Communication-Avoiding Parallel Recursive Algorithms for Matrix Multiplication

Benjamin Lipshitz

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2013-100
http://www.eecs.berkeley.edu/Pubs/TechRpts/2013/EECS-2013-100.html

May 17, 2013

Copyright © 2013, by the author(s).
All rights reserved.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission.

Communication-Avoiding Parallel Recursive Algorithms for Matrix Multiplication

by

Benjamin Lipshitz

A thesis submitted in partial satisfaction of the
requirements for the degree of
Master of Science

in

Computer Science

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Professor James Demmel, Chair
Professor Katherine Yelick
Professor Armando Fox

Spring 2013

Communication-Avoiding Parallel Recursive Algorithms for Matrix Multiplication

Copyright 2013
by
Benjamin Lipshitz

Contents

Contents i

List of Figures iii

List of Tables iv

1 Introduction 1
- 1.1 Communication Model . . . 2
- 1.2 Iterative Parallel Matrix Multiplication Algorithms . . . 3
- 1.3 Parallelizing Recursive Algorithms . . . 3
- 1.4 The Loomis-Whitney Inequality . . . 4

2 Strassen's Matrix Multiplication 5
- 2.1 Strassen-Winograd Algorithm . . . 6
- 2.2 Communication Lower Bounds . . . 6
- 2.3 Communication-Avoiding Parallel Strassen . . . 7
- 2.4 Analysis of Other Algorithms . . . 15
- 2.5 Performance Results . . . 20
- 2.6 Performance Model . . . 25
- 2.7 Implementation Details . . . 31
- 2.8 Numerical Stability . . . 35
- 2.9 Parallelizing Other Fast Matrix Multiplication Algorithms . . . 36

3 Classical Rectangular Matrix Multiplication 38
- 3.1 Communication Lower Bounds . . . 40
- 3.2 CARMA Algorithm . . . 44
- 3.3 Performance Results . . . 48
- 3.4 Remarks . . . 51

4 Sparse Matrix Multiplication 54
- 4.1 Preliminaries . . . 55
- 4.2 Communication Lower Bounds . . . 56
- 4.3 Algorithms . . . 59

4.4 Performance Results . . . 63

5 Beyond Matrix Multiplication **68**

5.1 Naïve n-body Interaction . . . 68
5.2 Dealing with Dependencies . . . 70
5.3 All-Pairs Shortest Paths . . . 70
5.4 Triangular Solve with Multiple Righthand Sides . . . 71
5.5 Cholesky Decomposition . . . 73

6 Discussion and Open Questions **75**

Bibliography **78**

List of Figures

1.1 The computation rectangular prism for matrix multiplication 4

2.1 Representation of BFS and DFS steps 8
2.2 An example matrix layout for CAPS 10
2.3 Bandwidth costs and strong scaling of matrix multiplication: classical vs. Strassen-based. Horizontal lines correspond to perfect strong scaling. $P_{\min}$ is the minimum number of processors required to store the input and output matrices. 18
2.4 Strong scaling results for CAPS 22
2.5 Efficiency at various numbers of Strassen steps 25
2.6 Comparison of the sequential model to the actual performance of classical and Strassen matrix multiplication 26
2.7 Time breakdown comparison between the sequential model 27
2.8 Comparison of the parallel model with the algorithms in strong scaling 29
2.9 Time breakdown comparison between the parallel model and the data 29
2.10 Predicted speedups of CAPS over 2.5D and 2D on an exascale machine 30
2.11 The memory and communication costs of all possible interleavings of BFS and DFS steps 34
2.12 Data layout for CAPS 34
2.13 Stability test: theoretical error bound versus actual error 36

3.1 Three ways to split rectangular matrix multiplication to get two subproblems . . 39
3.2 Examples of the three cases of aspect ratios: *one large dimension*, *two large dimensions*, and *three large dimensions* 41
3.3 CARMA compared to ScaLAPACK on Hopper 49
3.4 Data layout for a BFS step 53

4.1 How the cube is partitioned in 1D, 2D, and 3D algorithms 56
4.2 Graphical representation of V and ℓ^C_{ij}. 57
4.3 Two ways to split the matrix multiplication into four subproblems 62
4.4 Strong scaling of sparse matrix multiplication. 64
4.5 Time breakdown for sparse matrix multiplication 66
4.6 Possible interleavings of the recursive algorithm 67

List of Tables

2.1 Asymptotic computational and communication costs of Strassen-based and classical square matrix multiplication algorithms and corresponding lower bounds . . 16
2.2 The pattern of BFS and DFS steps and memory usage for CAPS. 23
2.3 Asymptotic bandwidth costs of classical and fast, square and rectangular matrix mul tiplication 37

3.1 Asymptotic bandwidth costs and lower bounds for rectangular matrix multiplication 44

4.1 Asymptotic expected communication costs and lower bonuds for sparse matrix multiplication 59

Chapter 1

Introduction

Matrix multiplication is one of the most fundamental algorithmic problems in numerical linear algebra, distributed computing, scientific computing, and high-performance computing. Parallelization of matrix multiplication has been extensively studied (*e.g.*, [21, 12, 24, 2, 51, 39, 36, 23, 45, 61]). It has been addressed using many theoretical approaches, algorithmic tools, and software engineering methods in order to optimize performance and obtain faster and more efficient parallel algorithms and implementations.

To design efficient parallel algorithms, it is necessary not only to load balance the computation, but also to minimize the time spent communicating between processors. The interprocessor communication costs are in many cases significantly higher than the computational costs. Moreover, hardware trends predict that more problems will become communication-bound in the future [38, 35]. Even matrix multiplication, which is widely considered to be computation-bound, becomes communication-bound when a given problem is run on sufficiently many processors.

Here we consider three cases of matrix multiplication: fast matrix multiplication algorithms, such as Strassen's, which compute the product of dense matrices using asymptotically fewer than the naïve number of scalar products and so run in $(o(n^3))$ time (Chapter 2); classical matrix multiplication, for which all of the naïve scalar products $A_{ik} \cdot B_{kj}$ must be computed (Chapter 3); and sparse matrix multiplication, where most of the entries of the input matrices are zero, and only nonzero products are computed (Chapter 4). In each case, we present a new recursive parallel algorithm. The new algorithms are communication-optimal: they asymptotically match communication lower bounds, and they communicate asymptotically less than previous algorithms. In the cases of sparse matrix multiplication and high aspect ratio rectangular matrices for classical matrix multiplication we present new, tight lower bounds. We also present benchmarking data that shows our new algorithms are faster than previous algorithms in practice. Compared to the best previous algorithm, we show speedups of up to 2.8× for Strassen's algorithm, 140× for classical matrix multiplication, and 8× for sparse matrix multiplication. In Chapter 5, we explain how to generalize our parallelization approach to other recursive algorithms, including those with more dependencies than matrix multiplication.

1.1 Communication Model

We model communication of distributed-memory parallel architectures as follows. We assume the machine has P processors, each with local memory of size M words, which are connected via a network. Processors communicate via messages, and we assume that a message of w words can be communicated in time $\alpha + \beta w$. Here α and β are machine parameters specifying the overhead time per message and the reciprocal bandwidth, respectively. The *bandwidth cost* of the algorithm is given by the word count and denoted by W, and the *latency cost* is given by the message count and denoted by S. Similarly the computational cost is given by the number of floating point operations and denoted by F. We call the time per floating point operation γ.

We count the number of words, messages and floating point operations along the *critical path* as defined in [66]. That is, two messages that are communicated between separate pairs of processors simultaneously are counted only once, as are two floating point operations performed in parallel on different processors. Note that we do not require the communication to be bulk-synchronous: some of the processors may be engaged in communication while others are performing computations. This metric is closely related to the total running time of the algorithm, which we model as

$$\alpha S + \beta W + \gamma F.$$

We assume that (1) the architecture is homogeneous (that is, γ is the same on all processors and α and β are the same between each pair of processors), (2) processors can send/receive only one message to/from one processor at a time and they cannot overlap computation with communication (this latter assumption can be dropped, affecting the running time by a factor of at most two), and (3) there is no communication resource contention among processors. That is, we assume that there is a link in the network between each pair of processors. Thus lower bounds derived in this model are valid for any network, but attainability of the lower bounds depends on the details of the network. One way to model more realistic networks is as a hierarchy with different values for α and β at each level.

Perfect Strong Scaling

We say that an algorithm exhibits *perfect strong scaling* if its running time for a fixed problem size decreases linearly with the number of processors; that is, if all three of F, W, and S decrease linearly with P. Several of the algorithms we will discuss exhibit perfect strong scaling within certain ranges. Our running time model naturally generalizes to give an energy model, and perfect strong scaling of runtime corresponds to getting linear speedup in P while using no extra energy [32].

1.2 Iterative Parallel Matrix Multiplication Algorithms

The first scalable algorithm for parallel matrix multiplication is due to Cannon in 1969 [21]. Cannon's algorithm multiplies two $n \times n$ matrices on P processors on a square grid with bandwidth cost $O\left(n^2/\sqrt{P}\right)$. SUMMA generalizes this to arbitrary matrix dimensions and a rectangular grid of processors [36] using broadcasts rather than cyclic shifts. We call these "2D" algorithms, because they use a two-dimensional processor grid and have bandwidth cost that scales as $1/\sqrt{P}$.

Another class of algorithms, known as "3D" [12, 2] because the communication pattern maps to a three-dimensional processor grid, uses more local memory and reduces communication relative to 2D algorithms. The bandwidth cost of these algorithms scales as $1/P^{2/3}$. Unfortunately, they can only be used if a factor of $\Omega(P^{1/3})$ extra memory is available.

The "2.5D" algorithm [52, 61] interpolates between 2D and 3D algorithms, using as much memory as is available to reduce communication. For square matrices, it asymptotically matches the lower bounds of [45] and [5], and so is communication-optimal. The recently proposed 3D-SUMMA algorithm [57] attempts to generalize the 2.5D algorithm to rectangular matrices. As our communication lower bounds in Section 3.1 show, 3D-SUMMA is communication-optimal for many, but not all, matrix dimensions.

1.3 Parallelizing Recursive Algorithms

An alternative to the iterative parallelization approach is what we call the *BFS/DFS* approach. BFS/DFS algorithms are based on sequential recursive algorithms, and view the processor layout as a hierarchy rather than a grid. Breadth-first steps (BFS) and depth-first steps (DFS) are alternate ways to solve the subproblems. At a BFS step, all of the subproblems are solved in parallel on independent subsets of the processors, whereas at a DFS all the processors work together to solve the subproblems serially. In general, BFS steps reduce communication costs, but may require extra memory relative to DFS steps. The extra memory at a BFS step is because each subset of the processors must have space for the entire input and output of its subproblem. With correct interleaving of BFS and DFS steps to stay within the available memory, we show that BFS/DFS gives communication-optimal algorithms for all the cases of matrix multiplication we consider. Because of their recursive structure, BFS/DFS algorithms are cache-, processor-, and network-oblivious in the sense of [34, 13, 25, 26]. Note that they are not oblivious to the aggregated local memory (DRAM) size, which is necessary to determine the optimal interleaving of BFS and DFS steps. Hence they should perform well without much tuning on hierarchical architectures, which are becoming more common.

Our primary focus is on matrix multiplication algorithms. In this case, subproblems are independent of each other, and hence may be computed simultaneously. In Chapter 5, we relax

this requirement, and show how to parallelize some recursive algorithms with dependencies between the subproblems.

1.4 The Loomis-Whitney Inequality

The lower bound proofs in Sections 3.1 and 4.2 make use of the geometric embedding techniques of [45, 10], so we review them here. This section may be safely skipped if the reader is only interested in parallelizing Strassen's fast matrix multiplication algorithm as described in Chapter 2.

Let A be an $m \times k$ matrix, B be a $k \times n$ matrix, and C be an $m \times n$ matrix. The mnk scalar multiplications that the classical algorithm performs when computing $C = A \cdot B$ may be arranged into a rectangular prism $\mathcal{V}$ of size $m \times n \times k$ with the three matrices as its faces. To perform a given multiplication, a processor must have access to the entries of A, B, and C corresponding to the projections onto the $m \times k$, $n \times k$, and $m \times n$ faces of the prism, respectively. If these entries are not assigned to that processor by the initial or final data layout, these entries correspond to words that must be communicated.

Given a set of voxels $V \subset \mathcal{V}$, the projections of the set onto three orthogonal faces corresponds to the input elements of A and B necessary to perform the multiplications and the output elements of C that the products must update. The computation cube and this relationship of voxels to input and output matrix elements is shown in Figure 1.1. The following lemma relates the volume of V to its projections:

Lemma 1.1. [49] *Let V be a finite set of lattice points in $\mathbf{R}^3$, i.e., points (x, y, z) with integer coordinates. Let V_x be the projection of V in the x-direction, i.e., all points (y, z) such that there exists an x so that $(x, y, z) \in V$. Define V_y and V_z similarly. Let $|\cdot|$ denote the cardinality of a set. Then $|V| \leq \sqrt{|V_x| \cdot |V_y| \cdot |V_z|}$.*

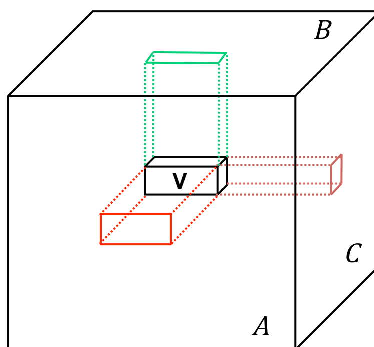


Figure 1.1: The computation rectangular prism for matrix multiplication, with a specified subset of voxels V along with its three projections. Each voxel corresponds to the multiplication of its projection onto A and B, and contributes to its projection onto C.

Chapter 2

Strassen's Matrix Multiplication

Strassen showed that 2×2 matrix multiplication can be performed using 7 multiplications and 18 additions, instead of the classical algorithm that does 8 multiplications and 4 additions [62]. By recursive application this yields an algorithm which multiplies two $n \times n$ matrices with $O(n^{\omega_0})$ flops, where $\omega_0 = \log_2 7 \approx 2.81$. Winograd improved the algorithm to use 7 multiplications and 15 additions in the base case, thus decreasing the hidden constant in the O notation [65]. Further reduction in the number of additions is not possible [56, 16].

In this chapter, we show how to parallelize Strassen's algorithm in a communication-optimal way, and benchmark and analyze its performance on several machines. The results in this chapter are joint work with Grey Ballard, James Demmel, Olga Holtz, and Oded Schwartz. The algorithm, analysis, and preliminary performance appear in [6]; more performance results, implementation details, and performance modeling appear in [48].

We use the term *parallel Strassen algorithm* for a parallel algorithm that performs exactly the same arithmetic operations as any variant of Strassen's (sequential) algorithm; that is, any algorithm based on 2×2 matrix multiplication using 7 scalar multiplications. We use the broader term *parallel Strassen-based algorithm* for a parallel matrix multiplication algorithm that is a hybrid of any variant of Strassen's and the classical algorithm. Examples of such hybrids are given in the next section. Note that Theorems 2.1 and 2.2 below apply to parallel Strassen algorithms, but not to all Strassen-based algorithms.

The rest of this chapter is organized as follows. We review the Strassen-Winograd algorithm in Section 2.1, and the communication-cost lower bounds for Strassen-like algorithms in Section 2.2. Section 2.3 presents our new communication-optimal algorithm. In Section 2.4 we analyze the communication costs of other approaches to parallelizing Strassen's algorithm, and show that none of them are communication-optimal. We present performance results for both classical and Strassen-based algorithms in Section 2.5 and compare the performance to a theoretical model in Section 2.6. We discuss details of our implementation, and numerical stability of Strassen's algorithm in Sections 2.7 and 2.8, respectively. Finally, in Section 2.9 we show how to generalize our approach to other fast matrix multiplication algorithms.

