Title
An I/O-Complexity Lower Bound for All Recursive Matrix Multiplication Algorithms by Path-Routing

Author
Scott, Jacob N.

Publication Date
2015

Peer reviewed|Thesis/dissertation

An I/O-Complexity Lower Bound for All Recursive Matrix Multiplication Algorithms by Path-Routing

by

Jacob N. Scott

A dissertation submitted in partial satisfaction of the

requirements for the degree of

Doctor of Philosophy

in

Mathematics

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Professor Olga V. Holtz, Chair
Professor James W. Demmel
Professor Satish B. Rao

Fall 2015

An I/O-Complexity Lower Bound for All Recursive Matrix Multiplication Algorithms by Path-Routing

Abstract

An I/O-Complexity Lower Bound for All Recursive Matrix Multiplication Algorithms by Path-Routing

by

Jacob N. Scott

Doctor of Philosophy in Mathematics

University of California, Berkeley

Professor Olga V. Holtz, Chair

Via novel path-routing techniques we prove a lower bound on the I/O-complexity of all recursive matrix multiplication algorithms computed in serial or in parallel and show that it is tight for all square and near-square matrix multiplication algorithms. Previously, tight lower bounds were known only for the classical $\Theta(n^3)$ matrix multiplication algorithm and those similar to Strassen's algorithm that lack multiple vertex copying. We first prove tight lower bounds on the I/O-complexity of Strassen-like algorithms, under weaker assumptions, by constructing a routing of paths between the inputs and outputs of sufficiently small subcomputations in the algorithm's CDAG. We then further extend this result to all recursive divide-and-conquer matrix multiplication algorithms, and show that our lower bound is optimal for algorithms formed from square and nearly square recursive steps. This requires combining our new path-routing approach with a secondary routing based on the Loomis-Whitney Inequality technique used to prove the optimal I/O-complexity lower bound for classical matrix multiplication.

Contents

List of Figures

List of Tables

Acknowledgments

Thanks to Olga Holtz for introducing me to this field of research, Oded Schwartz for assistance with the paper of Chapter 2, and James Demmel for his helpful comments.

Chapter 1

Introduction

In this paper we develop a new technique to prove lower bounds on the I/O-complexity of algorithms and use it to derive lower bounds on the I/O-complexity of recursive divide-and-conquer matrix multiplication algorithms. We show that our bound is tight for square matrix multiplication algorithms, as well as for "near-square" matrix multiplication algorithms, those whose multiplication subproblems each multiply matrices of dimensions within a constant factor. As explained in detail in Chapter 2, the I/O-complexity of an algorithm is the minimum number of cache reads/writes necessary to compute it; I/O-complexity lower bounds are therefore of great interest when designing "big data" algorithms where cache operations can be more costly than arithmetic computations. Because matrix multiplication underlies many problems in numerical linear algebra – upon which many big data algorithms are based – we prove lower bounds for any recursive divide-and-conquer matrix multiplication algorithm, with the understanding that the techniques developed can likely be applied to other linear algebra problems as well.

This dissertation is composed of two parts. First we present a paper by myself, Holtz, and Schwartz [16] in which we prove an optimal lower bound for the I/O-complexity of a subset of matrix multiplication algorithms (also under an assumption removed later in this dissertation), called *Strassen-like* via a new technique based on path routings. This novel approach overcomes many of the difficulties of existing techniques in this field, as detailed in Chapter 2. The text of this paper has been extended slightly to better introduce the concepts used in succeeding chapters.

Any recursive divide-and-conquer matrix multiplication algorithm is formed from a tree of recursive steps, not necessarily all taking the same form. Each recursive step – called a *base graph* in Chapter 2 – represents an algorithm for multiplying small matrices applied to blocks of larger matrices, with multiplications of blocks performed recursively. In a Strassen-like algorithm, each recursive step is identical

and multiplies square matrices, but the approach in Chapter 2 can easily be extended to apply even when the recursive steps are not uniform. However, if any recursive step is as slow as classical $\Theta(n^3)$ matrix multiplication, then the approach in Chapter 2 is not always sufficient. Many matrix multiplication algorithms used in practice involve recursing until the matrices to multiply are of sufficiently small size and then reverting to classical matrix multiplication; such algorithms are thus not covered by Chapter 2.

We then show how to further generalize the technique of Chapter 2 to any recursive divide-and-conquer matrix multiplication algorithm. First we extend the approach to rectangular matrix multiplication, potentially involving recursive steps of different dimensions. Then we show how to combine the path-routing approach of Chapter 2 with the Loomis-Whitney Inequality technique [12] [6] – used previously to prove I/O-complexity lower bounds for classical matrix multiplication – to yield our general result. The bound we derive has a dependence on the maximum number of subblocks into which a matrix is divided in any recursive step, which we assume is a constant. Finally, we show how to extend this line of reasoning to prove a lower bound on the I/O-complexity of recursive matrix multiplication algorithms computed in parallel. These bounds are asymptotically optimal for square matrix multiplication algorithms.

For convenience we now state the main results proved in this dissertation:

Theorem 1 (Main Theorem – see Theorem 7 of Chapter 3) *Let G be the computation graph (CDAG) for a recursive divide-and-conquer rectangular matrix multiplication that computes distinct products in each recursive step. If the exponent $\omega(G) \leq 3$ and the maximum number of blocks a matrix is divided into in any subcomputation is a small constant, then G has I/O-complexity*

$$IO(G) \geq \Omega\left(\frac{Mult}{\sqrt{M}^{\,\omega(G)}} \cdot M\right)$$

where Mult is the number of elementary multiplications computed by G. If $\omega(G) > 3$, the matrix multiplication algorithm can be "simplified" to one requiring fewer arithmetic operations and no more cache I/Os to which this result also applies.

Theorem 2 (Main Parallel Theorem – see Theorem 29 of Chapter 7) *Let G be as above. If G computes Mult elementary multiplications, then the I/O-complexity of G computed in parallel by P processors is*

$$IO(G) \geq \Omega\left(\frac{1}{P} \cdot \frac{Mult}{\sqrt{M}^{\,\omega(G)}} \cdot M\right)$$

The definition of CDAG and of the I/O-complexity of a CDAG computed in parallel appear in the following chapter. The definition of $\omega(G)$ appears in Chapter 3; intuitively, $\omega(G)$ is the number such that, if every step in the recursive matrix multiplication were identical to the least efficient one used, G would contain $\Theta\left(n^{\omega(G)}\right)$ elementary multiplications for input dimension n. Note that these results apply to classical matrix multiplication, as well as any algorithm interleaving classical and fast matrix multiplication recursive steps, and are independent of the implementation (ordering of intermediate computations) of the algorithm. These results are the natural generalizations of the current lower bounds in the literature [5] [12] [4] to any recursive matrix multiplication algorithm, and are optimal for square and nearly square matrix multiplication algorithms.

Chapter 2

Matrix Multiplication I/O-Complexity by Path Routing

This chapter is adapted from a paper I published [16] with coauthors Olga Holtz and Oded Schwartz: http://dx.doi.org/10.1145/2755573.2755594

In practice, most of the runtime of an algorithm is often due to the communication of data within memory hierarchy and between multiple processors, rather than the arithmetic computations. The amount of communication performed during an algorithm depends on the order in which intermediate values are computed and kept in/discarded from cache. While much work has gone into constructing implementations of algorithms that reduce communication, in this paper we show lower bounds on the communication of any implementation of a common class of fast (but not classical; see Lemma 1) matrix multiplication algorithms.

The I/O-complexity of an algorithm is defined as the minimum possible number of cache operations required to compute all outputs of the algorithm using a fixed cache size M. In 2011, Ballard, Demmel, Holtz, and Schwartz showed a tight lower bound on the I/O-complexity of Strassen's fast matrix multiplication algorithm [5]. We prove an analogous I/O-complexity bound via a more general technique for any fast square matrix multiplication algorithm based on a uniform recursive step that does not recompute any intermediate values, subject to the assumption that every intermediate linear combination is used in only one multiplication. We also claim, without proof, that this assumption can be lifted. Because algorithms achieving our I/O-complexity bounds have been found [3], our bounds are optimal.

Machine model

In this paper, we assume a 2-layer memory hierarchy for sequential computations

consisting of slow memory and fast memory. The slow memory is of unlimited size and represents, for example, the hard drive of a computer, while the fast memory, which we call cache, is of limited size M and may represent RAM. We model the I/O communication of an algorithm as follows: initially, all data resides in slow memory and the cache is empty. A single value may be input into cache from slow memory or output to slow memory from cache for the cost of one I/O. A computation in the algorithm may only be performed if all input values to that computation already reside in cache; when computed, the result is also put in cache. The algorithm halts when all outputs of the algorithm are stored in slow memory. In this model we assume that no arithmetic computation is ever performed more than once. See [10] for the formalization of this model as a pebble game played on the computation graph.

The number of cache I/Os required may depend on the order in which intermediate values of the algorithm are computed. The algorithm's *I/O-complexity* is thus defined as the minimum number of I/Os over all sequences of computations and I/Os that computes the algorithm's outputs.

For parallel computations we consider P processors, each having independent local memory of size M. As in [5] and [20], we define the bandwidth cost of an algorithm executed in parallel to be the number of values communicated between processors along the critical path. In other words, we count the total number of words (single values) sent between processors, except that words sent between processors simultaneously count as only one I/O. We call this the *bandwidth cost* of the algorithm.

Previous Work

In 1981 Hong and Kung [10] proved a tight lower bound on the I/O-complexity of the classical $\Theta(n^3)$ matrix multiplication algorithm (achieved by blocked multiplication) using S-partitions. A different proof of this result was given in [12] and later generalized in [6] via the Loomis-Whitney inequality [14]; this approach was also shown to apply to several other problems in numerical linear algebra. See [1] and [9] for further generalizations using other geometric bounds. However, these proofs do not easily apply to algorithms that use distributivity for cancellation, such as Strassen's algorithm.

The edge expansion approach detailed in [5] relates the I/O-complexity of an algorithm to the edge expansion properties of the underlying computation graph. This technique provides an I/O-complexity lower bound for Strassen's fast matrix multiplication algorithm, but fails for algorithms with base graphs (the computation graph representing one recursive step; see Section 2.2) containing disconnected encoding and decoding graphs and those involving multiple copying. In [4], this

approach is extended to fast recursive matrix multiplication algorithms for rectangular matrices whose base graphs consist of multiple equal-size connected components. This is sufficient to yield lower bounds for some common fast matrix multiplication algorithms, such as Bini's algorithm [8] and the Hopcroft-Kerr algorithm [11], but still does not address algorithms with general base graphs.

In this paper we present the first approach for proving I/O-complexity lower bounds for recursive fast matrix multiplication algorithms involving arbitrary base graphs, as long as the same base graph is used at each recursive step.

2.1 New Approach

Most previous lower bounds in this field are based on the Loomis-Whitney inequality (as in [12]), dominator sets/S-partitions (as in [10], [15], and [7]), or edge expansions (as in [5] and [4]). In this paper we apply a new technique, based on the existence of a routing of paths within the underlying computation graph. In particular, we show the existence of a set of paths between all the inputs and all the outputs of sufficiently large matrix multiplication subcomputations such that each vertex is hit relatively few times. We then show that if some, but not all, of these input and output vertices are to be computed in one computation segment, then there must exist many other vertices that contribute cache I/Os as a result. This new approach may generalize to other problems that have sufficient symmetry to guarantee the existence of an efficient routing.

2.2 Preliminaries

As in [5], we define the computation directed acyclic graph (CDAG) of an algorithm to be the directed graph that contains a vertex for every value in the computation (input, output, or intermediate value) and an edge, from input value to output value, whenever one value depends directly on another. For example, if $y = x_1 - 2x_2 + 4x_3$, then there are edges (x_1, y), (x_2, y), and (x_3, y) in the CDAG.

Strassen's matrix multiplication algorithm works as follows [17]: to multiply 2×2 matrices A and B, compute specific linear combinations of the entries of A and linear combinations of the entries of B, perform 7 multiplications of these linear combinations, and then take linear combinations of the results to get the entries of $C = AB$. For larger square input matrices, divide each input matrix in half horizontally and vertically and apply the above procedure, recursively computing the necessary products of submatrices.

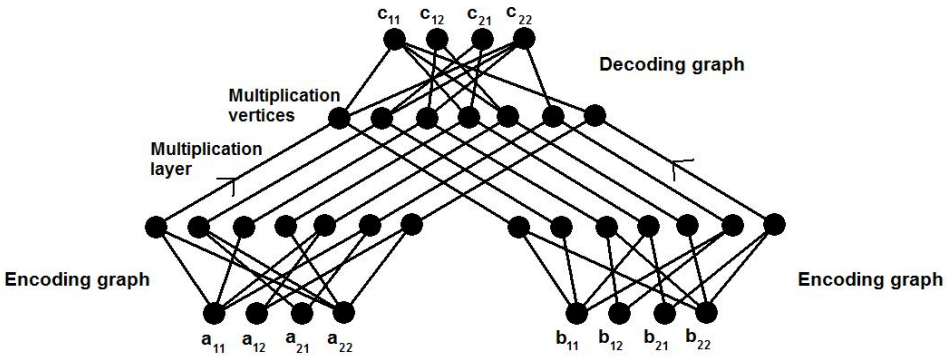
A *Strassen-like* algorithm is a square matrix multiplication algorithm that takes a similar form: to multiply matrices of dimensions $n_0 \times n_0$, take linear combinations of the input matrices, compute products, and take linear combinations of the results to yield the entries of the output matrix. For larger matrices, divide into blocks and recurse.

Let G_r be the CDAG of a Strassen-like algorithm for $n_0^r \times n_0^r$ square matrix multiplication $C = AB$, necessarily consisting of r recursive levels. We call G_1 the *base graph*. G_1 consists of two *encoding graphs*, which compute linear combinations of entries of A and of B, a *multiplication layer* with b *multiplication vertices*, which compute products of these linear combinations, and then a *decoding graph*, which takes linear combinations of these products to yield the entries of C. Note that G_1 has $2n_0^2$ inputs, n_0^2 from each input matrix. Further note that the same linear combination of input elements may be used as inputs in multiple product vertices. In this paper all figures show computations that proceed from bottom to top; we therefore omit the directions of edges. See Figure 2.1.

Note that G_r is a ranked graph, with inputs on rank 0 and outputs on rank $2r$. Ranks 0 through r lie in the encoding graphs and ranks $r + 1$ through $2r$ lie in the decoding graph; the multiplication layer occurs between ranks r and $r + 1$.

An intermediate vertex in G_r may have a single input vertex and, in this case, may have the same value as its one input. We call this *copying*; if the same value is copied to more than one child vertex, we call it *multiple copying*. We could consider this an artifact of our drawing of G_r and choose to identify these vertices. However, doing so would break the simple ranked, recursive structure of G_r. Instead, we group all vertices that represent the same value into a single *meta-vertex*. The vertices corresponding to each meta-vertex form a chain in the case of single copying and an

Figure 2.1: The base graph G_1 of Strassen's algorithm for multiplying two 2×2 matrices A and B. Here $b = 7$.



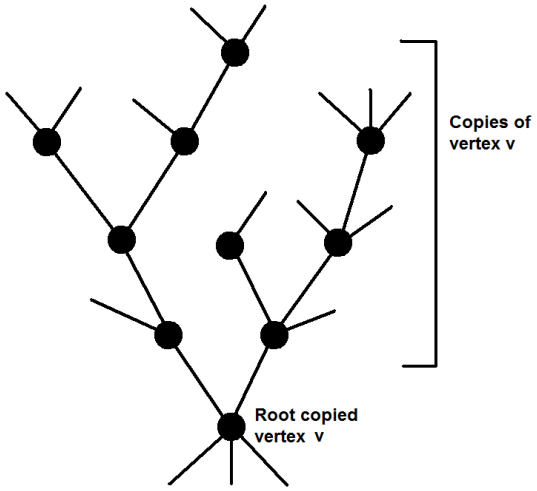
upwards-branching subtree of the CDAG in the case of multiple copying, where each vertex of the subtree apart from the root has no other edges entering it from below. See Figure 2.2 for a depiction of a meta-vertex in the case of multiple copying. For most of this chapter we consider only vertices, not meta-vertices, and then show that our technique still applies when copying or multiple copying occurs.

The approach in [5] fails when the decoding graph of the base graph G_1 has a disconnected encoding or decoding graph. Note that the entire CDAG G_r (and similarly G_1) must be connected simply because it computes matrix multiplication (this will be shown in greater detail in the process of proving Lemma 4), but the decoding graph and/or encoding graph may not be connected individually, as is the case for classical matrix multiplication.

In this paper, we first demonstrate a technique to derive the I/O-complexity bound for Strassen's algorithm presented in [5] more easily. We then show how to extend the technique, via use of Theorem 4, to the case of disconnected decoding and/or encoding graphs, allowing us to derive strong lower bounds for all Strassen-like matrix multiplication algorithms in which each linear combination is used in only one multiplication. This will prove Theorem 3, our main result. Finally, we present a proof of Theorem 4.

Theorem 3 (Main Theorem) *Let a and b be small constants. Consider a Strassen-like matrix multiplication algorithm for $n \times n$ matrices with arithmetic complexity $o(n^3)$ using cache size $M = o(n^2)$ in which the base graph has $2a$ inputs and b multiplication vertices. If in the base graph every nontrivial linear combination of elements of the input matrices is used in only one multiplication, then the algorithm*

Figure 2.2: The meta-vertex corresponding to copies of the vertex v. Edges whose endpoints are not shown denote edges to vertices not in the shown meta-vertex. If this meta-vertex is in the CDAG for Strassen-like matrix multiplication, the structure of the meta-vertex is actually more regular than depicted due to the simple recursion.



has I/O-complexity

$$\Omega\left(\left(\frac{n}{\sqrt{M}}\right)^{2\log_a b}\cdot M\right).$$

If run on P processors each of local cache size M, then the bandwidth cost is

$$\Omega\left(\left(\frac{n}{\sqrt{M}}\right)^{2\log_a b}\cdot \frac{M}{P}\right).$$

In other words, if a Strassen-like matrix multiplication algorithm performs $\Theta(n^{\omega_0})$ arithmetic operations with $\omega_0 < 3$, then its I/O-complexity is

$$\Omega\left(\left(\frac{n}{\sqrt{M}}\right)^{\omega_0}\cdot M\right).$$

If run on P processors, the bandwidth cost is

$$\Omega\left(\left(\frac{n}{\sqrt{M}}\right)^{\omega_0}\cdot \frac{M}{P}\right).$$

Furthermore, regardless of the cache size the bandwidth cost is

$$\Omega\left(\frac{n^2}{P^{2/\omega_0}}\right)$$

as long as computation is load balanced per rank of the computation graph.

In [3] an explicit divide-and-conquer algorithm is given that attains the bounds in Theorem 3. The sequential-to-parallel argument from [2] (as well as [5], [12], and [6]) allows us to take $P = 1$ – that is, work entirely in the serial model – and get the factor of $\frac{1}{P}$ in the parallel case with no additional work. Therefore, the remainder of this paper is devoted to proving a lower bound of $\Omega\left(\left(\frac{n}{\sqrt{M}}\right)^{2\log_a b} \cdot M\right)$ in the sequential case, from which Theorem 3 follows. By [3], the lower bounds in Theorem 3 are optimal.

2.3 Definitions

The proof presented in [5] relies on the notion of edge expansion; it shows a lower bound for the edge expansion of small subsets of vertices of the CDAG for Strassen's algorithm and then applies a lemma to yield a better edge expansion bound that relies on the fact that G_r contains as subgraphs many edge-disjoint copies of G_k for $k < r$. In our proof we bypass edge expansions entirely by explicitly cutting G_r into many copies of G_k for $k < r$. Both methods rely on the following fact, which is a consequence of the recursive definition of Strassen-like algorithms:

Fact 1 *For $0 \leq k \leq r$, let $G_{r,k}$ be the induced subgraph of G_r formed by the middle $2(k+1)$ levels of vertices (i.e. ranks $r-k$ through r of the encoding graphs and rank 0 through k of the decoding graph). Then $G_{r,k}$ consists of b^{r-k} vertex-disjoint copies of the graph G_k.*

In other words, the middle $2(k+1)$ layers of G_r are responsible for computing b^{r-k} independent matrix multiplications of square matrices of size $n_0^k \times n_0^k$.

Definition 1 *For any subset S of vertices of a computation graph G with directed edges E, define the following:*

1. $R(S) = \{v \in G - S \mid \text{ for some } w \in S,\ (v, w) \in E\}$
2. $W(S) = \{v \in S \mid \text{ for some } w \in G - S,\ (v, w) \in E\}$
3. $\delta(S) = R(S) \bigcup W(S)$

Note that $R(S)$ and $W(S)$ are disjoint, so $|\delta(S)| = |R(S)| + |W(S)|$. If S denotes a set of consecutively-computed vertices of G, then $R(S)$ denotes the set of vertices of G that must be read into cache, if not already present, during the computation of the vertices of S, and $W(S)$ the set of vertices of G that must be written to cache, if not to remain in cache after the computation of S. We assume that no vertex in G is ever computed more than once, meaning that if a vertex is used in the computations of multiple other vertices, it must either remain in cache until all the computations of vertices depending on it have finished or else be written to and read from cache.

We also assume that every linear combination of inputs in the base graph – except for the inputs themselves – is used in at most one multiplication in the base graph; this implies that every meta-vertex in the base graph is either a single vertex or else is rooted at one of the input vertices.

If S' is a subset of meta-vertices of G, we similarly define
$\delta'(S') = \{$meta-vertex v' of G not in $S' \mid$
for some $w' \in S',\ v'$ and w' are adjacent$\}$,
where two meta-vertices v' and w' are considered to be adjacent if for some vertex $v \in v'$ and vertex $w \in w'$, $(v, w) \in E$ or $(w, v) \in E$. In other words, $\delta'(S')$ is the set of meta-vertices adjacent to any of those in S'.

The main proof in this paper is based on finding routings of paths between sets of vertices in subgraphs of the CDAG that avoid using any vertex too many times. To this end we make the following definition:

Definition 2 *If X and Y are subsets of the vertices $V(G)$ of a directed graph G, define an m-*routing between X and Y *to be a collection R of $|X||Y|$ paths such that for any $x \in X$ and $y \in Y$ there exists a path, ignoring the directedness of edges, in G between x and y and such that every vertex of G is used collectively amongst all the paths in R at most m times. Similarly, if F is a subset of $V(G) \times V(G)$, define an m-*routing for F *to be a collection of paths, one for every $(v, w) \in F$ connecting v and w, such that every vertex of G is hit at most m times.*

We will consider only the case where X and Y are disjoint. Note that m-routings need not be unique, and in fact part of the challenge of our proof is constructing a canonical m-routing with sufficiently small m.

Definition 3 *Let $G = (V, E)$ and $S \subseteq V$. If p is a path in G that contains at least one vertex in S and at least one vertex in $G - S$, then we call p* boundary-crossing *with respect to S in G.*

Note that any boundary-crossing path contains at least one pair of adjacent vertices such that one is in S and the other is not. Our basic strategy will be to show the existence of m-routings for relatively small m, and then show that such a routing must contain many boundary-crossing paths, implying the existence of many vertices in $\delta(S)$ and thus many meta-vertices in $\delta'(S')$.

2.4 Simple Proof for Strassen's Algorithm

First we use our technique to rederive the lower bound on the I/O-complexity for Strassen's algorithm presented in [5], $\Omega\left(\left(\frac{n}{\sqrt{M}}\right)^{\log_2 7} \cdot M\right)$. As in [5], we consider the sequence of computations of vertices performed by the algorithm. In [5], this sequence is divided up into segments of sufficient length such that the I/O due to each segment is guaranteed to be at least M, the cache size. To do this, the smallest segment length s is found such that for any segment S of size s we are guaranteed that $|\delta(S)| \geq 3M$. All vertices present in $\delta(S)$ contribute to the I/Os due to S, except for vertices in $R(S)$ already present in cache (at most M) and vertices in $W(S)$ that need not be written to cache (at most M). Because [5] considers only the decoding graph of G_r, there are no concerns about vertex copying.

We use the same basic argument, but instead divide the sequence of vertex computations of the CDAG G_r into the smallest segments possible such that each segment S (except perhaps the last segment) contains $66M$ vertices from rank k of the decoding graph (rank $r+k$ of G_r) [1]. When a vertex v is in S we consider every vertex in the same meta-vertex as v to also be in S; however, because there is no copying in the decoding graph every meta-vertex can contain only one vertex from the decoding graph. Note that the size of each segment may be different; we care only about the number of vertices on this specific rank. We let $k = \lceil \log_4(132M) \rceil$, the smallest integer k such that $4^k \geq 2 \cdot 66M$. Because rank k of the decoding graph contains $4^k 7^{r-k}$ vertices, there are $\left\lfloor \frac{4^k 7^{r-k}}{66M} \right\rfloor$ such complete segments. Let S be one such complete segment and let $\overline{S}$ denote the vertices in S on rank k of the decoding graph of G_r. Thus we pick S as small as possible such that $|\overline{S}| = 66M$. If G_r is the CDAG for Strassen's algorithm for multiplying $n_0^r \times n_0^r$ matrices, recall that $G_{r,k}$ contains 7^{r-k} copies of the graph G_k. For $1 \leq i \leq 7^{r-k}$, let G_k^i be the ith such copy, S_i be the subset of S in G_k^i, and $\overline{S_i}$ be the subset of vertices of S_i on rank k of G_r.

Intuitively, we "count" S by the number of vertices of S on this particular rank. It is these vertices that will contribute, perhaps indirectly, to I/Os performed during the computation of S, regardless of what vertices on other ranks lie in S.

Let D_k be the decoding graph of G_k. We now claim that there exists a routing of paths between all the input vertices and output vertices of D_k such that no vertex of D_k is hit too often:

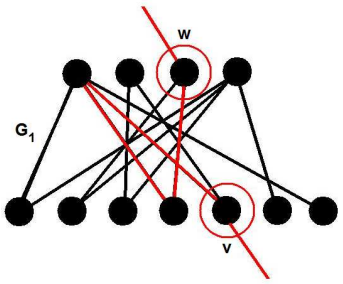
Claim 1 *There exists an $\left(11 \cdot 7^k\right)$-routing in D_k between the set of inputs of D_k and the set of outputs of D_k.*

[1] We did not optimize for the constant factor.

Proof. If D_1 were simply the complete graph $K_{7,4}$ consisting of all 28 edges between the 7 inputs and 4 outputs, there would exist a very natural routing of paths between inputs and outputs of D_k: for any input and output, there is a unique chain of vertices between them defined by the sequence of subcomputations the input lies in. A vertex on rank i of D_k is then hit $7^i 4^{k-i} \leq 7^k$ times in this routing, once for every pair consisting of an input vertex beneath it and an output vertex above it.

Unfortunately, D_1 is not a complete graph. However, because D_1 is connected there still exists a path within each copy of D_1 from any input vertex to any output vertex. Where each path previously went directly from an input vertex v to an output vertex w of each D_1, it will now take any path (that doesn't repeat vertices) through the same D_1 component from v to w. This idea is depicted in Figure 2.3. This multiplies the number of times a vertex is hit in the routing by at most the number of vertices in D_1, 11. □

Figure 2.3: One of the decoding graphs in G_1 for Strassen's algorithm. Because there is no edge from v to w, a chain must instead take a more indirect path, shown in red, through the encoding graph.



Each G_k^i contains a copy of D_k – for this proof we consider only the decoding piece D_k of G_k, but in the full proof we must consider G_k in its entirety in order to account for base graphs with disconnected encoding/decoding portions. Let D_k^i be the copy of D_k lying in G_k^i and note that $|\overline{S_i}| \leq \frac{1}{2}4^k$, so at most half of the vertices on the top rank of D_k^i are in S. For each $1 \leq i \leq 7^{r-k}$, fix an $\left(11 \cdot 7^k\right)$-routing in D_k^i between the 7^k inputs and 4^k outputs. See Figure 2.4. There are now two cases:

1. Fewer than half of the 7^k vertices on the bottom rank of D_k^i are in S. In this case, there exist at least $|\overline{S_i}|\frac{1}{2}7^k$ paths in the routing going from an input to D_k^i not in S to an output in S.

2. At least half of the vertices on the bottom rank of D_k^i are in S. In this case, there exist at least $\left(4^k - |\overline{S_i}|\right) \frac{1}{2} 7^k$ paths in the routing going from an input in S to an output not in S.

