tuning in two phases:

1. A larger software design space was explored to determine the code variant parameters which only depend on the hardware configuration of the processor core and excludes the remaining memory subsystem. This phase was used to determine the register tiling strategy, loop unroll factor, and other code generation parameters. Since the instruction throughput was being optimized here, a small dense matrix (small enough to fit in the L1 cache) was used, which enabled a much larger software design to be searched quickly.

2. The optimal code generation parameters were fixed and larger matrices of size 512×512 were benchmarked to explore other optimal software parameters like cache blocking, transposing, etc.

Figure 3.16 shows the performance results for GEMM on single-core. Figure 3.16(a) shows how performance improvement due to tuning varies as a function of chip area. As expected, tuning always improves performance but the improvement is dramatic (more than $100 \times$ for small chip area (which correspond to small cache configurations). Because of the dramatic improvements in performance for small area configurations (Figure 3.16(b), the small area configurations become much more competitive with the large area configurations due to tuning. In fact, performance improvements due to increased chip area are not nearly as impressive once tuning is taken into consideration. Figure 3.16(c) plots area and power efficiencies in a manner similar to Figure 3.11. While area efficiency improves by $51 \times$, power efficiency improvement is less dramatic at $5 \times$.

Figure 3.16(d) plots the normalized simulation throughput, i.e., the number of cycles simulated per FPGA core clock cycle—a direct implementation of hardware on the FPGA would have a normalized simulation throughput of 1. If the normalized simulation throughput is more than one then clock skipping was effective in speeding up the hardware simulation. Since the small area configurations perform extremely poorly, they benefit a lot (up to $6 \times$) from clock skipping as the core is stalled most of the time.

SpMV

The SpMV optimizations implemented here are the same as discussed in Section 3.2.1. Since the cores on the FPGA run $100 \times$ slower than modern cores, SpMV benchmarking was done in two phases: the data structure optimization, which tries to minimize the memory footprint of the sparse matrix, was done offline and the required memory image was dumped
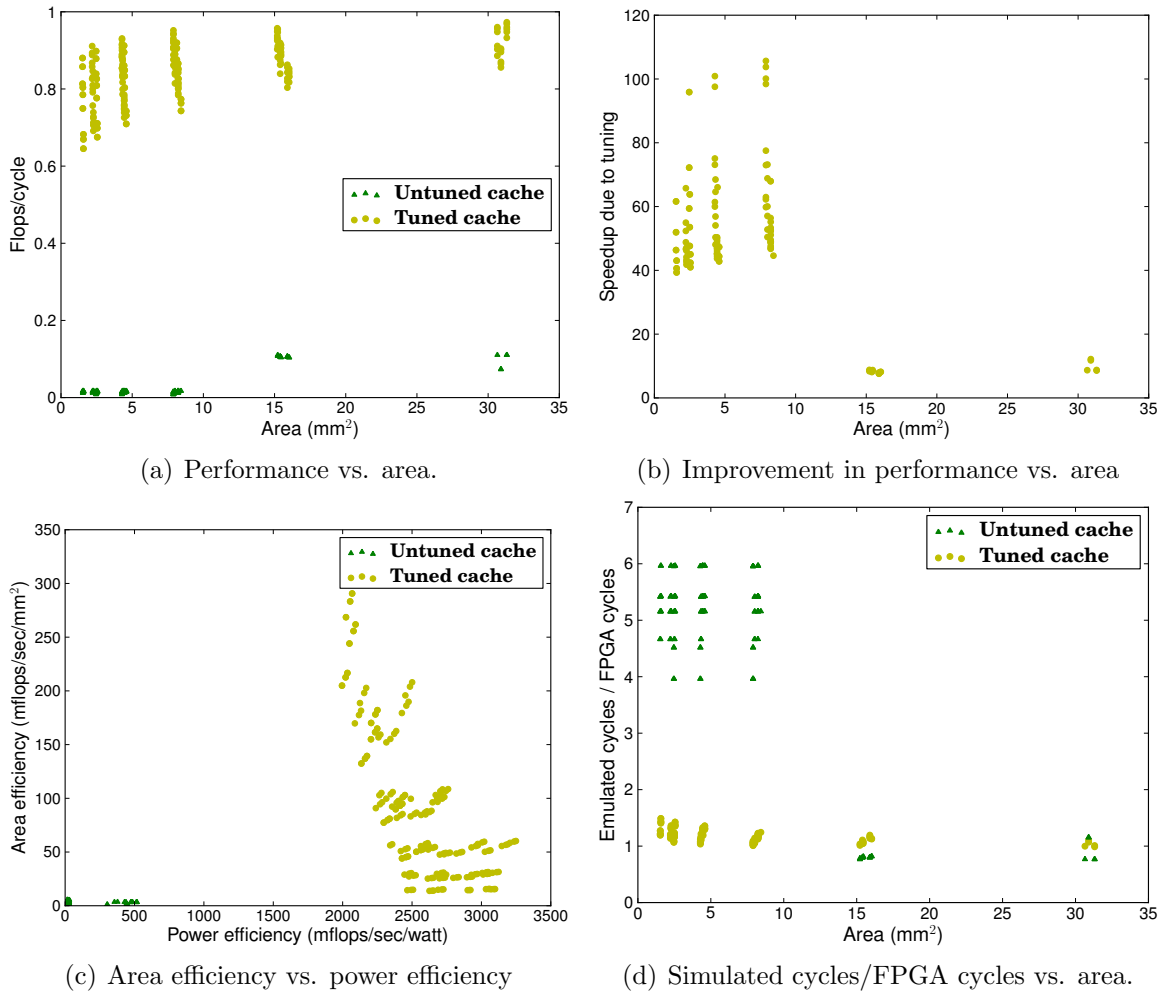
(a) Performance vs. area.

(b) Improvement in performance vs. area

(c) Area efficiency vs. power efficiency

(d) Simulated cycles/FPGA cycles vs. area.

Figure 3.16: SGEMM performance

on to the board memory when benchmarking. Since the date structure optimization is an expensive operation depending on the search space, it is done offline, just as in Section 3.2.1. FPGA emulation is only used for performance measurement—although it is much faster than software simulation, it is still 2 orders of magnitude slower than real machines.

Figures 3.17–3.28 show the (a) performance, (b) performance improvements, (c) efficiencies and (d) simulation throughput for different matrices—in all, 216 hardware configurations were considered. When compared to SGEMM, the performance and energy improvements are less dramatic, as expected, from our experience with the Smart Memories simulator, since it is more difficult to tune SpMV and the performance is strongly dependent on the sparsity pattern of the matrix. We note that the performance, speedups due to tuning and efficiencies are strongly dependent on the matrix. The matrices `econ`, `marcatcomm`, `mc2depi`, `scircuit` and `webbase` do not benefit much from tuning, and as a result, co-tuning doesn't add more value to the traditional approach of using untuned codes. In contrast, other matrices get significant improvements in performance (up to 2×) and ef-

ficiencies and as such demonstrate the effectiveness of co-tuning. It can also be seen that, in general, small area configurations benefited more from tuning due to poor data reuse for small caches.

Interestingly, we note that some of the poorly performing matrices, viz., `marcatcomm` and `mc2depi` were some of the good matrices for the matrix powers kernel (Section 2.10.3). The poor performance of these matrices can be attributed to the lack of structure as well small number of nonzeros per row. We also note that the simulation throughput doesn't show much variation across the different matrices. In contrast to SGEMM, where the normalized simulation throughput was up to 6, it is at most 2 for SpMV—this is due to the extremely poor performance of the naïve SGEMM code and the tremendous improvements in its performance due to tuning.

The results for FPGA-based simulation reaffirm the software-based simulation results. However, while it took several days on a small cluster for software-based simulation, it took less then 6 hours to collect results for a 3× larger hardware design space using a single FPGA board. Moreover, techniques like clock skipping provided further improvements in the simulation throughput.

Figure 3.29 shows the performance of the "median" matrix—for each hardware configuration, the matrix with the median tuned performance on that configuration was chosen to compute the representative energy and performance numbers. The "median" matrix is used to summarize the different performance and energy trends for different matrices using a single "matrix". Although the performance and energy improvements are near nil for the worst performing matrices like `scircuit` and `webbase`, the "median" matrix fares much better as tuning gives modest improvements in performance for most of the matrices–1.7× improvement in area efficiency and 1.4× improvement in power efficiency.
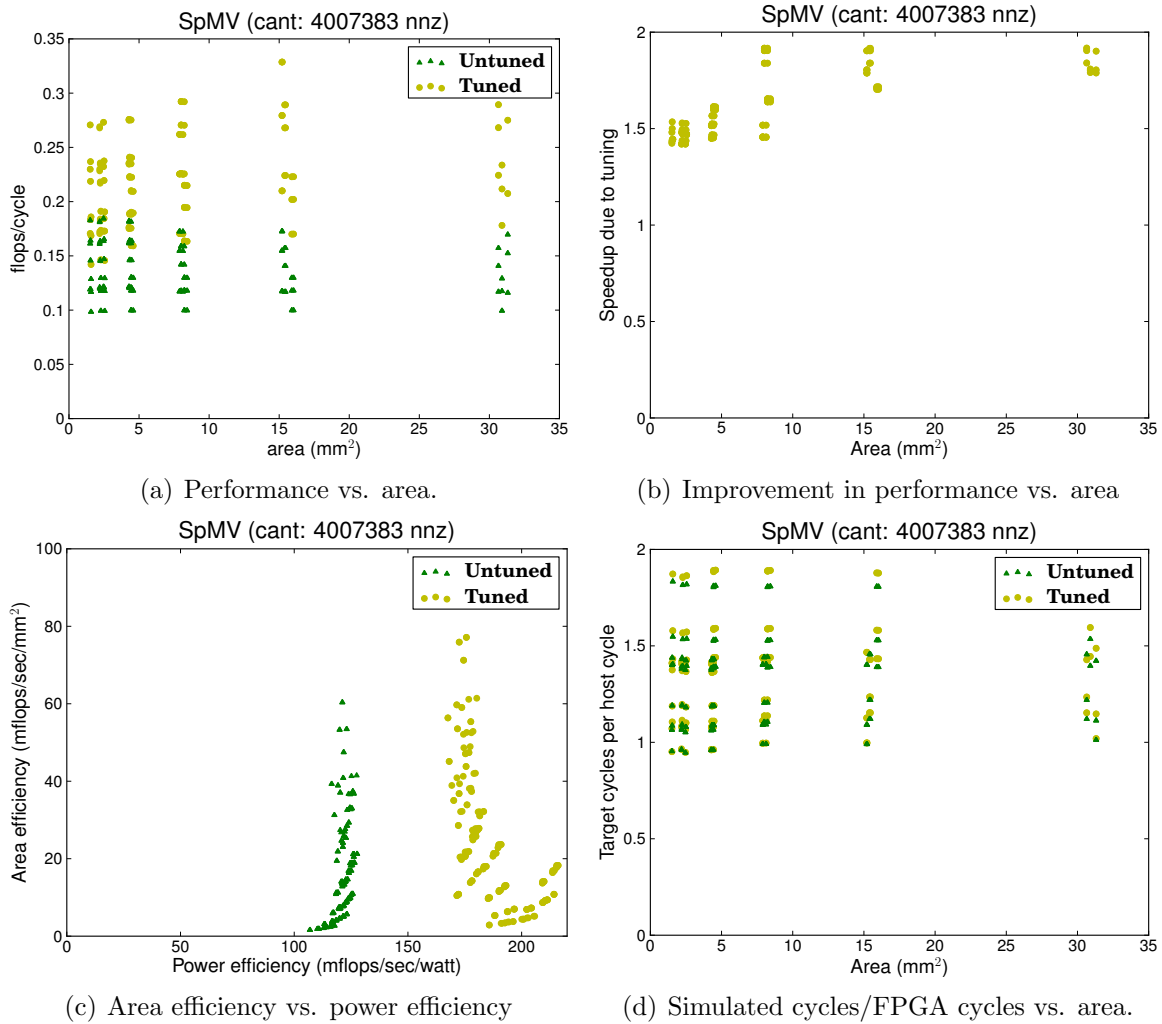
(a) Performance vs. area.