2.1 Strassen-Winograd Algorithm

To perform matrix multiplication using the Strassen-Winograd algorithm, first divide the input matrices A, B and output matrix C into 4 submatrices:

$$A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \; B = \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix} \; C = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix}$$

Then compute 7 linear combinations of the submatrices of each of A and B, call these T_i and S_i, respectively; multiply them pairwise; then compute the submatrices of C as linear combinations of these products:

$$\begin{array}{llll}
T_1 = A_{11} & S_1 = B_{11} & Q_1 = T_1 \cdot S_1 & U_1 \;= Q_1 + Q_4 \\
T_2 = A_{12} & S_2 = B_{21} & Q_2 = T_2 \cdot S_2 & U_2 \;= U_1 + Q_5 \\
T_3 = A_{21} + A_{22} & S_3 = B_{12} - B_{11} & Q_3 = T_3 \cdot S_3 & U_3 \;= U_1 + Q_3 \\
T_4 = T_3 - A_{11} & S_4 = B_{22} - S_3 & Q_4 = T_4 \cdot S_4 & C_{11} = Q_1 + Q_2 \\
T_5 = A_{11} - A_{21} & S_5 = B_{22} - B_{12} & Q_5 = T_5 \cdot S_5 & C_{12} = U_3 + Q_6 \\
T_6 = A_{12} - T_4 & S_6 = B_{22} & Q_6 = T_6 \cdot S_6 & C_{21} = U_2 - Q_7 \\
T_7 = A_{22} & S_7 = S_4 - B_{21} & Q_7 = T_7 \cdot S_7 & C_{22} = U_2 + Q_3
\end{array}$$

This is one step of Strassen-Winograd. The algorithm is recursive since it can be used for each of the 7 smaller matrix multiplications. In practice, one often uses only a few steps of Strassen-Winograd, although to attain $O(n^{\omega_0})$ computational cost, it is necessary to recursively apply it all the way down to matrices of size $O(1) \times O(1)$. The precise computational cost of Strassen-Winograd is

$$\mathrm{F}(n) = c_s n^{\omega_0} - 5n^2. \tag{2.1}$$

Here c_s is a constant depending on the cutoff point at which one switches to the classical algorithm. For a cutoff size of n_0, the constant is $c_s = (2n_0 + 4)/n_0^{\omega_0 - 2}$ which is minimized at $n_0 = 8$ yielding a computational cost of approximately $3.73n^{\omega_0} - 5n^2$.

2.2 Communication Lower Bounds

For parallel Strassen algorithms, the bandwidth cost lower bound has been proved using expansion arguments on the computation graph [9], and the latency cost lower bound is an immediate corollary. We believe the requirement of no recomputation is purely technical, but there is no known proof of the lower bounds without it.

Theorem 2.1. *(Memory-dependent lower bound) [9] Consider a parallel Strassen algorithm running on P processors each with local memory size M. Let $W(n, P, M)$ be the bandwidth cost and $S(n, P, M)$ be the latency cost of the algorithm. Assume that no intermediate values are computed twice. Then*

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

$$S(n, P, M) = \Omega\left(\left(\frac{n}{\sqrt{M}}\right)^{\omega_0} \cdot \frac{1}{P}\right).$$

We extended this to a memory-independent lower bound using the same expansion approach:

Theorem 2.2. *(Memory-independent lower bound) [5] Consider a parallel Strassen algorithm running on P processors. Let $W(n, P)$ be the bandwidth cost and $S(n, P)$ be the latency cost of the algorithm. Assume that no intermediate values are computed twice. Assume only one copy of the input data is stored at the start of the algorithm and the computation is load-balanced in an asymptotic sense. Then*

$$W(n, P) = \Omega\left(\frac{n^2}{P^{2/\omega_0}}\right),$$
$$S(n, P) = \Omega\left(1\right).$$

Note that when $M = O(n^2/P^{2/\omega_0})$, the memory-dependent lower bound dominates, and when $M = \Omega(n^2/P^{2/\omega_0})$, the memory-independent lower bound dominates.

2.3 Communication-Avoiding Parallel Strassen

In this section we present the CAPS algorithm, and prove it is communication-optimal. See Algorithm 1 for a concise presentation and Algorithm 2 for a more detailed description.

Theorem 2.3. *CAPS has computational cost*

$$\Theta\left(\frac{n^{\omega_0}}{P}\right),$$

bandwidth cost

$$\Theta\left(\max\left\{\frac{n^{\omega_0}}{PM^{\omega_0/2-1}}, \frac{n^2}{P^{2/\omega_0}}\right\}\right),$$

and latency cost

$$\Theta\left(\max\left\{\frac{n^{\omega_0}}{PM^{\omega_0/2}}\log P, \log P\right\}\right).$$

By Theorems 2.1 and 2.2, we see that CAPS has optimal computational and bandwidth costs, and that its latency cost is at most $\log P$ away from optimal. We prove Theorem 2.3 in Section 2.3.5.

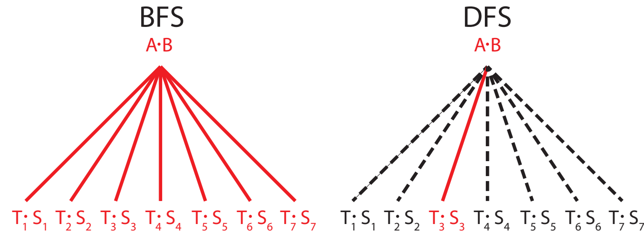


Figure 2.1: Representation of BFS and DFS steps. In a BFS step, all seven subproblems are computed at once, each on 1/7 of the processors. In a DFS step, the seven subproblems are computed in sequence, each using all the processors. The notation follows that of Section 2.1.

2.3.1 Overview of CAPS

Consider the recursion tree of Strassen's sequential algorithm. CAPS traverses it in parallel as follows. At each level of the tree, the algorithm proceeds in one of two ways. A "breadth-first-step" *(BFS)* divides the 7 subproblems among the processors, so that $\frac{1}{7}$ of the processors work on each subproblem independently and in parallel. A "depth-first-step" *(DFS)* uses all the processors on each subproblem, solving each one in sequence. See Figure 2.1.

In short, a BFS step requires more memory but reduces communication costs while a DFS step requires little extra memory but is less communication-efficient. In order to minimize communication costs, the algorithm must choose an ordering of BFS and DFS steps that uses as much memory as possible.

Let $k = \log_7 P$ and $s \geq k$ be the number of distributed Strassen steps the algorithm will take. For simplicity, in this section, we assume that n is a multiple of $2^s 7^{\lceil k/2 \rceil}$. If k is even, the restriction simplifies to n being a multiple of $2^s \sqrt{P}$. Since P is a power of 7, it is sometimes convenient to think of the processors as numbered in base 7. CAPS performs s steps of Strassen's algorithm and finishes the calculation with local matrix multiplication. The algorithm can easily be generalized to other values of n by padding or dynamic peeling (where, at each recursive step, if the matrix dimension is odd the last row and column are handled separately).

We consider two simple schemes of traversing the recursion tree with BFS and DFS steps. The first scheme, which we call the Unlimited Memory *(UM)* scheme, is to take k BFS steps in a row. This approach is possible only if there is sufficient available memory. It is also used in the second scheme, which we call the Limited Memory *(LM)* scheme, which is to take ℓ DFS steps in a row followed by k BFS steps in a row, where ℓ is minimized subject to the memory constraints.

It is possible to use a more complicated scheme that interleaves BFS and DFS steps to reduce communication. We show that the LM scheme is optimal up to a constant factor, and hence no more than a constant factor improvement can be attained from interleaving.

Algorithm 1 CAPS, in brief. For more details, see Algorithm 2.

Input: A, B, n, where A and B are $n \times n$ matrices
P = number of processors
Output: $C = A \cdot B$
▷ The dependence of the S_i's on A, the T_i's on B and C on the Q_i's follows the Strassen or Strassen-Winograd algorithm. See Section 2.1.

```
1: procedure C = CAPS(A, B, n, P)
2:     if enough memory then                              ▷ Do a BFS step
3:         locally compute the S_i's and T_i's from A and B
4:         parallel for i = 1 . . . 7 do
5:             redistribute S_i and T_i
6:             Q_i = CAPS(S_i, T_i, n/2, P/7)
7:             redistribute Q_i
8:         locally compute C from all the Q_i's
9:     else                                               ▷ Do a DFS step
10:        for i = 1 . . . 7 do
11:            locally compute S_i and T_i from A and B
12:            Q_i = CAPS(S_i, T_i, n/2, P)
13:            locally compute contribution of Q_i to C
```

2.3.2 Data layout

We require that the data layout of the matrices satisfies the following two properties:

1. At each of the s Strassen recursion steps, the data layouts of the four sub-matrices of each of A, B, and C must match so that the weighted additions of these sub-matrices can be performed locally. This technique follows [51] and allows communication-free DFS steps.

2. Each of these submatrices must be equally distributed among the P processors for load balancing.

There are many data layouts that satisfy these properties, perhaps the simplest being block-cyclic layout with a processor grid of size $7^{\lfloor k/2 \rfloor} \times 7^{\lceil k/2 \rceil}$ and block size $\frac{n}{2^s 7^{\lfloor k/2 \rfloor}} \times \frac{n}{2^s 7^{\lceil k/2 \rceil}}$. (When $k = \log_7 P$ is even these expressions simplify to a processor grid of size $\sqrt{P} \times \sqrt{P}$ and block size $\frac{n}{2^s \sqrt{P}}$.) See Figure 2.2.

Any layout that we use is specified by three parameters, (n, P, s), and intermediate stages of the computation use the same layout with smaller values of the parameters. A BFS step reduces a multiplication problem with layout parameters (n, P, s) to seven subproblems with layout parameters $(n/2, P/7, s-1)$. A DFS step reduces a multiplication problem with layout parameters (n, P, s) to seven subproblems with layout parameters $(n/2, P, s-1)$.

Note that if the input data is initially load-balanced but distributed using a different layout, we can rearrange it to the above layout using no more than $O\left(\frac{n^2}{P}\right)$ words and $O(P)$ messages. This has no asymptotic effect on the bandwidth cost but may significantly increase the latency cost.

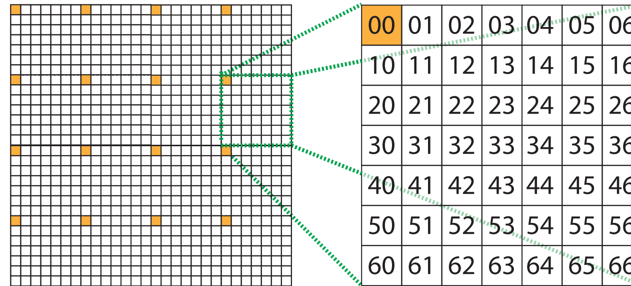


Figure 2.2: An example matrix layout for CAPS. Each of the 16 submatrices as shown on the left has exactly the same layout. The colored blocks are the ones owned by processor 00. On the right is a zoomed-in view of one submatrix, showing which processor, numbered base 7, owns each block. This is block-cyclic layout with some blocksize b, and matches our layout requirements with parameters $(n = 4 \cdot 7 \cdot b, P = 49, s = 2)$.

2.3.3 Unlimited Memory (UM) scheme

In the UM scheme, we take $k = \log_7 P$ BFS steps in a row. Since a BFS step reduces the number of processors involved in each subproblem by a factor of 7, after k BFS steps each subproblem is assigned to a single processor, and so is computed locally with no further communication costs. We first describe a BFS step in more detail.

The matrices A and B are initially distributed as described in Section 2.3.2. In order to take a recursive step, the 14 matrices $S_1, \ldots S_7, T_1, \ldots, T_7$ must be computed. Each processor allocates space for all 14 matrices and performs local additions and subtractions to compute its portion of the matrices. Recall that the submatrices are distributed identically, so this step requires no communication. If the layouts of A and B have parameters (n, P, s), the S_i and the T_i now have layout parameters $(n/2, P, s-1)$.

The next step is to redistribute these 14 matrices so that the 7 pairs of matrices (S_i, T_i) exist on disjoint sets of $P/7$ processors. This requires disjoint sets of 7 processors performing an all-to-all communication step (each processor must send and receive a message from each of the other 6). To see this, consider the numbering of the processors base-7. On the mth BFS step, the communication is between the seven processors whose numbers agree on all digits except the mth (counting from the right). After the mth BFS step, the set of processors working on a given subproblem share the same m-digit suffix. After the above communication is performed, the layout of S_i and T_i has parameters $(n/2, P/7, s-1)$, and the sets of processors that own the T_i and S_i are disjoint for different values of i. Since each all-to-all

Algorithm 2 CAPS, in detail

Input: A, B, are $n \times n$ matrices
P = number of processors
rank = processor number base-7 as an array
M = local memory size
Output: $C = A \cdot B$

1: **procedure** C = CAPS(A, B, P, rank, M)
2: $\ell = \left\lceil \log_2 \frac{4n}{P^{1/\omega_0} M^{1/2}} \right\rceil$ ▷ ℓ is number of DFS steps to fit in memory
3: $k = \log_7 P$
4: **call** DFS(A, B, C, k, ℓ, rank)

1: **procedure** DFS(A, B, C, k, ℓ, rank) ▷ Do $C = A \cdot B$ by ℓ DFS, then k BFS steps
2: **if** $\ell \leq 0$ **then call** BFS(A, B, C, k, rank); **return**
3: **for** $i = 1 \ldots 7$ **do**
4: locally compute S_i and T_i from A and B ▷ following Strassen's algorithm
5: **call** DFS(S_i, T_i, Q_i, k, $\ell - 1$, rank)
6: locally compute contribution of Q_i to C ▷ following Strassen's algorithm

---

1: **procedure** BFS(A, B, C, k, rank)
▷ Do $C = A \cdot B$ by k BFS steps, then local Strassen
2: **if** $k == 0$ **then call** localStrassen(A, B, C); **return**
3: **for** $i = 1 \ldots 7$ **do**
4: locally compute S_i and T_i from A and B ▷ following Strassen's algorithm
5: **for** $i = 1 \ldots 7$ **do**
6: target = rank
7: target[k] = i
8: send S_i to target
9: receive into L ▷ One part of L comes from each of 7 processors
10: send T_i to target
11: receive into R ▷ One part of R comes from each of 7 processors
12: **call** BFS(L, R, V, $k - 1$, rank)
13: **for** $i = 1 \ldots 7$ **do**
14: target = rank
15: target[k] = i
16: send i^{th} part of V to target
17: receive from target into Q_i
18: locally compute C from Q_i ▷ following Strassen's algorithm

communication only involves seven processors no matter how large P is, this algorithm does not have the scalability issues that typically come from an all-to-all communication pattern.