In either case, there are at least $\frac{1}{2}|\overline{S_i}|7^k$ boundary-crossing paths (between S_i and $D_k^i - S_i$) in the routing. Associate to each boundary-crossing path an edge in the path that crosses between S_i and $D_k^i - S_i$. The vertex of this edge that is not in S lies in $\delta(S_i)$. By the definition of m-routing[2],

$$|\delta(S_i)| \geq \frac{\frac{1}{2}|\overline{S_i}|7^k}{11 \cdot 7^k} = \frac{1}{22}|\overline{S_i}|$$

Adding this up over all the $\overline{S_i}$ yields

$$|\delta(S)| \geq \sum_{i=1}^{7^{r-k}} \frac{1}{22}|\overline{S_i}| = \frac{1}{22}|\overline{S}| \tag{2.1}$$

This step relies on the D_k being disjoint and the lack of copying in the decoding graph of Strassen's (or any Strassen-like) matrix multiplication algorithm. If multiple copying did occur, vertices in the different D_k^i need not correspond to distinct computations. This will add an additional layer of complexity to the upcoming proof.

Since $|\overline{S}|$ was chosen to be $66M$, this yields $\delta(S) \geq 3M$. Therefore the computation of S contributes at least M I/Os. Thus the total I/O is at least

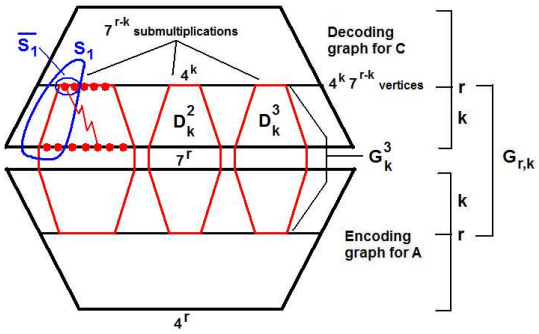
$$\left\lfloor \frac{4^k 7^{r-k}}{66M} \right\rfloor \cdot M = \Omega\left(7^r \left(\frac{4}{7}\right)^k\right) = \Omega\left(|V(G_r)| \frac{M}{M^{\log_4 7}}\right)$$

$$= \Omega\left(\left(\frac{n}{\sqrt{M}}\right)^{\log_2 7} \cdot M\right)$$

as long as $M = o\left(n_0^2\right)$ (which guarantees that $66M \leq 4^k 7^{r-k}$). $\square$

[2]And using the fact that the boundary-crossing paths in D_k^i are a subset of all the paths in the $\left(11 \cdot 7^k\right)$-routing in D_k.

Figure 2.4: An example of a path considered in the $\left(11 \cdot 7^k\right)$-routing between an input vertex (to D_k^1) that is not in S and an output vertex that is in S. The submultiplications are shown in red, S_1 is shown in blue, and $\overline{S_1}$ is circled. Note that the path zags up and down, as explained in Figure 2.3. For simplicity, only one encoding graph is shown and only 3 submultiplications are drawn.



2.5 Strassen-Like Algorithms

We now turn our attention to Strassen-like square matrix multiplication algorithms. Several nuances prevent our above proof from working as-is:

1. G_1 may have disconnected encoding or decoding graphs. This prevents us from finding an m-routing in the decoding graph D_k because D_k itself may no longer be connected. We will solve this problem by considering G_k, consisting of the decoding graph as well as the two encoding graphs. The paths in our m-routing will no longer be chains or even chains with length 1 "zags," but may need to bounce between inputs and outputs of G_k several times. See Figure 2.5.

2. Multiple copying may occur in the encoding graphs. This means a collection of m-routings for the G_k^i could potentially hit a meta-vertex more than m times. We will show via Theorem 4 that m-routings will only hit a meta-vertex entirely within G_k at most m times and then change the overall counting argument slightly to prevent meta-vertices between multiple G_k^is from being hit too often.

As before, we divide the sequence of vertex computations of G_r into segments such that each segment S contains enough vertices of a certain type. Again let G_k^i be the ith subcomputation of $G_{r,k}$ for $1 \leq i \leq b^{r-k}$. Let a *duplicated* vertex be a vertex of the CDAG G_r with at least one other copy (called a *duplicate*) in G_r, that is

one whose meta-vertex contains more than one vertex. We call two subcomputations *input-disjoint* if none of their inputs lie in the same meta-vertex.

Let S be a segment of the sequence of vertex computations. Recall that when $v \in S$ we consider every vertex w in the same meta-vertex as v to also be in S. For this argument we count only the vertices on rank k of the decoding graph of G_r and rank $r-k$ of either encoding graph that are in mutually input-disjoint subcomputations $G_{r,k}$. We choose $k = \lceil \log_a 72M \rceil$, the smallest integer k such that $a^k \geq 2 \cdot 36M$.

First we show that counting only vertices lying in subcomputations that do not share inputs reduces the number of vertices on the relevant ranks by only a constant factor. In Chapter 3 we will show a stronger analog of this lemma via a different technique; see Corollary 2. Note that this lemma does not apply to classical matrix multiplication, so our proof will not apply to classical matrix multiplication or algorithms with interleaved classical recursive steps; in later chapters we show how to extend our result to such algorithms.

Lemma 1 *Let $k \leq r-2$. If not every vertex in the encoding graph for A of G_1 is a duplicated vertex and similarly for the encoding graph for B of G_1, then at least a fraction $\frac{1}{b^2}$ of the subcomputations G_k^i are mutually input-disjoint.*

Proof. Consider the recursion tree of subcomputations computed by G_r. Let P_1 be the "grandparent" subcomputation of G_k^i – the subcomputation in the recursion tree two levels above G_k^i – and suppose P_1 multiplies matrices A_1 by B_1. Then at least one child subcomputation P_2 of P_1 multiplies matrices A_2 by B_2 such that A_2 shares no meta-vertices with A_1. Similarly, at least one child subcomputation of P_2 multiplies matrices A_3 by B_3 such that B_3 shares no meta-vertices with B_2, and hence with B_1. Thus at least one subsubcomputation of P_1 is input-disjoint from it. P_1 has b^2 subcomputations two levels down from it, so at least a fraction $\frac{1}{b^2}$ of all the subcomputations G_k^i are mutually input-disjoint. □

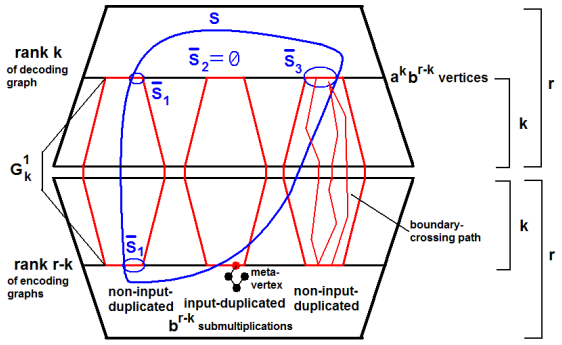
Fix a collection C of b^{r-k-2} mutually input-disjoint subcomputations G_k^i. Let $\overline{S}$ be the set of vertices of S on the aforementioned ranks in these subcomputations, which now consist of both encoding and decoding graphs. Formally, for $v \in S$ we let $v \in \overline{S}$ if both conditions below are met:

1. v lies on one of the following ranks: rank k of the decoding graph of G_r, rank $r-k$ of the encoding graph of G_r that encodes A, or rank $r-k$ of the encoding graph of G_r that encodes B.

2. The subcomputation G_k^i that v lies in (necessarily as an input or output of) is in C.

Divide the sequence of vertex computations into segments such that for each segment S we have $|\overline{S}| = 36M$. Let S_i be the subset of S in G_k^i and $\overline{S_i}$ be the subset of $\overline{S}$ in G_k^i. Note that if G_k^i is not one of the chosen input-disjoint subcomputations then $\overline{S_i} = \emptyset$. Intuitively, only the vertices in $\overline{S}$ "count" towards our I/O lower bound, regardless of how many other vertices lie in S, and we choose our segment divisions such that each segment has enough counted vertices. See Figure 2.5.

Figure 2.5: The overall idea of the main proof. For simplicity only one encoding graph is explicitly drawn. The set S is shown in blue. Note that only the elements on rank k of the decoding graph and rank $r-k$ of the encoding graphs in input-disjoint G_k^is lie in $\overline{S}$. A typical boundary-crossing path in G_k^3 is shown. (Not shown) The two vertices of the path on the bottom rank of G_k^3 lie in different encoding graphs.



Note that if the condition of Lemma 1 is not met, then the algorithm never computes linear combinations of one of the input matrices. It is known (see Theorem 14) that any matrix multiplication algorithm that computes linear combinations of only one of the input matrices performs no better than naive matrix multiplication and so does not have $o(n^3)$ arithmetic complexity (i.e., is not a fast matrix multiplication algorithm). Thus from now on we assume the condition of Lemma 1 is met.

Second, we show that our choice of partitioning the sequence of vertex computations into segments S exists. If meta-vertices contained multiple input and/or output vertices counted in $\overline{S}$, then including into S the next vertex v in the sequence of vertex computations – which by definition also includes into S every vertex in the same meta-vertex as v – could increase this count by more than one.

Lemma 2 *If G_k^i and G_k^j are input-disjoint, then the meta-vertices corresponding to the inputs and outputs of G_k^i and G_k^j are all distinct.*

Proof. Note that the decoding graph of G_1 cannot contain copying. If it did, then in the base case of $n_0 \times n_0$ matrix multiplication $C_1 = A_1 B_1$ some outputs would be identically equal, which is not the case. Hence the decoding graph of G_r contains no copying, and so every output vertex of G_k^i and G_k^j is non-duplicated. By definition, the input vertices of G_k^i and G_k^j are in distinct meta-vertices, proving the lemma. □

For the remainder of this proof we will consider, for each i, the entire subcomputation graph G_k^i (as opposed to just the decoding portion D_k^i). We must consider the decoding graph and both encoding graphs of G_k^i together because the decoding graph by itself, or even the decoding graph plus one encoding graph, may be disconnected. We now state the main theorem used in our proof, whose proof we defer until Section 2.6. Compare to the routing found in Section 2.4 between the input and output vertices of each D_k^i.

Theorem 4 (Routing Theorem) *Let G_k be the CDAG for $n_0^k \times n_0^k$ matrix multiplication, $a = n_0^2$, and let the encoding graph of the base graph G_1 have $2a$ inputs and b outputs. Assume every linear combination vertex in the base graph is used in only one multiplication. Then there exists a $6a^k$-routing between the set of inputs of G_k and the set of outputs of G_k. Furthermore, every meta-vertex in G_k is also hit by the routing at most $6a^k$ times.*

For each of the mutually input-disjoint G_k^i in C, fix a $6a^k$-routing guaranteed by the Routing Theorem between the inputs and outputs of G_k^i. Because the size of the top rank of G_k^i is a^k and the size of the bottom rank is $2a^k$ and $|\overline{S_i}| \leq |\overline{S}| \leq \frac{1}{2}a^k$, for every vertex v in $\overline{S_i}$ there exist at least $\frac{1}{2}a^k$ paths in the routing that go either:

1. between a vertex in S on the bottom rank of G_k^i and a vertex not in S on the top rank of G_k^i (if v is on the bottom rank)

2. between a vertex not in S on the bottom rank of G_k^i and a vertex in S on the top rank of G_k^i (if v is on the top rank).

Thus the routing in G_k^i contains at least $\frac{1}{2}a^k|\overline{S_i}|$ boundary-crossing paths; call the set of such paths P_i and let $P = \bigcup_i P_i$ be all these boundary-crossing paths in the above routings for all input-disjoint G_k^i. Then $|P| \geq \sum_i \frac{1}{2}a^k|\overline{S_i}| = \frac{1}{2}a^k|\overline{S}|$. By the Routing Theorem every meta-vertex contained entirely within G_k^i is hit by the routing at most $6a^k$ times. No meta-vertex in G_k^i extends beneath the bottom rank of G_k^i, and so every meta-vertex in G_r intersects at most one of the mutually input-disjoint G_k^i. Therefore every meta-vertex in G_r is hit at most $6a^k$ times by the paths in P.

Let S' be the set of meta-vertices represented by S, and recall that $\delta'(S')$ denotes all meta-vertices adjacent to S' that are not in S' itself. Then

$$|\delta'(S')| \geq \frac{\frac{1}{2}a^k|\overline{S}|}{6a^k} = \frac{1}{12}|\overline{S}| \tag{2.2}$$

This is a more general analogue of Equation 2.1.

Every meta-vertex adjacent to S necessarily contributes one to the I/Os due to computing S, except possibly for those meta-vertices already in memory (at most M) and those that need not be written to cache (at most M). Because $|\overline{S}| = 36M$, we have $|\delta'(S')| \geq 3M$, and so computing S requires at least M I/Os.

As indicated above, because G_r has $o(n^3)$ multiplications we may apply Lemma 1. Because rank k of the decoding graph of G_r and rank $r-k$ of the encoding graphs of

G_r together have size $3a^k b^{r-k}$ and $\frac{1}{b^2}$ of these vertices are in mutually input-disjoint subcomputations G_k^i, the total I/O from computing G_r is at least

$$\left\lfloor \frac{\frac{1}{b^2}3a^k b^{r-k}}{36M} \right\rfloor \cdot M = \Omega\left(b^r \left(\frac{a}{b}\right)^k\right) = \Omega\left(|V(G_r)|\frac{M}{M^{\log_a b}}\right)$$

$$= \Omega\left(\left(\frac{n}{\sqrt{M}}\right)^{2\log_a b} \cdot M\right)$$

as long as $M \leq o\left(n^2\right)$ (which guarantees that $36M \leq \frac{1}{b^2}3a^k b^{r-k}$ and $k \leq r-2$).

In the parallel case, we apply the above argument to a processor that computes an above-average number of vertices of $\overline{S}$, yielding a factor of $\frac{1}{P}$ as in [2]. The cache-independent result comes from instead picking $k = \Theta\left(\log_b \frac{n^{\omega_0}}{P}\right)$ and letting S represent the computations performed by just one processor. This proves Theorem 3. □

2.6 Proof of The Routing Theorem

In this section we prove Theorem 4. Let G_k be the CDAG for a square Strassen-like matrix multiplication algorithm for $C = AB$, let Out be the set of outputs of G_k (corresponding to entries of C), In be the set of inputs, In_A be the set of inputs to the encoding graph for A within G_k, and In_B be the inputs to the encoding graph for B. Then $|Out| = |In_A| = |In_B| = a^k = n_0^{2k}$. For $v \in In$ and $w \in Out$, we say that the input-output pair (v, w) is a *guaranteed dependence* if in any correct matrix multiplication algorithm there exists a chain from v to w, or equivalently if the output element corresponding to w explicitly depends on the input element corresponding to v. It is clear that if $v \in In_A$ represents the input a_{ij} and w represents the output $c_{i'j'}$ then there is a guaranteed dependence between v and w if and only if $i = i'$, and similarly if $v \in In_B$ represents the input b_{ij}, then there is a guaranteed dependence between v and w if and only if $j = j'$.

To prove the Routing Theorem we will combine the following two lemmas, whose proofs follow in the succeeding sections:

Lemma 3 *Let $F \subseteq V(G_k) \times V(G_k)$ be the set of all guaranteed dependencies (v, w) of G_k with $v \in In$ and $w \in Out$. Then there exists a $2n_0^k$-routing for F in G_k consisting only of chains.*

Intuitively, we can route chains between all pairs of input and output vertices where a chain is guaranteed to exist while using no vertex more than $2\sqrt{a^k}$ times. That

every path of the routing is a chain is not necessary to complete the proof of the Routing Theorem.

Lemma 4 *Fix a routing for F, where F is as defined in Lemma 3. Then there exists a routing between In and Out such that every path in the routing consists of the concatenation of chains in F – some reversed in direction – such that each chain in F is used $3n_0^k$ times.*

In other words, given any way of routing chains between all guaranteed dependencies, we can combine those chains, backwards and forwards, to give a path between every input and every output vertex while not using any such chain more than $3\sqrt{a^k}$ times.

Given these lemmas, the proof is simple:

Proof of the Routing Theorem. By Lemma 3, fix a $2n_0^k$-routing R_0 for the set of guaranteed dependencies F. By Lemma 4, there exists a routing R between the inputs and outputs of G_r composed of concatenations of chains (some reversed) in R_0 such that every chain in R_0 is used at most $3n_0^k$ times. Thus in the routing R every vertex of G is used at most $2n_0^k \cdot 3n_0^k = 6a^k$ times, and so R is a $6a^k$-routing, as desired.

Because every meta-vertex is an upward-facing subtree (see Figure 2.2), any path hitting a meta-vertex also hits the root vertex of the meta-vertex. Hence every meta-vertex is also hit at most $6a^k$ times. □

Proof of Lemma 4

In this section we prove the second, significantly easier, lemma. The proof of this lemma is constructive, yielding an explicit scheme for routing chains between all inputs and outputs given a routing for all guaranteed dependencies. This lemma holds for any correct matrix multiplication algorithm based only on the definition of matrix multiplication.

Proof of Lemma 4. For an input vertex v of G_k and output vertex w corresponding to element $c_{i'j'}$ of C, suppose first that $v \in In_A$. Let v then represent element a_{ij} of A. We form the following sequence of guaranteed dependencies:

$$a_{ij} \to c_{ij'} \to b_{jj'} \to c_{i'j'}$$

That is, $(a_{ij}, c_{ij'})$ is a guaranteed dependence, $(b_{jj'}, c_{ij'})$ is a guaranteed dependence, and $(b_{jj'}, c_{i'j'})$ is a guaranteed dependence. Note that every guaranteed dependence in this chain involves 3 out of the 4 variables i, i', j, and j'. Hence as i, j, i', and j'

vary between 1 and n_0^k, each guaranteed dependence above is used n_0^k times, once for each value of the missing variable (for each time it appears in the above sequence). For example, for any i, j, and j', the guaranteed dependence between a_{ij} and $c_{ij'}$ is used exactly once for every $1 \leq i' \leq n_0^k$. See Figure 2.6 for another interpretation of this pattern.

Similarly, if $v \in In_B$ let v correspond to element b_{ij} of B. The following sequence of guaranteed dependencies has the same properties:

$$b_{ij} \rightarrow c_{i'j} \rightarrow a_{i'i} \rightarrow c_{i'j'}$$

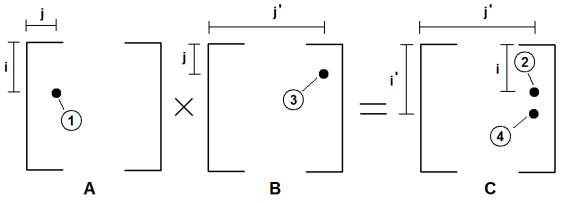
Amongst both these sequences, each guaranteed dependence between an element of A and one of C is used exactly $3 \cdot n_0^k$ times and similarly for every guaranteed dependence between B and C. This proves Lemma 4. $\square$

Note that these sequences are not unique. When routing a_{ij} to $c_{i'j'}$, any sequence of the form

$$a_{ij} \rightarrow c_{ij'} \rightarrow b_{\_j'} \rightarrow c_{i'j'}$$

where the blank is any value forms a set of sequences of guaranteed dependencies. However, unless the values that the blank takes are well-distributed over j for all choices of i, i', and j', this sequence will not have the desired property. This explains the odd use of j as a row index, and similarly the use of i as a column index when routing b_{ij} to $c_{i'j'}$.

Figure 2.6: The sequence of guaranteed dependencies between a_{ij} and $c_{i'j'}$ shown as elements in the matrices A, B, and C. Note the use of j as a row index.



Proof of Lemma 3

This lemma is significantly harder to prove. We use the following overall strategy: In order to prove there exists a $2n_0^k$-routing between all guaranteed dependencies, we show there exists a n_0-routing of guaranteed dependencies in the subgraph of

G_1 formed by the decoding graph together with the encoding graph for A; by the recursive structure of G_k, this is sufficient to prove it in general. Define a *middle-rank* vertex of G_1 to be a vertex on the top rank of the encoding graph of A. To show the lemma for this $\frac{2}{3}$ of G_1 – the encoding graph for A together with the decoding graph for C – we show a (several-to-one) matching between guaranteed dependencies and middle-rank vertices on some chain satisfying the dependence. By assumption, every vertex representing a linear combination of elements of A is adjacent to exactly one multiplication vertex; thus a routing of guaranteed dependencies that uses each middle-rank vertex at most n_0 times also uses each multiplication vertex at most n_0 times.

We will prove the existence of this matching via a version of Hall's Matching Theorem. In order to apply this theorem, we will need to show that for every set of d guaranteed dependencies, there exist chains between those dependencies collectively hitting at least $\frac{d}{n_0}$ middle-rank vertices. We demonstrate that if this is not the case, then setting some entries of the $n_0 \times n_0$ input matrix A to be identically 0 results in an algorithm that correctly computes many of the guaranteed dependencies between C and A using relatively few multiplications. Finally, we show that this implies the existence of an algorithm for multiplying a $n_0 \times n_0$ matrix by a length n_0 vector in fewer than n_0^2 operations, which is known to be impossible [19]. This will conclude the proof.

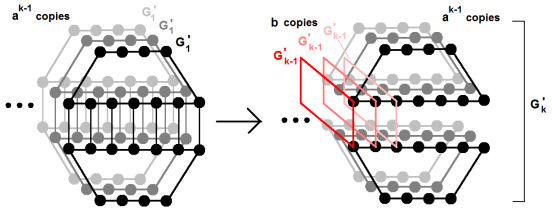
Let G'_k be the induced subgraph of G_k containing the vertices from the decoding graph of G_k and the encoding graph of G_k for A (excluding only the encoding graph for B). Let F' be the subset of F with both vertices lying in G'_k, that is the set of guaranteed dependencies (v, w) between inputs v of A and outputs w of C. For simplicity, we simply call F' the *guaranteed dependencies of* G'_k. We now consider m-routings for the set of guaranteed dependencies (that is, F') of G'_k. It then suffices to find an a^k-routing of guaranteed dependencies in G'_k.

Claim 2 *If there exists an m-routing for the guaranteed dependencies of G'_1, then there exists an m^k-routing for the guaranteed dependencies of G'_k.*

Proof. This lemma follows from the recursive structure of G'_k. Intuitively, the graph G'_k is formed by placing b copies of G'_{k-1} in parallel, connecting up their inputs with a^{k-1} copies of the encoding graph for A, and connecting up their outputs with a^{k-1} copies of the decoding graph for C. See Figure 2.7. In other words, take a^{k-1} copies of G'_1 and replace their middle two ranks with copies of G'_{k-1}. Any number of copies of G'_1 in parallel still have an m-routing for guaranteed dependencies, and replacing their middle ranks effectively replaces a pair of adjacent vertices on the middle ranks with a guaranteed dependence in G'_{k-1}. Thus if there exists an m^{k-1}-routing for

G_{k-1} then there exists an m^k routing for G_k. The claim then follows by induction. □

Figure 2.7: The construction of G'_k from b copies of G'_{k-1}. A pair of adjacent vertices on the middle two ranks is replaced with a guaranteed dependence in one of the G'_{k-1}.



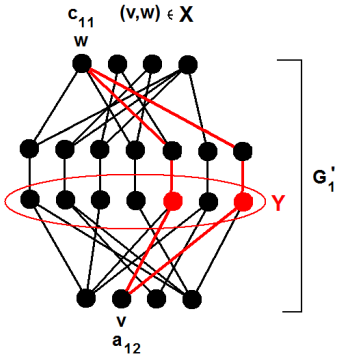
Therefore it will suffice to prove the existence of an n_0-routing for the guaranteed dependencies of G'_1. We now apply a version of Hall's Matching Theorem:

Theorem 5 (Hall's Matching Theorem) *(Many-to-one version) Let G be a bipartite graph with bipartition X and Y and for $D \subseteq V(G)$ let $N(D)$ denote the set of neighbors of D in G. If for every $D \subseteq X$ we have $|N(D)| \geq \frac{|D|}{p}$, then there exists a many-to-one matching between X and Y such that every vertex in X is used exactly once and every vertex in Y is used at most p times.*

This theorem follows from the standard form of Hall's Matching Theorem by simply duplicating all vertices in Y p times.

We now construct a graph $H = (X, Y)$ to which to apply Theorem 5. For every guaranteed dependence (v, w) in G_1' (with v an input representing an element of A and w an output representing an element of C), define a corresponding vertex in X. Let Y be the set of middle-rank vertices of G_1: all vertices on the top rank of the encoding graph for A. It suffices to assign to each guaranteed dependence in X a middle-rank vertex from Y through which its chain may pass. To this end, if $x \in X$ corresponds to the guaranteed dependence (v, w) and $y \in Y$ corresponds to the middle-rank vertex t, let there be an edge between x and y if there exists some chain between v and w passing through t. See Figure 2.8.

Figure 2.8: The vertices shown in red are those adjacent to the vertex in H corresponding to the guaranteed dependence (v, w), where v corresponds to the input a_{12} of A and w corresponds to the output c_{11} of C. The graph shown is the G_1' for Strassen's algorithm.



Lemma 5 *For any set $D \subseteq X$, we have $|N(D)| \geq \frac{|D|}{n_0}$.*

From Lemma 5, the proof of Lemma 3 follows, and thus our main result:

Proof of Lemma 3. By Hall's Matching Theorem (Theorem 5), there exists a many-to-one matching from X to Y using every vertex in Y at most n_0 times. Fix such a matching. For every guaranteed dependence (v, w) of G'_1, simply route a chain through the vertex of Y that (v, w) is matched with. Every vertex on the middle two ranks of G'_1 is thus hit at most n_0 times. Every vertex on the top and bottom ranks of G'_1 is hit exactly n_0 times by any routing for guaranteed dependencies that uses only chains, because in $n_0 \times n_0$ matrix multiplication every element of A influences n_0 elements of C, and every element of C depends on n_0 elements of A. Thus there exists a n_0-routing for the guaranteed dependencies in G'_1, and so by Claim 2 there exists a n_0^k-routing for the guaranteed dependencies of G'_k. The same holds for the induced subgraph of G_1 consisting of the decoding graph together with the encoding graph for B, yielding a $2n_0^k$-routing for the guaranteed dependencies of G_k. □

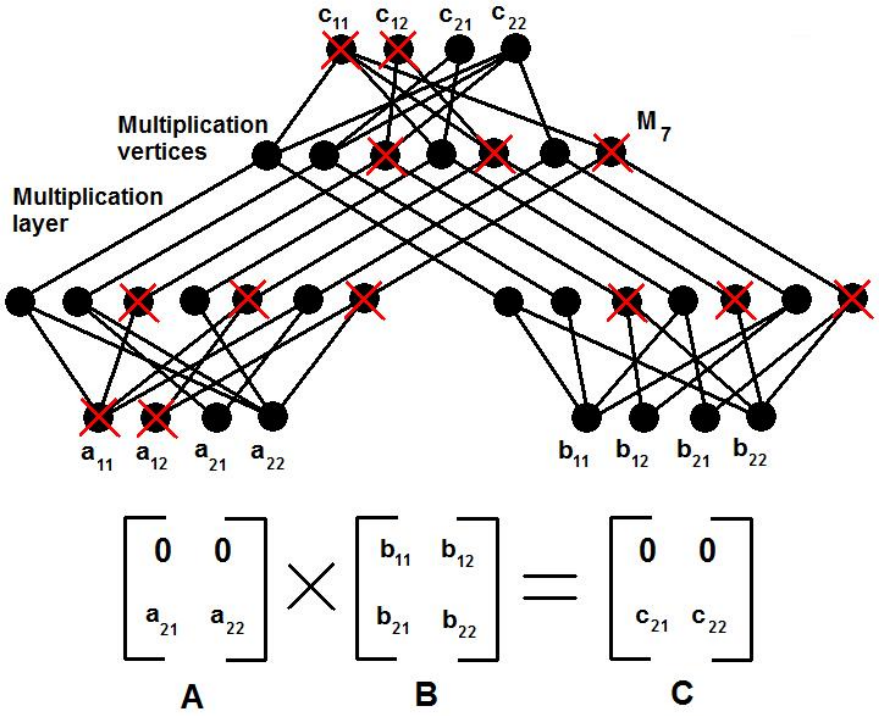
Proof of Lemma 5

Finally, we prove Lemma 5 to complete the proof of the Routing Theorem and thus our main result, Theorem 3:

Proof of Lemma 5. Suppose by way of contradiction that for some subset $D \subseteq X$ of guaranteed dependencies in G'_1 we have $|N(D)| < \frac{|D|}{n_0}$. Recall that a guaranteed dependence occurs between the vertex representing a_{ij} and the vertex representing $c_{i'j'}$ exactly when $i = i'$. We may thus partition D by the choice of i: let D_i be the subset of D consisting of all guaranteed dependencies between a_{ij} and $c_{ij'}$ for any j and j'. Because $1 \leq i \leq n_0$, for some i we have $|D_i| \geq \frac{|D|}{n_0}$. Since $N(D_i) \subseteq N(D)$, we have $|N(D_i)| < \frac{|D|}{n_0} \leq |D_i|$. In other words, the set of guaranteed dependencies D_i is computed using fewer than $|D_i|$ multiplication vertices.