(b) Improvement in performance vs. area

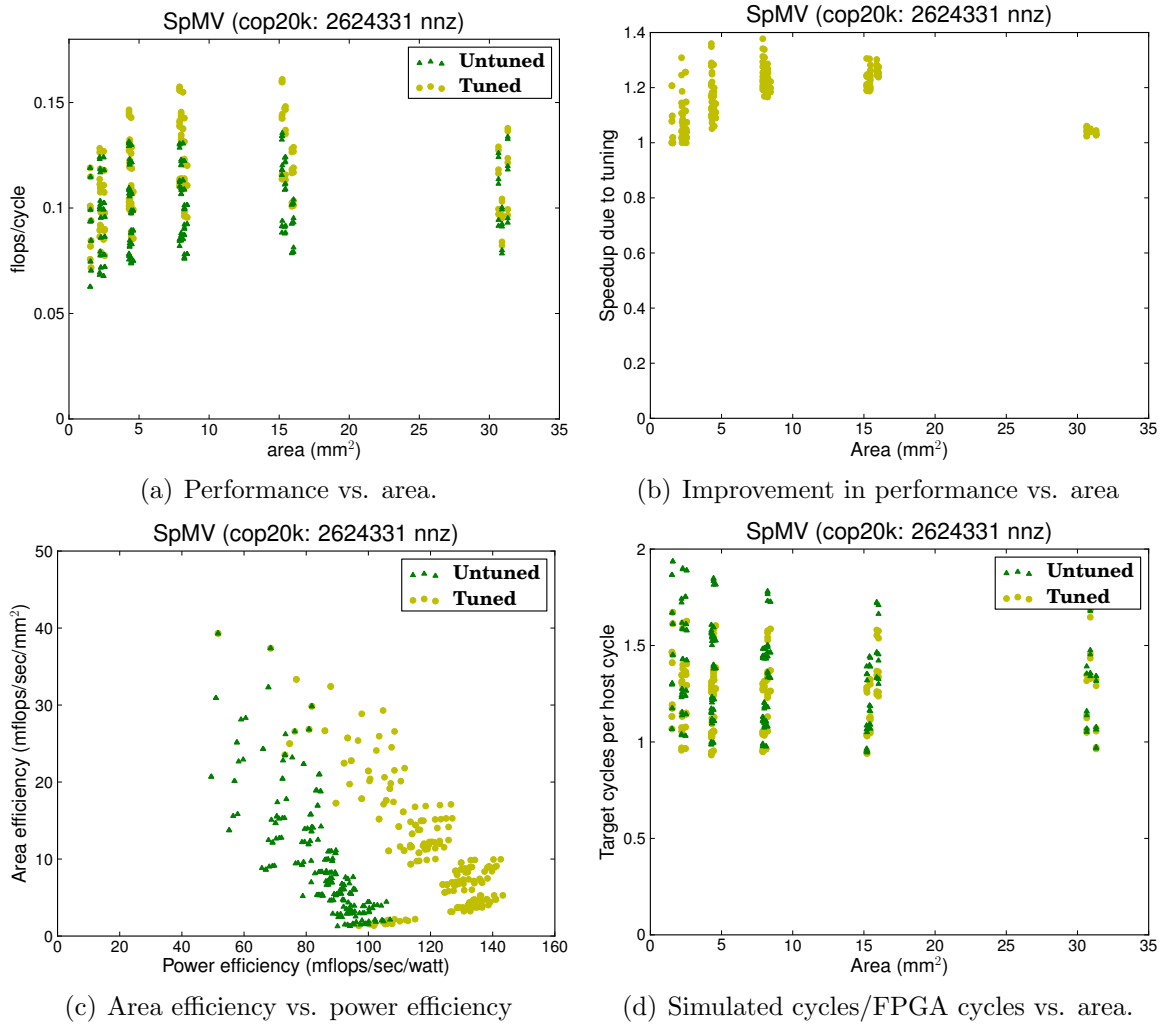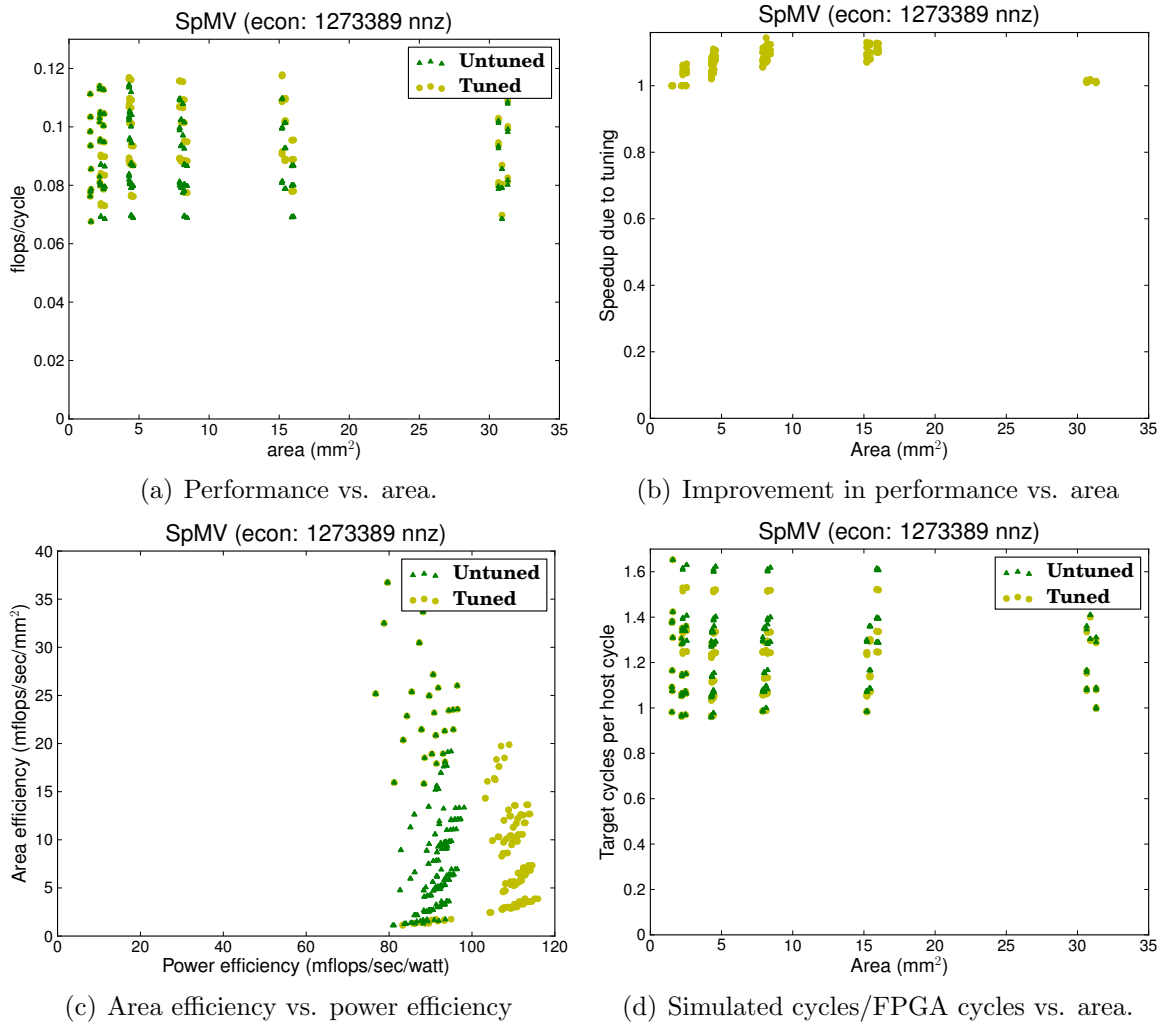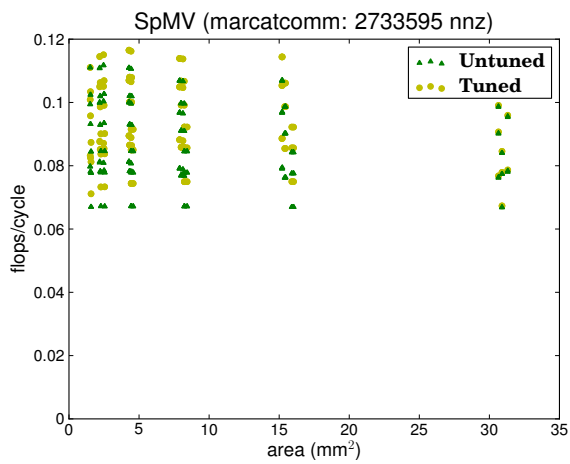(c) Area efficiency vs. power efficiency

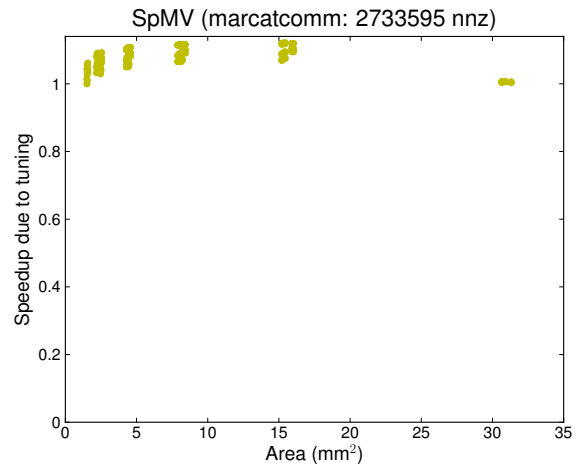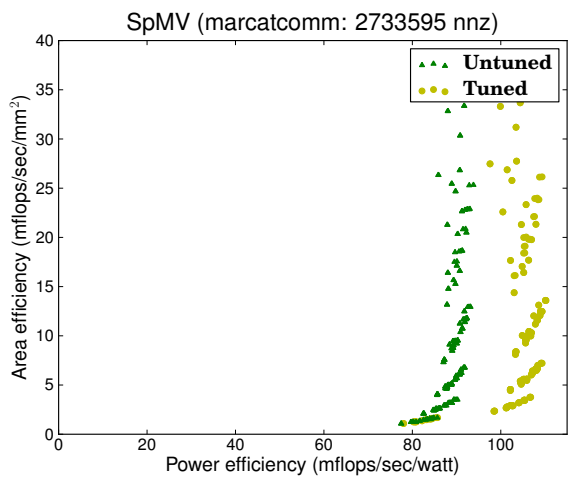(d) Simulated cycles/FPGA cycles vs. area.

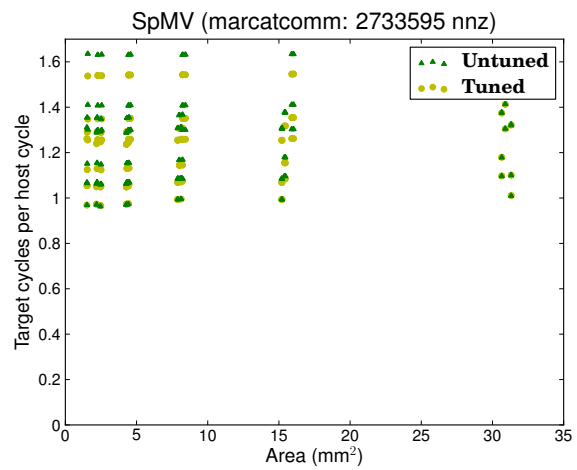Figure 3.17: SpMV: `cant` matrix performance

(a) Performance vs. area.