Memory requirements

The extra memory required to take one BFS step is the space to store all 7 triples S_j, T_j, Q_j. Since each of those matrices is $\frac{1}{4}$ the size of A, B, and C, the extra space required at a given step is 7/4 the extra space required for the previous step. We assume that no extra memory is required for the local multiplications.[1] Thus, the total local memory requirement for taking k BFS steps is given by

$$\text{Mem}_{\text{UM}}(n, P) = \frac{3n^2}{P} \sum_{i=0}^{k} \left(\frac{7}{4}\right)^i = \frac{7n^2}{P^{2/\omega_0}} - \frac{4n^2}{P} = \Theta\left(\frac{n^2}{P^{2/\omega_0}}\right).$$

Computational costs

The computation required at a given BFS step is that of the local additions and subtractions associated with computing the S_i and T_i and updating the output matrix C with the Q_i. Since Strassen-Winograd performs 15 additions and subtractions, the computational cost recurrence is

$$F_{\text{UM}}(n, P) = 15\left(\frac{n^2}{4P}\right) + F_{\text{UM}}\left(\frac{n}{2}, \frac{P}{7}\right)$$

with base case $F_{\text{UM}}(n, 1) = c_s n^{\omega_0} - 5n^2$, where c_s is the constant of Strassen-Winograd. See Section 2.1 for more details. The solution to this recurrence is

$$F_{\text{UM}}(n, P) = \frac{c_s n^{\omega_0} - 5n^2}{P} = \Theta\left(\frac{n^{\omega_0}}{P}\right).$$

Communication costs

Consider the communication costs associated with the UM scheme. Given that the redistribution within a BFS step is performed by an all-to-all communication step among sets of 7 processors, each processor sends 6 messages and receives 6 messages to redistribute $S_1, \ldots, S_7$, and the same for $T_1, \ldots, T_7$. After the products $Q_i = S_i T_i$ are computed, each processor sends 6 messages and receives 6 messages to redistribute $Q_1, \ldots, Q_7$. The size of each message varies according to the recursion depth, and is the number of words a processor owns of any S_i, T_i, or Q_i, namely $\frac{n^2}{4P}$ words.

---

[1] If one does not overwrite the input, it is impossible to run Strassen in place; however using a few temporary matrices affects the analysis here by a constant factor only.

As each of the Q_i is computed simultaneously on disjoint sets of $P/7$ processors, we obtain a cost recurrence for the entire UM scheme:

$$\begin{aligned} W_{\mathrm{UM}}(n,P) &= 36\frac{n^2}{4P} + W_{\mathrm{UM}}\left(\frac{n}{2},\frac{P}{7}\right) \\ S_{\mathrm{UM}}(n,P) &= 36 + S_{\mathrm{UM}}\left(\frac{n}{2},\frac{P}{7}\right) \end{aligned}$$

with base case $S_{\mathrm{UM}}(n,1) = W_{\mathrm{UM}}(n,1) = 0$. Thus

$$\begin{aligned} W_{\mathrm{UM}}(n,P) &= \frac{12n^2}{P^{2/\omega_0}} - \frac{12n^2}{P} = \Theta\left(\frac{n^2}{P^{2/\omega_0}}\right) \\ S_{\mathrm{UM}}(n,P) &= 36\log_7 P = \Theta(\log P). \end{aligned} \tag{2.2}$$

2.3.4 Limited Memory (LM) scheme

In this section we discuss a scheme for traversing Strassen's recursion tree in the context of limited memory. In the LM scheme, we take ℓ DFS steps in a row followed by k BFS steps in a row, where ℓ is minimized subject to the memory constraints. That is, we use a sequence of DFS steps to reduce the problem size so that we can use the UM scheme on each subproblem without exceeding the available memory.

Consider taking a single DFS step. Rather than allocating space for and computing all 14 matrices $S_1, T_1, \ldots, S_7, T_7$ at once, the DFS step requires allocation of only one subproblem, and each of the Q_i will be computed in sequence.

Consider the ith subproblem: as before, both S_i and T_i can be computed locally. After Q_i is computed, it is used to update the corresponding quadrants of C and then discarded so that its space in memory (as well as the space for S_i and T_i) can be re-used for the next subproblem. In a DFS step, no redistribution occurs. After S_i and T_i are computed, all processors participate in the computation of Q_i.

We assume that some extra memory is available. To be precise, assume the matrices A, B, and C require only $\frac{1}{3}$ of the available memory:

$$\frac{3n^2}{P} \leq \frac{1}{3}M. \tag{2.3}$$

In the LM scheme, we set

$$\ell = \max\left\{0, \left\lceil \log_2 \frac{4n}{P^{1/\omega_0}M^{1/2}} \right\rceil\right\}. \tag{2.4}$$

The following subsection shows that this choice of ℓ is sufficient not to exceed the memory capacity.

Memory requirements

The extra memory requirement for a DFS step is the space to store one subproblem. Thus, the extra space required at this step is 1/4 the space required to store A, B, and C. The local memory requirements for the LM scheme is given by

$$\begin{aligned}
\text{Mem}_{\text{LM}}(n,P) &= \frac{3n^2}{P}\sum_{i=0}^{\ell-1}\left(\frac{1}{4}\right)^i + \text{Mem}_{\text{UM}}\left(\frac{n}{2^\ell},P\right) \\
&\le \frac{M}{3}\sum_{i=0}^{\ell-1}\left(\frac{1}{4}\right)^i + \frac{7\left(\frac{n}{2^\ell}\right)^2}{P^{2/\omega_0}} \\
&\le \frac{127}{144}M < M,
\end{aligned}$$

where the last line follows from (2.4) and (2.3). Thus, the limited memory scheme does not exceed the available memory.

Computational costs

As in the UM case, the computation required at a given DFS step is that of the local additions and subtractions associated with computing each S_i and T_i and updating the output matrix C with the Q_i. However, since all processors participate in each subproblem and the subproblems are computed in sequence, the recurrence is given by

$$F_{\text{LM}}(n,P) = 15\left(\frac{n^2}{4P}\right) + 7\cdot F_{\text{LM}}\left(\frac{n}{2},P\right).$$

After ℓ steps of DFS, the size of a subproblems is $\frac{n}{2^\ell}\times\frac{n}{2^\ell}$, and there are P processors involved. We take k BFS steps to compute each of these 7^ℓ subproblems. Thus

$$F_{\text{LM}}\left(\frac{n}{2^\ell},P\right) = F_{\text{UM}}\left(\frac{n}{2^\ell},P\right),$$

and

$$F_{\text{LM}}(n,P) = \frac{15n^2}{4P}\sum_{i=0}^{\ell-1}\left(\frac{7}{4}\right)^i + 7^\ell\cdot F_{\text{UM}}\left(\frac{n}{2^\ell},P\right) = \frac{c_s n^{\omega_0} - 5n^2}{P} = \Theta\left(\frac{n_0^\omega}{P}\right).$$

Communication costs

Since there are no communication costs associated with a DFS step, the recurrence is simply

$$\begin{aligned}
W_{\text{LM}}(n,P) &= 7\cdot W_{\text{LM}}\left(\frac{n}{2},P\right) \\
S_{\text{LM}}(n,P) &= 7\cdot S_{\text{LM}}\left(\frac{n}{2},P\right)
\end{aligned}$$

with base cases

$$W_{\text{LM}}\left(\frac{n}{2^{\ell}}, P\right) = W_{\text{UM}}\left(\frac{n}{2^{\ell}}, P\right)$$
$$S_{\text{LM}}\left(\frac{n}{2^{\ell}}, P\right) = S_{\text{UM}}\left(\frac{n}{2^{\ell}}, P\right).$$

Thus the total communication costs are given by

$$W_{\text{LM}}(n, P) = 7^{\ell} \cdot W_{\text{UM}}\left(\frac{n}{2^{\ell}}, P\right) \leq \frac{12 \cdot 4^{\omega_0 - 2} n^{\omega_0}}{P M^{\omega_0/2-1}} = \Theta\left(\frac{n^{\omega_0}}{P M^{\omega_0/2-1}}\right).$$
$$S_{\text{LM}}(n, P) = 7^{\ell} \cdot S_{\text{UM}}\left(\frac{n}{2^{\ell}}, P\right) \leq \frac{(4n)^{\omega_0}}{P M^{\omega_0/2}} 36 \log_7 P = \Theta\left(\frac{n^{\omega_0}}{P M^{\omega_0/2}} \log P\right). \tag{2.5}$$

2.3.5 Communication optimality

Proof. (*of Theorem 2.3 on page 7*). In the case that $M \geq \text{Mem}_{\text{UM}}(n, P) = \Omega\left(\frac{n^2}{P^{2/\omega_0}}\right)$ the UM scheme is possible. Then the communication costs are given by Equation (2.2) on page 13 which matches the lower bound of Theorem 2.2 on page 7. Thus the UM scheme is communication-optimal (up to a logarithmic factor in the latency cost and assuming that the data is initially distributed as described in Section 2.3.2). For smaller values of M, the LM scheme must be used. Then the communication costs are given by Equation (2.5) on page 15 and match the lower bound of Theorem 2.1 on page 6, so the LM scheme is also communication-optimal. □

We note that for the LM scheme, since both the computational and communication costs are proportional to $\frac{1}{P}$ (up to a $\log P$ factor on S), we can expect *perfect strong scaling*: given a fixed problem size, increasing the number of processors by some factor will decrease each cost by the same factor. However, this strong scaling property has a limited range. For any fixed M and n, increasing P increases the global memory size PM. The limit of perfect strong scaling is exactly when there is enough memory for the UM scheme. See [5] for details.

2.4 Analysis of Other Algorithms

In this section we detail the asymptotic communication costs of other matrix multiplication algorithms, both classical and Strassen-based. These communication costs and the corresponding lower bounds are summarized in Table 2.1.

Many of the algorithms described in this section are hybrids of two different algorithms. We use the convention that the names of the hybrid algorithms are composed of the names of the two component algorithms, hyphenated. The first name describes the algorithm used at the top level, on the largest problems, and the second describes the algorithm used at the base level on smaller problems.

		Flops	Bandwidth	Latency
Classical	Lower Bound [5, 45]	$\frac{n^3}{P}$	$\max\left\{\frac{n^3}{PM^{1/2}}, \frac{n^2}{P^{2/3}}\right\}$	$\max\left\{\frac{n^3}{PM^{3/2}}, 1\right\}$
	2D [21, 36]	$\frac{n^3}{P}$	$\frac{n^2}{P^{1/2}}$	$P^{1/2}$
	3D [2, 12]	$\frac{n^3}{P}$	$\frac{n^2}{P^{2/3}}$	$\log P$
	2.5D [52, 61], CARMA [31]	$\frac{n^3}{P}$	$\max\left\{\frac{n^3}{PM^{1/2}}, \frac{n^2}{P^{2/3}}\right\}$	$\frac{n^3}{PM^{3/2}} + \log P$
Strassen	Lower Bound [5, 9]	$\frac{n^{\omega_0}}{P}$	$\max\left\{\frac{n^{\omega_0}}{PM^{\omega_0/2-1}}, \frac{n^2}{P^{2/\omega_0}}\right\}$	$\max\left\{\frac{n^{\omega_0}}{PM^{\omega_0/2}}, 1\right\}$
	2D-Strassen [51]	$\frac{n^{\omega_0}}{P^{(\omega_0-1)/2}}$	$\frac{n^2}{P^{1/2}}$	$P^{1/2}$
	Strassen-2D [39, 51]	$\left(\frac{7}{8}\right)^{\ell}\frac{n^3}{P}$	$\left(\frac{7}{4}\right)^{\ell}\frac{n^2}{P^{1/2}}$	$7^{\ell}P^{1/2}$
	2.5D-Strassen [6]	$\max\left\{\frac{n^3}{PM^{3/2-\omega_0/2}}, \frac{n^{\omega_0}}{P^{\omega_0/3}}\right\}$	$\max\left\{\frac{n^3}{PM^{1/2}}, \frac{n^2}{P^{2/3}}\right\}$	$\frac{n^3}{PM^{3/2}} + \log P$
	Strassen-2.5D [6]	$\left(\frac{7}{8}\right)^{\ell}\frac{n^3}{P}$	$\max\left\{\left(\frac{7}{8}\right)^{\ell}\frac{n^3}{PM^{1/2}}, \left(\frac{7}{4}\right)^{\ell}\frac{n^2}{P^{2/3}}\right\}$	$\left(\frac{7}{8}\right)^{\ell}\frac{n^3}{PM^{3/2}} + 7^{\ell}\log P$
	CAPS [6]	$\frac{n^{\omega_0}}{P}$	$\max\left\{\frac{n^{\omega_0}}{PM^{\omega_0/2-1}}, \frac{n^2}{P^{2/\omega_0}}\right\}$	$\max\left\{\frac{n^{\omega_0}}{PM^{\omega_0/2}}\log P, \log P\right\}$

Table 2.1: Asymptotic computational and communication costs of Strassen-basend and classical square matrix multiplication algorithms and corresponding lower bounds. Here $\omega_0 = \log_2 7 \approx 2.81$ is the exponent of Strassen; ℓ is the number of Strassen steps taken. The CAPS algorithm attains the lower bounds of Section 2.2, and thus is optimal. All of the other Strassen-based algorithms have asymptotically higher communication costs; see Section 2.4 for details.

2.4.1 Classical Algorithms

Any classical algorithm must communicate asymptotically more than an optimal parallel Strassen algorithm (see the lower bounds in[45]). To compare the communication cost upper and lower bounds of classical and parallel Strassen algorithms, it is necessary to consider three cases for the memory size: when the memory-dependent bounds dominate for both classical and Strassen, when the memory-dependent bound dominates for classical but the memory-independent bound dominates for Strassen, and when the memory-independent bounds dominate for both classical and Strassen. See Figure 2.3.

Case 1 $M = \Omega(n^2/P)$ and $M = O(n^2/P^{2/\omega_0})$. The first condition is necessary for there to be enough memory to hold the input and output matrices; the second condition puts both classical and Strassen algorithms in the memory-dependent case. Then the ratio of the bandwidth costs is:

$$R = \Theta\left(\frac{n^3}{PM^{1/2}} \Big/ \frac{n^{\omega_0}}{PM^{\omega_0/2-1}}\right) = \Theta\left(\left(\frac{n^2}{M}\right)^{(3-\omega_0)/2}\right).$$

Using the two bounds that define this case, we obtain $R = O(P^{(3-\omega_0)/2})$ and $R = \Omega(P^{3/\omega_0-1})$.

Case 2 $M = \Omega(n^2/P^{2/\omega_0})$ and $M = O(n^2/P^{2/3})$. This means that in the classical case the memory-dependent lower bound dominates, but in the Strassen case the memory-independent lower bound dominates. Then the ratio is:

$$R = \Theta\left(\frac{n^3}{PM^{1/2}} \Big/ \frac{n^2}{P^{2/\omega_0}}\right) = \Theta\left(\left(\frac{n^2}{M}\right)^{1/2} P^{2/\omega_0-1}\right).$$

Using the two bounds that define this case, we obtain $R = O(P^{3/\omega_0-1})$ and $R = \Omega(P^{2/\omega_0-2/3})$.

Case 3 $M = \Omega(P^{2/3})$. This means that both the classical and Strassen lower bounds are dominated by the memory-independent cases. Then the ratio is:

$$R = \Theta\left(\frac{n^2}{P^{2/3}} \Big/ \frac{n^2}{P^{2/\omega_0}}\right) = \Theta\left(P^{2/\omega_0-2/3}\right).$$

Overall, depending on the ratio of the problem size to the available memory, the factor by which the classical bandwidth costs exceed the Strassen bandwidth costs is $\Theta(P^a)$, where a ranges from $\frac{2}{\omega_0} - \frac{2}{3} \approx 0.046$ to $\frac{3-\omega_0}{2} \approx 0.10$. The same sort of analysis is used throughout Section 2.4 to compare each algorithm with the parallel Strassen lower bounds.