We now demonstrate that this is impossible by using this structure to create a matrix-vector multiplication algorithm that requires fewer than n_0^2 multiplications. For fixed D_i, define the computation graph G_1° as follows: G_1° is the induced subgraph of G_1 containing as inputs vertices corresponding to all the elements of B and their linear combinations, the elements a_{ij} of A for all j, and the elements $c_{ij'}$ of C for all j'. G_i° additionally contains all the middle-rank vertices in $N(D_i)$ and all vertices on the bottom rank of the decoding graph. G_1° may now contain "useless" vertices – we draw G_1° with these vertices additionally removed, but it does not matter for the bounds in this proof. See Figures 2.9 and 2.10.

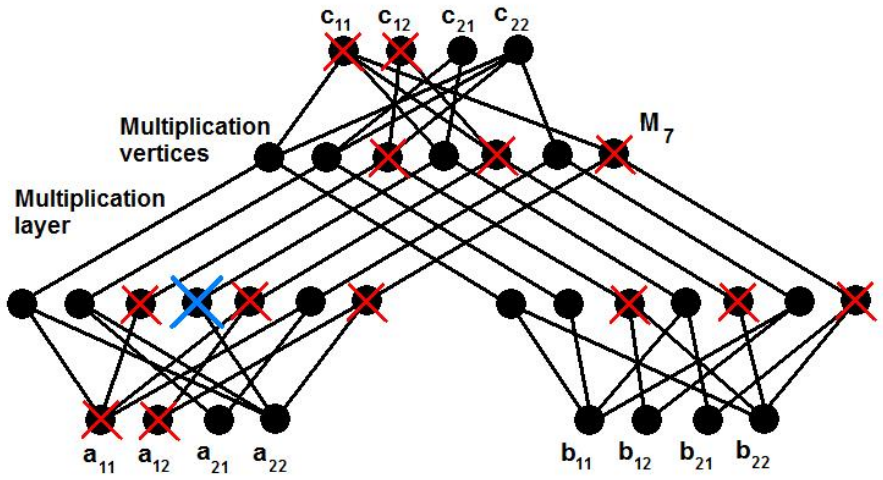
Figure 2.9: G_1° for Strassen's algorithm when $i = 2$ and $D_2 = \{(a_{21}, c_{21}), (a_{21}, c_{22}), (a_{22}, c_{21}), (a_{22}, c_{22})\}$. The crossed-out vertices are those removed from G_1 to construct this reduced computation graph G_1°. This is equivalent to setting all but the 2nd row of A to 0; the resulting algorithm computes the product of a row of A by the matrix B.



By the structure of G_1, every multiplication vertex multiplies a linear combination $\sum_{i,j} \lambda_{ij}^A a_{ij}$ by a linear combination $\sum_{i,j} \lambda_{ij}^B b_{ij}$ for some coefficients λ_{ij}^A and λ_{ij}^B in the ground field F ($\mathbb{R}$ or $\mathbb{C}$). We consider linear combinations of the a_{ij}s with coefficients in $F[b_{11}, b_{12}, \ldots, b_{n_0 n_0}]$. In other words, consider b_{ij}s to be coefficients and a_{ij}s to be variables. For $1 \leq j \leq n_0$, let a_{ij} and c_{ij} be the inputs and outputs of G_1° respectively. Note that for all $1 \leq j, j' \leq n_0$, c_{ij} depends on $a_{ij'}$. We now define a boolean-valued function f that represents whether the coefficient of each input is correct in each output: For $1 \leq j, j' \leq n_0$, define $f(j, j')$ to be 1 exactly when the coefficient of $a_{ij'}$ in c_{ij} is its correct value for matrix multiplication, namely $b_{j'j}$, and otherwise 0.

Let n_f denote the number of pairs (j, j') with $1 \leq j, j' \leq n_0$ at which f takes the

Figure 2.10: However, D_2 need not contain all possible guaranteed dependencies as in Figure 2.9, depending on the given subset D. In this figure $D_2 = \{(a_{21}, c_{21}), (a_{21}, c_{22}), (a_{22}, c_{22})\}$. Because the guaranteed dependence (a_{22}, c_{21}) is not included in D_2, the vertex crossed out in blue is removed, and so G_1° does not quite compute vector-matrix multiplication; the coefficient of a_{22} in the computation of c_{21} may not be correct. This necessitates adding additional multiplication vertices to "fix" errors of this form.



value 1 – that is, the number of coefficients correctly set by G_1°. By the definition of G_1° relative to the matching graph H, we have $n_f \geq |D_i|$ (for the i for which $|D_i| \geq \frac{|D|}{n_0}$): If the guaranteed dependence of c_{ij} on $a_{ij'}$ is represented in D_i, then the coefficient of $a_{ij'}$ in c_{ij} must be "correct" for matrix multiplication, since, by definition of G_1°, there exists no chain between the vertices corresponding to c_{ij} and $a_{ij'}$ contained in G_1 (which correctly computes matrix multiplication) but not in G_1°.

Finally, we use G_1° to construct a new, correct, vector-matrix multiplication algorithm. Define $\overline{G_1^\circ}$ to be the CDAG formed as follows: to the CDAG G_1° add $n_0^2 - n_f$ multiplication vertices, one for each pair (j, j') for which $f(j, j') = 0$. For $1 \leq j, j' \leq n_0$ let the coefficient of $a_{ij'}$ in c_{ij} computed by G_1° be $x_{j'j} \in F[b_{11}, b_{12}, \ldots, b_{n_0 n_0}]$ – a linear combination of the "coefficients" b_{ij}. For each such j and j' at which $f(j, j') = 0$, use a multiplication vertex to compute $a_{ij'}\,(b_{j'j} - x_{j'j})$ and add it to the output vertex representing c_{ij}. In other words, for every incorrect dependence of c_{ij} on $a_{ij'}$ we may use a single multiplication vertex to "fix" the dependence. Now $\overline{G_1^\circ}$ correctly computes $n_0 \times n_0$ vector-matrix multiplication. G_1° contained

fewer than $|D_i|$ multiplication vertices and we added at most $n_0^2 - n_f$, so $\overline{G_1^\circ}$ has $< |D_i| + n_0^2 - n_f \leq |D_i| + n_0^2 - |D_i| = n_0^2$ multiplication vertices. Thus we have constructed a correct algorithm for computing $n_0 \times n_0$ vector-matrix multiplication using fewer than n_0^2 multiplications, which is known to be impossible [19]. This concludes the proof of Lemma 5 and hence of our main result Theorem 3. □

We state the result we obtained in the proof of Lemma 5 as its own Lemma:

Lemma 6 *Let G_1° be a CDAG with inputs a_{ij} and b_{ij} and outputs c_{ij} for $1 \leq i, j \leq n_0$ where each c_{ij} is computed as a product of linear combinations of the a_{ij} and b_{ij}. If for d pairs (j, j'), $1 \leq j, j' \leq n_0$, the coefficient of $a_{ij'}$ in c_{ij} is $b_{j'j}$, then G_1° uses at least d multiplications.*

2.7 Conclusion

We have proven optimal lower bounds for the I/O-complexity of any Strassen-like square matrix multiplication algorithm in which every linear combination in the base graph is used in only one multiplication by proving the existence of a routing between the inputs and outputs of such an algorithm that uses every intermediate computation vertex relatively few times. The proof generalizes easily to algorithms composed of different base graphs, as long as each base graph performs square matrix multiplication with $2a$ inputs and b subcomputations and satisfies the conditions of Lemma 1. This bound holds regardless of the form of the base graph(s), including those that have disconnected encoding or decoding pieces and those that perform multiple copying. Our technique provides a novel alternative to the edge expansion argument in [5] that applies to less straightforward recursive computation graphs.

In Chapter 3 we remove the assumption that every linear combination is used in only one multiplication. Without this assumption Lemma 5 no longer holds; vertices representing linear combinations used in multiple multiplications may require too many paths routed through them. Thus a more general approach to routing guaranteed dependencies is required. This difficulty will be overcome by routing paths in response to the choice of S, where paths are now allowed to "jump" to other vertices that have the same membership in S. This extends our result to all fast Strassen-like matrix multiplication algorithms.

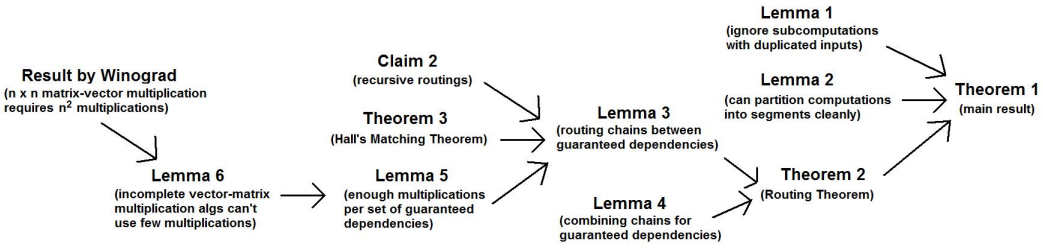
2.8 Acknowledgments

Research is supported by grants 1878/14, and 1901/14 from the Israel Science Foundation (founded by the Israel Academy of Sciences and Humanities) and grant 3-

10891 from the Ministry of Science and Technology, Israel. Research is also supported by the Einstein Foundation and the Minerva Foundation.

Finally, in Figure 2.11 we present a quick summary of all the results used or proved thus far:

Figure 2.11: All the main intermediate results used in the proof of Theorem 3.



Chapter 3

Generalization to Recursive Divide-and-Conquer Matrix Multiplication Algorithms

In this chapter we generalize the lower bound of Theorem 3 to all recursive divide-and-conquer matrix multiplication algorithms, defined below. This class of matrix multiplication algorithms does not necessarily use the same base graph G_1 at each recursive step and may even mix in classical matrix multiplication steps, as is often done in practice. This also requires removing the assumption of the previous chapter that every (nontrivial) linear combination be used in only only multiplication. All well-studied matrix multiplication algorithms are of this form. The proof of our main result lies in the following chapters. In this chapter we develop some machinery and show how to apply it to prove our general result for matrix multiplication algorithms that are efficient at every step. In Chapter 4 we introduce a powerful new technique for computing I/O-complexity lower bounds and show how it simplifies the proof in this special case. We prove our main result for general, square matrix multiplication algorithms in Chapter 5, and show how to extend it to rectangular matrix multiplication algorithms in Chapter 6. Finally, we extend our result to matrix multiplication algorithms run in parallel in Chapter 7. These bounds are tight for square matrix multiplication algorithms (and matrix multiplication algorithms all of whose submultiplications have their dimensions within a constant factor).

Throughout this paper we assume that every base graph computes distinct multiplications. If this is not the case, an algorithm that is more efficient, both in terms of arithmetic and in terms of I/O-complexity, can easily be derived by removing the extraneous multiplications. Without this assumption an algorithm of fixed I/O-complexity can be made to have arbitrarily high arithmetic complexity (by simply

adding extraneous multiplications that are never used), so this assumption is both reasonable and necessary.

To achieve these generalized bounds we further refine our technique through the use of a modified measure of I/O-complexity that we call *internal I/O-complexity*; see Definition 7. I/O-complexity is neither additive nor superadditive; one cannot simply partition a CDAG into two (non-parallel) subgraphs, compute the I/O-complexities of the subgraphs, and compute the sum to derive a lower bound on the I/O-complexity of the entire graph. Intuitively this is because the I/Os due to the inputs and outputs of each subgraph are "artificial" I/Os that should not count when computing the I/O-complexity of the entire graph. Internal I/O-complexity ignores these I/Os and, as proved in Theorem 17, is superadditive, as long as the graphs considered are disjoint. For non-disjoint graphs we conjecture that in many cases internal I/O-complexity is still superadditive, but this is not necessary for the proof of our main result.

First we generalize the technique of Chapter 2 to recursive steps – not necessarily all identical – that multiply rectangular matrices, as long as each base graph is more efficient than classical matrix multiplication. This assumption is important to guarantee the existence of sufficiently many mutually disjoint submultiplications, as in the proof of Chapter 2. This step requires redoing some of the proofs of Chapter 2, keeping all the matrix dimensions distinct. For this reason we present abbreviated versions of some of these generalized proofs.

We also show how to remove the assumption of Chapter 2 that intermediate linear combinations are used only once. By itself this extends the results of Chapter 2 to all Strassen-like fast matrix multiplication algorithms.

Next we use this result to provide a tight lower bound on the I/O-complexity of recursive divide-and-conquer matrix multiplication algorithms with no inefficient [1] recursive steps via an argument based around an analog of Lemma 1. As in Chapter 2, we prove the existence of sufficiently many disjoint parallel subcomputations whose I/O-complexities we may then add. This step requires proving that any fast matrix multiplication algorithm computes a product of nontrivial linear combinations; see Theorem 14.

However, many matrix multiplication algorithms used in practice contain recursive steps that perform no better than classical matrix multiplication. Most implementations of fast matrix multiplication, for instance, revert to classical matrix multiplication at recursive steps involving small enough matrices. Thus it is of interest to prove I/O-complexity lower bounds for recursive divide-and-conquer matrix

[1] At least as slow as classical matrix multiplication; see the definition of exponent below.

multiplication algorithms involving arbitrary recursive steps. The difficulty in this generalization arises because subcomputations may share multiply copied inputs.

We will show that within a computation interval S, every sufficiently small matrix multiplication subcomputation must either have most of its input and output vertices in S, or else contribute a proportional number of disjoint (internal) I/Os. We then separately analyze subcomputations most of whose input and output vertices are in S; in particular, because decoding graphs contain no vertex copying, it is possible to efficiently route paths between these input and output vertices to vertices that do not lie in S.

3.1 Main Theorems

Recall that in Chapter 2 we derived an I/O-complexity lower bound for Strassen-like matrix multiplication proportional to the total number of input vertices to dimension $n_0^k \times n_0^k$ submultiplications for a particular k. This k was chosen so that the number of input elements in each subcomputation (which was $\Theta\left(n_0^{2k}\right) = \Theta\left(a^k\right)$) was $\Theta(M)$. In the general case, we similarly derive an I/O lower bound proportional to the number of input vertices in all subcomputations of the recursion tree involving $\Theta(M)$ inputs/outputs. As a corollary, we derive the I/O-complexity bound of

$$\Omega\left(\frac{\text{number of arithmetic operations}}{\sqrt{M}^{\omega_0}} \cdot M\right),$$

where ω_0 represents the worst matrix multiplication exponent of any of the recursive steps used in the algorithm (see below). This holds as long as each base graph is no worse than classical matrix multiplication; if this is not the case, the algorithm can be simplified to one that is, as explained after Theorem 6.

Call a *recursive divide-and-conquer* matrix multiplication algorithm any algorithm for multiplying rectangular matrices that, in order to multiply (potentially padded) matrices A and B, divides A into x equal-width blocks vertically and y horizontally (for a total of xy blocks of equal dimensions), divides B into y equal-width blocks vertically and z horizontally (for a total of yz blocks of equal dimensions), computes some linear combinations of blocks of A and some linear combinations of blocks of B, multiplies some pairings of these linear combinations recursively, and finally computes linear combinations of these results to yield the blocks of C. The values of x, y, and z may be different for different recursive steps. In other words, a recursive divide-and-conquer matrix multiplication algorithm is a generalization of

a Strassen-like algorithm in which base graphs needn't be identical, multiply square matrices, or satisfy the conditions of Lemma 1. [2]

Effectively all matrix multiplication algorithms in the literature are recursive divide-and-conquer (or used to build up divide-and-conquer matrix multiplication algorithms via tensors [18]). Our bound is tight for square matrix multiplication algorithms, and therefore solves the problem of determining the I/O-complexity of matrix multiplication, given its arithmetic complexity, for any reasonable square matrix multiplication algorithm. There may exist rectangular matrix multiplication algorithms for which our bound is too low, but all such algorithms involve breaking the multiplication into submultiplications of very-far-from-square matrices, and thus are unlikely to be relevant in practice.

Just as before we will consider each recursive step to consist of a "base graph" G_1; however, now G_1 may be different at different steps. For ease of notation we will still refer to each recursive step as a base graph G_1; which recursive step we mean will be made clear by context. As before, each G_1 will consist of inputs corresponding to blocks of the overall input matrices A and B, multiplications of linear combinations of these inputs that correspond to submultiplications performed recursively (though not necessarily with the same base graph!), and linear combinations of the products that form the output matrix, which is used as a (scaled) summand in various blocks of the overall output matrix C.

Definition 4 *If a base graph G_1 divides A into x equal-width blocks vertically and y horizontally, divides B into y equal-width blocks vertically and z horizontally, and performs b submultiplications, then we define the* exponent [3] *of G_1 as* $\omega(G_1) = 3 \log_{xyz} b$.

Definition 5 *If G is the CDAG for a recursive divide-and-conquer matrix multiplication algorithm, define the* exponent *of G to be $\omega(G) = \max_{\text{base graph } G_1} \omega(G_1)$, the maximum exponent of a base graph in G.*

Definition 6 *If G_1 is a base graph of G, call G_1* fast *if $\omega(G_1) < 3$ and* slow *if $\omega(G_1) \geq 3$.*

[2] There is one subtlety in this definition with respect to the definition of I/O-complexity. It is assumed that all submultiplications are distinct as long as all elementary multiplications of each base graph are distinct; if not, then it is assumed that these identical submultiplication are recomputed as if they were different, or else this system of recursive steps is to be represented as one single recursive step with larger x, y, and z values.

[3] This is equivalent to the bound on the optimal matrix multiplication exponent ω derived from the base graph G_1, usually interpreted as a result stemming from tensor rank in the literature [13].

The exponent of a base graph is a measure of how inefficient that base graph is from the viewpoint of arithmetic complexity. If all base graphs have the same exponent ω_0, as in the case of Strassen-like algorithms, then the number of elementary multiplications performed is simply $\Theta\left(n^{\omega_0}\right)$, justifying the name $\omega(G_1)$. The exponent of a full matrix multiplication CDAG then represents the worst performance of any of its base graphs. A base graph G_1 representing classical $\Theta(n^3)$ matrix multiplication has $\omega(G_1) = 3$ and is thus slow, but not every slow base graph need represent classical matrix multiplication. However, it will turn out (Theorem 15) that every slow matrix multiplication algorithm can be reduced in some sense to classical matrix multiplication.

Let $\mathrm{IO}(G)$ be the I/O-complexity of G and for a submultiplication X of G – a subgraph of G computing a matrix multiplication – define $\mathrm{max\_size}(X)$ to be the number of elements in the largest matrix (either input or output) of X. We will prove the following results:

Theorem 6 *Let G be a CDAG for recursive divide-and-conquer rectangular matrix multiplication that computes distinct products in each recursive step. Let I be the set of subcomputations of G whose minimum matrix dimension is $\geq \Omega\left(\sqrt{M}\right)$ and none of whose subcomputations satisfy this property. If $\omega(G) \leq 3$ and the maximum number of blocks a matrix is divided into in any subcomputation is a small constant, then G has I/O-complexity*

$$IO(G) \geq \Omega\left(\sum_{X \in I} max\_size(X)\right).$$

The above result still holds even if $\omega(G) > 3$, as long as each recursive step of G with exponent > 3 has a disjoint submultiplication. If $\omega(G) > 3$ but this is not the case, the matrix multiplication algorithm can be transformed into one one requiring fewer arithmetic operations and no more cache I/Os to which this result also applies by modifying the base graphs of exponent greater than 3.

Theorem 7 (Main Theorem) *Let G be as in Theorem 6. If G computes Mult elementary multiplications, then*

$$IO(G) \geq \Omega\left(\frac{Mult}{\sqrt{M}^{\,\omega(G)}} \cdot M\right)$$

Both theorems hold for internal I/O-complexity as well (see Chapter 4). This theorem is our main theorem, relating the (internal) I/O-complexity of an arbitrary recursive matrix multiplication algorithm to the number of elementary multiplications it computes and the worst efficiency of any of its recursive steps. This theorem does require that every step of G be no more inefficient than classical matrix multiplication; clearly this is not a large assumption in practice. In fact, by (the discussion following) Theorem 15, every matrix multiplication step that is less efficient than classical matrix multiplication either has a disjoint submultiplication – in which case Theorem 6 is valid – or else can be "reduced" to classical matrix multiplication in a way that cannot possibly increase the I/O cost of G. Thus the bound of Theorem 6 applies to any recursive divide-and-conquer matrix multiplication algorithm that would ever be used in practice. In Chapter 7 we will generalize the proof of Theorem 6 to parallel algorithms computing any recursive divide-and-conquer matrix multiplication. The tightness of Theorem 6 is discussed in Section 3.2.

To prove Theorem 7 from Theorem 6, we require the following lemma:

Lemma 7 *If G computes the product of an $l \times m$ matrix by an $m \times n$ matrix, then G involves at most $(lmn)^{\omega(G)/3}$ elementary multiplications.*

Proof. We prove this inductively. The result is clearly true in the base case, since $\omega(G_1)$ for a base graph G_1 is defined as $\omega(G_1) = 3\log_{lmn} b$, where b is the number of elementary multiplications. Suppose G has subcomputations of dimensions $\frac{l}{x}$, $\frac{m}{y}$, and $\frac{n}{z}$. Then the number of subcomputations is given by $b = (xyz)^{\omega(G_1)/3} \leq (xyz)^{\omega(G)/3}$, where G_1 represents this top-level base graph. By the inductive hypothesis each subcomputation has no more than $\left(\frac{lmn}{xyz}\right)^{\omega(G)/3}$ elementary multiplications, and so the total number of elementary multiplications is no more than $(xyz)^{\omega(G)/3}\left(\frac{lmn}{xyz}\right)^{\omega(G)/3} = (lmn)^{\omega(G)/3}$.

□

Proof of Theorem 7 from Theorem 6. For each subcomputation in I of dimensions l, m, n, the number of elementary multiplications b performed within I satisfies $b \leq (lmn)^{\omega(G)/3}$ by Lemma 7. If Y is the contribution of one subcomputation X of I to the sum in Theorem 6, then $Y \geq \Omega\left(\frac{b}{(lmn)^{\omega(G)/3}}\max(lm, mn, ln)\right)$. Let $r = \frac{\max(lm,mn,ln)}{\min(l,m,n)^2}$. Then

$$Y \geq \Omega\left(\frac{b}{(r\min(l,m,n)^3)^{\omega(G)/3}} r\min(l,m,n)^2\right) \geq \Omega\left(\frac{b}{\sqrt{M}^{3\omega(G)/3}} \cdot \frac{r}{r^{\omega(G)/3}} \cdot M\right)$$

$$\geq \Omega\left(\frac{b}{\sqrt{M}^{\omega(G)}} \cdot M\right)$$

Thus by Theorem 6 the total number of I/Os is at least $\Omega\left(\frac{Mult}{\sqrt{M}^{\omega(G)}} \cdot M\right)$. □

Consider the recursion tree representing the subcomputations performed in the computation of G. In Theorem 6, the set I is formed by "cutting off" the recursion tree at the indicated size. Thus the number of I/Os from a matrix multiplication algorithm is bounded below by the total number of vertices of its CDAG on this "layer." [4] Note that Theorem 6 is stronger than Theorem 7 for matrix multiplication algorithms involving base graphs of different exponents; in particular, if the bottom (smaller) recursive steps are more efficient than those closer to the top of the recursive tree, then the bound of Theorem 7 is too low; for this reason we present the bound of Theorem 6 as its own theorem.

In order to prove this result, we will first consider the case where $\omega(G) < 3$ – that is, every base graph is strictly better than naive matrix multiplication – and

[4] Not necessarily a layer in the general case, since recursive steps needn't be of the same dimensions.

then approach the general case. Again, note that any optimal algorithm in terms of arithmetic complexity has $\omega(G) < 3$.

We face several difficulties in our generalization:

- The recursion tree of a divide-and-conquer matrix multiplication algorithm needn't have every leaf node at the same height.
- Even if the input matrices are square, blocks may become rectangular (if x, y, and z are different in some recursive step). Our proof will thus be forced to account for rectangular input matrices as well.
- Multiple copying in base graphs may cause subcomputations to have shared inputs (but not outputs). In the Strassen-like case we found that either every encoding base graph contained a disjoint subcomputation or else the entire matrix multiplication algorithm was essentially naive matrix multiplication; this does not hold in general.

Define duplicated vertices as in the Strassen-like case. Note the following fact:

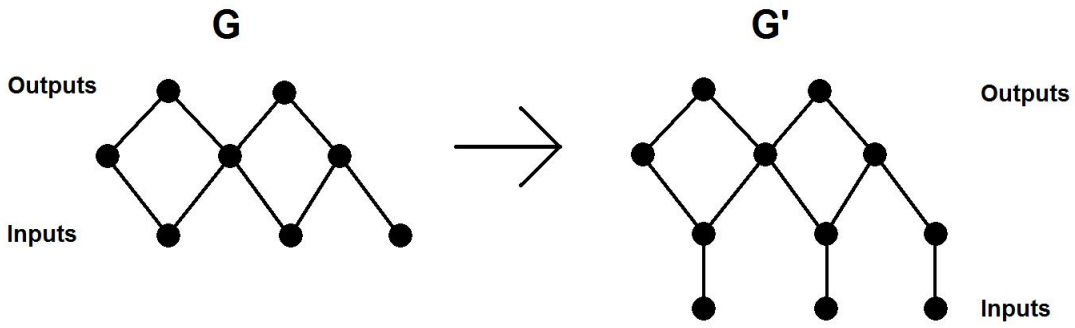
Fact 2 *Suppose that at the recursive step of a divide-and-conquer matrix multiplication algorithm corresponding to multiplying A and B that the submultiplication A_1B_1 is performed, where A_1 is a linear combination of blocks of A and similarly for B_1. Then either every input vertex of A_1 has a duplicate vertex outside A_1 or every input vertex of A_1 does not. The former case occurs exactly when A_1 is simply equal to one of the blocks of A. The same holds for B_1.*

In other words, the "A" matrix used in a submultiplication is either entirely copied from A (the input to the parent subcomputation) or entirely disjoint from A. It is never the case that some of the "A" inputs to a submultiplication are copies of those to the parent multiplication while others are not. We also make the following useful definition:

Definition 7 *If G is a CDAG, define the* internal I/O-complexity *$IO^+(G)$ as the minimum number of I/Os required to compute G', excluding I/Os of the input/output vertices of G', where G' is defined as follows: To each input vertex v in G' create a new input vertex v' with an edge from v' to v. See Figure 3.1. In other words, $IO^+(G)$ is the minimum number of I/Os required to compute G where the I/Os of the output vertices of G are "free" and every input vertex of G may be input once for free. Call an I/O counted by internal I/O-complexity an* internal I/O.

Note that $\text{IO}(G) \geq \text{IO}^+(G)$; every result in the following chapters on internal I/O-complexity thus applies to standard I/O-complexity as well. That every input vertex of G may be input for free only once follows from the assumption that no value is ever recomputed. In Chapter 4 we prove a few results about internal I/O-complexity that simplify our proof.

Figure 3.1: The internal I/O-complexity of a CDAG G is defined to be the minimum number of I/Os – excluding those of the input and output vertices – required to compute the CDAG G', as shown in this figure.



3.2 Tightness of the Main Theorems (Theorems 6 and 7)

Theorem 6 is asymptotically tight for any square recursive divide-and-conquer matrix multiplication algorithm (that is, involving only square recursive steps), in the sense that there exists an implementation of any such algorithm using a number of cache I/Os proportional to the bound of Theorem 6. To see this, consider the following implementation for computing the CDAG G of any recursive divide-and-conquer algorithm to multiply square matrices: [5]

[5]Here "algorithm implementation" refers to the specific sequence of vertex computations and cache I/Os used to realize an algorithm, while "algorithm" is functionally synonymous with CDAG.