(b) Improvement in performance vs. area

(c) Area efficiency vs. power efficiency

(d) Simulated cycles/FPGA cycles vs. area.
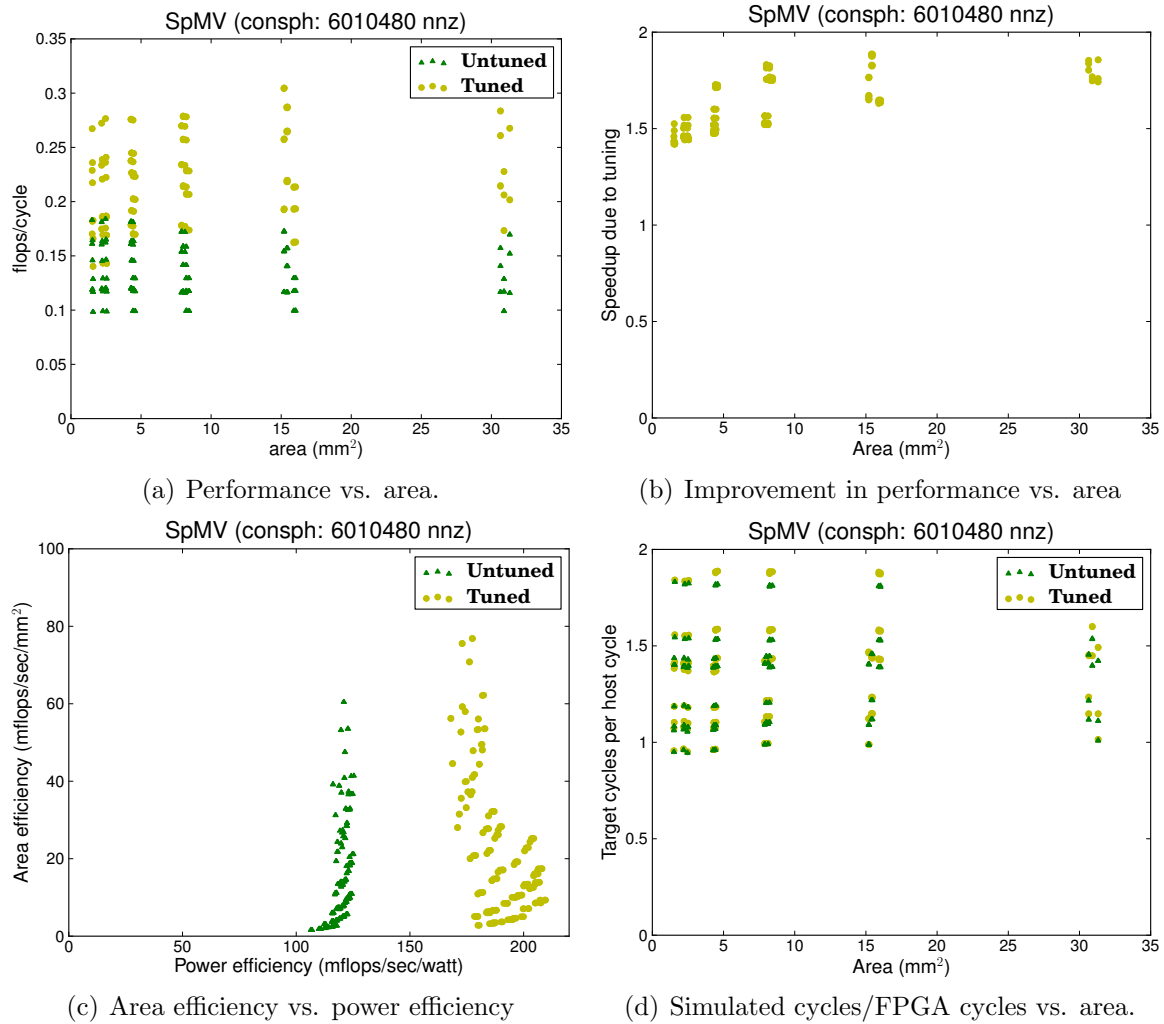
Figure 3.18: SpMV: `consph` matrix performance

(a) Performance vs. area.

(b) Improvement in performance vs. area

(c) Area efficiency vs. power efficiency

(d) Simulated cycles/FPGA cycles vs. area.

Figure 3.19: SpMV: `cop20k` matrix performance

149

(a) Performance vs. area.

(b) Improvement in performance vs. area

(c) Area efficiency vs. power efficiency

(d) Simulated cycles/FPGA cycles vs. area.

Figure 3.20: SpMV: `econ` matrix performance

(a) Performance vs. area.

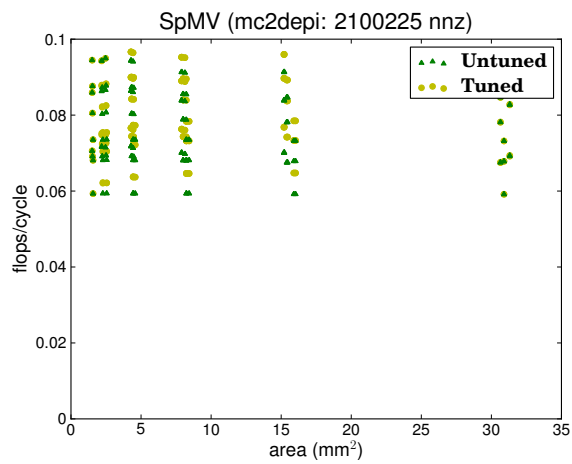(b) Improvement in performance vs. area
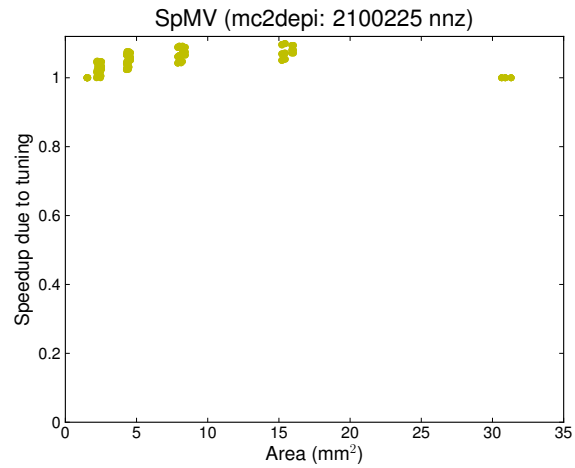
(c) Area efficiency vs. power efficiency

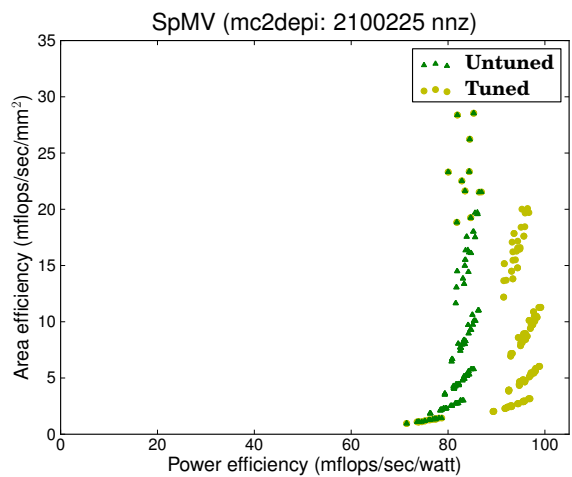(d) Simulated cycles/FPGA cycles vs. area.

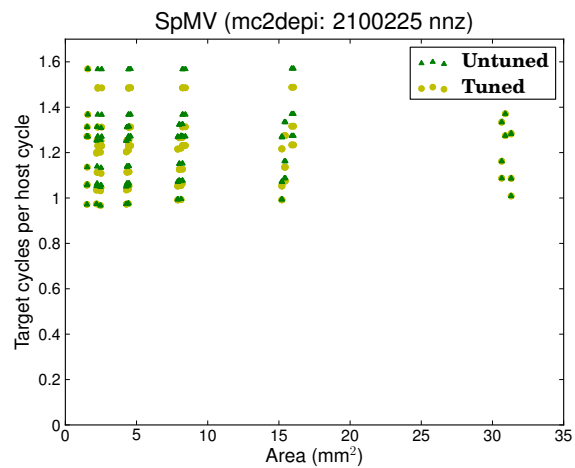Figure 3.21: SpMV: `marcatcomm` matrix performance

(a) Performance vs. area.
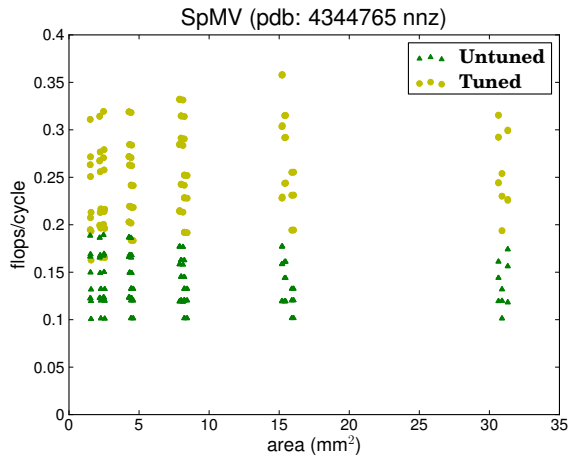
(b) Improvement in performance vs. area

(c) Area efficiency vs. power efficiency

(d) Simulated cycles/FPGA cycles vs. area.

Figure 3.22: SpMV: `mc2depi` matrix performance

(a) Performance vs. area.

(b) Improvement in performance vs. area

(c) Area efficiency vs. power efficiency

(d) Simulated cycles/FPGA cycles vs. area.

Figure 3.23: SpMV: `pdb` matrix performance

(a) Performance vs. area.

(b) Improvement in performance vs. area

(c) Area efficiency vs. power efficiency

(d) Simulated cycles/FPGA cycles vs. area.

Figure 3.24: SpMV: `pwtk` matrix performance

153

(a) Performance vs. area.

(b) Improvement in performance vs. area
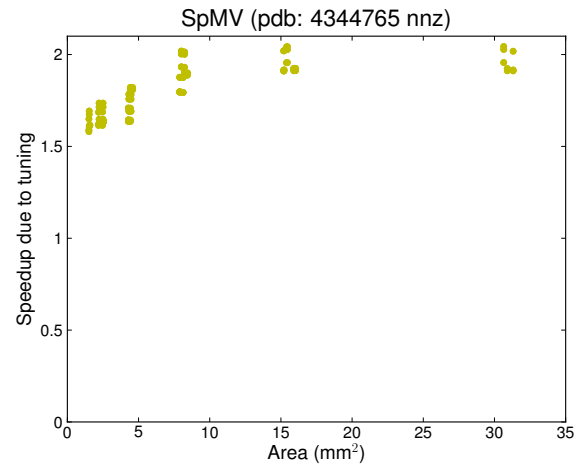
(c) Area efficiency vs. power efficiency

(d) Simulated cycles/FPGA cycles vs. area.

Figure 3.25: SpMV: `qcd` matrix performance

(a) Performance vs. area.

(b) Improvement in performance vs. area

(c) Area efficiency vs. power efficiency

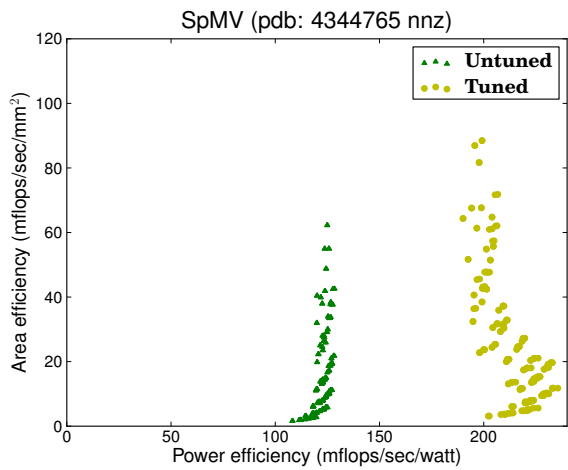(d) Simulated cycles/FPGA cycles vs. area.