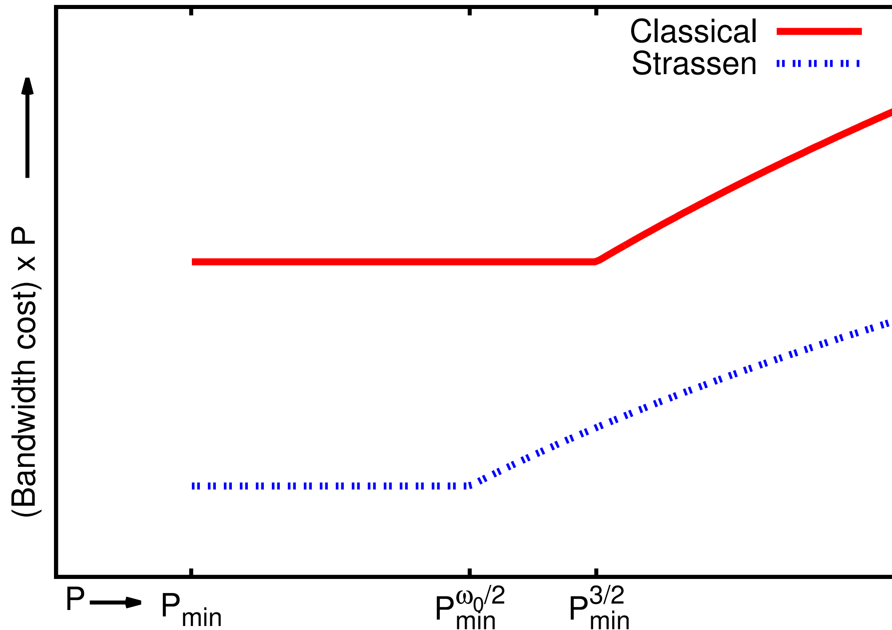


Figure 2.3: Bandwidth costs and strong scaling of matrix multiplication: classical vs. Strassen-based. Horizontal lines correspond to perfect strong scaling. P_{min} is the minimum number of processors required to store the input and output matrices.

2.4.2 2D-Strassen

One idea to construct a parallel Strassen-based algorithm is to use a 2D classical algorithm for the inter-processor communication, and use the fast matrix multiplication algorithm locally [51]. We call such an algorithm "2D-Strassen". It is straightforward to implement, but cannot attain all the computational speedup from Strassen since it uses a classical algorithm for part of the computation. In particular, it does not use Strassen for the largest matrices, when Strassen would provide the greatest reduction in computation. As a result, the computational cost exceeds $\Theta(n^{\omega_0}/P)$ by a factor of $P^{(3-\omega_0)/2} \approx P^{0.10}$. The 2D-Strassen algorithm has the same communication cost as 2D algorithms, and hence does not match the communication costs of CAPS. In comparing the 2D-Strassen bandwidth cost, $\Theta(n^2/P^{1/2})$, to the CAPS bandwidth cost in Section 2.3, note that for the problem to fit in memory we always have $M = \Omega(n^2/P)$. The bandwidth cost exceeds that of CAPS by a factor of P^a, where a ranges from $(3-\omega_0)/2 \approx .10$ to $2/\omega_0 - 1/2 \approx .21$, depending on the relative problem size. Similarly, the latency cost, $\Theta(P^{1/2})$, exceeds that of CAPS by a factor of P^a where a ranges from $(3-\omega_0)/2 \approx .10$ to $1/2 = .5$.

2.4.3 Strassen-2D

The "Strassen-2D" algorithm applies ℓ DFS steps of Strassen's algorithm at the top level, and performs the 7^ℓ smaller matrix multiplications using a 2D algorithm. By choosing certain data layouts as in Section 2.3.2, it is possible to do the additions and subtractions for Strassen's algorithm without any communication [51]. However, Strassen-2D is also unable to match the communication costs of CAPS. Moreover, the speedup of Strassen-2D in computation comes at the expense of extra communication. For large numbers of Strassen steps ℓ, Strassen-2D can approach the computational lower bound of Strassen, but each step increases the bandwidth cost by a factor of $\frac{7}{4}$ and the latency cost by a factor of 7. Thus the bandwidth cost of Strassen-2D is a factor of $\left(\frac{7}{4}\right)^\ell$ higher than 2D-Strassen, which is already higher than that of CAPS. The latency cost is even worse: Strassen-2D is a factor of 7^ℓ higher than 2D-Strassen.

One can reduce the latency cost of Strassen-2D at the expense of a larger memory footprint. Since Strassen-2D runs a 2D algorithm 7^ℓ times on the same set of processors, it is possible to pack together messages from independent matrix multiplications. In the best case, the latency cost is reduced to the cost of 2D-Strassen, which is still above that of CAPS, at the expense of using a factor of $\left(\frac{7}{4}\right)^\ell$ more memory.

2.4.4 2.5D-Strassen

A natural idea is to replace a 2D classical algorithm in 2D-Strassen with the superior 2.5D classical algorithm to obtain an algorithm we call 2.5D-Strassen. This algorithm uses the 2.5D algorithm for the inter-processor communication, and then uses Strassen for the local computation. When $M = \Theta(n^2/P)$, 2.5D-Strassen is exactly the same as 2D-Strassen, but when there is extra memory it both decreases the communication cost and decreases the computational cost since the local matrix multiplications are performed (using Strassen) on larger matrices. To be precise, the computational cost exceeds the lower bound by a factor of P^a where a ranges from $1 - \frac{\omega_0}{3} \approx 0.064$ to $\frac{3-\omega_0}{2} \approx 0.10$ depending on the relative problem size. The bandwidth cost exceeds the bandwidth cost of CAPS by a factor of P^a where a ranges from $\frac{2}{\omega_0} - \frac{2}{3} \approx 0.046$ to $\frac{3-\omega_0}{2} \approx 0.10$. In terms of latency, the cost of $\frac{n^3}{PM^{3/2}} + \log P$ exceeds the latency cost of CAPS by a factor ranging from $\log P$ to $P^{(3-\omega_0)/2} \approx P^{0.10}$, depending on the relative problem size.

2.4.5 Strassen-2.5D

Similarly, by replacing a 2D algorithm with 2.5D in Strassen-2D, one obtains the new algorithm we call Strassen-2.5D. First one takes ℓ DFS steps of Strassen, which can be done without communication, and then one applies the 2.5D algorithm to each of the 7^ℓ subproblems. The computational cost is exactly the same as Strassen-2D, but the communication cost will typically be lower. Each of the 7^ℓ subproblems is multiplication of $n/2^\ell \times n/2^\ell$ matrices. Each subproblem uses only $1/4^\ell$ as much memory as the original problem. Thus there may be a large amount of extra memory available for each subproblem, and the lower communication

costs of the 2.5D algorithm help. The choice of ℓ that minimizes the bandwidth cost is

$$\ell_{\text{opt}} = \max\left\{0, \left\lceil \log_2 \frac{n}{M^{1/2}P^{1/3}} \right\rceil\right\}.$$

The same choice minimizes the latency cost. Note that when $M \geq \frac{n^2}{P^{2/3}}$, taking zero Strassen steps minimizes the communication within the constraints of the Strassen-2.5D algorithm. With $\ell = \ell_{\text{opt}}$, the bandwidth cost is a factor of $P^{1-\omega_0/3} \approx P^{0.064}$ above that of CAPS. Additionally, the computational cost is not optimal, and using $\ell = \ell_{\text{opt}}$, the computational cost exceeds the optimal by a factor of $P^{1-\omega_0/3}M^{3/2-\omega_0/2} \approx P^{0.064}M^{0.096}$.

It is also possible to take $\ell > \ell_{\text{opt}}$ steps of Strassen to decrease the computational cost further. However the decreased computational cost comes at the expense of higher communication cost, as in the case of Strassen-2D. In particular, each extra step over ℓ_{opt} increases the bandwidth cost by a factor of $\frac{7}{4}$ and the latency cost by a factor of 7. As with Strassen-2D, it is possible to use extra memory to pack together messages from several subproblems and decrease the latency cost, but not the bandwidth cost.

2.5 Performance Results

We have implemented CAPS using MPI on three supercomputers, a Cray XE6 (Hopper[2]), an IBM BG/P (Intrepid[3]), and a Cray XT4 (Franklin[4]), and we compare it to various previous classical and Strassen-based algorithms. All our experiments are in double precision on random input matrices. CAPS performs less communication than communication-optimal classical algorithms, and much less than previous Strassen-based algorithms. As a result it outperforms all classical algorithms, both on large problems (because of the lower flop count of Strassen) and on small problems scaled up to many processors (which are communication bound, so the lower communication costs of CAPS make it superior). It also outperforms previous Strassen-based algorithms because of its lower communication costs.

Effective performance is a useful construct for comparing classical and fast matrix multiplication algorithms. It is the performance, normalized with respect to the arithmetic complexity of classical matrix multiplication, $2n^3$:

$$\text{Effective flop/s} = \frac{2n^3}{\text{Execution time in seconds}}.$$

For classical algorithms, this gives exactly the flop rate. For fast matrix multiplication algorithms it gives the relative performance, but does not accurately represent the number of floating point operations performed.

[2]For machine details, see http://www.nersc.gov/users/computational-systems/hopper

[3]For machine details, see http://www.alcf.anl.gov/intrepid

[4]For machine details, see http://www.nersc.gov/users/computational-systems/retired-systems/franklin

For each of the three machines, we present two types of plots. First, in Figures 2.4a, 2.4c, and 2.4e, we show strong scaling plots for a fixed, large matrix dimension where the x-axis corresponds to number of processor cores (on a log scale) and the y-axis corresponds to fraction of peak performance, as measured by the effective performance. Horizontal lines in the plots correspond to perfect strong scaling.

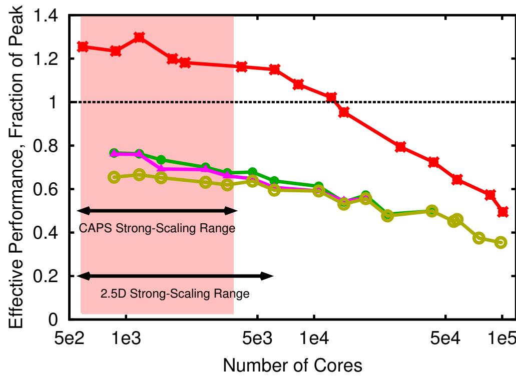
There are general trends for all algorithms presented in these plots. On the left side of the plots, the number of processors is small enough such that the input and output matrices nearly fill the memories of the processors. As the number of processors increases, both 2.5D and CAPS can exhibit perfect strong scaling within limited ranges. We demarcate the strong-scaling range of CAPS as defined in Section 1.1 with a shaded region. The slightly larger strong-scaling range for the classical 2.5D algorithm is also shown. To the right of the strong scaling range, CAPS must begin to lose performance, as per-processor communication no longer scales with $1/P$. While CAPS performance should theoretically degrade more slowly than classical algorithms, network resource contention can also be a limiting factor. The pattern of BFS and DFS steps used by CAPS for these benchmarks is shown in Table 2.2 when the number of MPI processes is a power of 7.

Second, we show execution time for fixed, small matrix dimension over an increasing number of processors. See Figures 2.4b, 2.4d, and 2.4f. For these problem sizes, the execution time is dominated by communication, and the speedup relative to classical algorithms is based primarily on decreases in communication. The optimal number of processors to minimize time to solution varies for each implementation and machine. These plots do not show strong scaling ranges. For both 2.5D and CAPS if a problem fits on one processor, that is $P_{\min} = 1$, then for 2.5D $P_{\max} = P_{\min}^{1.5} = 1$ and for CAPS $P_{\max} = P_{\min}^{\omega_0/2} = 1$, which means that there is no strong scaling range.

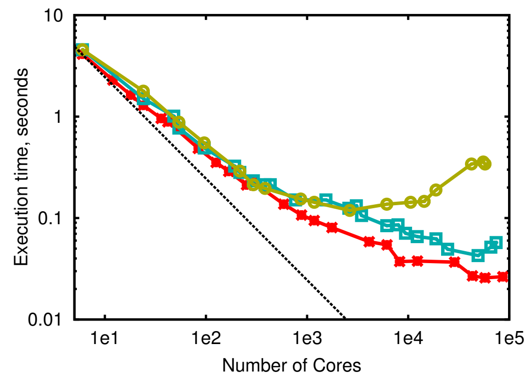
Note that because several of the implementations, including CAPS, are prototypes, each has its own requirement on the matrix size n and the number of MPI processes P. We have arranged for all algorithms in a given plot to use the same value of n, but the values of P usually do not match between algorithms. In both types of plots, we are comparing CAPS performance with the best classical implementations and previous Strassen-based approaches. To simplify the plots, we omit the performance of algorithms that are dominated by other similar algorithms.

2.5.1 Cray XE6 Hopper

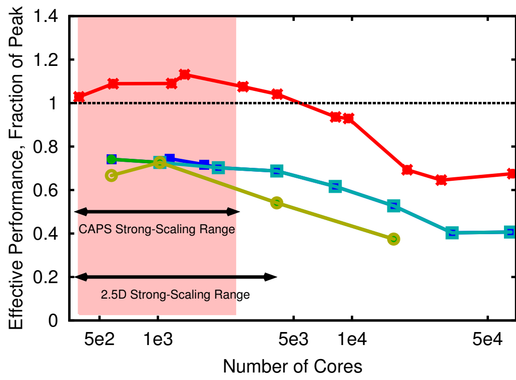
Hopper is a Cray XE6 at the National Energy Research Scientific Computing Center. It consists of 6,384 compute nodes, each of which has 2 twelve-core AMD "MagnyCours" 2.1 GHz processors, and 32 GB of DRAM (384 of the nodes have 64 GB of DRAM). The 24 cores are divided between 4 NUMA regions. Parallelism between the 6 cores in a NUMA region comes from the threaded BLAS implementation in Cray's LibSci version 11.0.05. Hopper's peak double precision rate is 50.4 Gflop/s per NUMA region or 1.28 Pflop/s for the entire machine. As of November 2011, it was ranked number 8 on the TOP500 list [53], with a LINPACK score of 1.05 Tflop/s on a matrix of dimension about 4.5 million.



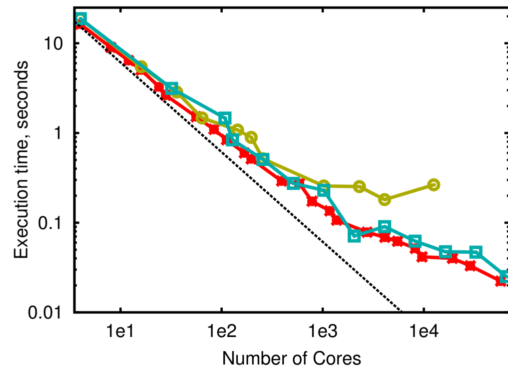
(a) Hopper (Cray XE6), $n = 131712$



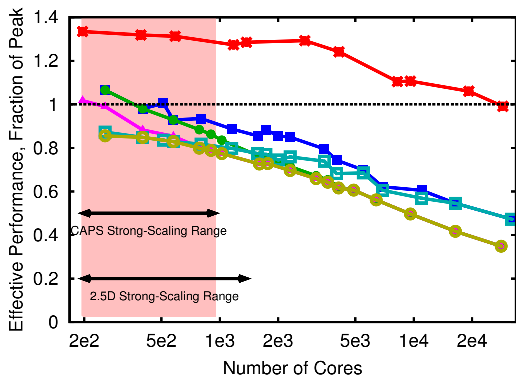
(b) Hopper (Cray XE6), $n = 4704$



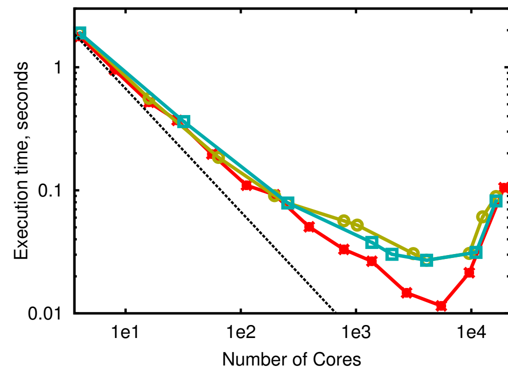
(c) Intrepid (IBM BG/P), $n = 65856$



(d) Intrepid (IBM BG/P), $n = 4704$



(e) Franklin (Cray XT4), $n = 94080$



(f) Franklin (Cray XT4), $n = 3136$

Figure 2.4: Strong scaling results on three machines. Left column: effective performance on large matrices (up is good). Right column: execution time on small matrices (down is good).