Algorithm Implementation 1

- *Let I be the set of subcomputations of G whose matrix dimension is $\geq \Theta\left(\sqrt{M}\right)$ and none of whose subcomputations satisfy this property.*
- *Compute, in arbitrary order, the vertices in the encoding graphs of G not in I, and then the vertices in the inputs of the subcomputations in I, performing cache I/Os when necessary.*
- *Compute each subcomputation in I via Algorithm Implementation 2 below.*
- *Using the outputs of the subcomputations in I, computed above, compute the vertices in the decoding graph of G not in I in arbitrary order, performing cache I/Os when necessary.*

Algorithm Implementation 2

- *To compute matrix multiplication X with base graph G_1, do the following:*
- *If X is simply an elementary multiplication, compute it.*
- *Otherwise, compute in order the matrix multiplications X_1, X_2, ..., X_n, each via Algorithm Implementation 2, where X_1, X_2, ..., X_n are the submultiplications G_1 depends on. Do not perform any internal I/Os.*

Note that the maximum number of vertices that must be in cache during a run of Algorithm Implementation 2 is within a constant factor of the number of input/output vertices in a chain of subcomputations, one at each recursive level, down from a subcomputation X in I. Since there are $\Theta(M)$ input/output vertices of X and at most half as many input/output vertices in a subcomputation relative to its parent computation, this implementation computes X using $\Theta(M)$ cache without any internal I/Os. Thus the maximum number of I/Os performed during Algorithm Implementation 1 is proportional to the number of input/output vertices of the subcomputations in I, plus the number of input/output vertices not in the subcomputations of I, which is itself proportional to the total number of input/output vertices of I.

Thus the maximum number of cache I/Os required by this implementation is at most a constant factor times the total number of vertices in the inputs and outputs of I, which is proportional to $\sum_{X \in I} \text{max\_size}(X)$. Hence Theorem 6 is tight up to a constant factor for square matrix multiplication algorithms. This bound is again

tight for all "near-square" matrix multiplication algorithms, algorithms whose submultiplications all (or at least for those of dimension $\sim \sqrt{M}$) have their largest and smallest matrix dimension within a constant factor. Theorem 7, a simplification of Theorem 6, is thus tight for square and near-square matrix multiplications when all base graphs have the same exponent. Note that this algorithm implementation is the serialization of the parallel CAPS algorithm implementation presented in [3].

For rectangular matrix multiplications, Theorem 6 might not be tight. If a submultiplication of dimensions l, m, and n has its minimum dimension (that is, $\min(l, m, n)$) less than $\Theta\left(\sqrt{M}\right)$ but its other dimensions significantly larger, it is unclear whether its I/O-complexity is proportional to the number of entries in its inputs/outputs, as in the above algorithm, or significantly larger, resulting in an algorithm of worse I/O-complexity than the bound of Theorem 6. However, if each submultiplication whose minimum matrix dimension is $= \Theta\left(\sqrt{M}\right)$ is computable without any internal I/Os, then the previous argument holds, yielding an algorithm implementation achieving the bound of Theorem 6 up to a constant factor.

3.3 Internal I/O Bounds for Submultiplications

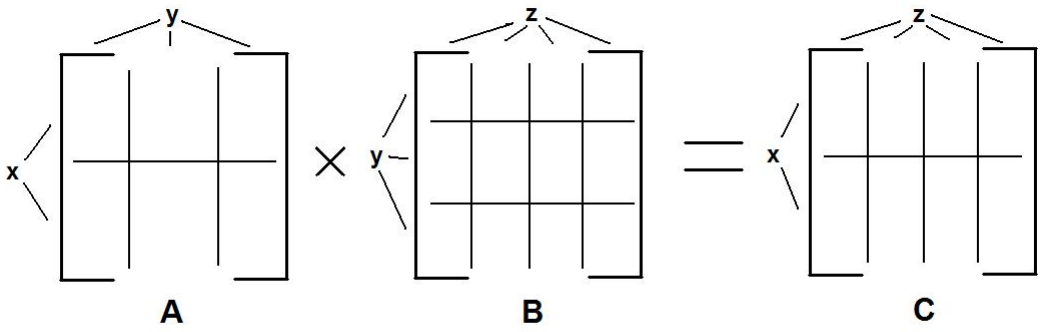
In this section we generalize the results and proofs of Chapter 2. For now we still assume that every intermediate nontrivial linear combination is used in only one multiplication within each base graph. As in the Strassen-like case, we first prove the existence of efficient routings within submultiplications and, using many such routings, prove Theorem 6 when $\omega(G) < 3$. We also derive an I/O bound for subcomputations of sufficient size; this was not done in Chapter 2, but will be necessary to prove Theorem 6 in the general case in the following chapter.

Several of the proofs in this section are analogous to those in Chapter 2; such proofs are abbreviated below, with only their differences with the corresponding Strassen-like proofs spelled out in detail. We will first give the natural generalizations of these proofs; unfortunately, the resulting theorem (Theorem 13) will not be sufficient in the case of rectangular submultiplications. We will then show how to further generalize our routing argument to give yet a stronger version of the Routing Theorem, and thus prove our result in general.

Let G_1 be the base graph corresponding to any recursive step in a divide-and-conquer matrix multiplication algorithm. Suppose that G_1 is applied to the multiplication of an $l \times m$ matrix A by an $m \times n$ matrix B to get an $l \times n$ matrix C, and suppose G_1 does so by dividing A vertically into x blocks and horizontally into y blocks, and dividing B vertically into y blocks and horizontally into z blocks (and

then the product C is divided into x blocks vertically and z blocks horizontally). G_1 then contains $xy + yz$ inputs and xz outputs. As before, let G_1' be the portions of G_1 corresponding to the encoding graph of A and the decoding graph of C (omitting only the encoding graph of B). See Figure 3.2.

Figure 3.2: To multiply two rectangular matrices A and B, a divide-and-conquer matrix multiplication algorithm divides them into blocks, computes linear combinations, multiplies those linear combinations recursively, and takes linear combinations of the products to find C. Here $x = 2$, $y = 3$, and $z = 4$.



We also remove the assumption that every intermediate nontrivial linear combination of inputs within a base graph is used in only one multiplication. This requires adding an extra layer of complexity to the previous proof, which also allows the path-routing counting argument to generalize to matrix multiplication algorithms with reused nontrivial linear combinations.

Previously we showed the existence of an n_0-routing between the guaranteed dependencies (a, c) of each base graph computing $C = AB$, where a is an input of A and c an output of C. Via Hall's Matching theorem, we were able to construct such a routing that hit every multiplication vertex at most n_0 times. However, a linear combination (of the inputs) vertex may be used in many multiplications, resulting in the linear combination vertex being hit more than n_0 times. Also, the inputs and output vertices of a base graph may be hit more/fewer times than the multiplication vertices, simply because the numbers of vertices in the input/output matrices may be different (they aren't both n_0^2).

To alleviate these issues, we allow paths to "jump" within the matrix multiplication CDAG between vertices with the same membership in S. Note that this does not alter the counting argument (of boundary-crossing paths) used in the proof of

Theorem 3. Many of the "jumps" used will be within the same rank, while some will involve jumping between an output and an input vertex of the same submultiplication with the same membership in S.

Lemma 8 *Let A be $x \times y$, B be $y \times z$, and C be $y \times z$, where A, B, and C are the inputs/outputs of a matrix multiplication subcomputation. Let G_1 be a base graph multiplying $C = AB$ and S be a set of vertices in G_1. Then there exists a routing of paths between A and C that hits every linear combination vertex at most $\min(y, z)$ times, where each path goes between a vertex of A in S and a vertex of C not in S, and which consists of a number of paths equal to the number of guaranteed dependencies (a, c) for which $a \in S$ and $c \notin S$. There is a similar routing of paths from vertices of A not in S to vertices of C in S, and similar for paths between B and C (replacing z with x).*

Intuitively, this lemma states that there exists an efficient routing of guaranteed dependencies (a, c) with $a \in S$ and $c \notin S$ where each path from a to c may start not at a itself, but at another vertex in A that is also in S.

Proof. As usual, we apply Hall's Matching theorem. Let D be a set of guaranteed dependencies (a, c) between a vertex $a \in S \bigcap A$ and a vertex $c \in \overline{S} \bigcap C$ and T be the set of vertices of C represented in D (which are by definition all in $\overline{S}$). Define V_D to be the set of linear combination vertices v for which there exists an edge from a vertex in $S \bigcap A$ to v and an edge from v to a vertex in T. Let c_i, $1 \leq i \leq x$, be the number of vertices in the ith row of C that are in T if row i of A has at least one vertex in S; otherwise let $c_i = 0$. Then $|D| \leq (c_1 + ... + c_x)y$, since for every row of c_i vertices of C in $\overline{S}$, there are at most y guaranteed dependencies counted by D for each such vertex, and no guaranteed dependencies counted by D if none of the corresponding vertices of A are in S.

Intuitively, each row i of C whose corresponding row in A has at least one vertex in S requires at least c_i linear combination vertices for the dependence of those c_i vertices of C in $\overline{S}$ on the vertex of A in S. Each such linear combination vertex is counted by V_D. More precisely, after setting every vertex of A not in S to 0, the span of the elements of C counted by the c_is – ignoring elements of C that are now identically 0 – is of size $c_1 + ... + c_x$, and thus requires at least $c_1 + ... + c_x$ linear combination vertices, each of which is by construction counted by V_D. Thus $|V_D| \geq c_1 + ... + c_x$. From this we see that $|D| \leq |V_D|y$, and so $|V_D| \geq \frac{|D|}{y}$.

By Hall's Matching theorem, there exists a matching between the guaranteed dependencies (a, c) (with $a \in S \bigcap A$ and $c \in \overline{S} \bigcap C$) and linear combination vertices they depend on that have some incoming edge from $A \bigcap S$, using each linear combination vertex at most z times. In other words, every guaranteed dependence (a, c)

is matched with a linear combination vertex v it depends on such that there exists some edge (u, v) where u is a vertex in A that is also in S. Simply route the path for each guaranteed dependence along this path. Every linear combination vertex (and thus also every multiplication vertex) is thus hit at most y times. A similar argument establishes a routing from vertices in A that lie $\overline{S}$ to vertices in C lying in S that hits every vertex at most y times.

The same construction applies swapping the roles of A and C, yielding a routing that hits every linear combination vertex at most z times. Simply picking the best of these two routing (depending on which is the minimum of y and z) yields the desired routing.

□

We will also need the following extension:

Lemma 9 *There exists a routing of paths between A and B that hits every linear combination vertex at most $\min(x, z)$ times, where each path goes between a vertex of A in S and a vertex of B not in S, and which consists of a number of paths equal to the number of guaranteed dependencies (a, b) for which $a \in S$ and $b \notin S$. There is a similar routing of paths from $\overline{S} \bigcap A$ to $S \bigcap B$.*

Proof. Analogous to the above proof. For every set D of guaranteed dependencies from $S \bigcap A$ to $\overline{S} \bigcap B$, define V_D and T similarly to before (replacing C with B). Let b_i, $1 \leq i \leq y$, be the number of vertices of the ith row of B that are in $\overline{S}$ and used in D, or 0 if all vertices in the ith column of A are in $\overline{S}$. Then $|D| \leq (b_1 + ... + b_y)x$.

Setting every vertex in $\overline{S}$ to 0, the span of the linear combinations of elements of A and of B is at least $b_1 + ... + b_y$, and so $|V_D| \geq b_1 + ... + b_y$. This yields $|V_D| \geq \frac{|D|}{x}$. By Hall's Matching theorem this yields a matching from guaranteed dependencies between A and B and linear combination vertices they depend on on a path from S to $\overline{S}$, yielding a routing that hits every linear combination vertex at most x times. A similar routing from vertices in $\overline{S} \bigcap A$ to $S \bigcap B$ hits every linear combination vertex at most x times, and an analogous argument yields routings from B to A hitting every vertex at most z times. □

Via the usual recursive argument we get the following analogue of Lemma 3:

Lemma 10 *If paths are allowed to jump between vertices with the same membership in S, then there exists a $\max(2ma_{max}, 2na_{max})$-routing of guaranteed dependencies between A and C and a $\max(2la_{max}, 2ma_{max})$-routing of guaranteed dependencies between B and C.*

Proof. Compare Lemma 3. Apply the routings constructed in Lemma 8 within each base graph. In each submultiplication step multiplying $C = AB$, we must route

paths between the guaranteed dependencies (a, c) of A and C. If a and c have the same membership in S, route the path by simply "jumping" from a to c directly. If $a \in S$ and $c \notin S$, use a routing constructed by Lemma 8 from S to $\overline{S}$. If $a \notin S$ and $c \in S$, use a routing constructed by Lemma 8 from $\overline{S}$ to S. This has the effect of doubling the number of times every vertex is hit just once, as opposed to once per depth of the recursion tree. [6]

Within the first base graph (representing the outermost submultiplication step), every input vertex in A is hit at most $2na_{max}$ times and every output vertex at most $2ma_{max}$ times (the a_{max} because "jumping" paths may hit every input/output vertex additional times, at most one for the number of inputs/outputs of the base graph). Suppose a submultiplication of size x, y, z performed within this outer base graph multiplies matrices $C_1 = A_1B_1$ of dimensions l_1, m_1, n_1, and assume that every input/output vertex of the submultiplication is hit at most $2\max(m_1, n_1)a_{max}$ times in a path routing of guaranteed dependencies between A_1 and C_1, with every non-input/output vertex hit at most $2\max(m_1, n_1)$ times.

To route paths between the guaranteed dependencies of A and C, from S to $\overline{S}$, every path in each of the routings between A_1 and C_1 (from S to $\overline{S}$ and $\overline{S}$ to S) is used at most $\min(y, z)$ times (Lemma 8). Thus every vertex within that submultiplication is used at most $2\max(m_1, n_1)a_{max}\min(y, z) \leq 2\max(m_1y, n_1z)a_{max} = 2\max(m, n)a_{max}$ times. By induction, every vertex within the matrix multiplication CDAG is hit at most $2\max(m, n)a_{max}$ times. Similar logic yields a $2\max(l, m)a_{max}$-routing of paths between the guaranteed dependencies of B and C. $\square$

Via the same argument, we can prove

Lemma 11 *If paths are allowed to jump between vertices with the same membership in S, then there exists a $\max(2la_{max}, 2na_{max})$-routing of guaranteed dependencies between A and B.*

We now prove a generalization of Lemma 4; this proof will require slightly more careful counting to yield the bounds we want, but is otherwise identical. Let In_A denote the set of inputs of G corresponding to elements of A, In_B those corresponding to elements of B, and Out denote the outputs of G (which correspond to elements of C).

Lemma 12 *Fix a routing F_A for the guaranteed dependencies of G between A and C and a routing F_B for the guaranteed dependencies of G between B and C. Then*

[6] This can actually be shown not to increase the number of I/Os at all, since the boundary-crossing edge (u, v) contributes one I/O only from the vertex u, not v, but this is not necessary in our proof.

there exists a routing in G between In_A and Out such that every path in the routing consists of the concatenation of chains in F_A and F_B – some reversed in direction – such that every chain in F_A is used l times and every chain in F_B is used $2l$ times. There similarly exists a routing between In_B and Out composed of chains in F_A and F_B such that each chain in F_A is used $2n$ times and every chain in F_B is used n times.

Proof. We use the same explicit sequences of guaranteed dependencies. To route a_{ij} to $c_{i'j'}$ we use

$$a_{ij} \to c_{ij'} \to b_{jj'} \to c_{i'j'}$$

where $1 \le i, i' \le l$, $1 \le j \le m$, and $1 \le j' \le n$. For every choice of i' the first $a_{ij} \to c_{ij'}$ guaranteed dependence is used, for every choice of i' the second $c_{ij'} \to b_{jj'}$ dependence is used, and for every choice of i the third $b_{jj'} \to c_{i'j'}$ dependence is used. Thus as i, j, i', and j' range over all possibilities, every guaranteed dependence between A and C is used l times and every guaranteed dependence between B and C is used $2l$ times.

Similarly, for the sequence of guaranteed dependencies

$$b_{ij} \to c_{i'j} \to a_{i'i} \to c_{i'j'}$$

used to route b_{ij} to $c_{i'j'}$ for $1 \le i' \le l$, $1 \le i \le m$, and $1 \le j, j' \le n$, we see that each guaranteed dependence in this chain is used n times, proving the lemma. □

Theorem 8 (Generalized Routing Theorem) *If G is the CDAG for divide-and-conquer matrix multiplication of $l \times m$ matrix A by $m \times n$ matrix B, then there exists a $6la_{max}\max(l,m,n)$-routing between In_A and Out in which paths may jump between vertices with the same membership in S.* [7] *There is similarly a $6na_{max}\max(l,m,n)$-routing between In_B and Out. The same bounds hold for the meta-vertices of G.*

Proof. Apply Lemmas 10 and 12. To route paths between In_A and Out we require l copies of a routing of guaranteed dependencies between A and C, each of which hits every vertex at most $2\max(m,n)a_{max}$ times, and $2l$ copies of a routing of guaranteed dependencies between B and C, each of which hits every vertex at most $2\max(l,m)a_{max}$ times. Thus every vertex is hit at most $(2l\max(m,n) + 4l\max(l,m))a_{max} \le 6l\max(l,m,n)a_{max}$ times. As before, the same bound applies to meta-vertices as well (since except for path "jumping," which results in only an

[7] Recall that this means there is a path in the routing between every vertex in In_A and every vertex in Out.

extra factor of a_{max}, every path passing through a meta-vertex passes through its root vertex as well). $\square$

Note that when $l = m = n = n_0^k$ and a_{max} is small, this yields a $\Theta(n_0^k)$-routing between the inputs and outputs of G, as before in Theorem 4.

Just as before we can apply the Generalized Routing Theorem to derive a bound on the internal I/O-complexity of a matrix multiplication subcomputation:

Theorem 9 *If G is the CDAG for divide-and-conquer matrix multiplication of $l \times m$ matrix A by $m \times n$ matrix B, then*

$$IO^+(G) \geq \frac{1}{144a_{max}} \min(lm, mn, ln)$$

as long as $M \leq \frac{1}{72a_{max}} \min(lm, ln, mn)$

Proof. As in the Strassen-like case, we break the computation of G into segments such that each segment (except perhaps the last) contains a sufficient number of input vertices; we need use only the input vertices of A.[8] If such a segment S contains I vertices among the inputs of A, then the Generalized Routing Theorem guarantees a $6la_{max} \max(l, m, n)$-routing of $I \cdot ln$ paths between those vertices and all the vertices representing the outputs of C such that every meta-vertex is also hit at most $6la_{max} \max(l, m, n)$ times. Throughout this proof we use the fact that if $x \geq 1$ then $\lfloor x \rfloor \geq \frac{1}{2}x$ and $\lceil x \rceil \leq 2x$; the constant in this bound could likely be improved by more careful counting, but is sufficient for our purposes.

Choose each segment (except maybe the last) to contain $I = \lceil \frac{72Ma_{max} \max(l,m,n)}{n} \rceil$ input vertices of A. Either S contains at least $\frac{ln}{2}$ output vertices of C or it does not. In the latter case there exist at least $I \cdot \frac{ln}{2}$ boundary-crossing paths in the routing between vertices of A in S and vertices of C not in S. In the former case, note that $I \leq \lceil \frac{lmn}{2n} \rceil = \lceil \frac{lm}{2} \rceil$, and so there are at least $\frac{lm}{4} \geq \frac{I}{4}$ vertices of A not in S. Thus there are at least $I \cdot \frac{ln}{4}$ boundary-crossing paths in the routing between vertices of A not in S and vertices of C in S. Either way,

$$|\delta'(S')| \geq \frac{\frac{1}{4}I \cdot ln}{6la_{max} \max(l, m, n)} \geq \frac{\frac{1}{4}\lceil \frac{72Ma_{max} \max(l,m,n)}{n} \rceil n}{6a_{max} \max(l, m, n)} \geq 3M$$

where S' is the set of meta-vertices in the segment S. Therefore S requires at least $3M - 2M = M$ I/Os. Each such I/O corresponds to an edge in G between a vertex in S and a vertex not in S; it is clear that none of these I/Os represent an output from

[8] Applying the same technique using a routing between B and C instead of between A and C yields exactly the same result.

G or the first inputting of an input vertex of G, and are thus counted by internal I/O-complexity. Then

$$\mathrm{IO}^+(G) \geq \left\lfloor \frac{lm}{\left\lceil \frac{72Ma_{max}\max(l,m,n)}{n} \right\rceil} \right\rfloor M \geq \left\lfloor \frac{lmn}{72Ma_{max}\max(l,m,n)} \right\rfloor M \geq \frac{1}{144a_{max}} \frac{lmn}{\max(l,m,n)}$$

$$= \frac{1}{144a_{max}} \min(lm, mn, ln)$$

as long as $M \leq \frac{1}{72a_{max}} \min(lm, ln, mn)$ (which implies there is at least one complete segment and thus this floor is nonzero). □

Corollary 1 *Every subcomputation of a divide-and-conquer matrix multiplication algorithm involving matrices that each have* $\geq 72a_{max}M$ *entries has internal I/O-complexity of at least* $\frac{1}{2}M$.

In other words, if a matrix multiplication subcomputation is reasonably large, then computing it requires at least as many I/Os that are not due to inputs/outputs as the size of smallest matrix, up to a constant factor. Compare Equation 2.2.

While not necessary to the rest of the proof for $\omega(G) < 3$, the following theorem gives some intuition about the I/O-complexity of small square matrix multiplications. This theorem is however vital to the more general proof of Theorem 6 given in Chapter 5.

Theorem 10 *Let* G *be a CDAG for divide-and-conquer matrix multiplication of* $l \times m$ *matrix* A *by* $m \times n$ *matrix* B *where* $L \leq l, m, n \leq rL$. *Let* S *be a subset of the vertices of* G *containing* s *inputs of* A, *inputs of* B, *and outputs of* C *in total and let* S' *be the corresponding set of meta-vertices. If* S *contains fewer than* $0.8 \cdot lm$ *input vertices of* A, *fewer than* $0.8 \cdot mn$ *input vertices of* B, *or fewer than* $0.8 \cdot ln$ *output vertices of* C, *then*

$$\delta'(S') \geq \frac{2}{225r^4}s.$$

Proof. This proof is similar in form to the above proof, except that it uses only one segment. Since $6lr\max(l,m,n) \leq 6r^3L^2$, the Generalized Routing Theorem guarantees a $6r^3L^2$-routing of paths between the inputs of A and outputs of C and similarly a $6r^3L^2$-routing between the inputs of B and outputs of C.

We will show the existence of at least $\frac{4}{75_{max}}sL^2$ boundary-crossing paths in one or the other of the routings. This proof is fairly technical, but the result is quite intuitive: unless most of the inputs and outputs of a matrix multiplication are all in

cache at the same time (in S), the number of adjacent meta-vertices is proportional to the number in cache.

Suppose first that $|I_A| \geq \frac{s}{3}$. If $|I_C| \leq 0.8ln$, then the number of boundary-crossing paths between A and C is $\geq \frac{s}{3} \cdot 0.8ln \geq \frac{4}{15}sL^2$. Otherwise, $|I_C| \geq 0.8ln \geq 0.8\frac{lm+mn+ln}{3r} \geq \frac{4}{15r}s$. Either $|I_A| \leq 0.8lm$ or $|I_B| \leq 0.8mn$ by assumption; either way, the number of boundary-crossing paths – either between A and C or between B and C – is at least $0.2L^2 \cdot \frac{4s}{15r} = \frac{4}{75r}sL^2$. A similar argument holds if $|I_B| \geq \frac{s}{3}$.

Finally, suppose $|I_C| \geq \frac{s}{3}$. If either $|I_A| \leq 0.8lm$ or $|I_C| \leq 0.8mn$ then there are $\geq \frac{s}{3} \cdot 0.2L^2 = \frac{1}{15}sL^2$ boundary-crossing paths. Otherwise, $|I_C| \leq 0.8ln$ by assumption and we have $|I_A| \geq 0.8lm \geq \frac{4}{15r}s$ (via the same logic as in the previous case). Then between A and C there are at least $0.2L^2 \cdot \frac{4s}{15r} = \frac{4}{75r}sL^2$ boundary-crossing paths.

In any case there are at least $\frac{4}{75_{max}}sL^2$ boundary-crossing paths in one of the routings, and so

$$\delta'(S') \geq \frac{\frac{4}{75r}sL^2}{6r^3L^2} = \frac{2}{225r^4}s.$$

□

Unfortunately, the previous two theorems will be insufficient for our study of rectangular matrix multiplication algorithms. Our strategy in the coming sections will be to consider many small submultiplications that each have internal I/O of at least $\Theta(M)$. But if these submultiplications involve matrices of wildly different sizes (very long and skinny matrices, for example), this bound is too small relative to the number of such submultiplications that must compose the overall matrix multiplication. We therefore prove a result analogous to Theorem 9 for matrices of significantly different sizes; this effectively replaces the "min" in Theorem 9 with a "max."

3.4 Improving the Routing Theorem

In this section we strengthen the Generalized Routing Theorem (Theorem 8) to the following:

Theorem 11 (Improved Routing Theorem) *Let G be a CDAG for divide-and-conquer matrix multiplication $C = AB$ of dimensions $l \geq m \geq n$ and let A' be a subset of A such that no row of A contains more than n vertices in A'. If $|A'| \leq mn$, then there exists a $2lmn$-routing of $|A'| \cdot |C|$ paths (perhaps jumping between vertices with the same membership in S), m for each guaranteed dependence (a', c) with $a' \in A'$ and $c \in C$. The same result holds if no column of A contains more than n vertices of A'. A similar result holds for any permutation of sizes l, m, and n.*

Clearly, a vertex a in A must be hit at least once for every pair (a, c) in D, and similarly for C. Thus this theorem states that as a result of constructing this routing between arbitrary pairs (a, c), no vertex is hit significantly more than the minimum number of times a vertex must be hit simply by necessity of routing all pairs in D. Proving this theorem will require generalizing the logic of the previous section. As before, only the changes from the previous arguments are highlighted. From the following generalization of Lemma 10 the proof is simple:

Theorem 12 *Let D be a set of guaranteed dependencies between A and C. If paths are allowed to jump between vertices with the same membership in S, then there exists a routing of guaranteed dependencies between A and C hitting every vertex no more than the maximum number of times a vertex appears in D. Analogous bounds hold for routings between B and C and between A and B.*

Proof of Theorem 11 from Theorem 12. First suppose that every row of A contains no more than n vertices of A'. Route every vertex in A' to every vertex in C that has a guaranteed dependence on it. Route every vertex in C to every vertex in B it has a guaranteed dependence on. Route back every vertex in B to every vertex in C with a guaranteed dependence on it. For every $a' \in A'$ and $c \in C$, this forms a path from a' to c passing through each of the m vertices of B that c has a guaranteed dependence on.

By Theorem 12, the vertex hit the maximum number of times occurs not in the middle of one of these routings, but at the end. Thus it is sufficient to count the number of times that each endpoint of these routing pieces is hit:

- Every vertex in A' is hit once for every vertex in c and vertex in b that it depends on, a total of $m|C| = lmn$ times.
- Every vertex in A but not in A' is never hit.
- Every vertex in B is hit once for every vertex in A' and every vertex in C that depends on it, a total of $l|A'| \leq lmn$ times.
- Every vertex in C is hit in two different situations: $m|A'|$ times as the end of a path, and in the middle of a path once for every vertex of A' in the same row and every vertex of C in the same column and every vertex of B in the same column. This is at most $m|A'| + lmn \leq 2lmn$ times.

Thus every vertex is hit at most $2lmn$ times.

Now suppose that every column of A contains no more than n vertices of A'. Route every vertex in A' to every vertex in B that has a guaranteed dependence on

it. Route every vertex in B to every vertex in C that has a guaranteed dependence on it. Route back every vertex in C back to every vertex in B it has a guaranteed dependence on. For every $a' \in A'$ and $b \in C$, this forms a path from a' to b passing through each of the l vertices of C that b has a guaranteed dependence on.

- Every vertex in A' is hit once for every vertex in B and vertex in C that it depends on, a total of $l|B| = lmn$ times.
- Every vertex in A but not in A' is never hit.
- Every vertex in C is hit once for every vertex in A' and every vertex in B that depends on it, a total of $m|A'| \leq lmn$ times.
- Every vertex in B is hit in two different situations: $l|A'|$ times as the end of a path, and in the middle of a path once for every vertex of A' in the corresponding column and every vertex of C in the same column and every vertex of B in the same column. This is at most $l|A'| + lmn \leq 2lmn$ times.

Hence every vertex is again hit at most $2lmn$ times.

□

From this we can prove a stronger result on the internal I/O-complexity of a rectangular submultiplication. From now on we revert to asymptotic notation for simplicity, and because the precise counting argument is analogous to the proof of Theorem 9.