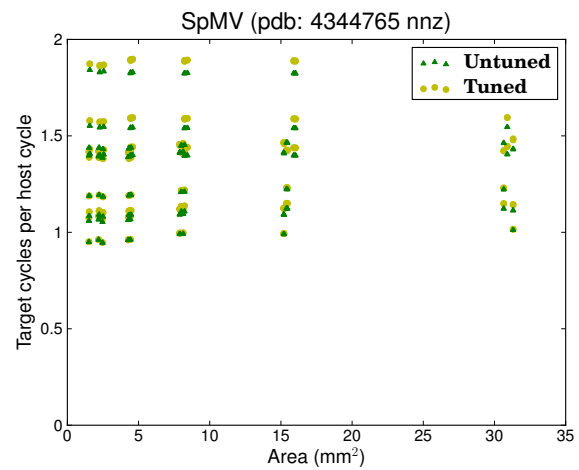Figure 3.26: SpMV: `scircuit` matrix performance
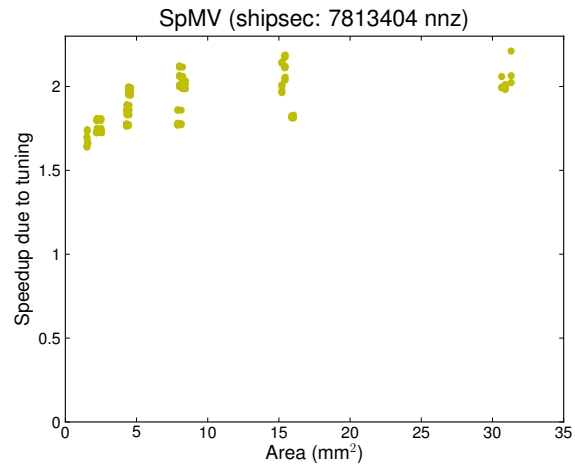
(a) Performance vs. area.

(b) Improvement in performance vs. area
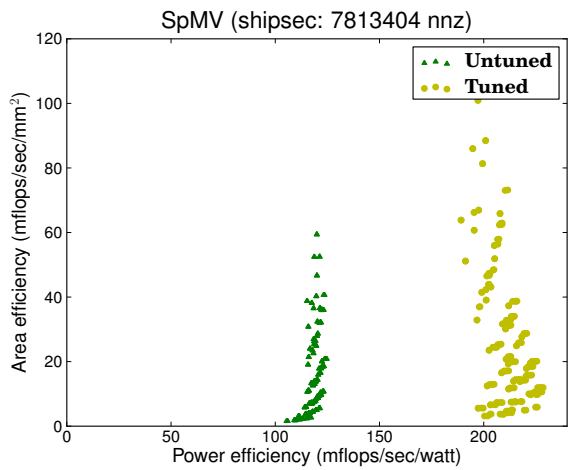
(c) Area efficiency vs. power efficiency
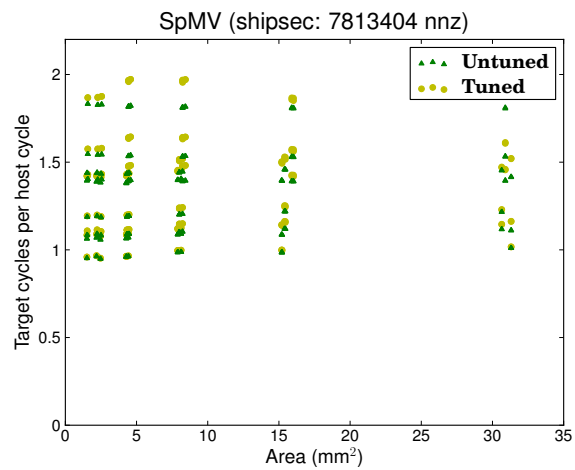
(d) Simulated cycles/FPGA cycles vs. area.

Figure 3.27: SpMV: `shipsec` matrix performance

(a) Performance vs. area.

(b) Improvement in performance vs. area

(c) Area efficiency vs. power efficiency

(d) Simulated cycles/FPGA cycles vs. area.
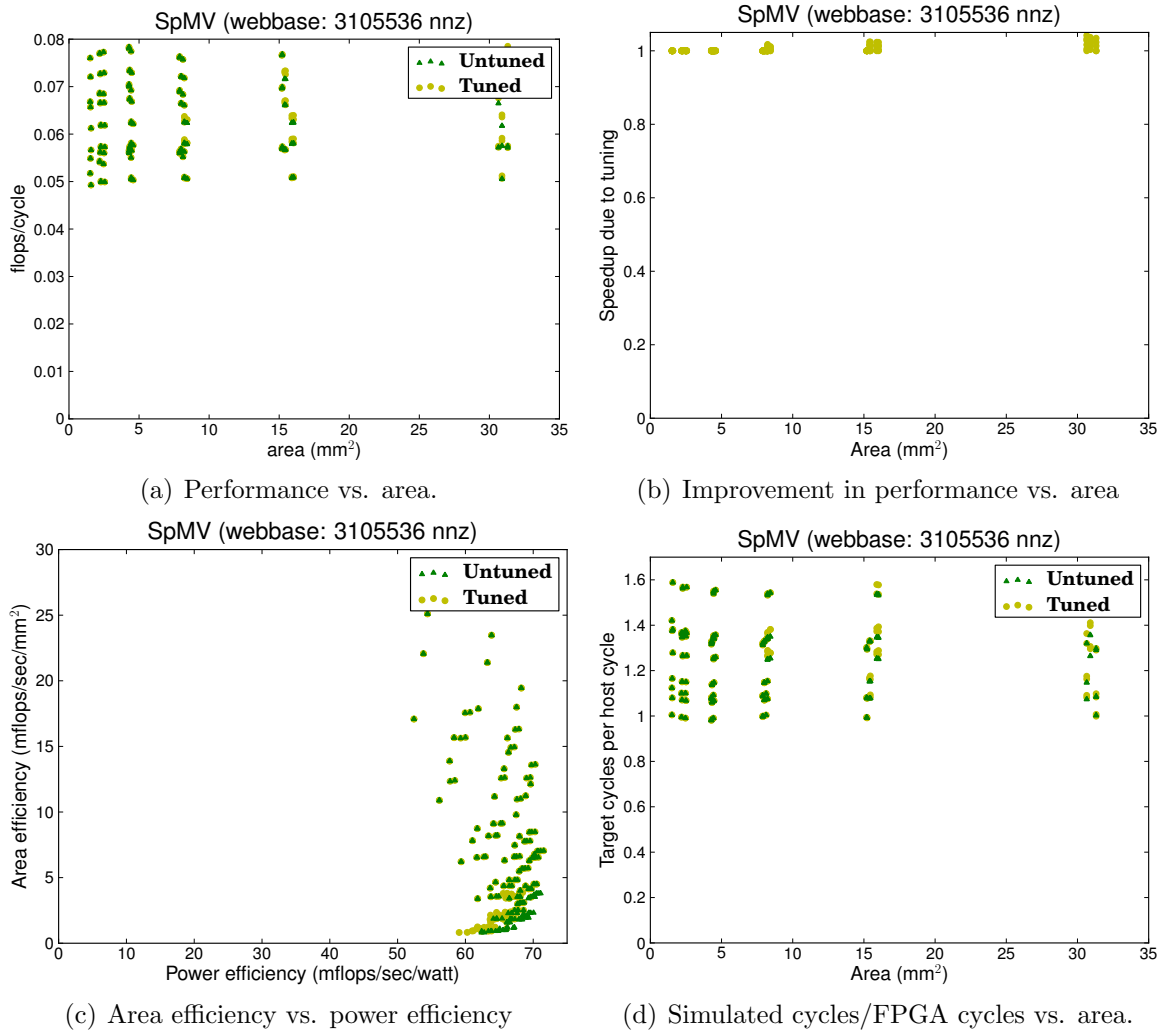
Figure 3.28: SpMV: `webbase` matrix performance

(a) Performance vs. area.