Machine, n	P	cores	pattern	memory use per process
Hopper, 131712	343	2058	D,B,B,B,D,D,D	5615 MB
Hopper, 131712	2401	14406	B,B,B,B,D,D,D	4816 MB
Intrepid, 65856	343	1372	D,B,B,B	1319 MB
Intrepid, 65856	2401	9604	B,B,B,B	1119 MB
Intrepid, 65856	16807	67228	B,B,B,B,B	289 MB
Franklin, 94080	49	196	D,D,B,B,D,D	6819 MB
Franklin, 94080	343	1372	B,B,D,B,D,D	5902 MB
Franklin, 94080	2401	9604	B,B,B,B,D,D	2449 MB

Table 2.2: The pattern of BFS and DFS steps and memory usage for CAPS.

CAPS outperforms all of the previous algorithms. For the large problem ($n = 131712$), it attains performance as high as 30% above the peak for classical matrix multiplication, 83% above 2D, and 75% above Strassen-2D. On this machine, we benchmark ScaLAPACK/PBLAS (part of Cray's LibSci version 11.0.03) as the 2D algorithm. Since we were not able to modify that code, the 2D-Strassen numbers are simulated based on single-node benchmarks of the corresponding local matrix multiplication size. We tested the 2.5D code tuned for Intrepid on Hopper; for large problems it performed worse than ScaLAPACK, and since it was not tuned for Hopper, we do not show results for 2.5D, Strassen-2.5D, or 2.5D-Strassen. We would expect that 2.5D code, properly tuned for Hopper, would outperform 2D. For the small problem ($n = 4704$), we observed speedups of up to 66% over 2.5D, which happened to be the best of the other algorithms for this problem size.

2.5.2 IBM BlueGene/P Intrepid

Intrepid is an IBM BG/P at the Argonne Leadership Computing Facility. It consists of 40,960 compute nodes, each of which has a quad-core IBM PowerPC 450 850 MHz processor, and 2 GB of DRAM. Intrepid's peak double precision rate is 13.6 Gflop/s per node, or 557 Tflop/s for the entire machine. We obtain on-node parallelism using the threaded BLAS implementation in IBM's ESSL version 4.4.1-0. As of November 2011, it was ranked number 23 on the TOP500 list [53], with a LINPACK score of 459 Tflop/s. Intrepid allows allocations only in powers of two nodes (with a few exceptions), but in our performance data we count only the nodes we use.

On Intrepid, the most efficient classical code is 2.5D and is well-tuned to the architecture. It consistently outperforms Strassen-2D and Strassen-2.5D, so we omit those algorithms in the performance plots. The 2D and 2.5D code are from [61]. For the large problem ($n = 65856$), CAPS achieves a speedup of up to 57% over 2.5D or 2.5D-Strassen; for the small problem ($n = 4704$), the best speedup is 12%.

2.5.3 Cray XT4 Franklin

Franklin is a recently retired Cray XT4 at the National Energy Research Scientific Computing Center. It consists of 9,572 compute nodes, each of which has a quad-core AMD "Budapest" 2.3 GHz processor, and 8 GB of DRAM. Franklin's peak double precision rate is 36.8 Gflop/s per node, or 352 Tflop/s for the entire machine. On each node, we use the threaded BLAS implementation in Cray's LibSci version 10.5.02. As of November 2011, it was ranked number 38 on the TOP500 list [53], with a LINPACK score of 266 Tflop/s on a matrix of dimension about 1.6 million.

CAPS outperforms all of the previous algorithms, and attains performance as high as 33% above the theoretical maximum for classical algorithms, as shown in Figure 2.4e. The largest speedups we observed for the large problem ($n = 94080$) was 103% faster than 2.5D, the fastest classical algorithm, and 187% faster than Strassen-2D, the best previous Strassen-based algorithm. For the small problem size ($n = 3136$), we observed up to 84% improvement over 2.5D, which was the best among the all other approaches. The 2D and 2.5D code are from [61].

For a matrix dimension of $n = 188160$, we observed an aggregate effective performance rate of 351 Tflop/s which exceeds the LINPACK score. Note that for this run CAPS used only 7203 (75%) of the nodes and a matrix of less than one eighth the dimension used for the TOP500 number. In fact, increasing the matrix size to $n = 263424$ increases its effective performance to 388 Tflop/s, higher than Franklin's theoretical peak for classical algorithms.

2.5.4 CAPS vs. Strassen-based algorithms

Figure 2.5 compares the performance of CAPS with the previous Strassen-based approaches on Intrepid. The plot shows, for a fixed matrix dimension and number of processors, both the effective and actual performance of the two previous Strassen-based algorithms and CAPS over various numbers of Strassen steps. For a given number of Strassen steps, the three algorithms do (almost) the same number of flops. Note that since the number of nodes is 49, CAPS is defined only for at least 2 Strassen steps.

For this matrix dimension, CAPS attains highest effective performance (shortest time to completion) at 4 Strassen steps. We see that the actual performance for CAPS (and the other two algorithms) decreases with the number of Strassen steps, as it becomes harder to do the fewer flops as efficiently.

In the case of 2D-Strassen, varying the number of Strassen steps means varying how each local matrix multiplication is performed. For the local matrix dimension of $n = 3136$, two Strassen steps is optimal, and the improvement in effective performance is modest because the matrix dimension is fairly small. In the case of Strassen-2D, both effective and actual performance degrade with each Strassen step. This is due to the increasing communication costs of the algorithm, which outweigh the computational savings.

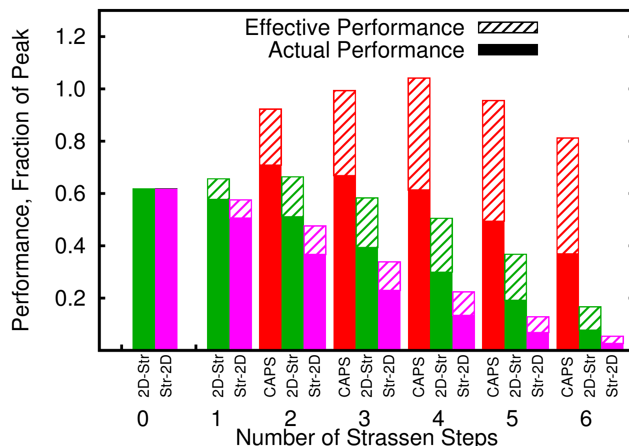


Figure 2.5: Efficiency at various numbers of Strassen steps, $n = 21952$, on 49 nodes (196 cores) of Intrepid. Effective performance is relative to $2n^3$ flops and actual performance is relative to the actual number of flops performed.

2.6 Performance Model

In this section, we introduce a performance model in order to predict performance on a distributed-memory parallel machine. We include a single-node performance model to more accurately represent local computation. The main goals of the performance model are to validate the theoretical analysis of CAPS to real performance, identify areas which might benefit from further optimization, and make predictions for performance on future hardware.

We choose to validate our model on Intrepid because its performance is very consistent (usually less than 1% variation in execution time, versus 10-20% on Hopper) and also because we believe there is opportunity for topology-aware optimizations, which we discuss in Section 2.6.4.

2.6.1 Single Node

Due to the sensitivity of Strassen performance to `DGEMM` performance and the difficulty of modeling `DGEMM` performance accurately for small problems, we use a third degree polynomial of best fit to match the measured time of ESSL's implementation of the classical algorithm (`DGEMM`). Besides making calls to `DGEMM`, Strassen's algorithm consists of performing matrix additions which are communication bound. Thus, we measured the time of `DAXPY` per scalar addition, which is fairly independent of matrix size.

Let $T_{\texttt{DGEMM}}(n)$ be the polynomial for the time cost of classical matrix multiplication of dimension n and $T_{\texttt{DAXPY}}$ be the cost per scalar addition for large vectors. We obtain the single node performance model for the time cost of Strassen's algorithm using s steps of Strassen on a problem of size n as

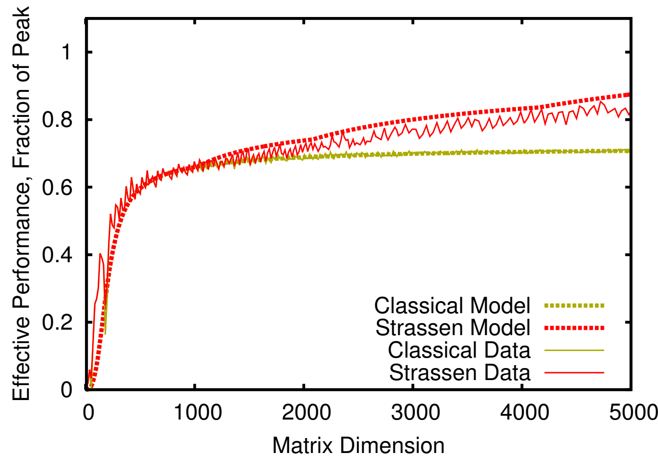


Figure 2.6: Comparison of the sequential model to the actual performance of classical and Strassen matrix multiplication on four cores (one node) of Intrepid.

$$T_{\text{seq}}(n) = \min_s \left\{ 7^s \cdot T_{\texttt{DGEMM}}\left(\frac{n}{2^s}\right) + \sum_{i=0}^{s-1} 9 \cdot T_{\texttt{D AXPY}} \cdot \left(\frac{7}{4}\right)^i \frac{n^2}{4} \right\} \tag{2.6}$$

The constant 9 comes from the fact that in Strassen-Winograd, for each of A and B, four sub-matrices must be read and four written (since three outputs are copies of inputs), and to compute C, seven input matrices must be read and four written; whereas $T_{\texttt{DAXPY}}$ is essentially the time to read two words and write one word. Alternately, one can make 15 calls to `DAXPY`, one for each matrix addition, which yields a constant of 15 but allows the use of a tuned subroutine. We found better performance using `DAXPY` on Intrepid, but with enough optimization, an implementation based on the first approach should be more efficient.

The parameters of our single node model (in seconds) are: $T_{\texttt{DGEMM}}(n) = 2.04 \cdot 10^{-10}n^3 + 2.14 \cdot 10^{-8}n^2 - 4.18 \cdot 10^{-6}n + 2.11 \cdot 10^{-3}$ and $T_{\texttt{DAXPY}} = 3.66 \cdot 10^{-9}$.

We present actual and modeled performance of both classical and Strassen performance on a single node in Figure 2.6. Note that the classical model is nearly indistinguishable from the data in the plot because it is a curve of best fit. By minimizing over s, the model from Equation (2.6) chooses the optimal cutoff point (around $n = 1000$) to switch to the classical algorithm, and the performance of Strassen matches the classical algorithm below that point.

In Figure 2.7 we show a breakdown of time between additions and multiplications (calls to `DGEMM`) for both the model and the actual implementation. For this problem size, the optimal number of Strassen steps is 2, where the time is almost completely dominated by the multiplications. Note that the model predicts better performance for the additions than the implementation achieves, but the main determining factor for optimal number of Strassen steps is the performance of `DGEMM` for the different problem sizes.

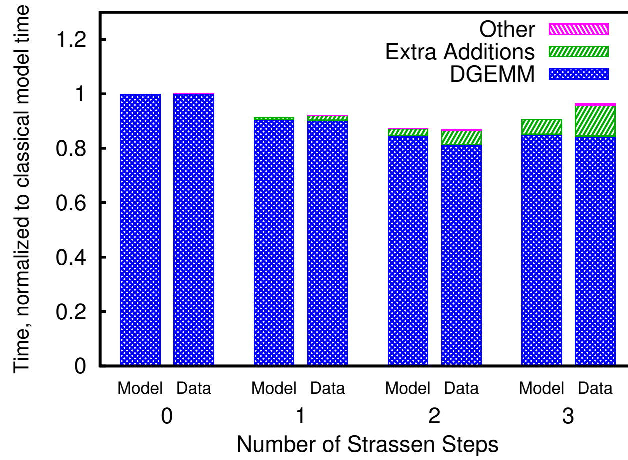


Figure 2.7: Time breakdown comparison between the sequential model and the data for $n = 4097$. Both model and data times are normalized to the modeled classical algorithm time.

2.6.2 Distributed Machine

We start with the conventional (α, β, γ) performance model for a distributed-memory parallel algorithm which uses three machine parameters: α as the latency between any two nodes, β as the inverse bandwidth between any two nodes, and γ as the time cost per flop on a single node [36, 10, 45]. By counting flops f, words w, and messages m along the critical path and summing up the three terms with corresponding coefficients, one can model the time cost of a parallel algorithm as $\alpha m + \beta w + \gamma f$. The main shortcomings of this model are that it assumes an all-to-all network (thus ignoring contention among processors for network links and the number of hops a message must take), it ignores overlap of computation and communication, and it assumes the cost per flop is constant on a node (ignoring on-node communication costs).

To overcome the third shortcoming, we modify the (α, β, γ) model by replacing the γ term with the single node model for the local multiplications (which may include more Strassen steps) and using the measured $T_{\texttt{DAXPY}}$ for the time cost of each scalar addition during the parallel Strassen steps. Then the time spent on computation is

$$T_f(n, P) = \frac{7^{k+\ell}}{P} T_{\text{seq}} \left(\frac{n}{2^{k+\ell}} \right) + \sum_{i=0}^{k+\ell-1} 9 \cdot T_{\texttt{DAXPY}} \cdot \left(\frac{7}{4} \right)^i \frac{n^2}{4P} \tag{2.7}$$

where $k = \log_7 P$ is the number of BFS steps taken, and

$$\ell = \log_2 \left(4 \sqrt{\frac{7n^2}{MP^{2/\omega_0}} - \frac{4n^2}{MP}} \right)$$

is the number of DFS steps necessary to fit in the available memory. Note that the computation is perfectly load balanced so that $T_f(n,P) = \frac{1}{P} \cdot T_{\text{seq}}(n)$. In the model we allow k and ℓ to be real valued to give a continuous function, even though the algorithm only makes sense for integral values.

The number of words and messages are exactly as in [6]:

$$w = \left(\frac{7}{4}\right)^\ell \left(\frac{12n^2}{P^{2/\omega_0}} - \frac{12n^2}{P}\right), \qquad m = 7^\ell 36k.$$

The distributed model is then:

$$T(n,P) = T_f(n,P) + 7^\ell 36k\alpha + \left(\frac{7}{4}\right)^\ell \left(\frac{12n^2}{P^{2/\omega_0}} - \frac{12n^2}{P}\right)\beta.$$

The parameters of the distributed model are $\beta = 2.13 \cdot 10^{-8}$ and $\alpha = 2 \cdot 10^{-6}$, measured in seconds.

We present actual and modeled strong scaling performance of CAPS, 2D and 2.5D in Figure 2.8 (see Appendix A in [60] for the classical performance model). The CAPS performance and model match quite well up to about 4116 cores, but for runs on more cores the actual performance drops significantly below the predictions of the model. We believe this is due to contention; we consider optimizing CAPS to a 3D-torus network in Section 2.6.4. The model also allows us to break down the time into communication time (the α and β terms), time spent in calls to `DGEMM` (the first term in Equation 2.7), and time spent in additions (the second term in Equation 2.7). We compare these times to the actual time breakdown (averaged over processors) in Figure 2.9. The model works well for small values of P, but understates the communication cost for large values of P, due to contention. In fact, at P=49, the communication is slightly faster than predicted by the model, which is possible because the model counts bandwidth along only one direction on one of the six links to a given node, and ignores communication hiding.