Theorem 13 *If G is the CDAG for divide-and-conquer matrix multiplication of $l \times m$ matrix A by $m \times n$ matrix B, then*

$$IO^+(G) \geq \Theta\left(\max(lm, mn, ln)\right)$$

as long as $M \leq \Theta\left(\min(l, m, n)^2\right)$

Proof. Assume first that $l \geq m \geq n$, and ensure $M \leq \frac{1}{8}n^2$. Partition the sequence of computations into segments containing $\Theta(M)$ vertices of A. For each such segment S, partition the rows of A into rows containing $\leq \frac{n}{2}$ vertices and those containing $> \frac{n}{2}$ vertices of S.

Suppose rows of the former type contain at least half of the specified vertices in S. Let A' be a set of vertices of A defined as follows: for every row of A containing $k \leq \frac{n}{2}$ vertices of S, add these k vertices of S to A', and then add another k vertices of $\overline{S}$ in the same row to A'. Apply Theorem 11 between A' and C (that is, all guaranteed dependencies between A' and C): in this routing exactly half

of all paths are boundary-crossing, so there are at least $\Theta(M)m|C| = \Theta(lmnM)$ boundary-crossing paths, resulting in at least $\Theta\left(\frac{lmnM}{2lmn}\right) = \Theta(n^2) = \Theta(3M)$ I/Os. After subtracting the $2M$ "free" I/Os (those uncounted by internal I/O-complexity), this results in $\Theta(M)$ I/Os for this segment.

If instead rows of the latter type contain half of the vertices of S, then $\Theta(M)$ vertices of A' are in columns of A that contain $\leq \frac{n}{2}$ vertices of A' each, and an analogous argument applies (routing to B instead of to C). This again yields $\Theta(M)$ I/Os for the segment. Thus the internal I/O-complexity of this matrix multiplication is $\Theta\left(\frac{lm}{M} \cdot M\right) = \Theta(lm)$. Similar arguments apply for any ordering of l, m, and n. See Section 6.2 for a graphical illustration of a similar proof. □

We now turn to proving this improved guaranteed-dependence-routing theorem (Theorem 12) that the improved Routing Theorem depends upon. Before we showed the existence of a matching from guaranteed dependencies to valid intermediate vertices they may pass through; it turned out that allowing paths to "jump" before beginning this routing was vital.

Now we must, in effect, match weighted guaranteed dependencies to valid intermediate vertices such that the total weight routed through each intermediate vertex is no more than the maximum total weight of an input/output vertex. For a set of pairs D, define mult(D) to be the maximum multiplicity of an element as a component of a pair in D.

Proof of Theorem 12. This proof is a generalization of those of Lemma 8 and Lemma 10. The routing generated by those two lemmas may hit intermediate vertices more than an input/output vertex is hit if for every vertex v, the set D does not contain all the guaranteed dependencies involving v. We first prove the following claim:

Claim 3 *If G_1 is a base graph for $C_1 = A_1B_1$ and D is a subset of the guaranteed dependencies between A_1 and C_1, there exists a routing of D (perhaps involving jumping paths, in the same sense as in Lemma 8) such that every linear combination vertex of G_1 is hit at most mult(D) times.*

Compare Lemma 8.

Proof. Assume without loss of generality that the minimum number of times a vertex appears in D occurs for a vertex in C (if not, apply the symmetric argument swapping A and C). Using the same notation as in the proof of Lemma 8, we now have $|D| \leq (c_1 + ... + c_x)\text{mult}(D)$, since every vertex in C has a guaranteed dependence in D on at most mult(D) vertices of A. This yields $|V_D| \geq \frac{|D|}{\text{mult}(D)}$. By Hall's Matching theorem there then exists a matching of guaranteed dependencies to intermediate vertices using each at most mult(D) times. □

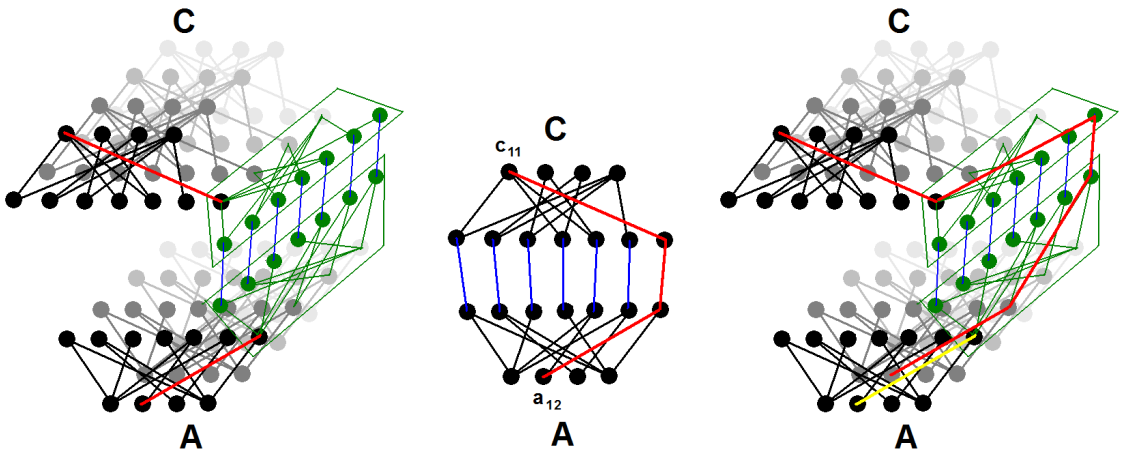
We now perform the usual recursive construction. Consider a computation X multiplying $C = AB$. For this computation, A and C (and B) are broken into blocks, and the products of certain linear combinations of these blocks are computed recursively. We must route the guaranteed dependencies between these blocks through these intermediate linear combinations. There are many base graph copies for this single recursive step, one for each position within the A blocks (this corresponds to the vertices on the first two ranks of the CDAG). Using the above routing for each such base graph copy, every input/output vertex of each submultiplication of X is hit no more times than the input/output vertices of X. By induction, this implies that every internal vertex of each submultiplication is hit no more than the maximum number of times any of its input/output vertices are hit, proving the claim. □

This proof can also be interpreted as constructing a routing of chains within G one CDAG level at at time, working out to in. To better illustrate this somewhat hard-to-visualize proof and give intuition about the form of this routing, we show an example of its logic in the case of Strassen's algorithm applied to 4×4 matrices, shown in Figure 3.3. Consider the base graph defined by the front-most, black, highest-level encoding and decoding graphs in Figure 3.3 (a). By the above claim, there exists an assignment of guaranteed dependencies in this base graph to multiplication vertices that uses each multiplication vertex no more times than one of its input/output is used.

For example, a path may be routed through this base graph as shown in red in (b). In the overall CDAG, this corresponds to the edges shown in red (and the analogs in the other top-level base graph copies shown in grey). A chain from A to C in the overall CDAG that follows one of these red edges needn't follow the other red edge; overall chains from A to C are formed by "splicing" routings for the middle-level base graphs (such as the one shown in green in (a)) inside the routings for each top-level base graph. Thus a path starting at C may follow the red edge down, go through the middle-layer (green) base graph to reach a different base graph copy than the one it started in, and follow an edge within the routing for that base graph (shown in red) to reach A, instead of taking the edge in the original base graph copy partnered with the first edge it transversed (shown in yellow in Figure 3.3 (c)). This path is shown in red in Figure 3.3 (c).

By Claim 3, within each of the highest-level base graph copies (shades of grey in Figure 3.3 (a) and (c)) the second-level vertices (those used as inputs/outputs to base graphs on the same level as the green base graph) are hit no more often than the inputs to each such base graph. The same argument applied to the green middle-level base graph, whose inputs/outputs are the aforementioned vertices, implies that the elementary multiplication vertices (and their inputs) are hit no more often than

any overall input/output vertex. This applies for every middle-level base graph (in parallel with the one shown in green). Note that this argument applies regardless of the subset of guaranteed dependencies to be routed, proving Theorem 12.



(a) (b) (c)

Figure 3.3: (a) The CDAG for Strassen matrix multiplication of 4×4 matrices, involving two recursive levels. Only the encoding graph for A and the decoding graph for C are shown. To make the construction easier to visualize, the four base graph copies on the top/bottom layer are shown in different shades of grey, while only one out of the 7 copies of the encoding and decoding graphs on the middle layer are shown (green). Edges denoting an input to an elementary matrix multiplication are shown in blue. (b) An example chain within one top-level base graph, shown in red; the corresponding edges in (a) are also shown in red. (c) An example chain between A and C within the overall graph shown in red; note that this chain uses only the top of the highlighted edges from (a) but not the bottom (shown in yellow).

3.5 Adding I/O Bounds When $\omega(G) < 3$

In the following chapter we use a more involved argument to give a full proof of Theorem 6 for any $\omega(G)$. In this chapter we develop some fundamental results about the structure of disjoint submultiplications and complete the proof of Theorem 6 for $\omega(G) < 3$ relying on Theorem 13. Recall that in Chapter 2 we first proved the existence of sufficiently many disjoint submultiplications of sufficient size and then

used the Routing Theorem to find efficient path routings in each, leading to a tight I/O-complexity lower bound.

The proof of Chapter 2 generalizes easily to divide-and-conquer matrix multiplication algorithms of exponent $\omega(G) < 3$. However, it first requires proving the existence of $\Omega(I)$ disjoint subcomputations, where I is as in Theorem 6. To that end, we prove that at least $\frac{1}{a_{max}^3}$ of the computations of I are mutually disjoint as long as $\Omega(G) < 3$, where a_{max} is again the maximum number of blocks a matrix is divided into horizontally or vertically per recursive step.

By Fact 2, in each subcomputation A_1B_1 either the vertices of A_1 are all copies of those input to G_1 or none of them are, and similarly for B_1. If the latter case holds for both A_1 and B_1, then A_1B_1 represents an input-disjoint subcomputation and thus a disjoint subcomputation. Call a linear combination $\alpha_1 v_1 + ... + \alpha_r v_r$ of $v_1, v_2, ..., v_r$ *trivial* if at most one $\alpha_i \neq 0$. Then it suffices to show that any base graph G_1 with $\omega(G_1) < 3$ contains a multiplication vertex that multiplies two nontrivial linear combinations of entries of its input matrices. Note that because $\omega(G) < 3$, every base graph G_1 computes some non-trivial linear combination of one of its inputs in the encoding steps, but this does not immediately prove that two non-trivial linear combinations are multiplied together.

Theorem 14 *If a matrix multiplication algorithm computes $C = AB$ (with dimensions l, m, and n as before) by computing products of the form $a_{ij} \cdot \sum_{j',k'} w_{j'k'} b_{j'k'}$ and/or products of the form $b_{jk} \cdot \sum_{i',j'} W_{i'j'} a_{i'j'}$ for some scalars $w_{j'k'}$ and $W_{i'j'}$ and then taking linear combinations to yield the entries of C, then the algorithm uses at least lmn multiplications.*

Intuitively, this lemma states that any matrix multiplication algorithm base graph that is better than classical matrix multiplication computes some product of nontrivial linear combinations: a nontrivial linear combination of entries of A multiplied by a nontrivial linear combination of entries of B.

Proof. Suppose C is computed using only products of the above forms; let V_j be the set of all computed products of the forms $a_{ij} \cdot \sum_{j',k'} w_{j'k'} b_{j'k'}$ and $b_{jk} \cdot \sum_{i',j'} W_{i'j'} a_{i'j'}$ as i ranges from 1 to l and k ranges from 1 to n. Note that this partitions all the computed products – except for those of the form $a_{ij}b_{lk}$ for $j \neq l$ – unambiguously into m sets, since the product $a_{ij}b_{jk}$ (which takes the form of both types of products) falls into V_j regardless of which form of product it is considered to be. Fix $1 \leq j' \leq m$. We will set most of A and B to 0 to zero out all products not in $V_{j'}$.

Considering a_{ij} and b_{jk} as indeterminates, set $a_{ij} = 0$ whenever $j \neq j'$ and similarly set $b_{jk} = 0$ whenever $j \neq j'$. In other words, "zero out" the matrix A to consist of only its j'th column (of indeterminates a_{ij} as before) and the matrix B to contain only its j'th row. Then the product matrix AB contains all the terms $c_{ik} = a_{ij'}b_{j'k}$ for all $1 \leq i \leq l$ and $1 \leq k \leq n$. For every j, let V_j° denote the set of products of V_j with the aforementioned indeterminates a_{ij} and b_{jk} zeroed out. Note that all products $a_{ij}b_{lk}$ with $j \neq l$ are 0. Then for $j \neq j'$ by assumption V_j contains only the trivial 0 product, and the multiplications of the form $a_{ij}b_{lk}$ not put in any V_j are also all 0. The products in $V_{j'}$ may be changed as well by this "zeroing out," but $|V_{j'}^\circ| \leq |V_{j'}|$.

Since some linear combination of all the products in all the V_j° – all but one of which contain no nonzero products – must yield $a_{ij'}b_{j'k}$, $a_{ij'}b_{j'k}$ is in the span of $V_{j'}^\circ$. This holds for each $1 \leq i \leq l$ and $1 \leq k \leq n$. These ln products are all linearly independent, and so $|V_{j'}| \geq |V_{j'}^\circ| \geq ln$. Thus the total number of products computed in all the V_j is at least lmn. □

This gives the desired result:

Corollary 2 *If $\omega(G) < 3$, then for every base graph G_1 in G at least one of the subcomputations of G_1 (one level down) is disjoint from G_1.*

Proof. The above lemma shows the existence of an input-disjoint subcomputation of G_1. It is clear that the outputs of this subcomputation are then disjoint from G_1 as well. □

In the following chapter we will require a variant of Theorem 14 along the following lines: every matrix multiplication involving no more than lmn multiplications (so no worse than classical matrix multiplication) either has a disjoint submultiplication or else essentially performs classical matrix multiplication, in the following sense:

Definition 8 *Call a matrix multiplication algorithm for multiplying $l \times m$ matrix A by $m \times n$ matrix B to get $l \times n$ matrix C* classical-like *if it satisfies the following conditions:*

- *It computes exactly lmn elementary multiplications*
- *Each elementary multiplication can be associated to a unique triple (i, j, k) such that a_{ij} and b_{jk} appear in the linear combinations representing the inputs to the multiplication and the result of the multiplication occurs in the linear combination of products yielding c_{ik}.*

Theorem 15 *Every matrix multiplication algorithm computing $C = AB$ (with dimensions l, m, and n) that uses at most lmn elementary multiplications is either classical-like or else involves an elementary multiplication both of whose linear combination inputs are nontrivial (involve multiple a_{ij}s or b_{jk}s).*

Proof. Suppose no elementary multiplication computes the product of two nontrivial linear combinations. By Theorem 14 this algorithm uses exactly lmn elementary multiplications. With notation as in the proof of Theorem 14, recall that the set V_j° of elementary multiplications spans the set of all products of the form $a_{ij'}b_{j'k}$ for fixed j' (after zeroing out all a_{ij}s and b_{jk}s with different j values). Thus $|V_j| = ln$. We now apply Hall's Matching theorem, matching every product in V_j to an appropriate triple (i, j, k). Consider a product to be associated to the triple (i, j, k) if a_{ij} and b_{jk} appear in the input linear combinations and c_{ik} depends on the product.

Let X be a subset of triples of the form (i, j', k) (with $1 \le i \le l$ and $1 \le k \le n$) and Y be the set of products in V_j compatible with some triple in X. If $|X| < |Y|$, then for some c_{ik}, c_{ik} is dependent on no product involving both $a_{ij'}$ and $b_{j'k}$, an impossibility (since $a_{ij'}b_{j'k}$ appears as a term in the value of c_{ik}). Thus there exists a matching from products in V_j to appropriate triples (i, j', k), and so there exists a matching from all the products computed in this matrix multiplication algorithm to triples (i, j, k). □

Note that if the matrix multiplication algorithm uses more than lmn elementary multiplications, the same proof still holds, except the matching constructed will not use every elementary multiplication. The matching will pair exactly lmn of the $> lmn$ elementary multiplications to all the triples (i, j, k). But this implies that, relative to this matrix multiplication algorithm, classical matrix multiplication performs fewer arithmetic operations while requiring no more I/Os (even when used as a base graph in a recursive construction). [9] Thus any matrix multiplication algorithm with $\omega(G) > 3$ that does not have a disjoint submultiplication can be replaced by one with $\omega(G) \le 3$, and so it suffices to consider matrix multiplication algorithms all of whose base graphs have this property.

We have now shown a tight (for square matrices) I/O bound for submultiplications of small size and shown that there are sufficiently many disjoint submultiplications at each recursive level. The remainder of the proof of Theorem 6 for matrix multiplication algorithms with $\omega(G) < 3$ is effectively unchanged from that in Chapter 2,

[9] Each elementary multiplication in our graph can be associated to one for classical matrix multiplication which depends on a subset of the inputs our elementary multiplication depends on, and similarly for the outputs. Thus changing the given matrix multiplication algorithm to classical matrix multiplication involves simply deleting edges of the CDAG, together with unnecessary vertices, which cannot increase its I/O cost.

giving a bound with a constant factor proportional to $\frac{1}{a_{max}^3}$. Because the following chapter subsumes this proof, we omit the details.

We make one final comment: while the proof of Theorem 13 may be somewhat complicated, if only square matrix multiplication steps are allowed it simplifies significantly. In that case, the only necessary argument in the above proof beyond that used in Chapter 2 is Lemma 8, to allow for "jumping" chains and thus remove the assumption of Chapter 2 that nontrivial linear combinations are used only once. This additionally streamlines the proof of Chapter 5, which proves Theorem 6 for $\omega(G) \leq 3$ for square matrix multiplications involving only square matrix multiplication recursive steps.

Chapter 4

Internal I/O-Complexity

Recall that the *internal I/O-complexity* $\mathrm{IO}^+(G)$ of a CDAG G is defined as the minimum number of I/Os required to compute G, excluding the first inputting of each input vertex and excluding the outputting of each output vertex of G. See Definition 7 and Figure 3.1. In this chapter we prove useful properties of internal I/O-complexity, such as its superadditivity and its relation to standard I/O-complexity. This yields a very short proof of Theorem 6 based on Theorem 13. We also compare the newly-developed internal I/O-complexity technique with several of those in the literature.

The internal I/O-complexity of an algorithm is a measure of how many I/Os are truly due to the inherent complexity of the algorithm, as opposed to simply having to read all inputs once and output the answer. For example, there is a simple but relatively useless cache-independent I/O-complexity lower bound for matrix multiplication: multiplying two $n \times n$ matrices requires at least $3n^2$ I/Os, since every input/output must be read/written into/from cache at least once. Recall that, from Theorem 9 (or Theorem 13) our I/O bound for $n \times n$ matrix multiplication was proportional to n^2, the same as this trivial bound![1] Meanwhile, adding two $n \times n$ matrices has the same simple lower bound, but intuitively performs much less work. Thus the bound proved in the previous chapter, in terms of standard I/O-complexity, is no better than the most trivial possible I/O bound, and in fact no better than the analogous bound for simply adding matrices.

The reason this happens is that bounds on standard I/O-complexity give no indication of how many of those I/Os are not simply due to the necessity of reading and writing the inputs and outputs. Internal I/O-complexity is defined to ignore these inputs and outputs, giving a clearer picture of where the I/Os of an algorithm

[1] Assuming M is smaller than a constant times the matrix size, n^2.

comes from. The internal I/O-complexity lower bound for matrix multiplication or matrix addition via the trivial argument is then 0, while the results of the previous chapter show a $\Omega(n^2)$ internal I/O-complexity lower bound for matrix multiplication, much better than the trivial bound. Thus multiplying two $n \times n$ matrices requires not just $\sim n^2$ I/Os, but $\sim n^2$ I/Os that are not just due to the "forced" inputs/outputs of the input/output vertices.

In Theorem 13 we proved the following internal I/O-complexity lower bound: multiplying two $n \times n$ matrices by a divide-and-conquer algorithm requires at least $\Theta(n^2)$ internal I/Os (those counted by internal I/O-complexity) as long as the cache size is $\leq \Theta(n^2)$. In other words, even ignoring the I/Os due to inputting the input matrices and outputting the result, at least $\Theta(n^2)$ I/Os are still required to compute matrix multiplication by divide-and-conquer. The same is clearly not true of matrix addition: the internal I/O-complexity cost of matrix addition is zero. Thus internal I/O-complexity provides a better measure of the difficulty of performing an algorithm in the sense of minimizing cache I/Os.

Internal I/O-complexity is also a valuable measure due to its additive properties. Consider partitioning the vertices of a computation graph G into two sets X and Y and forming their induced subgraphs. One may want to derive a bound on the I/O-complexity of the entire graph G by combining I/O bounds for each of X and Y (in a more practical scenario, one may partition the vertices of G into many more than two subsets). Unfortunately, the sum of the I/O-complexities of X and Y may be larger than the I/O-complexity of G. For example, suppose all vertices of X come before all vertices of Y, so that the output of the subcomputation X is used as the input to the subcomputation Y. Then the outputs of X (same as the inputs of Y) should not necessarily be counted in the I/Os of G, yet are counted both in the I/Os of X and in the I/Os of Y.

However, internal I/O-complexity does not suffer from this problem. We prove below that the sum of the internal I/O-complexities of X and Y is a lower bound on the I/O-complexity of G. One can also define the internal I/O-complexity of a CDAG in terms of the red-blue pebble game as per Hong and Kung [10]: a red pebble (representing a value in cache) may be placed on each input vertex once for free, and the algorithm is complete when a pebble of either color is placed on every output vertex.

To prove many of the results in this chapter we show that a sequence of computations that compute one CDAG G can be modified in some way to yield a sequence of computations for another CDAG H. To this end we make the following definition:

Definition 9 *Let a* computation sequence *of CDAG G be a sequence specifying computations and I/Os of vertices of G whose execution computes G. Each operation*

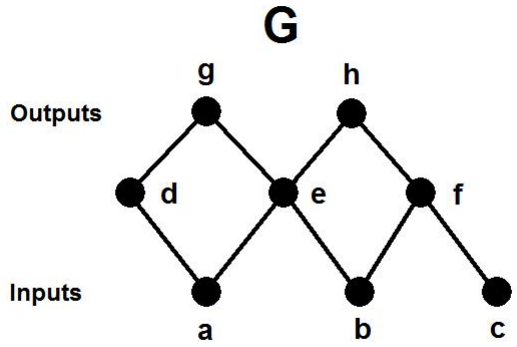
in this sequence is either a computation*, an* inputting *(moving a piece of data into cache), an* outputting *(moving a piece of data out of cache to slow memory), or a* deletion *(simply deleting a piece of data from cache).*

In the language of the red-blue pebble game devised by Hong and Kung [10], a computation sequence is a pebbling strategy. For ease of notation in the following proofs we refer to computation sequences instead of pebbling strategies, but the concepts are identical. The pebbling operations defined in [10] correspond to the operations of computation sequences as shown in Table 4.1. An example CDAG and computation sequence are shown in Figure 4.1. By assumption a vertex may be computed only once, so the operation `Compute v` may occur in a computation sequence at most once. Also, note that in this section we refer to the act of performing an I/O as "inputting" or "outputting" (instead of the more standard verbs "input" and "output") to help keep these operations separate from the input and output vertices of CDAGs.

Pebbling Operation	Computation Sequence Operation
R1 (Input)	Inputting
R2 (Output)	Outputting
R3 (Compute)	Computation
R4 (Delete)	Deletion [2]

Table 4.1: Pebbling strategies [10] correspond to computation sequences.

[2]We do not allow for the deletion of a blue (slow memory) pebble, since deleting a piece of data from the slow memory of unlimited size is never necessary and we assume no value is ever recomputed.



(a) An example CDAG G

```
1. Input a          9. Delete b
2. Input b          10. Compute h
3. Compute d        11. Output h
4. Compute e        12. Delete f
5. Output d         13. Input d
6. Input c          14. Compute g
7. Delete a         15. Output g
8. Compute f
```

(b) A computation sequence that computes G

Figure 4.1: A computation sequence is a sequence of operations – including cache I/Os – that computes a computation graph.

4.1 Mathematical Motivation

Before proving the necessary results about internal I/O-complexity – which will allow for a simple proof of Theorem 6 – we first provide some additional motivation about the properties of internal I/O-complexity relative to standard I/O-complexity. Let us now prove a not-quite-additive I/O theorem with standard I/O-complexity:

Theorem 16 *If G is a CDAG and A_1,A_2,..,A_r are vertex-disjoint subgraphs of G and $A_1, A_2, ..., A_r$ have T input and output vertices in total*[3]*, then $IO(G) \geq \sum_{i=1}^{r} IO(A_i) - T$.*

[3]Relative to each A_i, as opposed to the overall graph G.

Proof. Let S_G be a computation sequence that computes G. For $1 \leq i \leq r$, let S_i be the restriction of S_G to the vertices of A_i; that is, include into S_i only the operations of S_G referring to vertices of A_i. Note that S_i needn't be a computation sequence for A_i because input vertices of A_i are not necessarily inputted and output vertices of A_i are not necessarily outputted. For every input vertex v of S_i, replace the operation `Compute v` of S_i (if it occurs, i.e. if v was not also an input vertex of G) with the operation `Input v`. For every output vertex w of S_i add the operation `Output w` to S_i immediately after w is computed (or inputted) if w is not already outputted. Now S_i is a computation sequence for A_i and thus has at least as many I/Os as $\text{IO}(A_i)$. In modifying S_i we added several I/Os: potentially one for every input and output vertex of A_i. Thus $\text{IO}(G) \geq \text{IO}(A_1) + \text{IO}(A_2) + ... + \text{IO}(A_r) - T$. □

In this proof, every I/O performed when computing G corresponded to an I/O performed when computing one of the A_i. However, due to the necessity of inputting the input vertices and outputting the output vertices of each A_i, many additional I/Os were added. This motivates the study of internal I/O-complexity, which ignores these added I/Os. For simplicity, if H is a subgraph of CDAG G and S_G is a computation sequence that computes G, we define the *restriction of S_G to H*, denoted $S_G|_H$, to be the computation sequence for H formed as in the above proof: restrict S_G to only vertices in H, replace every computation of an input vertex of H with the inputting of the same vertex instead, and add the outputting of every output vertex of H immediately after its computation (or inputting), when not already outputted.

Define an *internal vertex* of a CDAG G to be any vertex of G that is neither an input nor an output vertex of G.

Theorem 17 *If G is a CDAG and A_1,A_2,..,A_r are vertex-disjoint subgraphs of G, then $IO^+(G) \geq \sum_{i=1}^{r} IO^+(A_i)$.*

Proof. Again let S_G be a computation sequence that computes G. The restriction $S_G|_{A_i}$ of S_G to A_i is a computation sequence for A_i; for every I/O in $S_G|_{A_i}$, there are three cases [4]:

1. The I/O is the inputting of an input vertex v of A_i. If this is the first inputting of v in $S_G|_{A_i}$ then it may have come from the operation `Compute v` in S_G and is not counted by internal I/O-complexity; otherwise, it must have come from a corresponding operation `Input v` in S_G and is counted by internal I/O-complexity.

[4] This holds even if some vertices are both input and output vertices of A_i, or even of G.

2. The I/O is the outputting of an output vertex w of A_i. In this case the I/O may or may not have come from an I/O operation in S_G and is not counted by internal I/O-complexity.

3. The I/O is from an internal vertex of A_i. The I/O must have come from an I/O operation in S_G and is counted by internal I/O-complexity.

Note that every I/O counted by internal I/O-complexity for one of the A_is corresponds to an I/O in S_G, and that each of these operations in S_G is itself counted by internal I/O-complexity (of the entire CDAG G). The number of internal I/Os performed in $S_G|_{A_i}$ is at least $\text{IO}^+(A_i)$, so $\text{IO}^+(G) \geq \text{IO}^+(A_1) + \text{IO}^+(A_2) + ... + \text{IO}^+(A_r)$. Table 4.2 summarizes this logic. □

Input type in H	Corresponds to internal I/O of S_G	Counted by internal I/O-complexity of H
Inputting input vertex for first time in $S_G\|_H$	Maybe	No
Inputting input vertex, not first time	Yes	Yes
Outputting output vertex	Maybe	No
I/O of internal vertex	Yes	Yes

Table 4.2: Every internal I/O of $S_G|_H$ – an I/O counted by the internal I/O-complexity of H – corresponds to an internal I/O of S_G. Every time an I/O is counted by the internal I/O-complexity of H (a "yes" in the third column), it corresponds to an internal I/O of S_G (a "yes" in the second column) and is thus counted in the internal I/O-complexity of G. From this it follows that the internal I/O-complexity of G is at least equal to the sum of those of the A_i.