(b) Improvement in performance vs. area

(c) Area efficiency vs. power efficiency

(d) Simulated cycles/FPGA cycles vs. area.
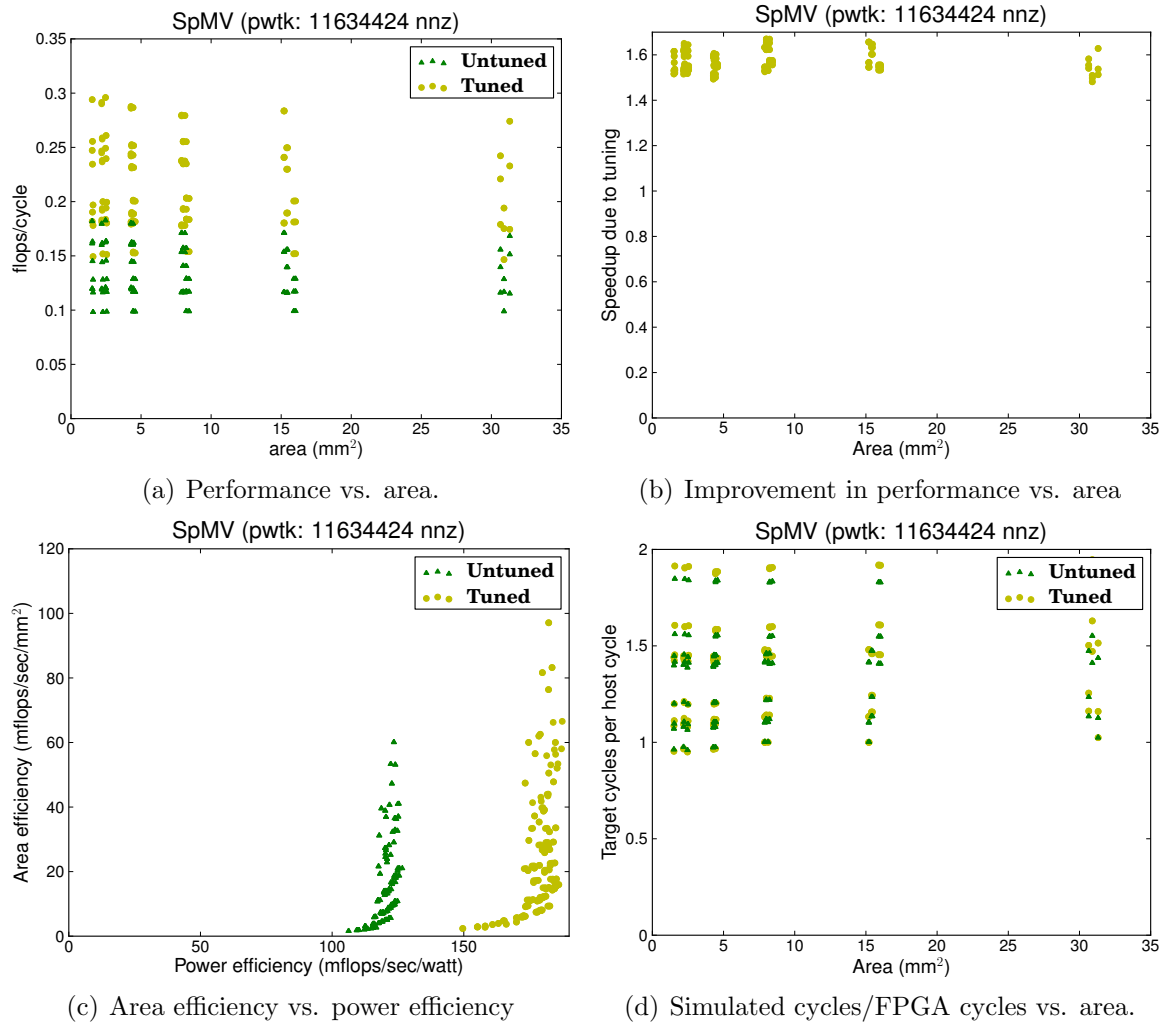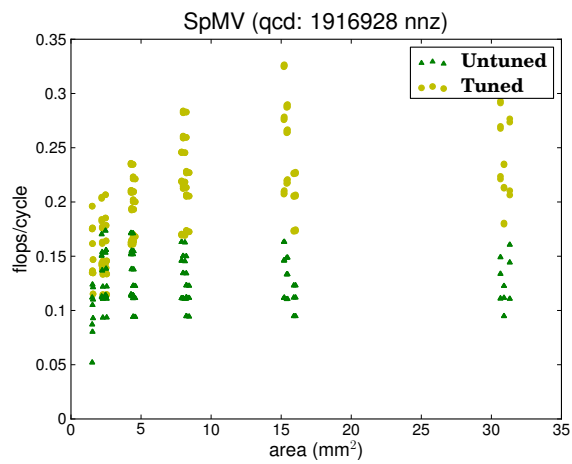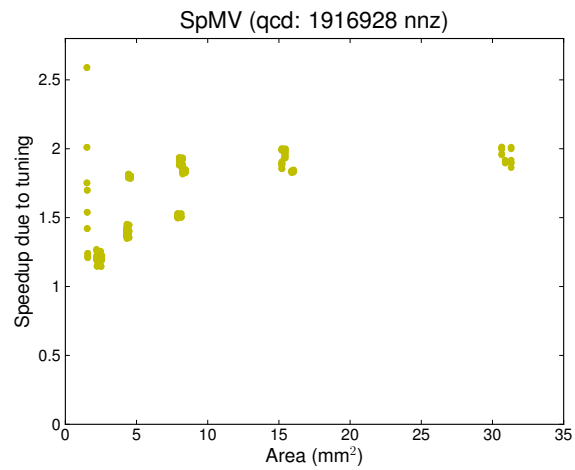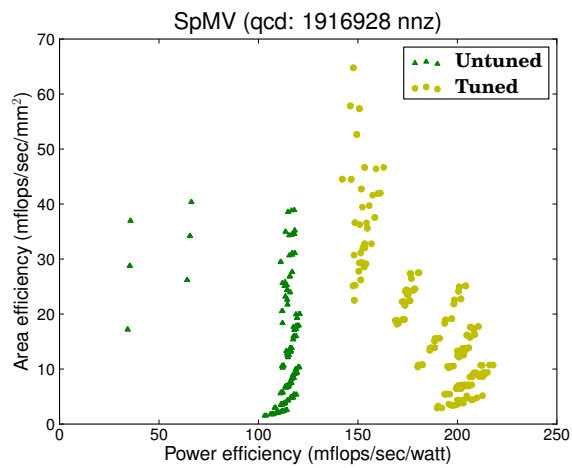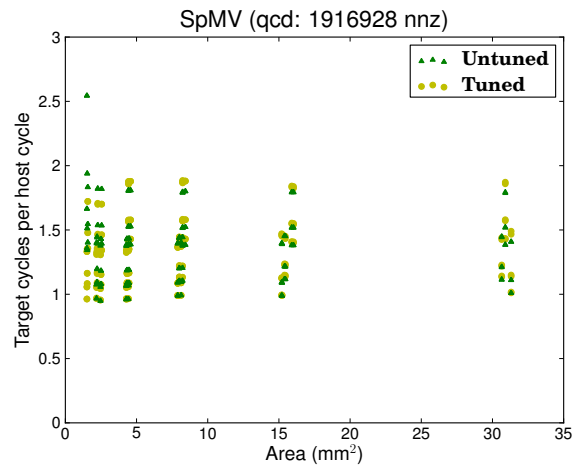
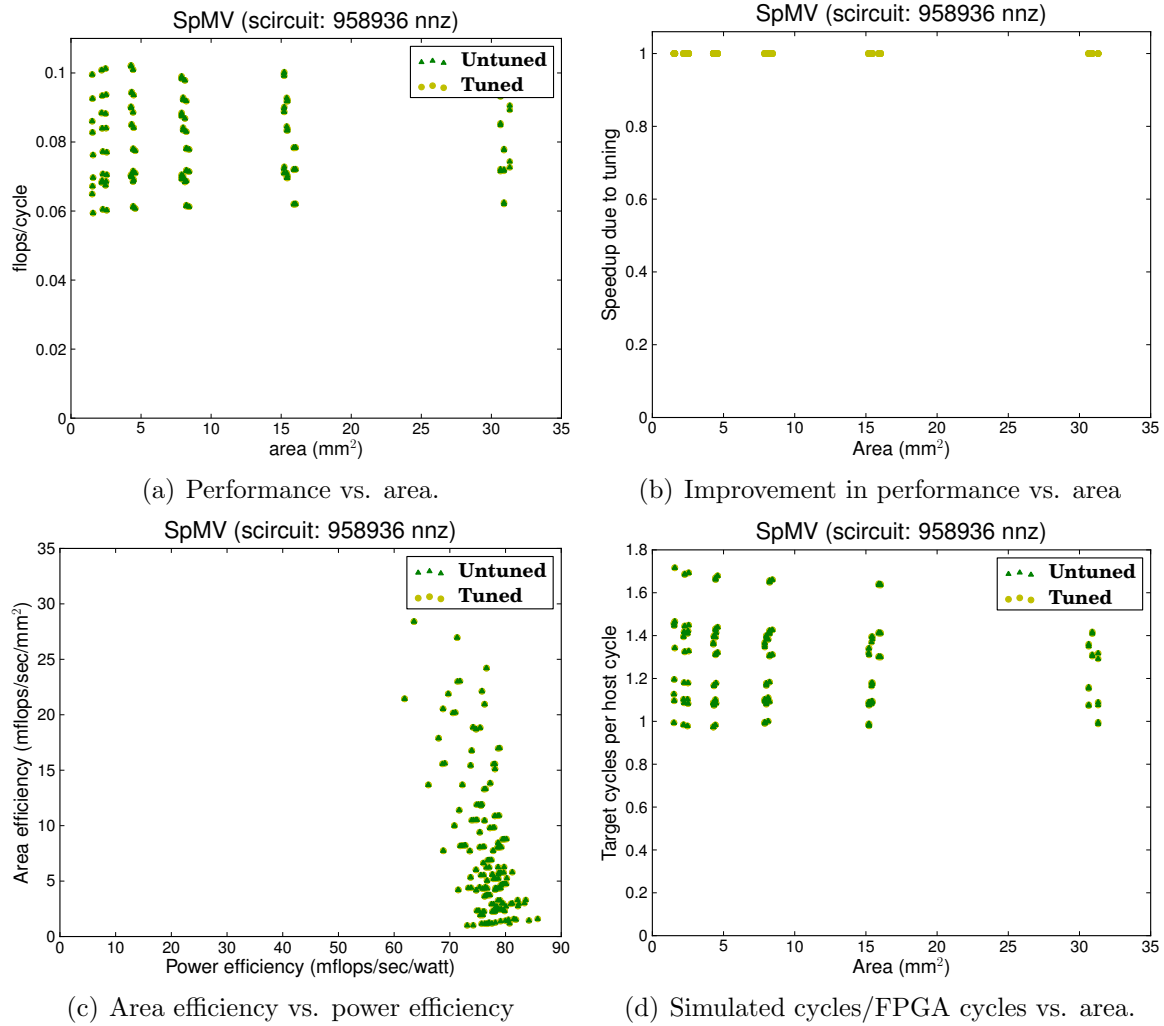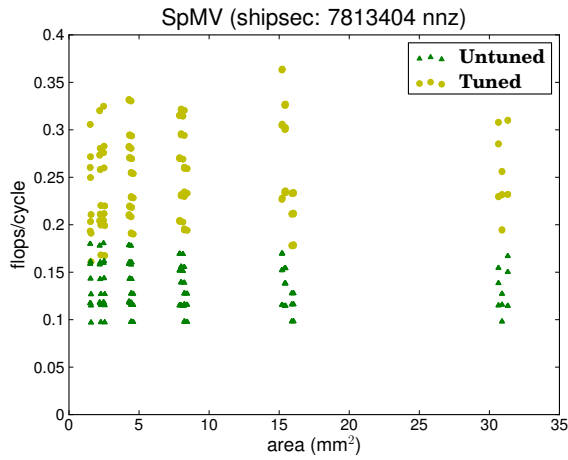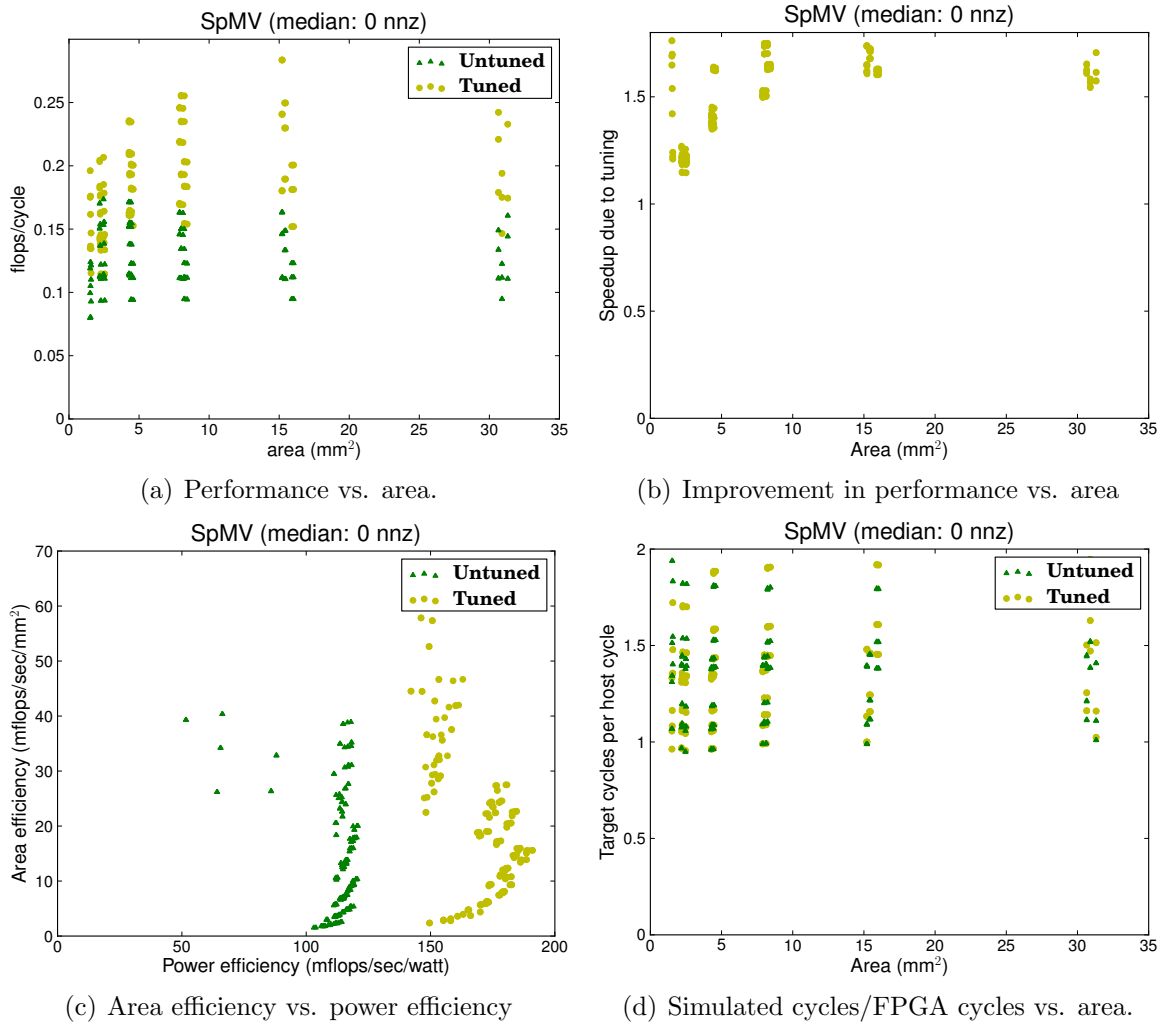Figure 3.29: SpMV: median matrix performance

3.8   Summary

In this chapter, we proposed hardware/software co-tuning as a novel approach to hardware design. We demonstrated co-tuning on the Smart Memories multiprocessor and three important scientific computing kernels. In addition to the results from a software-based hardware simulator, we also demonstrated co-tuning using an FPGA-based hardware simulator which ran orders of magnitude (nearly $250\times$) faster than the software-based simulator. Results showed that significant improvements in area and power efficiencies are possible due to co-tuning when compared to traditional design space exploration which ignores software tuning. These improvements range from $50\times$-$100\times$ for SGEMM to nearly $2\times$ for bandwidth-limited kernels like stencil and SpMV. Significant improvements in area and power efficiencies translate to lower procurement and running costs for a given target performance, thus demonstrating the effectiveness of co-tuning.