2.6.3 Exascale Predictions

We model performance on a hypothetical exascale machine by counting words communicated on the network, words transferred between DRAM and cache, and flops computed per processor. For the machine parameters, we use values from the 2018 Swimlane 1 extrapolation in [58]:

Number of nodes	2^{20}
Flop rate per node	1 Tflop/s
Cache size per node	512 MB
DRAM per node	32 GB
Node memory bandwidth	0.4 TB/s
Network link bandwidth	20 GB/s
Network latency	1 μs

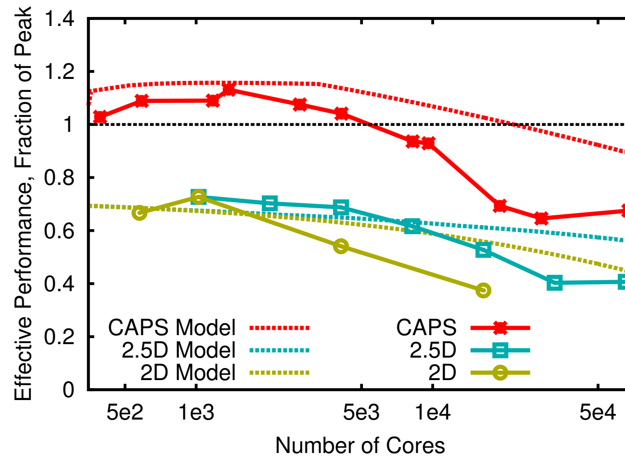


Figure 2.8: Comparison of the parallel model with the algorithms in strong scaling of $n = 65856$ on Intrepid.

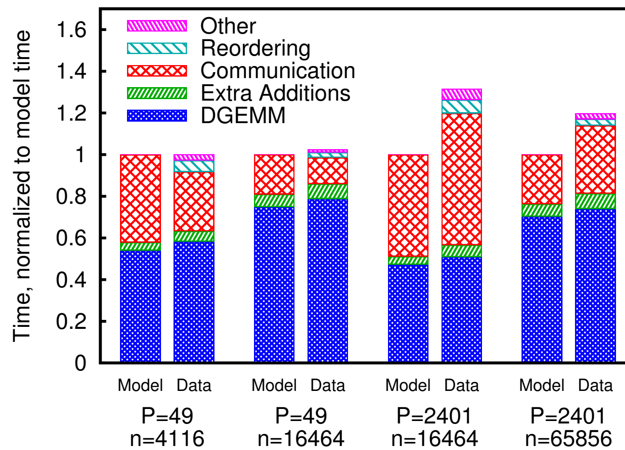


Figure 2.9: Time breakdown comparison between the parallel model and the data. In each case the entire modeled execution time is normalized to 1.

The projected speedups of CAPS over 2.5D and 2D are shown in Figure 2.10. The horizontal scale is the (log of the) number of nodes, and the vertical scale is the (log of the) amount of memory per node used to store a single matrix. Thus moving horizontally in the plot corresponds to weak scaling, and moving diagonally downward corresponds to strong scaling. Compared to 2.5D, our largest speedup is 5.45× at the top-right of the plot: very large matrices run using the entire machine. Although CAPS communicates asymptotically less than 2.5D, the advantage is very slight, and the constants for CAPS are larger than for 2.5D in our model. For small problems (bottom of the figure), CAPS is slightly faster when using the entire machine but slower for fewer processors. Comparing to 2D, which communicates

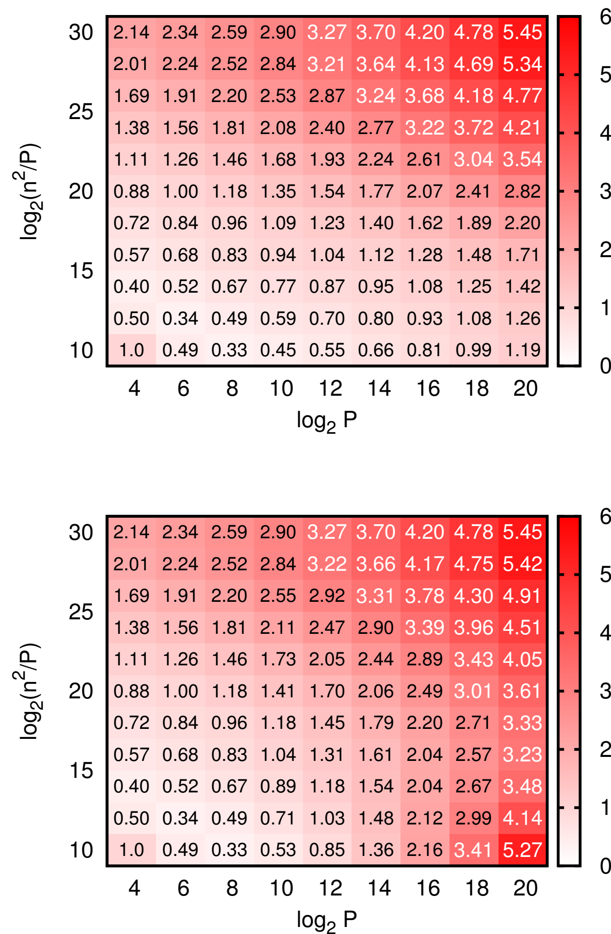


Figure 2.10: Predicted speedups of CAPS over 2.5D (top) and 2D (bottom) on an exascale machine.

much more for small problems, there are substantial speedups of 5.45× in the top right, and 5.27× in the communication-bound regime at the bottom right.

2.6.4 Areas of possible performance improvement

Based on our performance models and benchmarks, we believe there are several areas in which further performance optimizations will be effective. First, since local computation dominates the execution time for many problems, improving the on-node performance of Strassen can help overall. By writing more efficient addition code which exploits the shared operands and decreases reads from DRAM, we believe it is possible to match our modeled on-node performance (an improvement of around 10%). Further improving the performance of `DGEMM` for small problems would also boost on-node Strassen performance. If the performance curve

for the classical algorithm reaches its peak for smaller matrices, then the cutoff point can be decreased; more Strassen steps implies greater computational savings, so the effective performance will be improved for large matrices (using one more Strassen step can improve performance up to about 14% since it only does 7 instead of 8 multiplications on matrices of half the size).

Second, we believe there are important topology-aware optimization possibilities. On Intrepid, where the topology is known, one can map processors to nodes in order to minimize contention and also maximize the use of a node's links in each of the three dimensions, as in [61]. In most cases we achieve the best performance by laying out 7 processes onto 7 of the 8 nodes in a $2 \times 2 \times 2$ cube, and then recursively using this layout for higher powers of 7. Another natural mapping is to place the 7^k processes in a k-dimensional grid so that the communication occurs only in disjoint pencils (lines of 7 axis-aligned processors). The contention will then never be worse than for 7 processors communicating around a ring, although only $1/k$ of the links will be active at any time. On Intrepid this works for $k \leq 3$ since it has a 3-dimensional topology ($k = 3$ implies $1372 = 4 \cdot 7^3$ cores).

A more systematic approach of finding optimal mappings may yield significant improvements. Avoiding contention completely would enable performance to match the performance model (an improvement of around 30% for large P). Since the model is based on one link's bandwidth, optimizing the mapping to take advantage of multiple links can yield performance which exceeds the model. For small matrices and communication-bound problems, this can lead to significant performance improvements.

Our implementation is somewhat sensitive to matrix dimension and number of processors. There are many optimizations which could help smooth the performance curve for arbitrary n and P which we did not consider in this work.

2.7 Implementation Details

The implementation of CAPS follows the algorithm presented in Section 2.3. This section fills in several of the details of the implementation.

2.7.1 Base-case Multiplication

Until now, we have assumed that Strassen will be performed all the way down to 1×1 matrices. In practice, however, it is faster to use a classical matrix multiplication algorithm on sizes below some threshold. See section 2.6 for a discussion of what constitutes a reasonable threshold.

So that the base-case doesn't increase the communication cost, we demand that all base-case matrix multiplies are local calls to `DGEMM`. This means that we must take exactly $k = \log_7 P$ BFS steps, which requires that there are at least k Strassen steps. This requirement is not very demanding, and in practice any matrix size that scales well to P processors will benefit from more than k Strassen steps. We count on the `DGEMM` implementation to provide

good shared-memory parallel performance, which is reasonable on today's computers, but may not be as the number of cores on each node increases.

2.7.2 Running on $P = m \cdot 7^k$ Processors

In this section, we generalize the assumption that the number of processors is exactly a power of 7. This assumption is not realistic in practice, and if we set P to be the largest power of 7 no larger than a given allocation, we might lose up to a factor of 7 in performance, making Strassen slower than classical matrix multiplication in many cases. As shown in Section 2.7.3, making the algorithm more practical and capable of running on $m \cdot 7^k$ processors does not sacrifice the theoretical communication optimality. Figure 2.4 shows that actual performances with these generalizations is comparable.

If we take $P = m \cdot 7^k$, then after k BFS steps (and perhaps some DFS steps so there is enough memory), the problem is reduced to multiplying smaller matrices on $P = m$ processors. We have implemented two schemes for Strassen in such cases: perform either DFS steps or what we call *hybrid* steps, followed by a distributed classical matrix multiplication at the base case. In our implementation the classical multiplication uses a 1D processor grid, which performs well for small m.

Using only DFS steps, the number of words communicated grows by a factor of 7/4 for every DFS step. If more than one or two DFS steps are taken the increase in the communication cost can be too large. If too few Strassen steps are taken we may miss the arithmetic savings that they can provide. The situation is analogous to that of 2D-Strassen.

The alternative is a hybrid step on $1 < m < 7$ processors. In a hybrid step, the 7 matrix multiplies of a Strassen step are performed locally in groups of m, and any leftovers are run on all m processors. For example if $m = 2$ then 3 of the 7 multiplications are performed locally on each processor, and the remaining one is performed on both processors. Using hybrid steps recursively, most of the subproblems are computed locally by one processor, and so there is a lower communication cost.

In practice, the choice between hybrid steps and DFS steps on m processors is best regarded as a tuning parameter. Hybrid steps are provably optimal (see Section 2.7.3), but the extra communication from DFS steps overlaps more easily with the calls to `DGEMM` (see Section 2.7.6).

2.7.3 Optimality of hybrid steps

In this section we prove that CAPS running on $P = m \cdot 7^k$ using hybrid steps (as defined in Section 2.7.2) is communication optimal, up to a constant factor, if m is regarded as a constant. Given the optimality of CAPS using BFS and DFS steps proved in [6], we need only consider the case $P = m$.

Theorem 2.4. *Performing Strassen's matrix multiplication using hybrid steps on $m = 2, 3, 4, 5, 6$ processors communicates $O(n^2)$ words and requires $O(n^2)$ memory. Combined with*

the lower bounds of [9], this shows that the algorithm is communication optimal.

Proof. The bandwidth cost recurrence for a hybrid step on m processors is $W(n,m) = O(n^2) + \left(7 - m\left\lfloor\frac{7}{m}\right\rfloor\right) W\left(\frac{n}{2}, m\right)$, where the first term is the words communicated to redistribute the first $m\lfloor 7/m\rfloor$ subproblems to the m processors, and the second term is the words required to compute the remaining subproblems in parallel. Note that for $m = 2, 3, 4, 5, 6$, we have $7 - m\lfloor 7/m\rfloor < 4$, and so the solution to this recurrence is $W(n,m) = O(n^2)$. Further, the extra memory used by the algorithm is simply the amount of memory used to store the data each processor receives, and so the memory usage is also $M = O(n^2)$. □

It is similarly possible to show that the algorithm is communication-optimal, up to a constant factor, for any constant m that has prime factors $2, 3, 5$, by recursively using hybrid steps. However the constant in the communication cost grows with m, and so it is probably not practical to run on $P = m \cdot 7^k$ processors for m much larger than 6. In general, given a number of processors that is not a power of 7, the choice of how many processors to ignore and how large to allow m to be is left to tuning.

2.7.4 Interleaving BFS and DFS steps

As argued in [6], it is possible to achieve the bandwidth lower bound, up to a constant factor, using only a simple scheme of ℓ DFS steps, followed by $k = \log_7 P$ BFS steps, followed by local Strassen. Our implementation allows arbitrary interleaving of BFS and DFS steps, which in some cases provides a reduction in the bandwidth costs. Computation of the optimal interleaving patterns can be done once, offline, for each value of k.

For example, when running on $16807 = 7^5$ processors, the simple interleaving patterns are all optimal for certain memory sizes. However for intermediate memory sizes it is possible to reduce the volume of communication by up to about 25% by choosing a different interleaving; see Figure 2.11.

2.7.5 Data Layout

The data layout can be naturally divided into two levels: the global data layout specifies which process owns each part of each matrix, and the local data layout specifies in what order the data is stored in the local memory of a given process. For the global layout with 7^k processors, we use a cyclic distribution with a processor grid of size $7^{\lfloor k/2\rfloor} \times 7^{\lceil k/2\rceil}$. Note that this satisfies the properties given in Section 3.2 of [6]. Thus the communication cost analysis given there holds no matter what choice we make for the local data layout. Additionally, transformations between different local layouts can be done quickly and without any inter-processor communication. The local layout we choose is that blocks of size $\frac{n}{2^s 7^{\lfloor k/2\rfloor}} \times \frac{n}{2^s 7^{\lceil k/2\rceil}}$ are stored contiguously, and these blocks are ordered relative to each other following recursive И-Morton ordering [54].

The entire layout can also be thought of as s levels of recursive Morton ordering, followed by cyclic layout in each of the sub-matrices of size $\frac{n}{2^s}$. We choose Morton ordering because it

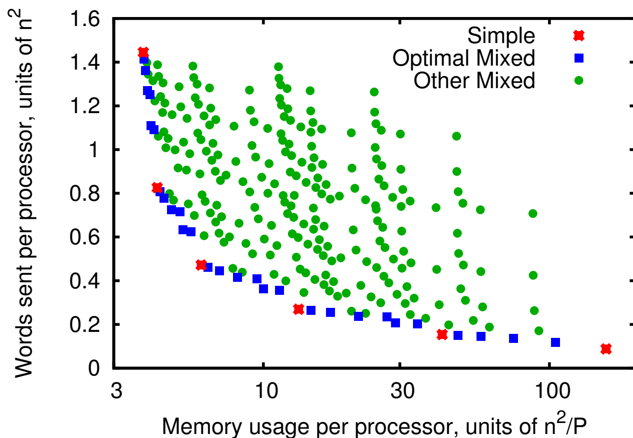


Figure 2.11: The memory and communication costs of all 252 possible interleavings of BFS and DFS steps for multiplying matrices of size n=351232 on P=16807 processors using 10 Strassen steps. The optimal ones show the trade-off between memory size and communication cost. Simple interleavings are those for which all k BFS steps are performed as a block.

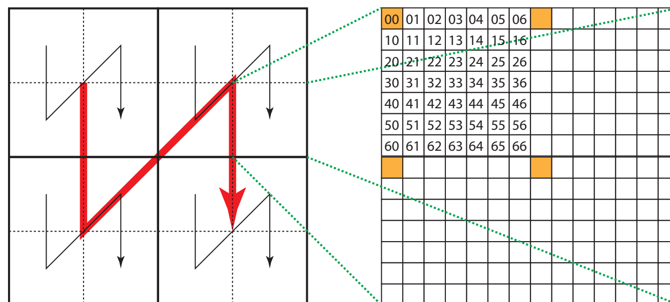


Figure 2.12: Data layout: s levels of Morton ordering are used at the top level, block-cyclic distribution is used at the bottom. Numbers correspond to processors in base 7.

is a very good fit to Strassen's algorithm both conceptually and to enhance locality [1]. Since we choose to pack messages together to minimize the number of message sent, it is necessary to re-order the data for each communication step to maintain this data layout.

If the matrix dimension is a multiple of $2^s 7^{\lfloor k/2 \rfloor}$, this layout is the same as cyclic layout. Thus it is compatible, up to local re-ordering, with the block-cyclic layout of ScaLAPACK in this case if the block size is one. Moreover, CAPS can work with any block size that divides $\frac{n}{2^s 7^{\lceil k/2 \rceil}}$, after local re-arrangement. If matrices are stored in another layout, re-arranging them to the layout that CAPS uses would only increase the bandwidth cost of the multiplication by a subleading term.