From this proof, it is apparent that Theorem 17 applies even if the input vertices of one A_i are the same as the output vertices of another A_j; again, this reinforces the notion that internal I/O-complexity, as its name suggests, counts only cache I/Os occurring "internally" in the CDAG:

Corollary 3 *If G is a CDAG and A_1,A_2,...,A_r are subgraphs of G such that for $i \neq j$ every vertex of $A_i \bigcap A_j$ is either an input vertex of A_i and an output vertex of A_j, or vice-versa, then $IO^+(G) \geq \sum_{i=1}^{r} IO^+(A_i)$.*

Applying Internal I/O-Complexity

We now show how to apply Theorem 17 to quickly prove Theorem 6. First observe the following link between standard and internal I/O-complexity:

Claim 4 *If G is a CDAG then*

$$IO^+(G) = IO(G) - (\textit{number of input and output vertices of } G)$$

Proof. Any computation of G with respect to I/O-complexity corresponds to a computation of G with respect to internal I/O-complexity (and vice-versa) by simply removing the outputting of each output vertex of G and replacing the first inputting of each input vertex by its "free" computation. The former thus contains one extra I/O for each input and output vertex of G. □

Proof of Theorem 6 (and thus of our main theorem). By Corollary 2, a constant fraction of the vertices of subcomputations in I are in vertex-disjoint subcomputations of I. By Theorem 13, the internal I/O-complexity of each such subcomputation is bounded below by $\Theta(M)$, and thus by Theorem 17 the internal I/O-complexity of G is $\Omega(|I| \cdot M)$. By Claim 4, this also implies that the standard I/O-complexity of G is $\Omega(|I| \cdot M)$.

□

We applied this theorem to a set of disjoint submultiplications; the same technique can be applied in the Strassen-like case (or even to Strassen's algorithm), but is less necessary there because of the simpler recursive structure. This proves Theorem 6 in the case that $\omega(G) < 3$.

4.2 Open Questions

While Theorem 17 is more elegant and applicable than the equivalent theorem using standard I/O-complexity, it still is not sufficient to prove Theorem 6 in general, wherein the subcomputations whose internal I/Os we hope to add are not disjoint (and we cannot necessarily find a constant fraction of subcomputations which are). To this end we pose the following open questions and suggest that a positive result to either would have significant applications for computing I/O-complexities:

Question 1 *Let G be a CDAG and A_1,A_2,..,A_r be not-necessarily-disjoint subgraphs of G. Under what conditions is it true that $IO^+(G) \geq \Omega\left(\sum_{i=1}^{r} IO^+(A_i)\right)$?*

Question 2 *More specifically, if for every* $1 \leq i < j \leq r$ *it holds that* $A_i \bigcap A_j$ *contains only input and/or output vertices of* G*, is it true that* $IO^+(G) \geq \Omega\left(\sum_{i=1}^{r} IO^+(A_i)\right)$*?*

Because of the difficulty associated with answering these questions in general, we take a different approach in the remainder of this paper to prove Theorem 6 in the case of $\omega \geq 3$.

Chapter 5

Proof of Theorem 6 in General for Square Matrix Multiplication Steps

We now present a significantly more involved proof of Theorem 6 – our main result – for any $\omega(G)$, for example when some base graphs perform naive matrix multiplication. In this chapter we consider matrix multiplication algorithms involving only square matrix multiplication recursive steps; in Chapter 6 this final assumption will be lifted.

First we summarize the Loomis-Whitney Inequality approach to proving the $\Omega\left(\frac{n^3}{\sqrt{M}}\right)$ I/O-complexity bound for the classical matrix multiplication algorithm, as in [12]. We then present a different, path-routing based, interpretation of this proof. Using our interpretation of the proof, we show how to combine this idea with the ideas of the previous chapters to prove Theorem 6 for square matrix multiplication steps.

5.1 The Loomis-Whitney Inequality

Consider the CDAG for classical, $\Theta(n^3)$-work, matrix multiplication. The bound of Theorem 13 still holds for any submultiplication in the CDAG of sufficient size, but Theorem 17 cannot be used to add these I/O bounds due to potentially multiply copied vertices. Instead, the canonical proof [12] of the optimal $\Omega\left(\frac{n^3}{\sqrt{M}}\right)$ I/O-complexity lower bound for classical matrix multiplication applies a geometrical result called the Loomis-Whitney Inequality:

Theorem 18 (Loomis-Whitney Inequality (simple form)) *Let S be a subset of $\{(x, y, z) \mid x, y, z \in \mathbb{Z}$ and $1 \leq x \leq l,\ 1 \leq y \leq m,\ 1 \leq z \leq n\}$ and S_x, S_y, and S_z be the projections of S onto coordinates 1 and 2, onto coordinates 1 and 3, and onto coordinates 2 and 3 respectively. Then*

$$|S| \leq \sqrt{|S_x| \cdot |S_y| \cdot |S_z|}$$

Geometrically, if S represents a set of lattice points in an $l \times m \times n$ box, then this theorem upper-bounds the size of S by the sizes of the projections of S onto 3 orthogonal faces of the box. For example, if S is a $n \times n \times n$ sub-cube, then $|S| = n^3$ and $|S_x| = |S_y| = |S_z| = n^2$, resulting in an equality. In [12] this theorem is used to provide a tight lower bound on the I/O-complexity of classical matrix multiplication and further generalized in [6] to yield bounds for other problems in numerical linear algebra. However, this technique relies on the lack of distributivity in classical matrix multiplication, and thus does not apply to general matrix multiplication steps.

For comparison purposes we present the I/O-complexity bound for classical matrix multiplication derived from the Loomis-Whitney Inequality together with its proof (stated for square matrices, for simplicity):

Theorem 19 *If G is the CDAG for classical matrix multiplication of $n \times n$ matrices, then*

$$IO(G) \geq \Omega\left(\frac{n^3}{\sqrt{M}}\right)$$

as long as $M \leq O(n^2)$.

Proof. For every $1 \leq i, j, k \leq n$ the product $a_{ij}b_{jk}$ is computed by G at a multiplication vertex. Divide the computation of G into segments such that each segment (except perhaps the last) contains $\lceil M^{3/2} \rceil$ such product vertices. Let S be the set of products in one such complete segment. Let S_A be the number of elements of A used in the products in S, S_B the number of elements of B used, and S_C the number of entries of C the computed products are required for. By Theorem 18,

$$|S| \leq \sqrt{|S_A| \cdot |S_B| \cdot |S_C|}$$

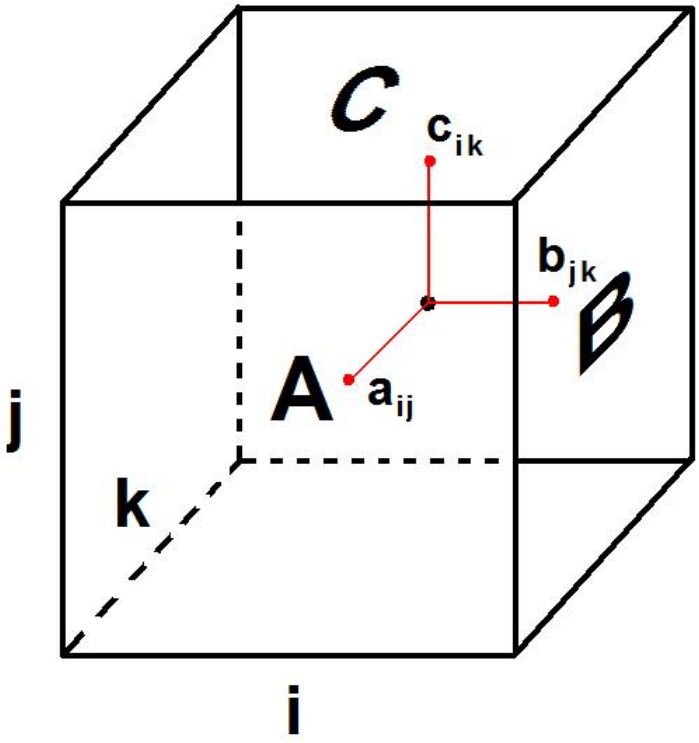
The number of I/Os performed during this segment is at least $|S_A|+|S_B|+|S_C|-2M$ (the $-2M$, as usual, because up to M required inputs may already be in cache, and similarly for outputs). It is easy to see that $|S_A| + |S_B| + |S_C|$ takes its minimum value subject to $|S_A| \cdot |S_B| \cdot |S_C| \geq |S|^2 \geq M^3$ when $|S_A| = |S_B| = |S_C| = |S|^{2/3} \geq M$,

resulting in at least $M + M + M - 2M = M$ I/Os due to this segment. The number of complete segments is $\Theta\left(\frac{n^3}{M^{3/2}}\right)$, giving an I/O bound of

$$IO(G) \geq \Omega\left(\frac{n^3}{M^{3/2}} \cdot M\right)$$

This proves the lower bound. See Figure 5.1. □

Figure 5.1: Each elementary multiplication performed within classical matrix multiplication can be associated to a unique lattice point in a three-dimensional box, as shown. The point representing the multiplication $a_{ij} \cdot b_{jk}$ lies at the intersection defined by the elements a_{ij}, b_{jk}, and c_{ik} on the three labeled orthogonal faces of the box. In other words, the elementary multiplications of classical matrix multiplication can be embedded in a box.



5.2 Motivation Behind the Proof

Following our paradigm of path routing, the above proof can be thought of in a different way: construct an efficient routing of paths within the CDAG for classical matrix multiplication from every multiplication vertex to the A inputs, another efficient routing from the multiplications to the B inputs, and a third efficient routing from the multiplications to the C outputs. These routings – which for classical matrix multiplication are quite trivial, simply following the chains of vertex copying – have the property that any $\Theta\left(M^{3/2}\right)$ multiplication vertices composing the computation segment S are collectively routed to at least $\Omega\left(M\right)$ input/output vertices.

This yields $\Omega\left(M\right)$ disjoint paths, each going from a multiplication vertex in S to an input/output vertex. Each of these disjoint paths is boundary-crossing, yielding an I/O, unless the input/output vertex ending the path happens to also lie in S. But this can happen at most $\Theta(n^2)$ times across all computation segments S, once for each input/output vertex, which is small relative to $\Theta\left(\frac{n^3}{\sqrt{M}}\right)$, and thus does not impact the overall asymptotic bound.

This argument can generalize to any matrix multiplication CDAG in which routings of a similar property exist. Of course, the only reason these routings in the classical case have this property is that the paths emanating from each multiplication vertex may be associated to a point within a 3-dimensional box, as in Figure 5.1. However, this property still holds for any *classical-like* matrix multiplication step – intuitively, a matrix multiplication step that can be simplified to classical matrix multiplication (see Definition 8 and Theorem 15). As a result of Theorem 15, every matrix multiplication base graph either has a disjoint submultiplication – to which we can apply the results of Chapters 2 and 3 – or else admits routings with the desired property.

Thus to prove Theorem 6 for $\omega(G) \leq 3$ we will combine the path-routing results of the preceding chapters with this new path-routing argument; the former argument applies when $\omega(G_1) < 3$, while the latter applies when $\omega(G_1) = 3$. Combined, this will be sufficient to prove Theorem 6. Before beginning the details of this proof, we give a slightly more in-depth overview. The following discussion in this section is oversimplified and imprecise, and included only to impart a more intuitive understanding of the proof to follow.

Consider a submultiplication step in I (defined in Theorem 6), a submultiplication of size just slightly larger than $\sqrt{M} \times \sqrt{M}$. If this submultiplication has a child subcomputation disjoint from it, we can simply add its child's internal I/Os to those of the remainder of the CDAG. [1] If not, this submultiplication must have

[1] And by the right definition of "slightly," we can guarantee that the child subcomputation is

n_0^3 child submultiplications, if this submultiplication breaks its inputs into n_0 blocks horizontally and vertically.

We may apply the same logic to each of its child subcomputations, and so on, but with one caveat: if a child subcomputation lower in the recursive tree is disjoint from its parent, it's possible that all/most of its inputs/outputs will exist in a computation segment S simultaneously. Theorem 13 (or Theorem 9, which was simpler to prove and more than sufficient for square matrix multiplication steps) required that at most a constant fraction of the inputs/outputs be in S (for square matrices; for rectangular matrices this becomes more challenging).

If most of the inputs/outputs of this disjoint submultiplication are *not* in S, Theorem 10 yields an additive I/O bound. If most of the inputs/outputs *are* in S, we will argue that there are many triples of the form (i, j, k) for which a_{ij}, b_{jk}, and c_{ik} (for this submultiplication $C = AB$) are all in S. In other words, if most of the inputs/outputs of this submultiplication are in S, then this submultiplication "looks like" classical matrix multiplication! This holds regardless of what recursive steps – classical-like or not – are actually used within this submultiplication.

We can then use the Loomis-Whitney routing idea presented above to find enough disjoint paths from these "terminal" disjoint submultiplications (or elementary multiplications, if every step above them is classical-like) to the layer of the CDAG our analysis began at, the layer containing the submultiplication steps in I. If a submultiplication X in I involves only classical-like subcomputations all the way down the recursive tree, then it must have $\Theta\left(M^{3/2}\right)$ elementary multiplications, and so we would expect it to contribute $\Theta\left(\frac{M^{3/2}}{\sqrt{M}}\right) = \Theta(M)$ I/Os by the Loomis-Whitney Inequality argument described above. That is, there exist efficient routings of paths from these elementary multiplications up to the input/output vertices of X. This is the desired number of I/Os per subcomputation in I, and so adding this up over all submultiplications in I would give the correct bound for Theorem 6. Of course, if one of the subcomputations in the recursive tree beneath X is more efficient than classical matrix multiplication it needn't have this many elementary multiplications, but by the above argument we can still find enough disjoint paths as if it did.

This isn't quite enough, because the multiplications at this initial level (the level on which the subcomputations of I lie) may still not be disjoint. We will continue this routing of paths outward in the CDAG, either ending with an input/output to the overall multiplication algorithm or with a non-duplicated vertex. The paths routed through the outputs (the decoding piece of the overall CDAG) immediately hit non-duplicated vertices, since vertex copying cannot happen within the decoding graph. The paths routed through the inputs may pass through copied vertices, but because

large enough to apply Theorem 13 or 9 to.

only ab products can be computed using a inputs from A and b inputs from B, each path can be associated with a unique point not in a 3-dimensional box, but within a 2-dimensional square. Each point represents the paths emanating from vertices beneath one submultiplication in I; geometrically, this corresponds to associating to every point in this square the 3-dimensional box representing the (i, j, k) path-routing triples for one submultiplication in I. To this 3-dimensional geometric structure we can apply the Loomis-Whitney Inequality (Theorem 18); see Figure 5.4.

Thus we must add up the I/Os from three different sources:

- Disjoint child multiplications of multiplications in I (near the top level of our analysis).
- Lower disjoint child submultiplications without too many vertices in S.
- Disjoint child submultiplications with most of their vertices in S, via efficient (embeddable in a box) routings to the inputs/outputs of the multiplications in I, and beyond.

See Figure 5.5. Finally, just as in the proof of Theorem 19, we must subtract the number of input/output vertices terminating the efficient routing we constructed. This is equal to the number of vertices among all the submultiplications in I. Unfortunately, this is too large; it is on the order of the I/O bound we are trying to prove. To fix this final problem, we will begin our analysis a constant number of recursive levels higher, using the submultiplications a few recursive steps prior to those in I. This decreases the number of I/Os to be subtracted from the final count. Because the classical-like routing argument discussed above need only apply to submultiplications of size less than $\Theta\left(\sqrt{M} \times \sqrt{M}\right)$, making this adjustment will not increase the number of submultiplications to be analyzed via this classical-like routing argument, and so the number of I/Os subtracted will become a constant fraction of the total number of I/Os counted, proving Theorem 6. We now present this proof in full detail for square matrix multiplication steps:

5.3 Proof of Theorem 6 for Square Matrix Multiplication Algorithms

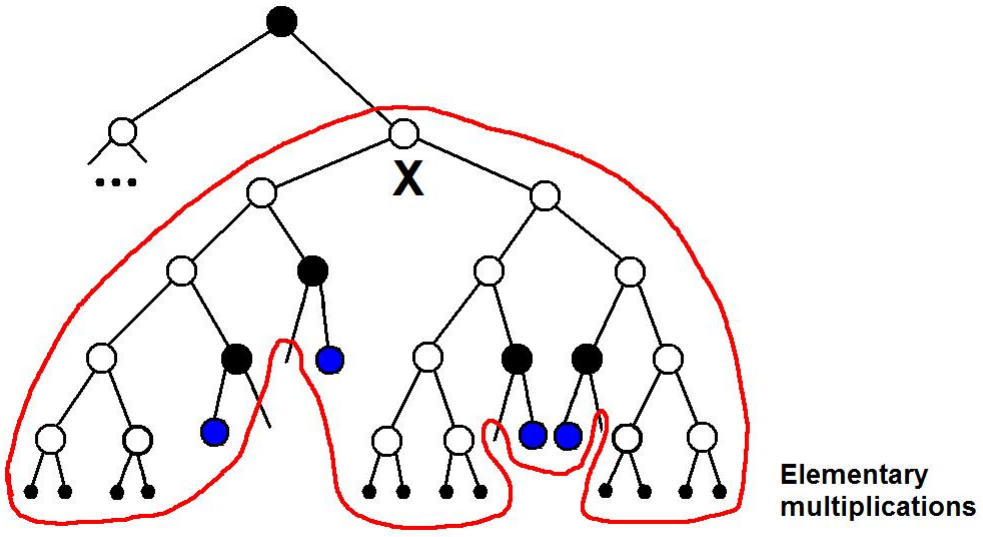
Let T be the set of subcomputations in G whose dimension is $\geq r\sqrt{M}$ and none of whose child subcomputations have this property. The value of r will be chosen later (in response to the multiplicative constants hidden in the asymptotic notation used in this section). It follows by the definition of a_{max} that every subcomputation in T

has smallest dimension $\leq ra_{max}\sqrt{M}$. Define the *cutoff layer* of G to consist of the input and output vertices of the subcomputations in T.

Definition 10 *Define a* classical tree rooted at subcomputation X *to be a tree T of subcomputations in the recursion tree representing the computation of G (see Figure 5.2) formed as follows: Add X to the tree. If X has a child subcomputation Y disjoint from it, add Y to the tree. Otherwise, for each child subcomputation Y of X, recursively add the classical tree rooted at subcomputation Y to the tree.*

Thus by Theorem 15, every multiplication step in a classical tree in G is classical-like except perhaps for the recursive steps forming the leaves and those just above the leaves. Intuitively, a classical tree is formed by following down the recursive tree subcomputation by subcomputation until a non-classical-like subcomputation is found (shown in black in Figure 5.2), and terminating on one of the guaranteed disjoint children of each non-classical-like subcomputations found in this way (shown in blue in Figure 5.2).

Figure 5.2: A classical tree rooted at subcomputation X, shown in red. Large white vertices represent classical-like subcomputations, large black vertices represent non-classical-like subcomputations, blue vertices represent (non-elementary) subcomputations disjoint from their parents, and small black vertices represent elementary multiplications. For simplicity only two subcomputations are shown per vertex.



In the proof of Theorem 13 we counted the number of vertices in the input/output of an individual submultiplication. Thus in effect, the proof of Theorem 6 relied on splitting the sequence of vertex computations into segments containing a sufficient number of vertices on the cutoff layer of G. In this section, we will instead count the vertices forming the leaves of the classical trees rooted at the subcomputations in T. To this end let L be the set of leaf computations among all these classical trees, and define the *power* of each such leaf computation to be m^3 if the leaf computation computes the product of $m \times m$ matrices (recall that in this section we are considering only square matrix multiplication recursive steps):

Definition 11 *If X is a square matrix multiplication algorithm of dimension m, define the* power *of X to be m^3.*

Definition 12 *If $L' \subseteq L$, define the* total power $p(L')$ *to be the sum of the powers of all submultiplications in L'.*

Theorem 20 *The total power $p(L)$ of subcomputations in L within one classical tree rooted at subcomputation X is at least $\frac{m^3}{a_{max}^3}$, if X multiplies $m \times m$ matrices.*

Proof. This result follows by induction. If X is an elementary multiplication it clearly holds. If X is disjoint from its parent, it contributes m^3 total power. If not, suppose X involves dividing its input matrices into k blocks vertically and horizontally. Then by Theorem 15 either it has k^3 submultiplications or else it has a disjoint submultiplication. In the latter case, by the inductive hypothesis the disjoint submultiplication has power $\left(\frac{m}{k}\right)^3 \geq \frac{m^3}{a_{max}^3}$. In the former case, by the induction hypothesis each of the k^3 submultiplications themselves have at least $\frac{1}{a_{max}^3} \cdot \left(\frac{m}{k}\right)^3$ power, for a total of at least $k^3 \frac{1}{a_{max}^3} \cdot \left(\frac{m}{k}\right)^3 = \frac{m^3}{a_{max}^3}$ power. Thus by induction X has at least $\frac{m^3}{a_{max}^3}$ power. □

This theorem justifies our use of power in our counting argument. We will end up proving a bound on the total internal I/O of G proportional to the total power of L divided by n. These I/Os come from three different sources, but each source of I/Os yields I/Os proportional to the power of L due to vertices in the computation segment S in subcomputations of that type.

Definition 13 *If X is a square submultiplication in L of dimension m, define the* active power $p_S(X)$ *of X relative to computation segment S to be the product of m with the number of vertices of the A input matrix of X that lie in S. If $L' \subseteq L$, define the* total active power $p_S(L')$ *to be the sum of the active powers of all submultiplications in L'.*

As before, split the sequence of computations of G into segments; this time, split into segments such that each segment contains $\Omega\left(M^{3/2}\right)$ total power. Each segment gets at most $2M$ "free" I/Os; for simplicity, we use the word "I/O" in the remainder of this proof to denote any vertex that need be input or output during the computation of S, including those already in cache before the beginning of S or in cache after the end. Subtracting $2M$ from the number of I/Os gives a lower bound on the number of "true" I/Os due to segment S. For a fixed full segment S, we now divide the submultiplications in L into three disjoint sets with respect to S:

- Let L_a consist of all submultiplications in L multiplying matrices of dimension $\geq \sqrt{72a_{max}M}$.
- Let L_b consist of all submultiplications in L multiplying matrices of dimension $< \sqrt{72a_{max}M}$ with the property that less than 80% of their A inputs, less than 80% of their B inputs, or less than 80% of their C ouputs are in S.
- Let L_c consist of all submultiplications in L multiplying matrices of dimension $< \sqrt{72a_{max}M}$ such that at least 80% of their A inputs, at least 80% of their B inputs, and at least 80% of their C outputs are in S. Include in L_c all elementary multiplication vertices that lie in L which are computed in S.

In other words, L_a consists of all submultiplications that are leaves of the classical trees – and thus are disjoint from their parents – whose I/Os can be analyzed by Theorem 13. [2] Meanwhile L_b contains the submultiplications with too few inputs/outputs to analyze via Theorem 13 (they are too small), but which nevertheless have enough inputs/outputs not in S for similar path-routing logic to apply, captured by Theorem 10. Finally, L_c contains those submultiplications which cannot be analyzed by either Theorem 13 or Theorem 10, and for good reason: most/all of their inputs/outputs *are* in S, preventing the usual path-routing arguments. This final set requires the additional Loomis-Whitney path-routing ideas mentioned above, which will form the majority of the remainder of this proof.

The following theorem shows that regardless of how much active power due to S lies in subcomputations of each of these three types, a proportional number of I/Os always result:

[2] In this section, because of the assumption that every recursive step performs a square matrix multiplication, we could just as well apply Theorem 9 instead of Theorem 13. However, when we generalize the results of this section to non-square matrix multiplication recursive steps it will be necessary to use Theorem 13.

Theorem 21 *The total internal I/O of L_a due to S is at least $\Omega\left(\frac{p(L_a)}{r\sqrt{M}}\right)$. The total internal I/O of L_b due to S is at least $\Omega\left(\frac{p(L_b)}{\sqrt{M}}\right)$. For L_c the same result holds, except for a total of t I/Os subtracted amongst all computation segments S, where t is at most half of the bound found in Theorem 6, assuming that r is a sufficiently large constant.*

Proof of Theorem 21 for L_a. Let X be a submultiplication in L_a. Then the dimension m of the matrices involved in X is bounded by $\sqrt{72a_{max}M} \leq m \leq ra_{max}\sqrt{M}$. The former inequality implies that Theorem 9 applies, while the latter implies that the active power of X is at most $i \cdot ra_{max}\sqrt{M}$, where i is the number of vertices of the A input matrix to X that lie in S. By Theorem 9 (or Theorem 13), X has internal I/O-complexity of at least $\Omega(i)$. Thus the internal I/O count contributed by X is at least $\Omega\left(\frac{p_S(X)}{r\sqrt{M}}\right)$. This holds for every submultiplication X in L_a, and because internal I/Os and active powers are additive thus holds for L_a as well. □

Proof of Theorem 21 for L_b. Analogous to the above proof, but using Theorem 10 [3]. Let X be a submultiplication in L_b of dimension m. By the definition of L_b Theorem 10 applies, yielding an I/O bound of $\Omega(i)$, where i is the number of vertices of the A input matrix to X that lie in S [4]. Since $m \leq ra_{max}\sqrt{M}$, X has active power of at most $\Omega\left(i \cdot ra_{max}\sqrt{M}\right)$, and so the internal I/O count contributed by X is again at least $\geq \Omega\left(\frac{p_S(X)}{\sqrt{M}}\right)$ (and the smaller of a matrix multiplication X is, the more this is an underestimate). □

Proof of Theorem 21 for L_c. This subproof is significantly more challenging and requires the use of the Loomis-Whitney path-routing ideas mentioned above. Let X be a submultiplication of dimension m in L_c that is not an elementary multiplication. By the definition of L_c, Theorem 10 does not apply. However, the following claim does hold:

Claim 5 *Let A, B, and C be the inputs and output of L_c. Then there exist at least $0.4m^3$ triples of the form (i, j, k) for which $a_{ij} \in S$, $b_{jk} \in S$, and $c_{ik} \in S$.*

[3] Unfortunately, this theorem does not generalize as easily to non-square matrix multiplications. See the above footnote. The proof of an analogue of this theorem will constitute the majority of the next chapter.

[4] This bound could be far too low, if there are no vertices in the A input but are vertices in B or C. It will prove simpler in the non-square case to count active power not by all three matrices per subcomputation, but just by the largest matrix.

Geometrically, picture A, B, and C as three orthogonal faces of a $m \times m \times m$ cube as in Figure 5.1, as in the standard interpretation of the Loomis-Whitney Inequality. Then this claim states that at least 40% of the m^3 unit cubes within this box align with elements of S in all three directions.

Proof. If U_1, U_2, and U_3 are subsets of finite set T, then $|U_1 \bigcap U_2 \bigcap U_3| \geq |U_1| + |U_2| + |U_3| - 2|T|$ (this follows by induction for n sets, replacing 2 with $n-1$) [5]. Let U_1 be the set of all triples (i, j, k) for which a_{ij} is in S, U_2 be the set of all triples (i, j, k) for which $b_{jk} \in S$, and U_3 be the set of all triples for which $c_{ik} \in S$. By the definition of L_c, there are at least $0.8m^2$ choices of (i, j) for which $a_{ij} \in S$, so $|U_1| \geq 0.8m^3$, and similarly for U_2 and U_3. Thus $|U_1 \bigcap U_2 \bigcap U_3| \geq 3 \cdot 0.8m^3 - 2m^3 = 0.4m^3$. Every triple $(i, j, k) \in U_1 \bigcap U_2 \bigcap U_3$ satisfies $a_{ij}, b_{jk}, c_{ik} \in S$, proving the claim. □

Bottom Part of the Routing

Let Y be a classical-like matrix multiplication. By the definition of classical-like matrix multiplication (see Definition 8), there exists an assignment from submultiplications performed by Y (one level down) to compatible triples (i, j, k), in the sense that a_{ij}, b_{jk}, and c_{ij} were used in/dependent on that submultiplication. This holds for each matrix multiplication recursive step in the classical trees constructed above (except the leaves and subcomputations just above the leaves).

Let X be a subcomputation in T of dimension m with input matrices A and B and output matrix C. Suppose first that every submultiplication beneath X is classical-like. Then by performing the aforementioned assignment at every recursive step, the following holds: Each elementary multiplication performed beneath X can be assigned to a unique triple (i, j, k) such that there exists a chain of dependencies from the elementary multiplication to a_{ij} (the input to X itself), a chain to b_{jk}, and a chain to c_{ik}.