Chapter 4

Conclusions and Future Work

4.1 Conclusions

To address both current and future gaps between computational speed and communication speed, we are developing a set of communication avoiding algorithms to minimize data movement within local memory hierarchies and between processors. Our work on the matrix powers kernel falls under the broad umbrella of communication-avoiding algorithms. In this work we presented both serial and parallel algorithms for the matrix powers kernel, which can be used in place of individual sparse matrix vector multiplication in Krylov Subspace Methods and elsewhere. The powers kernel amortizes the bandwidth cost of reading the sparse matrix A by breaking A into blocks that fit in a single processor or a fast memory system and taking multiple steps on those blocks. We present algorithms with minimal communication costs, which send a single message (or slow matrix read) for k matrix-vector products computed in the matrix powers kernel, compared to k such operations in the conventional approach. We also show variations of the algorithms that trade off communication cost for redundant work.

Our algorithms are of practical as well as theoretical interest. We developed detailed performance models for both serial and parallel algorithms, instantiating the parallel model with parameters that are expected to be typical for a Petascale machine and for computing across the Grid. The serial model is instantiated with numbers from a current processor memory hierarchy and for an out-of-core setting. Our detailed performance model predicts more than $4\times$ speedups for high latency parallel machines (Grid) for moderate sized stencils. In the sequential case, our proposed algorithm avoids both latency and bandwidth (in contrast to the parallel machine, where only latency is avoided), which gives speedups for all problem sizes. The performance of our sequential out-of-core implementation is promising indeed, with speedups of $3\times$ over the conventional algorithm. Our auto-tuned multi-core implementation integrated the parallel and sequential algorithms for matrix powers to get significant improvements of up to $2.6\times$ over the naïve implementation, which happens to be based on a highly optimized implementation of sparse matrix vector multiplication. Our multi-core implementation of matrix powers was integrated into a communication-avoiding iterative solver with significant speedups over the state of the art. In addition to the specific

Architecture	cores per chip	Power per chip	Sustained MFlop/s per chip			Sustained PFlop/s with 10MW of chip power		
			DGEMM	Stencil	SpMV	DGEMM	Stencil	SpMV
Opteron	4	95W	32000	3580	1980	3.37	0.38	0.21
Blue Gene/P	4	16W	10200	520	590	6.38	0.33	0.37

Table 4.1: Performance of the double-precision implementations of our three key kernels on petascale computers. Note, power requirements are for the chip only assuming perfect scaling.

results in this work, we believe this work reflects a shift in algorithm design that will be necessary for future systems; this approach carefully counts communication costs and may favor algorithms with higher computational cost if they avoid communication.

We noted that power efficiency is rapidly becoming the primary concern for HPC system design. Conventionally designed ultra-scale platforms constructed with the conventional-wisdom approach based on using commodity server-oriented processors, will draw tens to hundreds of Megawatts—making the cost of powering these machines impractically high. Therefore, it is critical to develop design tools and technologies that improve the power efficiencies of future high-end systems.

We have proposed a novel co-tuning methodology—traditional architecture space exploration is tightly coupled with software auto-tuning—for high-performance system design, and demonstrated that it provides substantial efficiency benefits by customizing the system's architecture to software and software to the system's architecture. Our study applies this approach to a multi-core processor design with three heavily used kernels from scientific applications spanning a wide variety of computational characteristics. Based on the optimization results for the individual kernels, we demonstrate power and area efficiency gains of 1.2–2.4× and 1.5–3× respectively, due to co-tuning—when compared to using auto-tuned software on the fastest, embedded processor configuration. Additionally, we show that these improvements can also be attained in multi-kernel application environments. As high-lighted in Table 4.1, this increased efficiency can translate into hundreds of Teraflops, if not Petaflops, of additional performance for next-generation power-constrained HPC systems.

Building platforms from pre-verified parameterized core designs in the embedded space enables programmability and accelerated system design compared to a full-custom logic design, while providing higher efficiencies than general purpose processors tailored for serial performance. Furthermore, our hardware/software co-tuning methodology is a tool for assisting and automating the optimization of programmable HPC systems for energy efficiency. Tools for automatic design space exploration in the context of ad-hoc architectures do not exist, and the design space is intractably large. However, basing architectures on programmable multi-core processors constrains the design space, making the search space tractable and verification costs reasonable—as the same core can be replicated millions of times.

4.2   Future Work

4.2.1   Matrix Powers

There are several opportunities for further work on the matrix powers kernel—we list some of them:

- Better algorithms for multi-cores: In our hybrid algorithms for multi-cores each thread tries to minimize the traffic to DRAM independently of other threads. While we were able to get significant speedups, we believe further performance improvements are possible if the hybrid algorithms are better matched to the underlying hardware where multiple threads may share cores, caches and sockets. Although communication is costly, on-chip communication is still significantly cheaper than off-chip communication. Thus, we envision hybrid algorithms where more sophisticated composition of parallel and sequential algorithms is done, e.g., multiple threads which share an on-chip cache, work on the same cache block instead of working independently (which is the case with the hybrid algorithms in this work). Sharing cache blocks also has the benefit of fewer and larger cache blocks (since more cache memory is available per block) which amounts to a decrease in redundant computation. Thus, instead of the two-level composition (sequential algorithms running in parallel), a more hierarchical composition may be implemented (parallel algorithms running under sequential algorithms running in parallel!) to better match the underlying hardware.

- Better partitioner: A good partition of the matrix is the key to getting a good performance. Current graph/matrix partitioners target SpMV, so we built our own partitioner for matrix powers on top of METIS, which can be very inefficient as well as generate poor quality partitions. Recently, there has been work in developing a better partitioner for matrix powers using hypergraphs which better capture the communication-minimizing constraint of matrix powers. Part of the problem in developing an efficient partitioner for matrix powers is that the exact formulation of the partitioning problem would require actual computation of the powers of the matrix—if the matrix is bad, the partitioner itself may run out of memory. In fact, in our work, the choice of whether to use the powers of the matrix to setup the partitioning problem, was a parameter but we decided not to use it as it was expensive and would have failed for some matrices.

- Better ordering for the matrix and the vector entries and blocks: While we studied the theoretical aspects of the ordering problem (see the Traveling Salesman formulation in Section 2.5.5), it would be interesting to see how much of a benefit can be gotten from it in an actual implementation.

- Better auto-tuning: Currently, we exhaustively search the parameter space for matrix powers. Not only is this expensive, but a more sophisticated search may be able to cut down the search space. We envision using additional performance counter data, e.g., measured bandwidth, cache hit/miss rates, to guide the search space exploration.

In addition, a larger parameter space may be searched during auto-tuning using tools from machine learning. We note that the problem of designing better auto-tuners is more general and not just limited to matrix powers.

Our work on the matrix powers kernel was a comprehensive effort involving the design of algorithms, detailed performance modeling, implementation on a variety of platforms and finally integrating into an actual solver to demonstrate the effectiveness of communication-avoiding algorithms at the theoretical as well as the practical level. Nonetheless, there is still significant work to be done in the future as can be gauged from the list above.

4.2.2   Co-Tuning

Our c-otuning studies in this work were very limited and more of a proof of concept than developing a framework for such a methodology. There a lot of avenues for future work, some of them being:

- Co-tuning studies targeting FPGA-based simulation of multi-cores: While we set out to demonstrate such a study, we were unsuccessful due to problems debugging our FPGA-based implementation. Since frameworks for FPGA-based multi-core simulation already exist, it would be interesting to see co-tuning demonstrated on some of them.

- Explore a larger hardware/software design space: Our hardware design space was limited in the sense of exploring high-level parameters like the number of cores, the cache sizes and memory bandwidth. Future work could examine more complex architectural designs that can potentially improve power efficiency such as VLIW, SIMD, vector, streaming and hardware multi-threading. While Tensilica's XTensa processors can be customized heavily by defining new instructions, we used fixed cores. It would be interesting to see how such low-level customizations can affect auto-tuned applications. In addition, more auto-tuned kernels should be included for a more complete study.

- Multi-kernel/application co-tuning: Our multi-kernel co-tuning discussion in Section 3.5.3 was very simplistic as it ignored the interactions that may occur between different kernels. A successful co-tuning framework would attempt to solve the problem of co-tuning for a set of interacting kernels (and even applications) to make better hardware design decisions.

- Intelligent design space exploration for co-tuning: Current hardware DSE studies already explore very large design spaces using machine learning tools. Given that the co-tuning design space is orders of magnitude bigger due to the incorporation of the software design space, such tools are a necessity in order for co-tuning to be practically viable—this is despite the two orders of magnitude speedups in simulation offered by FPGA-acceleration. Thus, statistical models may be used to jointly model the hardware/software parameter space, in order to better predict the performance metrics as a function of the hardware/software parameter values. It would be interesting

to determine which modeling techniques would work for the joint hardware/software parameter space.

Our proposed co-tuning strategy offers a promising trade-off between the additional design cost of architectural customization and the portability and programmability of off-the-shelf microprocessors. Moreover, existing toolchains of companies like Tensilica enable a large space of hardware configurations, and the evolving maturity of auto-tuners for scientific kernels provides the ability to extract near-peak performance from these designs. Overall, this approach can provide a quantum leap in hardware utilization and energy efficiency, the primary metrics driving the design of the next-generation HPC systems.

Bibliography

[1] E. Agullo, J. Demmel, J. Dongarra, B. Hadri, J. Kurzak, J. Langou, H. Ltaief, P. Luszczek, and S. Tomov. Numerical linear algebra on emerging architectures: The plasma and magma projects. *Journal of Physics: Conference Series*, 180(1):012037, 2009.

[2] N. Azizi, I. Kuon, and A. Egier. Reconfigurable Molecular Dynamics Simulator. In *IEEE Symposium on Field-Programmable Custom Computing Machines*, April 2004.

[3] O. Azizi, A. Mahesri, B. C. Lee, S. J. Patel, and M. Horowitz. Energy-performance tradeoffs in processor architecture and circuit design: a marginal cost analysis. In *Proceedings of the 37th annual international symposium on Computer architecture*, ISCA '10, pages 26–36, New York, NY, USA, 2010. ACM.

[4] Z. Bai, D. Hu, and L. Reichel. A Newton basis GMRES implementation. *IMA Journal of Numerical Analysis*, 14:563–581, 1994.

[5] D. Bailey. Little's Law and High-Performance Computing, 1997.

[6] D. Bailey, E. Barszcz, J. Barton, et al. The NAS Parallel Benchmarks. Technical Report RNR-94-007, NASA, 1994.

[7] D. H. Bailey, J. Chame, C. Chen, J. Dongarra, M. Hall, J. K. Hollingsworth, P. Hovland, S. Moore, K. Seymour, J. Shin, A. Tiwari, S. Williams, and H. You. Peri auto-tuning. *Journal of Physics: Conference Series*, 125(1):012089, 2008.

[8] Z. K. Baker and V. K. Prasanna. Efficient Hardware Data Mining with the Apriori Algorithm on FPGAs. In *IEEE Symposium on Field-Programmable Custom Computing Machines*, April 2005.

[9] G. Ballard, J. Demmel, O. Holtz, and O. Schwartz. Minimizing Communication in Numerical Linear Algebra. *SIAM Journal of Matrix Analysis and Applications*, 32(3):866–901, 2011.