2.7.6 Overlapping Computation and Communication

We attempt to overlap computation and communication as long as it can be done without breaking the recursive structure of the algorithm. First, during BFS steps the additions are overlapped with the communication. For Strassen-Winograd, 6 of the 14 factors require no computation, so the additions for the other 8 can be done while those are transferred. We do not attempt to overlap the communication of the seven products with the additions to convert them into entries of C, because it is not clear how to do this without degrading cache performance. Second, for base-case multiplies with $m > 1$, we overlap the communication with the calls to `DGEMM`. Finally, there is some overlap in hybrid BFS steps, where the details of how much we can overlap depend on the exact value of m.

In principle, it should be possible to hide more of the communication cost, ideally by performing some `DGEMM` calls during the communication of each BFS step. However these `DGEMM` calls only appear deeper in the recursion tree of the algorithm, so to do this would require breaking the recursive structure of the algorithm (that is, breaking one recursive call up into several parts, which are performed when the data they require has been communicated).

2.8 Numerical Stability

CAPS has the same stability properties as sequential versions of Strassen. For a complete discussion of the stability of fast matrix multiplication algorithms, see [41, 30]. We highlight a few main points here. The tightest error bounds for classical matrix multiplication are component-wise: $|C - \hat{C}| \leq n\epsilon|A| \cdot |B|$, where $\hat{C}$ is the computed result and ϵ is the machine precision. Strassen and other fast algorithms do not satisfy component-wise bounds but do satisfy the slightly weaker norm-wise bounds: $\|C - \hat{C}\| \leq f(n)\epsilon\|A\|\|B\|$, where $\|A\| = \max_{i,j} A_{ij}$ and f is polynomial in n [41]. Accuracy can be improved with the use of diagonal scaling matrices: $D_1CD_3 = D_1AD_2 \cdot D_2^{-1}BD_3$. It is possible to choose D_1, D_2, D_3 so that the error bounds satisfy either $|C_{ij} - \hat{C}_{ij}| \leq f(n)\epsilon\|A(i,:)\|\|B(:,j)\|$ or $\|C - \hat{C}\| \leq f(n)\epsilon\||A| \cdot |B|\|$. By scaling, the error bounds on Strassen become comparable to those of many other dense linear algebra algorithms, such as LU and QR decomposition [29]. Thus using Strassen for the matrix multiplications in a larger computation will often not harm the stability at all.

Using fewer than $\log_2 n$ Strassen steps improves the theoretical constant in the error bound. More precisely, using s Strassen steps, the error bound for Strassen-Winograd given in [41] is

$$\|C - \hat{C}\|_{\max} \leq \left(18^s\left(\left(\frac{n}{2^s}\right)^2 + \frac{6n}{2^s}\right) - 6n\right)\|A\|_{\max}\|B\|_{\max}\epsilon$$

where ϵ is the machine precision and $\|A\|_{\max} := \max_{i,j}|A_{ij}|$ is the max-norm of A.

However, as illustrated in [41], this theoretical bound is too pessimistic. In Figure 2.13, we show the measured max-norm absolute error compared to the theoretical bound for a single matrix of size $n = 16384$ in double precision where each entry is chosen uniformly at

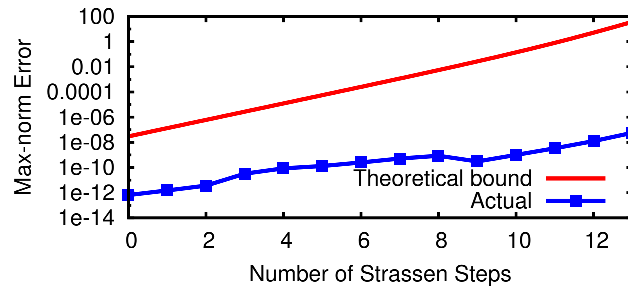


Figure 2.13: Stability test: theoretical error bound versus actual error for n=16384. Zero Strassen steps corresponds to the classical algorithm. Double precision machine epsilon is $2.2 \cdot 10^{-16}$.

random from $[-1, 1]$, varying the number of Strassen steps taken. For the "exact" answer we compute the product in quadruple precision. To maximize performance on a single node of Hopper or Intrepid, for example, the optimal number of Strassen steps for $n = 16834$ is 4, where the result loses about two decimal digits (measured by norm-wise error) compared to classical matrix multiplication.

2.9 Parallelizing Other Fast Matrix Multiplication Algorithms

Our approach of executing a recursive algorithm in parallel by traversing the recursion tree in DFS or BFS manner is not limited to Strassen's algorithm. Recursive matrix multiplication algorithms are typically built out of ways to multiply $n_0 \times n_0$ matrices using $q < n_0^3$ multiplications. As with Strassen and Strassen-Winograd, they compute q linear combinations of entries of each of A and B, multiply these pairwise, then compute the entries of C as linear combinations of these. CAPS can be easily generalized to any such multiplication, with the following modifications:

- The number of processors P is a power of q.
- The data layout must be such that all n_0^2 blocks of A, B, and C are distributed equally among the P processors with the same layout.
- The BFS and DFS determine whether the q multiplications are performed simultaneously or in sequence.

The communication costs are then exactly as above, but with $\omega_0 = \log_{n_0} q$. One special case is the classical square recursive algorithm that splits A, B, and C each into 4 square blocks and solves the problem with 8 multiplications of those blocks. Generalizing CAPS to this

	Square	Rectangular
Classical	$\frac{n^2}{P^{2/3}}$	$\min\left\{n_0^t p_0^t, \sqrt{\frac{m_0^t n_0^t p_0^{2t}}{P}}, \left(\frac{m_0^t n_0^t p_0^t}{P}\right)^{2/3}\right\}$
Fast	$\frac{n^2}{P^{2/\omega_0}}$	$\frac{m_0^t n_0^t}{P^{\log_q m_0 n_0}}$

Table 2.3: Asymptotic bandwidth costs of classical and fast, square and rectangular matrix multiplication in the case of unlimited memory. See the text for a description of the fast rectangular algorithm and problem size.

algorithm gives a communication-optimal classical square matrix multiplication algorithm with the same asymptotic costs as the 2.5D algorithm [31].

Using the techniques from Theorem 3.3 of [30], one can convert any fast square matrix multiplication algorithm into one that is nearly as fast and is of the form above. Using the CAPS parallelization approach this gives a communication-avoiding parallel algorithm corresponding to any fast matrix multiplication algorithm. We conjecture that there is a matching lower bound, making these algorithms optimal.

CAPS can also be generalized to fast rectangular matrix multiplication algorithms (see [42, 14, 27, 50, 44, 43, 28] and further details in [20]), which are built out of a base case for multiplying an $m_0 \times n_0$ matrix A with an $n_0 \times p_0$ matrix B to obtain an $m_0 \times p_0$ matrix C using only $q < m_0 n_0 p_0$ scalar multiplications. As with the square case, a lower bound is known for some of these algorithms [7], and its generalization is a conjecture. Implementing and benchmarking the BFS/DFS approach on any of these algorithms remains to be done.

It is instructive to compare the communication costs of fast rectangular matrix multiplication with those of the communication-optimal classical algorithm that is presented in Chapter 3. Consider a fast algorithm as above, with $m_0 \geq n_0 \geq p_0$ applied t times to multiply matrices with dimensions $m_0^t \geq n_0^t \geq p_0^t$ in the unlimited memory regime. The bandwidth costs of the generalization of CAPS and of CARMA (see Chapter 3) in this situation are shown in Table 2.3.

Note that since $p_0 \leq m_0, n_0$,

$$q < m_0 n_0 p_0 \leq (m_0 n_0)^{3/2},$$

and so the exponent of P in the denominator in the fast rectangular case is $\log_q m_0 n_0 > 2/3$. This means that for any given problem, on sufficiently many processors, any fast algorithm will communicate less than the classical algorithm. On a small number of processors, however, fast rectangular algorithms may communicate more than the classical algorithm because fast algorithms require the largest matrix to be communicated, whereas, depending on the number of processors, the classical algorithm may not.

Chapter 3

Classical Rectangular Matrix Multiplication

In this chapter we focus on the classical (three nested loops) matrix multiplication algorithm. For square matrices, the CAPS algorithm immediately generalizes to give an optimal algorithm as described in Section 2.9. However for rectangular matrices, there are additional complications. Using the classical algorithm, there are several ways to split up the problem into smaller matrix multiplications. To obtain a communication-optimal algorithm, it is necessary to combine three of these, corresponding to splitting each of the three dimensions, as illustrated in Figure 3.1.

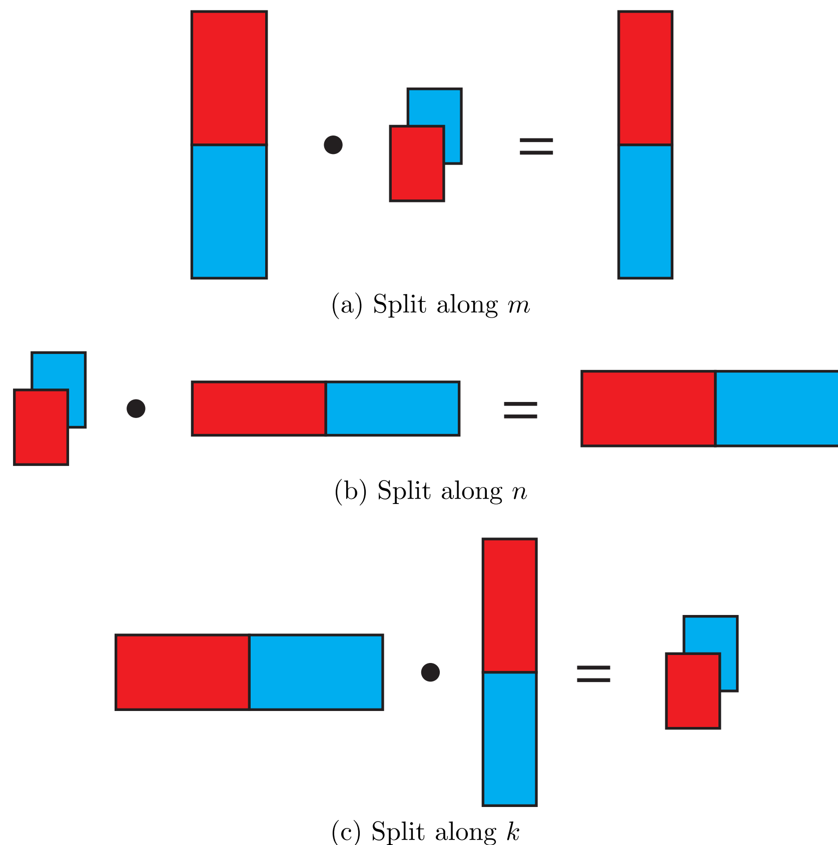
For the sequential case, it is shown in [34] that splitting the largest dimension at each recursive step asymptotically minimizes the communication costs. We apply the BFS/DFS approach to the dimension-splitting recursive algorithm to obtain a communication-avoiding recursive matrix multiplication algorithm, *CARMA*, which is asymptotically communication-optimal for any matrix dimensions, number of processors, and memory size. CARMA is a simple algorithm. However, because it is optimal across the entire range of inputs, we find cases where it significantly outperforms ScaLAPACK.

The basic idea of CARMA is that at each recursive step, the largest of the three dimensions is split in half, yielding two subproblems. Depending on the available memory, these subproblems are solved by either a BFS step or a DFS step. A simplified version of its pseudocode appears as Algorithm 3 (for more details, see Algorithm 4).

The results in this chapter are joint work with James Demmel, David Eliahu, Armando Fox, Shoaib Kamil, Oded Schwartz and Omer Spillinger, and appear in [31].

Notation

We consider the case of computing $C = AB$ where A is an $m \times k$ matrix, B is a $k \times n$ matrix, and C is an $m \times n$ matrix. In the next two subsections, it will be convenient to have an ordered notation for the three dimensions. Hence we define $d_1 = \min(m, n, k)$, $d_2 = \text{median}(m, n, k)$, and $d_3 = \max(m, n, k)$. Both the lower bounds and the communication



(a) Split along m

(b) Split along n

(c) Split along k

Figure 3.1: Three ways to split rectangular matrix multiplication to get two subproblems, shown in red and blue. In each case, two of the three matrices are split between the subproblems, and one is needed for both subproblems. The matrix that is needed for both subproblems is shown as two overlapping copies.

Algorithm 3 CARMA, in brief

Input: A is an $m \times k$ matrix, B is a $k \times n$ matrix
Output: $C = AB$ is $m \times n$

1: Split the largest of m, n, k in half, giving two subproblems
2: **if** Enough memory **then**
3: Solve the two problems recursively with a BFS
4: **else**
5: Solve the two problems recursively with a DFS

costs of CARMA depend only on the values of the three dimensions, not on their order. Similarly, let M_3, M_2, M_1 be the three matrices in increasing size, so M_3 has dimensions d_1, d_2, M_2 has dimensions d_1, d_3, and M_1 has dimensions d_2, d_3.

3.1 Communication Lower Bounds

To prove lower bounds on the communication costs, we will assume that the initial/final data layout of each matrix consists of one copy of the matrix load balanced among the P processors. Recall the arrangement of the mnk scalar products that must be computed into a rectangular prism from Section 1.4. At least one processor will perform at least $\frac{mnk}{P}$ multiplications. Consider such a processor, and let V be the set of voxels corresponding to the multiplications it performs. Let $|V_i|$ be the size of the projection of V onto matrix M_i. The Loomis-Whitney inequality [49] gives a lower bound on the product of these projections:

$$|V_1| \cdot |V_2| \cdot |V_3| \geq \left(\frac{d_1 d_2 d_3}{P}\right)^2 \tag{3.1}$$

There are three cases to consider, depending on the aspect ratio of the matrices (see Figure 3.2 for graphical representations of the cases).

One large dimension

First, consider the case of one very large dimension, so that

$$2\frac{d_3}{d_2} > P.$$

We consider three possibilities depending on the sizes of $|V_1|$ and $|V_2|$.

If $|V_1| \geq \frac{5d_2d_3}{4P}$, then the processor needs access to at least this many entries of M_1. We assumed that the input and output data layouts are exactly load balanced, so at most $\frac{d_2d_3}{P}$ of these entries are stored by that processor in the initial/final data layout. As a result, the remaining $\frac{d_2d_3}{4P}$ must be communicated by that processor.