Intuitively, this embeds all the submultiplications beneath X as unit cubes inside the $m \times m \times m$ cube with orthogonal faces given by the input and output matrices of X. This is exactly analogous to the case where every submultiplication performs classical matrix multiplication (hence the name "classical-like"). This implies that the Loomis-Whitney Inequality technique still applies: if U is a set of a^3 of these elementary multiplications, then their total projection onto the three faces is of size at least $3a^2$. There thus exist at least $3a^2$ disjoint chains going from the elementary multiplications in U to any of the matrices A, B, and C.

[5] Compare Bonferroni's Inequality for probabilities, which states that if $e_1, ..., e_n$ are events, then $P(e_1 \text{ and } e_2 \text{ and } ... \text{ and } e_n) \geq P(e_1) + P(e_2) + ... + P(e_n) - n + 1$.

The above analysis assumed that every submultiplication performed classical-like matrix multiplication, but the leaves (actually the subcomputations just above the leaves) of the classical-tree beneath X do not. Regardless, a similar construction applies: perform the aforementioned assignment to every classical-like matrix multiplication within the classical tree rooted at X. In other words, construct chains from the triples (i, j, k) for each subcomputation in this tree that is not a leaf (or just above a leaf, or an elementary multiplication; in other words, for every white vertex of Figure 5.2). Instead of forming chains from the elementary multiplications to the inputs/outputs of X, this forms chains from the leaf matrix multiplications – those in L – up to the inputs/outputs of X.

Let X have dimension n and let Y be a leaf subcomputation (of the classical tree rooted at X) in L_c of dimension m beneath X that is not an elementary multiplication. Suppose Y multiplies A' by B' to get C'. Each submultiplication above Y and beneath X admits the same assignment as above, so each triple (i, j, k), $1 \leq i, j, k \leq m$, can be assigned to a unique unit cube in the box used above with which it is compatible relative to the vertices a'_{ij}, b'_{jk}, and c'_{ik}. Thus one can embed not just the elementary multiplications of this tree in this box, but the elementary multiplications along with an appropriately large set of triples for each leaf computation.

Note that this embedding is one-to-one and fills at least $\frac{1}{a_{max}^3}$ of the entire $n \times n \times n$ box [6]; for each triple (i, j, k) with $1 \leq i, j, k \leq n$ (the dimension of X), this embedding assigns a unique point in the $n \times n \times n$ box to every elementary multiplication or triple (i', j', k') of a leaf computation:

Theorem 22 *Let X be a multiplication step $C = AB$. Then the embedding specified above has the following properties:*

- *Every elementary multiplication in L_c beneath X is mapped to a unit cube indexed by (i, j, k) for which there exists a chain from the multiplication to $a_{ij,}$, a chain to b_{jk}, and a chain to c_{ik}.*
- *Let $Y \in L_c$ be a leaf subcomputation beneath X of dimension m with inputs A' and B' and output C'. For every triple (i', j', k') with $1 \leq i', j', k' \leq m$, (i', j', k') is mapped to a unit cube (i, j, k) for which there exists a chain from $a_{i'j'}$ to a_{ij}, a chain from $b_{j'k'}$ to b_{jk}, and a chain from $c_{i'k'}$ to c_{ik}.*

[6] In a classical-tree leaf vertices are disjoint subcomputations, while subcomputations just above the leaves are thus non-classical-like; this means that the other child subcomputations of non-classical-like subcomputations are ignored, resulting in this factor of $\frac{1}{a_{max}^3}$.

The geometric picture of this routing is the same as in Figure 5.1, with the nuance that each classical-like recursive step has effectively been "simplified" to classical matrix multiplication. Before extending this routing beyond X, we make one remark: the above embedding argument can embed only triples (i', j', k'); it cannot embed triples $(a'_{i_a j_a}, b'_{j_b,k_b}, c'_{i_c,k_c})$, triples of arbitrary elements of A, B, C. An arbitrary set S may however contain elements of A, B, and C for which there are no triples (i', j', k') for which $a_{i'j'}$, $b_{j'k'}$, and $c_{i'k'}$ are in S. Recall that it is necessary for all these inputs/outputs to lie in S to find enough chains emanating from S (and, except for a small number of times, ending not in S) to yield a good I/O bound. For this reason we analyzed submultiplications with relatively few inputs/outputs of Y in S separately as L_b; as long as S contains many elements of A, many elements of B, and many elements of C (as do all subcomputations in L_c), the above claim shows that many of these (i', j', k') triples must indeed exist.

Top Part of the Routing

If the submultiplications in T (the collection of all submultiplications at the cutoff layer) were all disjoint, the above routing would be sufficient. It follows immediately from the Loomis-Whitney Inequality that if the total active power $p_S(L_c)$ is a^3, then there exist at least $0.4 \cdot 3a^2$ disjoint chains from elements of S (either elementary multiplications, or the inputs/outputs of a leaf multiplication represented in a triple (i', j', k')) up to vertices of the cutoff layer. Were each of these vertices distinct, each disjoint chain would yield an internal I/O, except when the cutoff vertex hit lay in S. This happens at most once for each vertex on the cutoff layer; the total size of the cutoff layer is smaller than the I/O bound produced by this method, and so is asymptotically negligible. This would yield the desired I/O bound.

Unfortunately, this analysis breaks down if the vertices on the cutoff layer are not distinct, because the above logic may overcount I/Os. To fix this issue, we will continue the routing constructed above; each chain will continue upwards/downwards through the overall CDAG, passing through subcomputations above the cutoff layer T, until it either hits an overall input/output to G or hits a vertex that is distinct from those on the next level.

Recall that every matrix multiplication step in G was assumed to compute distinct products. Let X be the base graph for a submultiplication for $C = AB$, I_A be a set of linear combinations of the elements of A computed by X, and I_B be similarly for B. Then it follows that the total number of multiplications computed by X using only inputs from I_A and I_B is at most $|I_A||I_B|$. In other words, submultiplications can be embedded within a 2-dimensional rectangle, as opposed to a 3-dimensional

box, if only the A and B inputs (not the C output) are to specify the embedding. From this simple observation comes the following theorem analogous to Theorem 22:

Theorem 23 *Let X be a matrix multiplication step $C = AB$. If I_A and I_B denote the sets of linear combinations of elements of A and of B computed in X, then there exists an injective embedding from the submultiplications of X into $I_A \times I_B$ with the following property: if the product Y is mapped to $a \times b$, then there exists a (nearly trivial) chain from Y to a and from Y to b.*

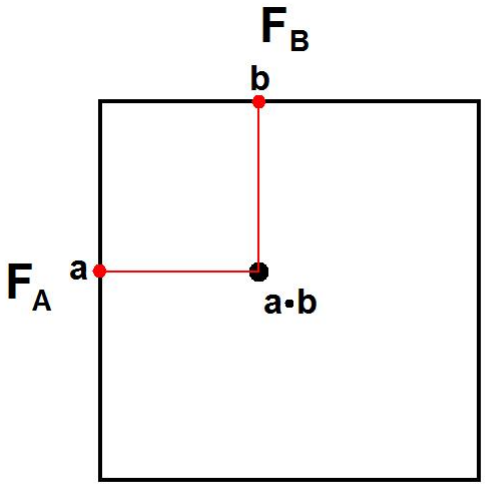
This theorem simply states that there cannot be two identical submultiplications performed beneath X, an assumption of Theorem 6 (which is clearly necessary for any meaningful bound). From this claim, we see that every submultiplication of X above the cutoff layer can be embedded not in a 3-dimensional box, but a 2-dimensional rectangle, with edges indexed by the distinct linear combinations of A and B computed in X. [7]

Suppose that for each submultiplication with inputs A' and B', every element in $I_{A'}$ and in $I_{B'}$ are duplicated elements from A' and B' (in other words, are trivial linear combinations with coefficient 1). Then via a similar recursive construction as before, each submultiplication in T can be associated to a distinct pair (a, b), where a is a block of inputs of the overall input matrix A and b is a block of inputs of the overall input matrix B to G. Alternatively, in the matrix multiplication algorithm formed by truncating the recursive construction of G at the cutoff layer (and thus interpreting each block of the original algorithm as a single element), each elementary multiplication can be associated to a distinct pair (a, b), where a is an input from A of the overall multiplication and b is an input from B of the overall multiplication. See Figure 5.3.

If the elements of $I_{A'}$ and $I_{B'}$ are not necessary duplicates of the inputs, a similar result still holds: every submultiplication in T can be associated to a distinct pair (a, b), where a is either a block of inputs of the overall input matrix A *or* a block of inputs to a matrix A' used in some subcomputation that is distinct from the inputs to its parent computation, and similarly for B. As before, the following property holds: For each submultiplication in T, there exists a routing of parallel (non-intersecting) chains from the submultiplication to the block a and a routing of chains to the block b, where (a, b) is the associated pair to the submultiplication. For each of these sets of chains, the "parallel" chains simply represent the fact that the A input to the

[7] Unlike above, the (a, b) pair associated to a submultiplication need not be internally compatible, in the sense that $a = a_{ij}$ and $b = b_{jk}$ for the same j. All that is required is that the submultiplication depend on a and b as inputs.

Figure 5.3: Every elementary multiplication performed within any matrix multiplication algorithm (that does not duplicate work) can be associated to a unique lattice point in a two-dimensional rectangle, dependent on the linear combinations of elements of the input matrices A and B that are multiplied. In this embedding the outputs this submultiplication plays a part in are not considered. The point representing the multiplication $a \cdot b$ lies at the intersection defined by the elements $a \in F_A$ and $b \in F_B$, where a and b are linear combinations of the input elements of A and B respectively.



submultiplication behaves as a block in all higher subcomputations, and similarly for B.

Combining the Routings

We have thus constructed a set of efficient routings from the elementary multiplications and triples (i, j, k) for leaf multiplications beneath every multiplication in T up to the inputs/outputs of T, and a set of efficient routings from the inputs/outputs of T up to either the inputs/outputs of the entire multiplication G or to inputs/outputs disjoint from those on the next level. Concatenate these routings. As before each elementary multiplication or triple (i, j, k) is routed to a triple (a, b, c), where c is an element of an output matrix on the cutoff layer (since the continuation of the routings constructed above do not route paths through C) and a and b are either inputs to G or are distinct values (from those on the next level). We formalize this

observation in the following theorem:

Theorem 24 *Let U consist of, for each submultiplication X in L_c of dimension m (including elementary multiplications), the set of all triples (i, j, k) with $1 \leq i, j, k \leq m$. Let F_A be the set of all distinct A input values at/above the cutoff layer, F_B be the set of all distinct B input values at/above the cutoff layer, and F_C be the set of all output values on the cutoff layer. Then there exists an injective mapping $U \to F_A \times F_B \times F_C$ [8] with the following properties:*

- *If $(i, j, k) \to (a, b, c)$, then there exists a canonical chain from a_{ij} to a, a chain from b_{jk} to b and a chain c_{ik} to c (where a_{ij}, b_{jk}, c_{ik} are the thusly indexed inputs/outputs of the submultiplication (i, j, k) belongs to).*
- *For each $a \in F_a$, either a is an input to G itself or else is disjoint from all the A inputs to the submultiplication above the one that a lies in. The same holds for each $b \in F_b$, and trivially holds for each $c \in F_c$ (replacing input with output).*
- *Suppose $(i, j, k) \to (a, b, c)$ and $(i', j', k') \to (a', b', c')$. Then the chain from (i, j, k) to a and the chain from (i', j', k') to a' are disjoint if $a \neq a'$ and $(i, j) \neq (i', j')$, and similarly for b with b' and c with c'.*

Informally, this theorem states that just like in the classical matrix multiplication case, every "elementary multiplication" (which every leaf multiplication in L_c "looks like" a cube of) can be embedded in a 3-dimensional box! The faces of this box are labeled not by the overall inputs/outputs to G, but by the distinct linear combinations of elements of A, B, and C at/above the cutoff layer (actually only the linear combinations of elements of C on the cutoff layer itself are necessary). See Figure 5.4.

From here the proof of Theorem 21 for L_c is simple: Let U_S be the set of all triples $(i, j, k) \in S$ for which $a_{ij}, b_{jk}, c_{ik} \in S$. By Claim 5, $|U_S| \geq 0.4 p_S(L_c)$. If $U_S \to V$ under the above mapping, let V_A, V_B, and V_C be the projections of V onto F_A, F_B, and F_C. Then by the Loomis-Whitney Inequality (Theorem 18),

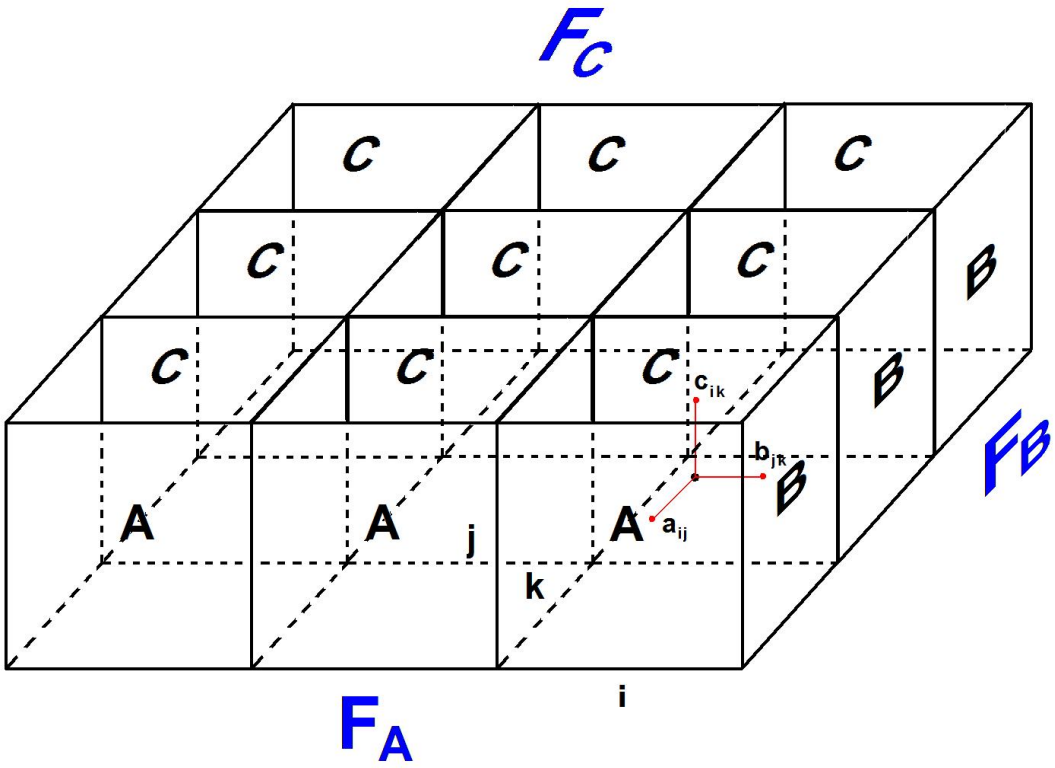
$$|U_S| \leq \sqrt{|V_A| \cdot |V_B| \cdot |V_C|},$$

from which it follows that

$$|V_A| + |V_B| + |V_C| \geq 3|U_S|^{2/3} \geq p_S(L_c)^{2/3}$$

[8] Formally, consider each element of F_A and F_B to be the vertex, not meta-vertex in G, with the specified value in the highest level submultiplication. This is necessary so that one may talk about the next-higher submultiplication.

Figure 5.4: Every elementary multiplication and triple (i, j, k) in U can be associated to a unique lattice point within a two-dimensional rectangle of three-dimensional boxes. Overall, this forms a large box whose faces are labeled by F_A, F_B, and F_C. The vertices of the C faces of the embedded boxes are all distinct (and lie in F_C), so chains emanating through the C side of one of the boxes simply stop there; the vertices of the A and B sides of the small boxes are not distinct, and so must continue onwards to a distinct element of F_A or F_B via the square embedding represented in Figure 5.3. Because this overall embedding forms a three-dimensional box – just as in the proof for classical matrix multiplication – the Loomis-Whitney inequality still applies!



Thus there are at least $p_S(L_c)^{2/3}$ disjoint chains, each from a vertex in S (of the form a_{ij}, b_{jk}, or c_{ik} for some triple $(i, j, k) \in U_S$) to a vertex in F_A, F_B, or F_C. Assuming the latter vertex is not in S, this yields $p_S(L_c)^{2/3}$ I/Os. If X is a submultiplication of dimension n in L_c, then $n \leq \sqrt{72 a_{max} M}$, so the active power

of X satisfies $p_S(X) \leq (72a_{max}M)^{3/2}$, and thus $p_S(X)^{1/3} \leq \sqrt{72a_{max}M}$. Since $p_S(L_c) = \sum_{X \in L_c} p_S(X)$, this is at least $p_S(L_c)^{2/3} \geq \frac{p_S(L_c)}{\sqrt{72a_{max}M}}$ I/Os, as desired.

This analysis counted some I/Os that should not have been included, I/Os formed by a path from an element in S to an element in F_A, F_B, or F_C that was also in S. We now upper-bound the total number of such "fake" I/Os, which are to be subtracted in total from the $\Omega\left(\frac{p_S(L_c)}{\sqrt{M}}\right)$ over all computation segments S. Each element of F_A, F_B, and F_C can lie in S in only one computation segment, so $|F_A| + |F_B| + |F_C|$ is an upper bound for this number of fake I/Os. Recall that F_A consists of all distinct A linear combinations found at/above the cutoff layer. Each submultiplication has at most a_{max}^3 child multiplications, bounding the exponent of any submultiplication away from 2. This implies that the total number of A linear combinations in F_A is proportional to the number of A linear combinations at the cutoff layer, and similarly for B. For sufficiently large choice of the constant r, $|F_A| + |F_B| + |F_C|$ is at most half the desired I/O bound of Theorem 6. [9] This concludes the proof of Theorem 21 for L_c.

□

From Theorem 21, the proof of Theorem 6 is simple:

Proof of Theorem 6 for Square Matrix Multiplication Steps. By Theorem 17, internal I/Os are additive. Thus each computation segment S of $\Omega\left(M^{3/2}\right)$ total active power contributes at least

$$\Omega\left(\frac{p_S(L_a)}{r\sqrt{M}} + \frac{p_S(L_b)}{\sqrt{M}} + \frac{p_s(L_c)}{\sqrt{M}}\right)$$

I/Os, except for the t total I/Os to be subtracted at the end of this computation (and the $2M$ free I/Os per segment) from the third term in this sum. Suppose that for at least half of the s full computation segments S the first term in the above sum comprises at least half of the total.[10] In that case, for each of these computation segments there are at least

$$\Omega\left(\frac{p_S(L_a)}{r\sqrt{M}}\right) = \Omega\left(\frac{p_S(L)}{r\sqrt{M}}\right) = \Omega\left(\frac{M^{3/2}}{r\sqrt{M}}\right) = \Omega\left(\frac{M}{r}\right)$$

[9] Because we used asymptotic notation for this section for simplicity, it can only be said that a constant r resulting in a small enough I/O subtraction exists.

[10] This split in the argument is necessary for the following reason: the larger the constant r is, the smaller the number of I/Os that must be subtracted from the total contribution of L_cs in Theorem 21 over all segments S. But the larger r is, the smaller the contribution of L_a over all segments S. Intuitively one would like to choose a large enough r for the former to be significantly less than the latter, but this need not exist. Instead, we note the the contributions from L_a do not overcount any I/Os, while the contributions from L_c do not have this $\frac{1}{r}$ dependence on r.

I/Os, except for the $2M$ values in memory before/after S. Since r is a small constant, this is bounded below by $3M$ (more precisely, the constant present in the $\Omega\left(M^{3/2}\right)$ in the definition of S can be chosen to be large enough so that this holds), resulting in at least $3M-2M=M$ true I/Os from S. In this case, no I/Os need be subtracted (only I/Os due to the L_c submultiplications may be overcounted). Thus the total true I/O count is $\Omega(sM)$. Since every computation segment has total active power $\Omega\left(M^{3/2}\right)$ and the total power $p(L)$ of all subcomputations in L is $\Omega\left(|I|M^{3/2}\right)$ by Theorem 20, it follows that there are $s \geq \Omega(|I|)$ complete segments, and so the total true I/O count is $\Omega(|I|M)$, from which the bound of Theorem 6 follows (since every submultiplication in I has dimension $\leq \Omega\left(r\sqrt{M}\right)$, and r is a constant).

Suppose instead that for at least half of the s full computation segments S the former term contributes less than half of the total. Then for each such computation segment, there are at least

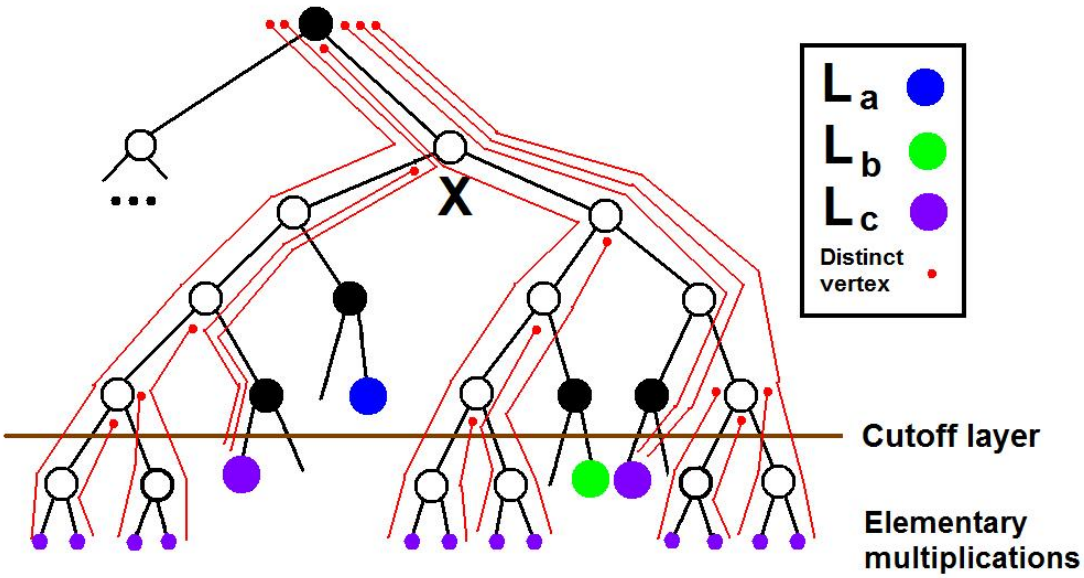
$$\Omega\left(\frac{p_S(L_b)}{\sqrt{M}}+\frac{p_s(L_c)}{\sqrt{M}}\right)=\Omega\left(\frac{p_S(L)}{\sqrt{M}}\right)=\Omega\left(\frac{M^{3/2}}{\sqrt{M}}\right)=\Omega(M)$$

I/Os, except for the $2M$ values in memory before/after S and except for a total of t I/Os total across all such segments. Thus, apart from these overcounted I/Os, each such segment contributes at least $3M-2M=M$ true I/Os. The total true I/O count is hence $\Omega(sM-t)=\Omega(|I|M-t)=\Omega(|I|M)$ by the definition of t, again yielding the bound of Theorem 6. This proves Theorem 6 for square matrix multiplication recursive steps.

□

Figure 5.5 gives a pictorial representation of the definitions of L_a, L_b, and L_c and the form of the secondary routing constructed.

Figure 5.5: The definitions of L_a, L_b, and L_c with respect to a computation segment S. Disjoint submultiplications above the cutoff layer lie in L_a (blue). Disjoint submultiplications beneath the cutoff layer may either lie in L_b (green) or in L_c (purple). Which set they lie in depends entirely on how many of their inputs are in S; if not too many inputs are in S such a submultiplication lies in S_b and contributes an appropriate number of internal I/Os by itself. If many inputs lie in S, it lies in S_c. All elementary multiplications in L computed during S also lie in L_c. We constructed a routing, shown in red, of sufficiently many chains from the inputs and outputs of L_c up to either the inputs and outputs of the entire graph (not in the example shown), or the first vertex distinct from those on the next layer. This may happen even at a subcomputation that is not disjoint from its parent (such as the children of the classical-like matrix multiplications shown as hollow circles), if some but not all of its values are distinct from its parent's.



Chapter 6

Modifications to the Proof of Theorem 6 for Rectangular Matrix Multiplication Steps

The proof of the preceding chapter assumed for simplicity that each matrix multiplication step performed square matrix multiplication. This made dimension arguments much simpler and allowed for the use of Theorem 10. In this chapter we show how to lift this assumption. The vast majority of the above proof remains unchanged, so we elaborate only on the details that change when rectangular matrix multiplication steps are allowed.

6.1 Proof of Theorem 6 for General Matrix Multiplication Algorithms

Recall that the "power" of a square matrix multiplication of dimension n was previously defined to be n^3, while the "active power" was defined to be the number of input vertices (of the A input) in S times n. Intuitively, this is a way of counting matrix multiplications by their number of A inputs: every (at most) $\sqrt{M}$ count results in one new internal I/O. From the proof of Theorem 13, we see that for a rectangular matrix multiplication of dimensions l, m, and n, we get a total internal I/O of $\max(lm, mn, ln)$ as long as not too many of the vertices of the largest matrix are in memory at once. This motivates the following more general definitions:

Definition 14 *If X is a rectangular matrix multiplication algorithm of dimensions l, m, and n, define the* power *of X to be lmn.*

Definition 15 *If X is a rectangular submultiplication in L of dimension m, define the* active power $p_S(X)$ *of X relative to computation segment S to be the product of* $\min(l, m, n)$ *with the number of vertices of the largest input/output matrix of X that lie in S.*

Let T consist of all submultiplications in G whose *smallest* dimension (so $\min(l, m, n)$) is $\geq r\sqrt{M}$ and none of whose child subcomputations have this same property. Let L be as before. From these definitions the following generalized result is clear:

Theorem 25 *The total power $p(L)$ of subcomputations in L within one classical tree rooted at subcomputation X is at least $\frac{lmn}{a_{max}^3}$, if X multiplies matrices of dimensions l, m, and n.*

Let X be a square submultiplication of dimension n. In the previous chapter, we considered two cases:

- Not too many of the input/output vertices of A lie in S. In this case, Theorem 9 gave a good internal I/O bound.
- Most of the input and output vertices of A lie in S. In this case, Theorem 10 showed us that X "looks like" classical matrix multiplication in the sense that it has many (i, j, k) triples compatible with S.

For a rectangular matrix multiplication step X we require an analogous argument. Previously the definition of "too many" was simple: if a constant fraction of the number of inputs and/or outputs lay in S apply the second argument, while if a constant fraction did not lie in S, apply the first argument. For rectangular matrix multiplication algorithms, if the matrix multiplication is sufficiently large, Theorem 13 applies. If not, we will show that either a sufficiently high number of I/Os come from a path routing within X, or else there are sufficiently many triples (i, j, k) compatible with S in X:

Theorem 26 *Let X be a submultiplication step of dimensions $l \geq m \geq n$ with i elements of the size lm matrix*[1] *in S. Then at least one of the following holds:*

1. *There exist at least $\Omega(i)$ internal I/Os within X due to S.*
2. *There exist at least $\Omega(i \cdot n)$ triples of the form (i, j, k) such that $a_{ij}, b_{jk}, c_{ik} \in U_0$, and U_0 satisfies the following property: for every vertex $v \in U_0$, either $v \in S$ or else there exist $\Omega(n)$ chains going from v to a vertex in X that lies in S, and*

[1] If there are multiple matrices of size lm, choose one arbitrarily.

no vertex in X is hit more than $\Omega(n)$ times by these chains across all vertices in U_0.

The second case of this theorem is more complicated than in the square matrix multiplication case; we defer the proof of this theorem until the following section. Intuitively, this second case states that the vertices in U, which are those lying on the "surface" of the box above a triple (i, j, k) (in any of the three dimensions) either lie in S themselves or else can be routed in an almost one-to-one way to vertices lying in S within the subcomputation X. This routing is not actually one-to-one, but yields the same I/O count as if it were. More precisely, these m vertices not in S have a $\Omega(n)$-routing of $\Omega(m \cdot n)$ chains to vertices in X that do lie in S. From this theorem, the analogous definitions of L_a, L_b, and L_c are clear:

- Let L_a consist of all submultiplications in L multiplying matrices whose smallest dimension is $\geq \Theta\left(\sqrt{M}\right)$.
- Let L_b consist of all submultiplications in L satisfying condition (1) of Theorem 26.
- Let L_c consist of all submultiplications in L satisfying condition (2) of Theorem 26, together with all elementary multiplication vertices that lie in L which are computed in S.