[10] M. Berger and J. Oliger. Adaptive Mesh Refinement for Hyperbolic Partial Differential Equations. *Journal of Computational Physics*, 53:484–512, 1984.

[11] J. Bilmes, K. Asanovic, C.-W. Chin, and J. Demmel. Optimizing matrix multiply using phipac: a portable, high-performance, ansi c coding methodology. Technical report, 1996.

[12] J. Bilmes, K. Asanovic, C. W. Chin, and J. Demmel. Optimizing matrix multiply using PHiPAC: a portable, high-performance, ANSI C coding methodology. In *ICS '97: Proceedings of the 11th International Conference on Supercomputing*, pages 340–347, New York, NY, USA, 1997. ACM Press.

[13] J. Bilmes, K. Asanovic, C.-W. Chin, and J. Demmel. Optimizing matrix multiply using phipac: a portable, high-performance, ansi c coding methodology. In *Proceedings of the 11th international conference on Supercomputing*, ICS '97, pages 340–347, New York, NY, USA, 1997. ACM.

[14] S. Borkar. Design Challenges of Technology Scaling. *IEEE Micro*, 19(4):23–29, 1999.

[15] P. N. Brown, R. D. Falgout, and J. E. Jones. Semicoarsening multigrid on distributed memory machines. *SIAM J. Sci. Comput.*, 21:1823–1834, December 1999.

[16] A. Buluc, S. Williams, L. Oliker, and J. Demmel. Reduced-bandwidth multithreaded algorithms for sparse matrix-vector multiplication. In *Proceedings of the 2011 IEEE International Parallel & Distributed Processing Symposium*, IPDPS '11, pages 721–733, Washington, DC, USA, 2011. IEEE Computer Society.

[17] B. Catanzaro, S. Kamil, Y. Lee, K. Asanovic, J. Demmel, K. Keutzer, J. Shalf, K. A. Yelick, and A. Fox. SEJITS: Getting Productivity and Performance With Selective Embedded JIT Specialization. Technical Report UCB/EECS-2010-23, EECS Department, University of California, Berkeley, March 2010.

[18] W. Chelton and M. Benaissa. Fast Elliptic Curve Cryptography on FPGA. *IEEE Transactions on VLSI Systems*, 16:198–205, February 2008.

[19] C. Chen, J. Chame, and M. Hall. CHiLL: A framework for composing high-level loop transformations. Technical report, University of Southern California, 2008.

[20] J. W. Choi, A. Singh, and R. W. Vuduc. Model-driven autotuning of sparse matrix-vector multiply on gpus. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '10, pages 115–126, New York, NY, USA, 2010. ACM.

[21] A. T. Chronopoulos and C. W. Gear. *s*-step iterative methods for symmetric linear systems. *J. Comput. Appl. Math.*, 25(2):153–168, 1989.

[22] E. S. Chung, M. K. Papamichael, E. Nurvitadhi, J. C. Hoe, K. Mai, and B. Falsafi. ProtoFlex: Towards Scalable, Full-System Multiprocessor Simulations Using FPGAs. *ACM Trans. Reconfigurable Technol. Syst.*, 2:15:1–15:32, June 2009.

[23] S. P. E. Corporation. SPEC Benchmarks, 2009. `http://www.spec.org/benchmarks.html`.

[24] K. Datta, M. Murphy, V. Volkov, et al. Stencil Computation Optimization and Auto-Tuning on State-of-the-art Multicore Architectures. In *SC '08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, 2008.

[25] K. Datta, M. Murphy, V. Volkov, S. Williams, J. Carter, L. Oliker, D. Patterson, J. Shalf, and K. Yelick. Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, SC '08, pages 4:1–4:12, Piscataway, NJ, USA, 2008. IEEE Press.

[26] J. Davis, C. Thacker, and C. Chang. BEE3: Revitalizing Computer Architecture Research. Technical Report MSR-TR-2009-45, Microsoft Research, 2009.

[27] J. D. Davis, J. Laudon, and K. Olukotun. Maximizing cmp throughput with mediocre cores. In *Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques*, PACT '05, pages 51–62, Washington, DC, USA, 2005. IEEE Computer Society.

[28] E. de Sturler. A parallel variant of GMRES(m). In J. J. H. Miller and R. Vichnevetsky, editors, *Proceedings of the 13th IMACS World Congress on Computation and Applied Mathematics*, Dublin, Ireland, 1991. Criterion Press.

[29] J. Demmel, J. Dongarra, V. Eijkhout, et al. Self Adapting Linear Algebra Algorithms and Software. In *Proc. of the IEEE: Special Issue on Program Generation, Optimization, and Adaptation*, February 2005.

[30] J. Demmel, J. Dongarra, V. Eijkhout, E. Fuentes, A. Petitet, R. Vuduc, C. Whaley, and K. Yelick. Self-adapting linear algebra algorithms and software. *Proceedings of the IEEE*, 93(2):293–312, February 2005.

[31] J. Demmel, M. Hoemmen, M. Mohiyuddin, and K. Yelick. Avoiding communication in computing Krylov subspaces. Technical Report UCB/EECS-2007-123, University of California Berkeley EECS, October 2007.

[32] J. Demmel, M. Hoemmen, M. Mohiyuddin, and K. Yelick. Avoiding Communication in Sparse Matrix Computations. In *Proceedings of IPDPS*, April 2008.

[33] J. Demmel, J. Langou, L. Grigori, and M. Hoemmen. Communication-Optimal Parallel and Sequential QR and LU Factorizations. *SIAM Journal of Scientific Computing*, 34(1), February 2012.

[34] C. Ding and Y. He. A ghost cell expansion method for reducing communications in solving PDE problems. In *Proceedings. of SC2001*, Nov. 2001.

[35] C. C. Douglas, J. Hu, M. Kowarschik, U. Rüde, and C. Weiss. Cache optimization for structured and unstructured grid multigrid. *Electronic Transaction on Numerical Analysis*, 10:21–40, Feb. 2000.

[36] C. Dubach, T. Jones, and M. O'Boyle. Microarchitectural design space exploration using an architecture-centric approach. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 40, pages 262–271, Washington, DC, USA, 2007. IEEE Computer Society.

[37] T. El-Ghazawi, W. Carlson, T. Sterling, and K. Yelick. *UPC: Distributed Shared-Memory Programming*. Wiley-Interscience, May 2005.

[38] J. Erhel. A parallel GMRES version for general sparse matrices. *Electronic Transactions on Numerical Analysis*, 3:160–176, 1995.

[39] M. Frigo and S. G. Johnson. The Design and Implementation of FFTW3. In *Proc. of the IEEE*, 2004.

[40] P. Gelsinger. Microprocessor for the New Millennium - Challenges, Opportunities and New Frontiers. In *ISSCC Digest of Technical Papers*, 2001.

[41] A. George. Nested dissection of a regular finite element mesh. *SIAM J. Num. Anal.*, 10:345–363, 1973.

[42] J. R. Gilbert and R. E. Tarjan. The analysis of a nested dissection algorithm. *Numerische Mathematik*, 50:377–404, 1986. 10.1007/BF01396660.

[43] K. Goto and R. Van De Geijn. High-performance implementation of the level-3 blas. *ACM Trans. Math. Softw.*, 35:4:1–4:14, July 2008.

[44] K. Goto and R. A. van de Geijn. Anatomy of high-performance matrix multiplication. *Transactions on Mathematical Software*, 34(3), 2008.

[45] M. Gschwind, V. Salapura, and D. Maurer. Fpga prototyping of a risc processor core for embedded applications. *IEEE Transactions on VLSI Systems*, 9:241–250, April 2001.

[46] M. Hoemmen. *Communication-Avoiding Krylov Subspace Methods*. PhD thesis, EECS Department, University of California, Berkeley, 2010.

[47] J.-W. Hong and H. T. Kung. I/O complexity: The red-blue pebble game. In *Proc. 13th Ann. ACM Symp. on Theory of Computing (May 11-13, 1981)*, pages 326–333, 1981.

[48] E.-J. Im, K. Yelick, and R. Vuduc. SPARSITY: An Optimization Framework for Sparse Matrix Kernels. *Int. J. HPCA*, February 2004.

[49] E. Ipek, S. A. McKee, K. Singh, R. Caruana, B. R. d. Supinski, and M. Schulz. Efficient architectural design space exploration via predictive modeling. *ACM Trans. Archit. Code Optim.*, 4:1:1–1:34, January 2008.

[50] F. Irigoin and R. Triolet. Supernode partitioning. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 319–329. ACM Press, 1988.

[51] 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.

[52] P. J. Joseph, K. Vaswani, and M. J. Thazhuthaveetil. A predictive performance model for superscalar processors. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 39, pages 161–170, Washington, DC, USA, 2006. IEEE Computer Society.

[53] W. D. Joubert and G. F. Carey. Parallelizable restarted iterative methods for nonsymmetric linear systems, Part I: Theory. *International Journal of Computer Mathematics*, 44:243–267, 1992.

[54] C. Kamath, R. Ho, and D. P. Manley. DXML: A High-performance Scientific Subroutine Library, 1994.

[55] S. Kamil, C. Chan, L. Oliker, J. Shalf, and S. Williams. An auto-tuning framework for parallel multicore stencil computations. In *Proceedings of the International Symposium on Parallel and Distributed Processing*, IPDPS '10, pages 1–12, 2010.

[56] S. Kamil, K. Datta, S. Williams, L. Oliker, J. Shalf, and K. Yelick. Implicit and explicit optimizations for stencil computations. In *Memory Systems Performance and Correctness*, San Jose, CA, Oct. 2006.

[57] S. Kamil, P. Husbands, L. Oliker, J. Shalf, and K. Yelick. Impact of modern memory subsystems on cache optimizations for stencil computations. In *3rd Annual ACM SIGPLAN Workshop on Memory Systems Performance*, Chicago, IL, 2005.

[58] G. Karypis and V. Kumar. A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs. In *International Conference on Parallel Processing*, 1995.

[59] P. Kogge, K. Bergman, S. Borkar, D. Campbell, W. Carlson, W. D. M. Denneau, P. Franzon, W. Harrod, K. Hill, J. Hiller, S. Karp, S. Keckler, D. Klein, R. Lucas, M. Richards, A. Scarpelli, S. Scott, A. Snavely, T. Sterling, R. S. Williams, and K. Yelick. ExaScale Computing Study: Technology Challenges in Achieving Exascale Systems, 2008. `http://users.ece.gatech.edu/~mrichard/ExascaleComputingStudyReports/exascale_final_report_100208.pdf`.

[60] R. Kumar, V. Zyuban, and D. M. Tullsen. Interconnections in Multi-core Architectures: Understanding Mechanisms, Overheads and Scaling. In *Proceedings of the International Symposium on Computer Architecture*, pages 408–419, June 2005.

[61] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh. Basic linear algebra subprograms for fortran usage. *ACM Trans. Math. Softw.*, 5:308–323, September 1979.

[62] B. C. Lee and D. M. Brooks. Accurate and efficient regression modeling for microarchitectural performance and power prediction. *SIGARCH Comput. Archit. News*, 34:185–194, October 2006.

[63] C. E. Leiserson, S. Rao, and S. Toledo. Efficient out-of-core algorithms for linear relaxation using blocking covers (extended abstract). In *IEEE Symposium on Foundations of Computer Science*, pages 704–713, 1993.

[64] J. Leverich, H. Arakida, A. Solomatnikov, A. Firoozshahian, M. Horowitz, and C. Kozyrakis. Comparing Memory Systems for Chip Multiprocessors. In *Proceedings of the International Symposium on Computer Architecture*, 2007.

[65] I. T. Li, W. Shum, and K. Truong. 160-fold acceleration of the Smith-Waterman algorithm using a field programmable gate array (FPGA). *BMC Bioinformatics*, 8(185), 2007.

[66] Y. Li, B. Lee, D. Brooks, Z. Hu, and K. Skadron. CMP design space exploration subject to physical constraints. In *High-Performance Computer Architecture, 2006*, pages 17–28, Feb. 2006.