With these definitions, the exact same result holds as in the previous chapter:

Theorem 27 *The total internal I/O of L_a due to S is at least $\Omega\left(\frac{p(L_a)}{r\sqrt{M}}\right)$. The total internal I/O of L_b due to S is at least $\Omega\left(\frac{p(L_b)}{\sqrt{M}}\right)$. For L_c the same result holds, except for a total of t I/Os subtracted amongst all computation segments S, where t is at most half of the bound found in Theorem 6, assuming that r is a sufficiently large constant.*

Proof of Theorem 27 for L_a. Let X be a submultiplication in L_a of dimensions $l \geq m \geq n$. Then $\Theta\left(\sqrt{M}\right) \leq n \leq \Theta\left(r\sqrt{M}\right)$. From the first inequality 9 applies; from the second inequality, the active power of X is at most $i \cdot \Omega\left(r\sqrt{M}\right)$, where i is the number of vertices of the size lm input matrix to X that lie in S. By Theorem 13, X has at least $\Omega(i)$ internal I/Os. Thus the internal I/O count contributed by X is at least $\geq \Omega\left(\frac{p_S(X)}{r\sqrt{M}}\right)$. This holds for every submultiplication X in L_a, and because internal I/Os and active powers are additive thus holds for L_a as well. $\square$

Proof of Theorem 27 for L_b. Similar to the above proof, using case (1) of Theorem 26 instead of Theorem 13. Each submultiplication X in L_b of dimensions $l \geq m \geq n$ and i vertices of the size lm matrix in S contributes $i \cdot n$ active power, while contributing $\Omega(i)$ internal I/Os. Since $n \leq \Omega\left(\sqrt{M}\right)$, it follows that the internal I/O count contributed by X is at least $\geq \Omega\left(\frac{p_S(X)}{\sqrt{M}}\right)$, and similarly for L_b. $\square$

Proof of Theorem 27 for L_c. Similar proof as in the previous chapter, applying case (2) of Theorem 26 instead of Claim 5. Theorem 24 still holds, so the same Loomis-Whitney Inequality technique yields a similar result as in the square matrix multiplication case. Given $p_S(L_c)^{2/3}$ disjoint chains from U to a vertex in $F_A \bigcup F_B \bigcup F_C$, we apply case (2) of Theorem 26. Recall that every vertex $v \in U$ lies in the set U_0 of some submultiplication X. For each such submultiplication X in L_c, every vertex $v \in U_0 \bigcap S$ from which one of these chains emanates forms a boundary-crossing path. If $|U_0 \bigcap \overline{S}| = m$, then there are $\Omega(m \cdot n)$ boundary-crossing paths within X, resulting in $\Omega\left(\frac{m \cdot n}{n}\right) = \Omega(m)$ internal I/Os in X. Adding these two sources of I/Os still yields at least $\Omega\left(p_S(L_c)^{2/3}\right) \geq \Omega\left(\frac{p_S(L_c)}{\sqrt{M}}\right)$ I/Os, as before. Note that $|F_A| + |F_B| + |F_C|$ is still (for large enough r) significantly smaller than the bound of Theorem 6. $\square$

From Theorem 27 the same proof of Theorem 6 applies, with a slight modification to take into account the differing dimensions:

Proof of Theorem 6 for Rectangular Matrix Multiplication Steps. Each computation segment S of $\Omega\left(M^{3/2}\right)$ total active power contributes at least

$$\Omega\left(\frac{p_S(L_a)}{r\sqrt{M}} + \frac{p_S(L_b)}{\sqrt{M}} + \frac{p_s(L_c)}{\sqrt{M}}\right)$$

I/Os, except for the t total I/Os to be subtracted (and the $2M$ "free" I/Os for each segment S). If for at least half of the s full computation segments S the first term in the above sum contributes at least half the total, we again have

$$\Omega\left(\frac{p_S(L_a)}{r\sqrt{M}}\right) = \Omega\left(\frac{M}{r}\right)$$

I/Os, except for the $2M$ values in memory before/after S. This again yields at least M true I/Os per segment, for the right choice of the constant in the definition of the "size" of S by its active power. The total power $p(L)$ of all subcomputations in L is now

$$\Omega\left(\sum_{X \in L} \frac{l_X m_X n_X}{a_{max}^3}\right) = \Omega\left(\sum_{X \in I} l_X m_X n_X\right) \geq \Omega\left(\sum_{X \in I} \max(l_X m_X, m_X n_X, l_X n_X)\sqrt{M}\right)$$

from Theorem 25 and the definition of L in terms of I (and r being a small constant). Thus there are at least

$$s \geq \Omega\left(\sum_{X \in I} \text{max\_size}(X) \frac{1}{M}\right)$$

complete segments, resulting in a total true I/O count of

$$\Omega\left(\sum_{X \in I} \text{max\_size}(X)\right),$$

as desired.

If fewer than half of the s full computation segments have this property, then for at least half of the s computation segments the second and third terms contribute at least half of the total. For each such computation segment, there are at least

$$\Omega\left(\frac{p_S(L_b)}{\sqrt{M}} + \frac{p_s(L_c)}{\sqrt{M}}\right) = \Omega\left(M\right)$$

I/Os, except for the $2M$ "free" I/Os per segment and the overcounted t I/Os across all segments. This yields M true I/Os per segment, from which the same argument yields a total true I/O count of

$$\Omega\left(\sum_{X \in I} \text{max\_size}(X)\right).$$

This proves Theorem 6 for rectangular matrix multiplication recursive steps.

□

6.2 Proof of Theorem 26

Finally, we prove Theorem 26. Recall that this theorem is the analog of Theorem 10 together with Claim 5; it allows us to analyze submultiplications X of L that have many of their input vertices (of the largest matrix, the matrix that "counts" for active power) in S. Just as before, in some cases X will contribute internal I/Os due to S directly, while in other cases it will contribute many triples to the set U for use with Theorem 24. Previously which case applied depended only on the number of vertices of S within the desired input matrix; in this more general setting it will additionally depend on the placement of the vertices of S within the input matrix.

Let X be a submultiplication step multiplying $l \times m$ matrix A by $m \times n$ matrix B to get $l \times n$ matrix C and assume without loss of generality that $l \geq m \geq n$ (if not, analogous arguments apply based on the new largest face of the three-dimensional box described below). See Figure 6.1. Let S_A consist of all the vertices of A that lie in S.

We make the following claim:

Claim 6 *At least one of the following statements holds:*

- *At least $\frac{1}{4}$ of the elements of S_A lie in rows of A that contain $\leq \frac{n}{2}$ elements of S_A each.*
- *At least $\frac{1}{4}$ of the elements of S_A lie in columns of A that contain $\leq \frac{n}{2}$ elements of S_A each.*
- *At least $\frac{1}{2}$ of the elements of S_A lie simultaneously in a row of A that contains $\geq \frac{n}{2}$ elements of S_A and in a column of A that contains $\geq \frac{n}{2}$ elements of S_A.*

Proof. Suppose statements (1) and (2) both fail to hold. Let T_1 be the set of all elements of S_A lying in rows of A containing $\geq \frac{n}{2}$ elements of S_A each, and let T_2 be similarly for the columns of A. Then $|T_1|, |T_2| \geq \frac{3}{4}|S_A|$, so $|T_1 \bigcap T_2| \geq \frac{6}{4}|S_A| - |S_A| = \frac{1}{2}|S_A|$. This implies that statement (3) holds. □

In other words, either a constant fraction of S_A lies in sparsely-populated rows, or a constant fraction of S_A lies in sparsely-populated columns, or a constant fraction of $S_!$ lies at the intersections of well-populated rows with well-populated columns.

Suppose case (1) of Claim 6 holds. Let S'_A be the subset of elements of S_A having the specified property. Then by Theorem 12, there exists a n-routing R of all guaranteed dependencies between the vertices of S'_A and vertices of C on which they depend. In other words, for every vertex $a_{ij} \in S'_A$, there is a chain in this routing from a_{ij} to c_{ik} for each k. Let R' be the n^2-routing constructed as follows: for each routing from a_{ij} to c_{ik} and each routing from $a_{ij'}$ to c_{ik}, concatenate the former chain with the reverse of the latter to yield a path from a_{ij} to $a_{ij'}$. The paths of this routing route a_{ij} to every $a_{ij'}$.

For each row i of A containing $m_i \leq \frac{n}{2}$ elements of S_A, there are at least $m_i \cdot \frac{n^2}{2}$ paths in R' routing from a vertex of the form a_{ij} in S to a vertex of the form $a_{ij'} \notin S$ (n choices for the k in c_{ik} to route through, and $\geq \frac{n}{2}$ elements of $\overline{S}$ in A to end at). It thus follows that there are at least $\sum_i m_i \cdot \frac{n^2}{2} = |S'_A|\frac{n^2}{2} \geq |S_A|\frac{n^2}{8}$ paths from vertices of S_A to vertices in S that lie in X. There are thus $\Omega\left(\frac{|S_A|n^2/8}{n^2}\right) = \Omega\left(|S_A|\right) = \Omega\left(i\right)$ internal I/Os within X due to S. See Figure 6.1(b).

If case (2) of Claim 6 holds, the proof is similar, routing paths from A to B back to A, instead of through C.

Suppose now that case (3) of Claim 6 holds. Let S'_A again be the subset of elements of S_A having this property. Let U_0 be the set consisting of, for each vertex $a_{ij} \in S'_A$, the elements of A, B, and C representing the projections of the triples (i, j, k) for each $1 \leq k \leq n$ – that is, the elements a_{ij}, b_{jk}, and c_{ik}. Clearly there are at least $|S'_A| \cdot n \geq \frac{|S_A|}{2} n \geq \Omega(i \cdot n)$ triples as desired. Next, we show that the desired routing property exists between U_0 and vertices of S in X.

By construction, every vertex $a_{ij} \in U_0$ lies in S, and so need not have any paths routed from it. Let U_B consist of all vertices of the form $b_{jk} \in U_0$. For every j, let V_j be a set of $\lfloor \frac{n}{2} \rfloor$ vertices[2] in column j of A that lie in S, if column j of A contains at least this many vertices of S. By Theorem 12, there exists an n-routing of all guaranteed dependencies between vertices $b_{jk} \in U_0$ and vertices in V_j. In other words, for each $b_{jk} \in U_0$, there exist $\Omega(n)$ chains from b_{jk} to vertices in A that lie in S, and every vertex of X is hit at most n times by this routing. Restrict this routing to include only chains beginning at vertices b_{jk} that do not themselves lie in S. Construct a similar routing of chains from vertices $c_{jk} \in U_0$ to vertices of A in S and simply union these routings. The resulting routing has the properties specified by case (2) of Theorem 26. See Figure 6.1(c); the blue squares denote the vertices b_{jk} from which paths are routed to the red squares of A in S. This proves Theorem 26.

This concludes the proof of Theorem 6, and thus the proof of our main theorem in the serial case, Theorem 7. We now turn to adapting the arguments of this and the preceding chapters to handle recursive matrix multiplication algorithms run in parallel on P processors.

[2] If $n = 1$, round up.

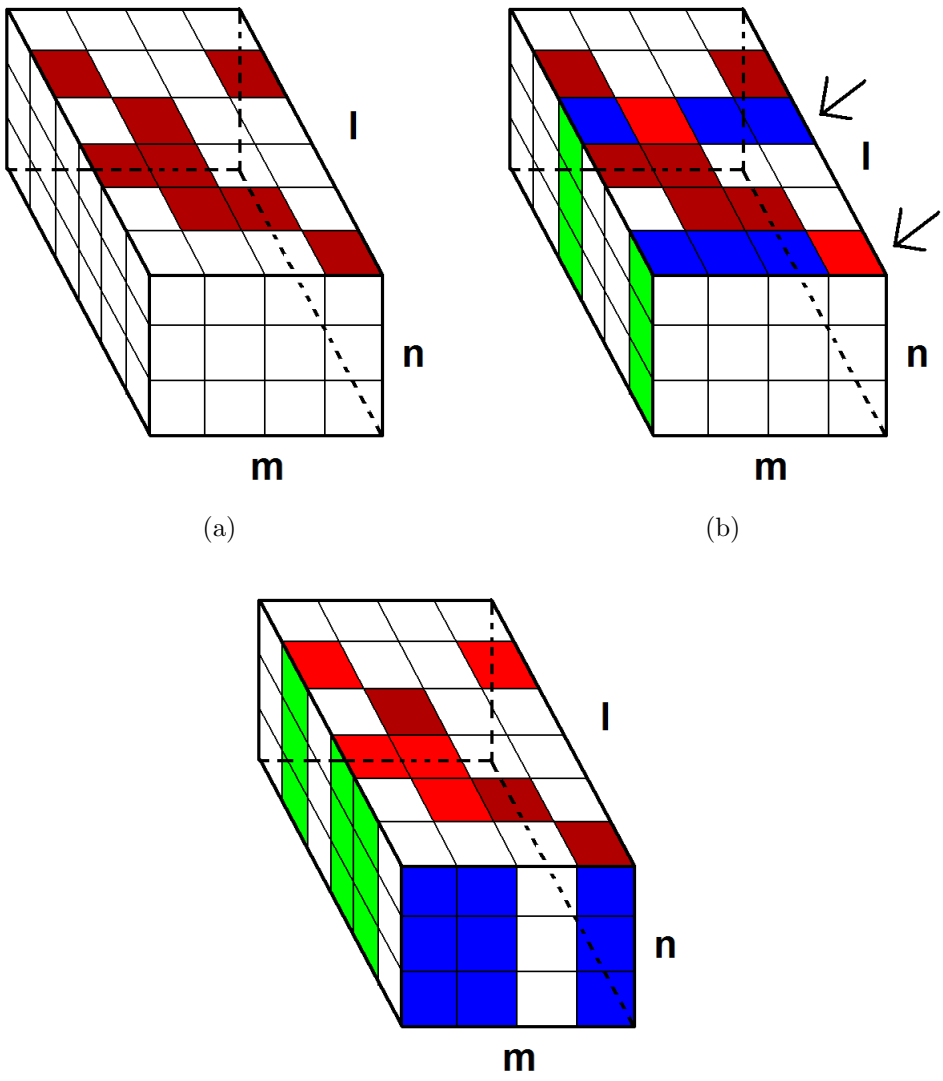


Figure 6.1: (a) The inputs and outputs of a matrix multiplication X can be thought of as composing three orthogonal faces of a cube consisting of triples (i, j, k). If the dark red squares indicate the elements S_A of A in S, then for this set S both statements (1) and (3) of Claim 6 hold. (b) Because statement (1) holds, there exists an efficient routing from the bright red squares in the indicated rows to the green squares on the side and back to the blue squares in the indicated rows. (c) Because statement (3) holds, there exists an efficient routing from the green squares to the (bright and dark) red squares, and an efficient routing from the blue squares to the (bright and dark) red squares. There are $\Omega(i \cdot n)$ triples (i, j, k) in the corresponding U_0, represented by the n unit cubes directly beneath each bright red square.

Chapter 7

Parallel Divide-And-Conquer Matrix Multiplication Algorithms

Finally, we show how to apply the logic of the preceding chapters to yield an I/O-complexity bound for divide-and-conquer matrix multiplication algorithms computed in parallel. Typically in the literature the bound for a parallel algorithm run on P processors is simply $\frac{1}{P}$ of the I/O-complexity bound in the serial case. The usual argument goes as follows: instead of letting S be a minimum-size[1] computation segment that yields M true I/Os, let S the set of computations performed by the processor performing the most work (performing the most vertex computations out of a prespecified set). Usually the number of I/Os attributable to a segment S ends up being linear in the "size" of S – not necessarily the total number of vertices of S, but the number of vertices we "count." As we saw, this linearity implies that in the serial case breaking the computation into segments of size proportional to M is sufficient. It also implies, by linearity, that the bound in the parallel case due to S representing the computation performed by the processor with the largest "size" computation is $\frac{1}{P}$ of that in the serial case.

7.1 Parallel Bound

The problem at hand is no exception to the above rule. However, due to the nuance in the preceding chapters of needing to subtract up to $|F_A| + |F_B| + |F_C|$ I/Os across all computation segments, we give a slightly more careful proof of the $\frac{1}{P}$ bound in the parallel case. In the following theorem we restate the main finding of the previous chapters that allowed us to prove Theorem 6:

[1] As small as possible while still guaranteeing an I/O bound of $3M$.

Theorem 28 *Let $S_1, S_2, ..., S_m$ be a partition of the vertices of the CDAG G, let q_{S_j} denote the total active power of S_j, and let $q = \sum_j q_{S_j}$ be the total active power of all the S_j (which depends only on G). Then at least one of the following statements holds:*

- *There exists a constant c_1 and a subset $U \subseteq \{S_1, ..., S_m\}$ such that $\sum_{S_j \in U} q_{S_j} \geq \frac{1}{4}q$ and for each $S_j \in U$ there exist at least $c_1 \frac{q_{S_j}}{\sqrt{M}}$ I/Os between S_j and $\overline{S_j}$.*[2]
- *There exists a constant c_2 and a subset $U \subseteq \{S_1, ..., S_m\}$ such that $\sum_{S_j \in U} q_{S_j} \geq \frac{1}{4}q$ and for each $S_j \in U$ there exist at least $c_2 \frac{q_{S_j}}{\sqrt{M}} - t_j$ I/Os between S_j and $\overline{S_j}$ where $\sum_{S_j \in U} t_j \leq \frac{1}{8} c_2 \frac{q}{\sqrt{M}}$.*

Proof. See the preceding chapters. Recall that the proof of Theorem 6 required casework because of the $\frac{1}{r}$ dependence in the first I/O term. This results in a different constant in the two cases; c_1 has an extra dependence on $\frac{1}{r}$ (which is still a constant). Instead of splitting into cases based on how many S_j have their first I/O term ($\Omega\left(\frac{p_S(L_a)}{r\sqrt{M}}\right)$) contributing at least half of the total I/Os, split into cases based on the sum of the active powers of segments S_j satisfying this condition versus the sum of the active powers of segments S_j not satisfying this condition. The remainder of the proof is analogous to the logic of the proof of Theorem 6 $\square$

We will use this result to prove the expected $\frac{1}{P}$ bound in the parallel case:

Theorem 29 (Main Parallel Theorem) *Let G be as in Theorem 6. If G computes Mult elementary multiplications, then the I/O-complexity of G (see Chapter 2 for the definition) computed in parallel by P processors is*

$$IO(G) \geq \Omega\left(\frac{1}{P} \cdot \frac{Mult}{\sqrt{M}^{\,\omega(G)}} \cdot M\right)$$

Proof. Let S_i be the set of vertices of G computed by processor i. Suppose case (1) of Theorem 28 holds. Let processor j be the processor indexed amongst U whose S_j has the largest total active power. Then $q_{S_j} \geq \frac{1}{P} \cdot \frac{q}{4}$, so processor j has at least

[2]That is, vertices $u \in G$ for which there exists an edge (u, v) such that u and v have different membership in S_j.

$c_1 \frac{1}{\sqrt{M}} \cdot \frac{1}{P} \cdot \frac{q}{4} = \Omega\left(\frac{1}{P} \cdot \frac{q}{\sqrt{M}}\right)$ I/Os. As before, $\frac{q}{\sqrt{M}} \geq \Omega\left(\frac{Mult}{\sqrt{M}^{\omega(G)}} \cdot M\right)$, proving Theorem 29.

Suppose instead that case (2) of Theorem 28 holds. We make the following claim:

Claim 7 *For some j indexed by U we have $c_2 \frac{q_{S_j}}{\sqrt{M}} - t_j \geq \frac{c_2}{8} \frac{q}{P\sqrt{M}}$.*

Proof. Suppose not. Then for each such j it is true that $c_2 \frac{q_{S_j}}{\sqrt{M}} < t_j + \frac{c_2}{8} \frac{q}{P\sqrt{M}}$; summing over all j represented by U yields

$$\frac{c_2}{4} \frac{q}{\sqrt{M}} < \frac{c_2}{8} \frac{q}{\sqrt{M}} + \frac{c_2}{8} \frac{q}{\sqrt{M}},$$

an impossibility. □

Pick j to be some processor represented in U satisfying the above claim. Then processor j has at least $\frac{c_2}{8} \frac{q}{P\sqrt{M}}$ I/Os, again yielding $\Omega\left(\frac{Mult}{\sqrt{M}^{\omega(G)}} \cdot M\right)$ parallel I/Os. This proves Theorem 29.

□

7.2 Tightness of Theorem 29

The parallel algorithm in [3] shows that the parallel bound of Theorem 29 is optimal for Strassen's algorithm. This algorithm generalizes naturally to any square recursive divide-and-conquer matrix multiplication algorithm, thus showing that Theorem 29 is asymptotically tight for any square matrix multiplication algorithm (See Section 6.6 of [3]). Intuitively, the CAPS algorithm in [3] is the parallel analogue of the serial algorithm described in Section 3.2. Instead of computing the submultiplications of the matrix multiplications of I in serial, they are computed in parallel, by assigning to each processor a submultiplication that just barely fits entirely in the processor's local cache.[3] For rectangular matrices this bound may not be tight, as in Section 3.2.

7.3 Conclusion

We have thus proved Theorem 6 in general, for any recursive divide-and-conquer matrix multiplication algorithm, and shown how to generalize the logic to apply

[3] And it was shown in Section 3.2 that this is sufficient for each processor to compute its piece of the computation with its limited cache size of M.

to matrix multiplication algorithms computed in parallel. In both the serial and the parallel cases the bounds we proved are the expected, significantly stronger, generalizations of those for Strassen-like algorithms (see [5]).

To prove these bounds we developed several new techniques based around the idea of path routing. First we showed how to construct a routing of paths within a small matrix multiplication graph from every input vertex to every output vertex in a way that does not hit any vertex too many times. We argued by a simple counting argument that this implies the existence of many paths crossing the boundary between a computation segment and its complement, resulting in many I/Os due to each computation segment S of "size" $\Omega(M)$.

We next showed how this path-routing idea generalizes in the case of rectangular matrix multiplication. When the matrix dimensions differ significantly, it may no longer be possible to find quite as simple of a routing; instead several routings between the vertices of S and those of $\overline{S}$ may be needed depending on the precise locations of the elements of S within the subcomputation. Each such routing, however, is as "sparse" as possible; no vertex in the CDAG is hit any more than is necessary simply by virtue of the guaranteed dependencies one wishes to route.

Next, we introduced the concept of internal I/O-complexity; this idea allows for much simpler I/O addition, and better represents the true I/O difficulty of computing an algorithm by "ignoring" the first inputting/outputting of the overall inputs and outputs of the algorithm. We proved that internal I/Os are additive for disjoint graphs, and conjectured that internal I/Os are additive under significantly less restrictive conditions.

We then showed how to handle matrix multiplication algorithms involving classical (or classical-like) recursive steps. This generalization relied on the intermixing of two ideas: adding internal I/Os due to disjoint submultiplications, and embedding non-disjoint submultiplications within a three-dimensional box for use with the Loomis-Whitney Inequality. Intuitively, this seems reasonable: if all subcomputations are disjoint, adding internal I/Os is sufficient (see Chapter 4). If all subcomputations have no disjoint children, then each subcomputation can be reduced to classical matrix multiplication (Theorem 15), and the standard proof for such an algorithm is the geometric one based on the Loomis-Whitney Inequality. This yields a second routing of paths, this time from the elementary multiplications (or triples of small disjoint submultiplications) up to the overall inputs/outputs of the algorithm (except stopping early if they hit distinct vertices). For algorithms mixing these steps, the Loomis-Whitney Inequality-based path routing we constructed is no longer sufficient by itself, but applying the concept of internal I/O-complexity to the submultiplications without too many vertices of S in them does turn out be enough.

Finally, we argued that the results of this line of reasoning also yielded a strong

bound in the parallel case. Our bounds are tight for square and near-square matrix multiplication algorithms, in the sense that one can find an implementation of a matrix multiplication algorithm achieving them (see [3]). We expect that the path-routing ideas developed in these chapters will likely have applications in proving the internal I/O-complexities of other algorithms.

Bibliography

[1] G. Ballard et al. "Communication lower bounds and optimal algorithms for numerical linear algebra". In: *Acta Numerica* 23 (May 2014), pp. 1–155. ISSN: 1474-0508. DOI: 10.1017/S0962492914000038. URL: http://journals.cambridge.org/article_S0962492914000038.

[2] Grey Ballard et al. "Brief announcement: strong scaling of matrix multiplication algorithms and memory-independent communication lower bounds". In: *Proc. 24th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. SPAA '12. Pittsburgh, Pennsylvania, USA: ACM, 2012, pp. 77–79. ISBN: 978-1-4503-1213-4. DOI: 10.1145/2312005.2312021. URL: http://doi.acm.org/10.1145/2312005.2312021.

[3] Grey Ballard et al. "Communication-Optimal Parallel Algorithm for Strassen's Matrix Multiplication". In: *Proceedings of the 24th ACM Symposium on Parallelism in Algorithms and Architectures, SPAA 2012* (2012).

[4] Grey Ballard et al. "Graph Expansion Analysis for Communication Costs of Fast Rectangular Matrix Multiplication". In: *Design and Analysis of Algorithms* 7659 (2012), pp. 13–36.

[5] Grey Ballard et al. "Graph Expansion and Communication Costs of Fast Matrix Multiplication". In: *Journal of the ACM* 59.6 (2012).

[6] Grey Ballard et al. "Minimizing Communication in Numerical Linear Algebra". In: *SIAM J. Matrix Anal. & Appl.* 32.3 (2011), pp. 866–901.

[7] G. Bilardi, A. Pietracaprina, and P. D'Alberto. "On the space and access complexity of computation dags". In: *Proceedings of the 26th International Workshop on Graph-Theoretic Concepts in Computer Science, London, UK* (2000), pp. 47–58.

[8] D. Bini et al. "$O(n^{2.7799})$ complexity for $n \times n$ approximate matrix multiplication". In: *Information processing letters* 8.5 (1979), pp. 234–235.

[9] Michael Christ et al. *Communication Lower Bounds and Optimal Algorithms for Programs That Reference Arrays - Part 1.* Tech. rep. UCB/EECS-2013-61. EECS Department, University of California, Berkeley, 2013. URL: `http://www.eecs.berkeley.edu/Pubs/TechRpts/2013/EECS-2013-61.html`.

[10] J. W. Hong and H. T. Kung. "The red-blue pebble game". In: *STOC 1981: Proceedings of the thirteenth annual ACM symposium on theory of computing* (1981), pp. 326–333.

[11] J. Hopcroft and L. Kerr. "On minimizing the number of multiplications necessary for matrix multiplication". In: *SIAM Journal on Applied Mathematics* 20.1 (1971), pp. 30–36.

[12] D. Irony, S. Toledo, and A. Tiskin. "Communication Lower Bounds for Distributed-Memory Matrix Multiplication". In: *J. Parallel Distrib. Comput.* 64.9 (2004), pp. 1017–1026.

[13] J.M. Landsberg. *Geometry and Complexity Theory.* Published online, accessed Sept. 2015. URL: `https://simons.berkeley.edu/sites/default/files/simonsclass.pdf`.

[14] L. H. Loomis and H. Whitney. "An Inequality Related to the Isoperimetric Inequality". In: *Bulletin of the American Mathematical Society* 55.10 (1949).

[15] John Savage. "Space-time tradeoffs in memory hierarchies". In: *Technical report, Brown University, Providence, RI, USA* (1994).

[16] Jacob N. Scott, Olga Holtz, and Oded Schwartz. "Matrix Multiplication I/O-Complexity by Path Routing". In: SPAA '15 (2015), pp. 35–45.

[17] Volker Strassen. "Gaussian Elimination is Not Optimal". In: *Numerische Mathematik* 13 (4 1969), pp. 354–356.

[18] Virginia Vassilevska Williams. "An overview of the recent progress on matrix multiplication". In: *ACM SIGACT Newsletter* 43.4 (), pp. 57–59.

[19] S. Winograd. "On the Number of Multiplications Required to Compute Certain Functions". In: *Proceedings of the National Academy of Science* 58.5 (1967).

[20] C.-Q. Yang and B.P. Miller. "Critical path analysis for the execution of parallel and distributed programs". In: *Proceedings of the 8th International Conference on Distributed Computing Systems* (1988), pp. 366–373.