[67] C. Liao, D. Quinlan, R. Vuduc, and T. Panas. Effective source-to-source outlining to support whole program empirical optimization. In G. Gao, L. Pollock, J. Cavazos, and X. Li, editors, *Languages and Compilers for Parallel Computing*, volume 5898 of *Lecture Notes in Computer Science*, pages 308–322. Springer Berlin / Heidelberg, 2010. 10.1007/978-3-642-13374-9.

[68] M. Lin, I. Lebedev, and J. Wawrzynek. High-throughput bayesian computing machine with reconfigurable hardware. In *Proceedings of the 18th annual ACM/SIGDA international symposium on Field programmable gate arrays*, FPGA '10, pages 73–82, New York, NY, USA, 2010. ACM.

[69] R. J. Lipton, D. J. Rose, and R. E. Tarjan. Generalized nested dissection. *SIAM Journal on Numerical Analysis*, 16(2):346–358, 1979.

[70] K. Mai, T. Paaske, N. Jayasena, R. Ho, W. Dally, and M. Horowitz. Smart Memories: A Modular Reconfigurable Architecture. In *Proc. of the International Symposium on Computer Architecture*, pages 161–171, 2000.

[71] J. McCalpin and D. Wonnacott. Time skewing: A value-based approach to optimizing for memory locality. Technical Report DCS-TR-379, Department of Computer Science, Rutgers University, 1999.

[72] Micron Inc. Calculating Memory System Power for DDR2, June 2006. `http://download.micron.com/pdf/technotes/ddr2/TN4704.pdf`.

[73] M. Mohiyuddin, M. Hoemmen, J. Demmel, and K. Yelick. Minimizing communication in sparse matrix solvers. In *Proceedings of Supercomputing*, November 2009.

[74] M. Mohiyuddin, M. Murphy, J. Shalf, L. Oliker, J. Wawrzynek, and S. Williams. A Methodology for Domain-Optimized Power-Efficient Supercomputers. In *Proceedings of Supercomputing*, November 2009.

[75] M. Monchiero, R. Canal, and A. González. Design space exploration for multicore architectures: a power/performance/thermal view. In *ICS '06: Proceedings of the International Conference on Supercomputing*, pages 177–186, New York, NY, USA, 2006. ACM.

[76] E. Musoll and M. Nemirovsky. Design Space Exploration of High-Performance Parallel Architectures. In *Journal of Integrated Circuits and Systems*, 2008.

[77] R. Nishtala and K. Yelick. Optimizing collective communication on multicores. In *First USENIX Workshop on Hot Topics in Parallelism*, 2009.

[78] D. Patterson. Latency lags bandwidth. *CACM*, 47(10):71–75, Oct 2004.

[79] J.-K. Peir. *Program partitioning and synchronization on multiprocessor systems*. PhD thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, Mar. 1986.

[80] M. Pellauer, M. Adler, M. Kinsy, A. Parashar, and J. Emer. HAsim: FPGA-Based High-Detail Multicore Simulation Using Time-Division Multiplexing. In *Proceedings of the International Symposium on High Performance Computer Architecture*, pages 406–417, February 2011.

[81] C. J. Pfeifer. Data flow and storage allocation for the PDQ-5 program on the Philco-2000. *Communications of the ACM*, 6(7):365–366, 1963.

[82] M. Püschel, J. M. F. Moura, B. Singer, J. Xiong, J. Johnson, D. Padua, M. Veloso, and R. W. Johnson. SPIRAL: A Generator for Platform-Adapted Libraries of Signal Processing Algorithms. In *Journal of High Performance Computing Applications*, 2004.

[83] G. Rivera and C. Tseng. Tiling Optimizations for 3D Scientific Computations. In *Proceedings of SC'00*, Dallas, TX, November 2000.

[84] E. J. Rosser. *Fine-grained analysis of array computations*. PhD thesis, Dept. of Computer Science, University of Maryland, Sept. 1998.

[85] Y. Saad. Practical use of polynomial preconditionings for the conjugate gradient method. *SIAM J. Sci. Stat. Comput.*, 6(4), Oct. 1985.

[86] Y. Saad and M. H. Schultz. GMRES: a generalized minimal residual algorithm for solving nonsymmetric linear systems. *SIAM J. Sci. Stat. Comput.*, 7(3), 1986.

[87] G. Schelle, J. Collins, E. Schuchman, P. Wang, X. Zou, G. Chinya, R. Plate, T. Mattner, F. Olbrich, P. Hammarlund, R. Singhal, J. Brayton, S. Steibl, and H. Wang. Intel Nehalem Processor Core Made FPGA Synthesizable. In *Proceedings of the 18th annual ACM/SIGDA international symposium on Field programmable gate arrays*, FPGA '10, pages 3–12, New York, NY, USA, 2010. ACM.

[88] M. Schordan and D. Quinlan. A source-to-source architecture for user-defined optimizations. In *JMLC'03: Joint Modular Languages Conference*, volume 2789 of *Lecture Notes in Computer Science*, pages 214–223. Springer Verlag, Aug. 2003.

[89] S. Seo, J. Lee, and Z. Sura. Design and Implementation of Software-Managed Caches for Multicores with Local Memory. In *Proceedings of the International Symposium on High Performance Computer Architecture*, pages 55–66, February 2009.

[90] A. Shrivastava, I. Issenin, and N. Dutt. A Compiler-in-the-Loop Framework to Explore Horizontally Partitioned Cache Architectures. In *Proceedings of ASPDAC 2008*, pages 328–333, March 2008.

[91] H. Simon, R. Stevens, T. Zacharia, et al. Modeling and Simulation at the Exascale for Energy and the Environment (E3). Technical report, DOE ASCR Program Technical Report, 2008.

[92] M. Snir and S. Graham, editors. *Getting up to speed: The Future of Supercomputing*. National Research Council, 2004. 227 pages.

[93] Y. Song and Z. Li. New tiling techniques to improve cache temporal locality. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation*, Atlanta, GA, 1999.

[94] M. M. Strout. Communiation avoidance for sparse applications using full sparse tiling. Minisymposium talk at the SIAM Parallel Processing 2008 conference, 2008.

[95] M. M. Strout, L. Carter, and J. Ferrante. Rescheduling for locality in sparse matrix computations. In V. N. Alexandrov and J. J. Dongarra, editors, *Lecture Notes in Computer Science*. Springer, 2001.

[96] D. Takahashi. A Blocking Algorithm for FFT on Cache-Based Processors. In B. Hertzberger, A. Hoekstra, and R. Williams, editors, *High-Performance Computing and Networking*, volume 2110 of *Lecture Notes in Computer Science*, pages 551–554. Springer Berlin / Heidelberg, 2001.

[97] Z. Tan, A. Waterman, R. Avizienis, Y. Lee, H. Cook, D. Patterson, and K. Asanovic. RAMP Gold: An FPGA-based Architecture Simulator for Multiprocessors. In *Proceedings of the ACM/IEEE Design Automation Conference*, DAC'10, pages 463–468, 2010.

[98] Tensilica Inc. Xtensa Architecture and Performance. Whitepaper, October 2005. `http://www.tensilica.com/pdf/xtensa_arch_white_paper.pdf`.

[99] S. Thoziyoor, N. Muralimanohar, J. H. Ahn, and N. P. Jouppi. CACTI 5.1. Technical Report HPL-2008-20, HP Labs, 2008.

[100] S. Toledo. *Quantitative performance modeling of scientific computations and creating locality in numerical algorithms.* PhD thesis, Massachusetts Institute of Technology, June 1995.

[101] Top500.org. TOP500 List, June 2008. `http://www.top500.org`.

[102] J. van Rosendale. Minimizing inner product data dependence in conjugate gradient iteration. In *Proc. IEEE Internat. Confer. Parallel Processing*, 1983.

[103] V. Volkov and J. Demmel. Benchmarking GPUs to tune dense linear algebra. In *Proceedings of Supercomputing*, SC '08, pages 1–11, 2008.

[104] R. Vuduc. *Automatic Performance Tuning of Sparse Matrix Kernels.* PhD thesis, University of California Berkeley, December 2003.

[105] R. Vuduc, J. Demmel, and K. Yelick. OSKI: A library of automatically tuned sparse matrix kernels. In *Proc. of SciDAC 2005, J. of Physics: Conference Series.* Institute of Physics Publishing, June 2005.

[106] R. Vuduc, J. Demmel, and K. A. Yelick. OSKI: A library of automatically tuned sparse matrix kernels. In *Proceedings of SciDAC*, Journal of Physics: Conference Series, San Francisco, CA, June 2005.

[107] C. D. Wait. IBM PowerPC 440 FPU with Complex-Arithmetic Extensions. *IBM Journal of Research and Development*, 49(2-3):249–254, 2005.

[108] H. F. Walker. Implementation of the GMRES method using Householder transformations. *SIAM J. Sci. Stat. Comput.*, 9(1):152–163, 1988.

[109] P. H. Wang, J. D. Collins, C. T. Weaver, B. Kuttanna, S. Salamian, G. N. Chinya, E. Schuchman, O. Schilling, T. Doil, S. Steibl, and H. Wang. Intel Atom Processor Core Made FPGA-Synthesizable. In *Proceedings of the ACM/SIGDA international symposium on Field programmable gate arrays*, FPGA '09, pages 209–218, New York, NY, USA, 2009. ACM.

[110] J. Wawrzynek, D. Patterson, M. Oskin, S.-L. Lu, C. Kozyrakis, J. C. Hoe, D. Chiou, and K. Asanovic. RAMP: A Research Accelerator for Multiple Processors. *IEEE Micro*, 27(2), 2007.

[111] M. Wehner, L. Oliker, J. Shalf, D. Donofrio, L. Drummond, R. Heikes, S. Kamil, C. Konor, N. Miller, H. Miura, M. Mohiyuddin, D. Randall, and W.-S. Yang. Hardware/Software Co-design of Global Cloud System Resolving. *Journal of Advances in Modeling Earth Systems*, 3(12), 2011.

[112] R. C. Whaley, A. Petitet, and J. Dongarra. Automated Empirical Optimizations of Software and the ATLAS Project. *Parallel Computing*, 27(1):3–25, 2001.

[113] S. Williams, L. Oliker, J. Shalf, and K. Yelick. Optimization of Sparse Matrix-Vector Multiplication on Emerging Multicore Platforms. In *Proceedings of Supercomputing*, 2007.

[114] S. Williams, L. Oliker, R. Vuduc, J. Shalf, K. Yelick, and J. Demmel. Optimization of sparse matrixvector multiplication on emerging multicore platforms. *Parallel Computing*, 35(3):178 – 194, 2009.

[115] M. E. Wolf. *Improving locality and parallelism in nested loops*. PhD thesis, Stanford University, 1992.

[116] D. Wonnacott. Using time skewing to eliminate idle time due to memory bandwidth and network limitations. In *Proceedings of the Fourteenth International Parallel and Distributed Processing Symposium (IPDPS)*, pages 171–180, 2000.

[117] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *Proceedings of ISCA*, 1995.

[118] N. A. Woods and T. VanCourt. FPGA acceleration of quasi-Monte Carlo in finance. In *Proceedings of International Conference on Field Programmable Logic and Applications*, September 2008.

[119] P. R. Woodward and S. E. Anderson. Scaling the Teragrid by latency tolerant application design. In *Proc. of NSF / Department of Energy Scaling Workshop*, Pittsburg, CA, May 2002.

[120] R. Wunderlich and J. Hoe. In-system FPGA prototyping of an Itanium microarchitecture. In *IEEE International Conference on Computer Design*, October 2004.

[121] Q. Yi, K. Seymour, H. You, R. Vuduc, and D. Quinlan. POET: Parameterized Optimizations for Empirical Tuning. In *Workshop on Performance Optimization for High-Performance Languages and Libraries*, 2007.

[122] S. Zierke and J. D. Bakos. FPGA acceleration of the phylogenetic likelihood function for Bayesian MCMC inference methods. *BMC Bioinformatics*, 11(184), 2010.