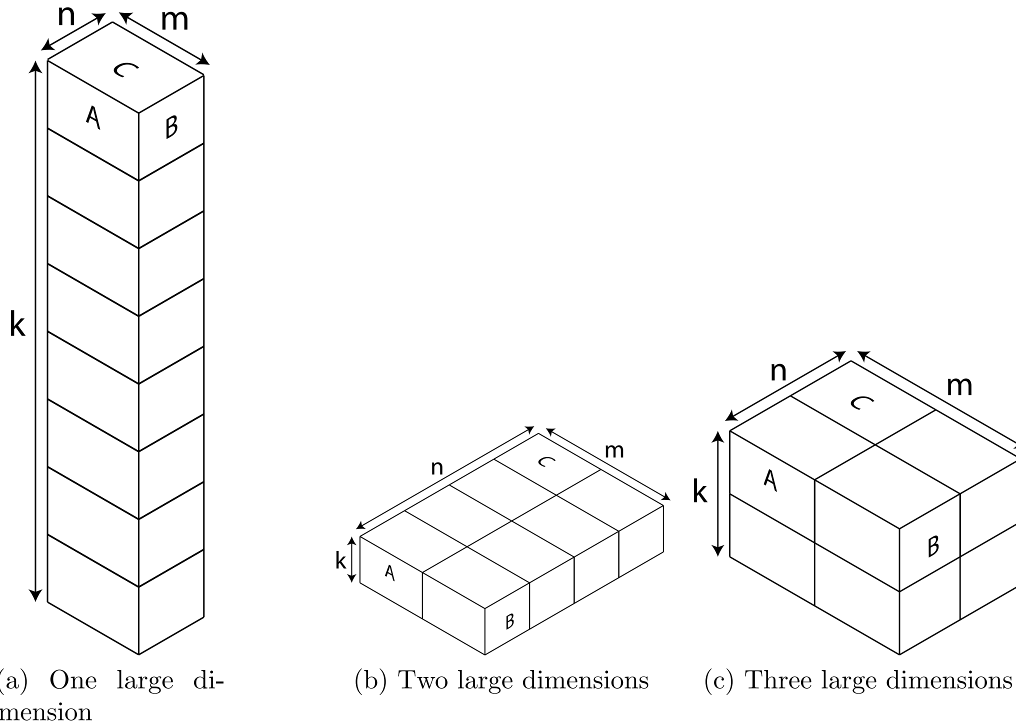
Similarly, if $|V_2| \geq \frac{5d_1d_3}{4P}$, then the processor needs access to at least this many entries of M_2. We assumed that the input and output data layouts are exactly load balanced, so at most $\frac{d_1d_3}{P}$ of these entries are stored by that processor in the initial/final data layout. As a result, the remaining $\frac{d_1d_3}{4P}$ must be communicated by that processor.

If $|V_1| < \frac{5d_2d_3}{4P}$ and $|V_2| < \frac{5d_1d_3}{4P}$, we may substitute into Inequality 3.1 to obtain

$$|V_3| \geq \frac{16}{25}d_1d_2.$$

Since M_3 is load balanced, only $\frac{d_1d_2}{P}$ of these entries can be owned by the processor in the initial/final data layout, so for $P \geq 2$, at least

$$W \geq \frac{7}{50}d_1d_2 = \Omega(d_1d_2)$$



(a) One large dimension (b) Two large dimensions (c) Three large dimensions

Figure 3.2: Examples of the three cases of aspect ratios on $P = 8$ processors. The lowest communication cost is attained if an algorithm divides the $m \times n \times k$ prism of multiplications into 8 equal sub-prisms that are as close to cubical as possible. If that division involves dividing only one of the dimensions, we call that case *one large dimension*. If it involves dividing only two of the dimensions, we call that case *two large dimensions*. If all three dimensions are divided, we call that case *three large dimensions*. Note that in the first two cases, only one processor needs access to any given entry of the largest matrix, so with the right data layout an algorithm only needs to transfer the smaller matrices.

words must be communicated. The lower bound is the minimum of these three possibilities:

$$W = \Omega\left(\min\left\{d_1 d_2, \frac{d_2 d_3}{P}, \frac{d_1 d_3}{P}\right\}\right).$$

By the assumption that $2\frac{d_3}{d_2} > P$, this simplifies to

$$W = \Omega\left(d_1 d_2\right). \tag{3.2}$$

Since this lower bound depends only on the size of the smallest matrix, it is only attainable if the two larger matrices are distributed such that each processor owns corresponding entries of them.

The latency lower bound is the trivial one that there must be at least one message: $L = \Omega(1)$.

Two large dimensions

Next consider the case that

$$2\frac{d_2 d_3}{d_1^2} > P \geq 2\frac{d_3}{d_2}.$$

We consider two posibilities depending on the size of $|V_1|$.

If $|V_1| \geq \frac{3d_2d_3}{2P}$, then the processor needs access to at least this many entries of M_1. We assumed that the input and output data layouts are exactly load balanced, so at most $\frac{d_2d_3}{P}$ of these entries are stored by that processor in the initial/final data layout. As a result, the remaining $\frac{d_2d_3}{2P}$ must be communicated by that processor.

If $|V_1| < \frac{3d_2d_3}{2P}$, we may substitute for $|V_1|$ in Inequality 3.1 to obtain

$$|V_2| \cdot |V_3| \geq \frac{2d_1^2 d_2 d_3}{3P}. \tag{3.3}$$

It follows that

$$\max\{|V_2|, |V_3|\} \geq \sqrt{\frac{2}{3}}\sqrt{\frac{d_1^2 d_2 d_3}{P}}.$$

The amount of data a processor stores of M_2 or M_3 in the initial/final layout is at most $\frac{d_1d_3}{P}$ (recall that $d_1 \leq d_2 \leq d_3$), thus we obtain a bandwidth lower bound of

$$W \geq \sqrt{\frac{2}{3}}\sqrt{\frac{d_1^2 d_2 d_3}{P}} - \frac{d_1 d_3}{P}.$$

By the assumption that $P \geq 2\frac{d_3}{d_2}$, the first term dominates and this simplifies to

$$W \geq \left(\sqrt{\frac{2}{3}} - \frac{1}{\sqrt{2}}\right)\sqrt{\frac{d_1^2 d_2 d_3}{P}} = \Omega\left(\sqrt{\frac{d_1^2 d_2 d_3}{P}}\right).$$

The lower bound is the minimum of these two possibilities:

$$W = \Omega\left(\min\left\{\sqrt{\frac{d_1^2 d_2 d_3}{P}}, \frac{d_2 d_3}{P}\right\}\right).$$

By the assumption that $\frac{d_2d_3}{d_1^2} > P$, this simplifies to

$$W = \Omega\left(\sqrt{\frac{d_1^2 d_2 d_3}{P}}\right). \tag{3.4}$$

Note that this lower bound is only attainable if the largest matrix is distributed among the processors in such a way that it doesn't need to be communicated.

The latency lower bound is the trivial one that there must be at least one message: $L = \Omega(1)$.

Three large dimensions

If

$$P \geq 2\frac{d_2 d_3}{d_1^2},$$

then by Theorem 3.1 of [45], we have

$$W \geq \frac{d_1 d_2 d_3}{2\sqrt{2}P\sqrt{M}} - M.$$

In the case that

$$M \leq \left(\frac{d_1 d_2 d_3}{4P}\right)^{2/3},$$

this bound can be simplified to

$$W = \Omega\left(\frac{d_1 d_2 d_3}{P\sqrt{M}}\right). \tag{3.5}$$

Additionally, following the methods of [5], Inequality 3.1 implies that

$$\max\{|V_1|, |V_2|, |V_3|\} \geq \left(\frac{d_1 d_2 d_3}{P}\right)^{2/3}.$$

By the assumption of load balance, the amount of data available to one processor in the initial/final layout of any of the matrices is at most $\frac{d_2 d_3}{P}$ (recall that $d_1 \leq d_2 \leq d_3$), so this corresponds to a bandwidth cost of at least

$$W \geq \left(\frac{d_1 d_2 d_3}{P}\right)^{2/3} - \frac{d_2 d_3}{P}.$$

By the assumption that $P \geq 2\frac{d_2 d_3}{d_1^2}$, the first term dominates, and

$$W = \Omega\left(\left(\frac{d_1 d_2 d_3}{P}\right)^{2/3}\right). \tag{3.6}$$

Equation 3.5 only applies for $M \leq \left(\frac{d_1 d_2 d_3}{4P}\right)^{2/3}$, but for larger M, the bound in Equation 3.6 dominates it. Thus the lower bound may be concisely expressed as their sum:

$$W = \Omega\left(\frac{d_1 d_2 d_3}{P\sqrt{M}} + \left(\frac{d_1 d_2 d_3}{P}\right)^{2/3}\right). \tag{3.7}$$

This bound is attainable for any load balanced data layout, since it is larger than the cost of $\frac{d_2 d_3}{P}$ words to redistribute the data.

	$P < \frac{d_3}{d_2}$ "1 large dimension"	$\frac{d_3}{d_2} < P < \frac{d_2 d_3}{d_1^2}$ "2 large dimensions"	$\frac{d_2 d_3}{d_1^2} < P$ "3 large dimensions"
Lower Bound [45, 31]	$d_1 d_2$	$\sqrt{\frac{d_1^2 d_2 d_3}{P}}$	$\frac{d_1 d_2 d_3}{P\sqrt{M}} + \left(\frac{d_1 d_2 d_3}{P}\right)^{2/3}$
2D SUMMA [2, 36]	$\sqrt{\frac{d_1^2 d_2 d_3}{P}}$	$\sqrt{\frac{d_1^2 d_2 d_3}{P}}$	$\sqrt{\frac{d_1^2 d_2 d_3}{P}}$
3D SUMMA [57]	$\sqrt{\frac{d_1^2 d_2 d_3}{P}}$	$\sqrt{\frac{d_1^2 d_2 d_3}{P}}$	$\frac{d_1 d_2 d_3}{P\sqrt{M}} + \left(\frac{d_1 d_2 d_3}{P}\right)^{2/3}$
CARMA [31]	$d_1 d_2$	$\sqrt{\frac{d_1^2 d_2 d_3}{P}}$	$\frac{d_1 d_2 d_3}{P\sqrt{M}} + \left(\frac{d_1 d_2 d_3}{P}\right)^{2/3}$

Table 3.1: Asymptotic bandwidth costs and lower bounds for rectangular matrix multiplication on P processors, each with local memory size M, where the matrix dimensions are $d_1 \leq d_2 \leq d_3$. 2D SUMMA and 3D SUMMA have the variant and processor grid chosen to minimize the bandwidth cost.

The latency lower bound is the combination of two trivial bounds: at least one message must be sent, and the maximum message size is M. Therefore the number of messages is:

$$L = \Omega\left(\frac{d_1 d_2 d_3}{PM^{3/2}} + 1\right)$$

Note that whenever $P = \Theta\left(\frac{d_2 d_3}{d_1^2}\right)$, Equations 3.7 and 3.4 give the same bound. Similarly, whenever $P = \Theta\left(\frac{d_3}{d_2}\right)$, Equations 3.4 and 3.2 give the same bound. We may thus drop the factors of 2 in the definitions of one, two, and three large dimensions.

3.2 CARMA Algorithm

Detailed pseudocode for CARMA is shown in Algorithm 4 on page 46. The recursion cuts the largest dimension in half to give two smaller subproblems. At each level of the recursion in CARMA, a decision is made between making a depth-first step (DFS) or a breadth-first step (BFS) to solve the two subproblems. A BFS step consists of two disjoint subsets of processors working independently on the two subproblems in parallel. In contrast, a DFS step consists of all processors in the current subset working on each subproblem in sequence. A BFS step increases memory usage by a constant factor, but decreases future communication costs. On the other hand, a DFS step decreases future memory usage by a constant factor, but increases future communication costs. On P processors, with unlimited memory, we show that the algorithm only needs BFS steps and is communication-optimal.

If the execution of only BFS steps causes the memory requirements to surpass the bounds of available memory, it is necessary to interleave DFS steps within the BFS steps to limit memory usage. We show that the resulting algorithm is still communication-optimal, provided the minimal number of DFS steps is taken. As we show in the analysis in Section 3.2.1, at

most a factor of $3\times$ extra memory is needed in the cases of one or two large dimensions, so DFS steps are not necessary in these cases. When all dimensions are large, the extra memory requirement may grow asymptotically. Therefore the memory may become a limiting resource in that case and DFS steps may be necessary. In our experiments, we used relatively small matrices on many processors so that the problem is communication-bound rather than computation-bound. As a result, the matrices are small enough that memory was not a limiting factor, and only BFS steps were used.

See Section 3.4.3 and Figure 3.4 for a description of the data layout in the distributed-memory case.

3.2.1 Communication cost of CARMA

CARMA will perform a total of $\log_2 P$ BFS steps, possibly with some DFS steps interleaved. There are again three cases to consider. In each case, CARMA attains the bandwidth lower bound up to a constant factor, and the latency lower bound up to a factor of at most $\log P$. In the previous subsection, we defined one large dimension, two large dimensions, and three large dimensions asymptotically. The lower bounds are "continuous" in the sense that they are equivalent for parameters within a constant factor of the threshold, so the precise choice of the cutoff does not matter. In this section, we define them precisely by CARMA's behavior.

One large dimension

If $P \leq \frac{d_3}{d_2}$, then there are no DFS steps, only one dimension is ever split, and the smallest matrix is replicated at each BFS step. The communication cost is the cost to send this matrix at each step:

$$W = O\left(\sum_{i=0}^{\log_2 P-1} \frac{d_1 d_2}{P} 2^i\right) = O\left(d_1 d_2\right),$$

since $d_1 d_2/P$ is the initial amount of data of the smallest matrix per processor, and it increases by a factor of 2 at each BFS step. In this case the BFS steps can be thought of as performing an all-gather or reduce-scatter on the smallest matrix. By having the initial/final data layout be distributed across the processors, the BFS approach avoids the $\log P$ factor in the bandwidth that a broadcast or reduce would incur. When $d_1 d_2 < P$, there is still a logarithmic factor, and the cost is $W = O\left(d_1 d_2 \log \frac{P}{d_1 d_2}\right)$.

The memory use is the memory required to hold the input and output, plus the memory required to hold all the data received, so

$$M = O\left(\frac{d_1 d_2 + d_1 d_3 + d_2 d_3}{P} + d_1 d_2\right) = O\left(\frac{d_2 d_3}{P}\right).$$

At most a constant factor of extra memory is required. In fact the constant factor is quite small: since $d_1 d_2 \leq d_1 d_3/P \leq d_2 d_3/P$, it uses at most a factor of $3/2$ as much memory as is required to store the input and output.

Algorithm 4 CARMA(A,B,C,m,k,n,P)

Input: A is an $m \times k$ matrix and B is a $k \times n$ matrix
Output: $C = AB$

```
 1: if P = 1 then
 2:     SequentialMultiply( A, B, C, m, k, n )
 3: else if Enough Memory then                                    ▷ Do a BFS
 4:     if n is the largest dimension then
 5:         Copy A to disjoint halves of the processors.
                  ▷ Processor i sends and receives local A from processor i ± P/2
 6:         parallel for  do
 7:             CARMA(A, B_left, C_left, m, k, n/2, P/2)
 8:             CARMA(A, B_right, C_right, m, k, n/2, P/2)
 9:     if m is the largest dimension then
10:         Copy B to disjoint halves of the processors.
11:               ▷ Processor i sends and receives local B from processor i ± P/2
12:         parallel for  do
13:             CARMA(A_top, B, C_top, m/2, k, n, P/2)
14:             CARMA(A_bot, B, C_bot, m/2, k, n, P/2)
15:     if k is the largest dimension then
16:         parallel for  do
17:             CARMA(A_left, B_top,C, m, k/2, n, P/2)
18:             CARMA(A_right, B_bot,C, m, k/2, n, P/2)
19:         Gather C from disjoint halves of the processors.
                  ▷ Processor i sends C and receives C' from processor i ± P/2
20:         C ← C + C'
21: else                                                          ▷ Do a DFS
22:     if n is the largest dimension then
23:         CARMA(A, B_left, C_left, m, k, n/2, P)
24:         CARMA(A, B_right, C_right, m, k, n/2, P)
25:     if m is the largest dimension then
26:         CARMA(A_top, B, C_top, m/2, k, n, P)
27:         CARMA(A_bot, B, C_bot, m/2, k, n, P)
28:     if k is the largest dimension then
29:         CARMA(A_left, B_top, C, m, k/2, n, P)
30:         CARMA(A_right, B_bot, C, m, k/2, n, P)
```

The number of messages sent at each BFS is constant, so the latency cost is $L = O(\log P)$.

Two large dimensions

Next consider the case that

$$\frac{d_3}{d_2} < P \leq \frac{d_2 d_3}{d_1^2}.$$

There will be two phases: for the first $\log_2 \frac{d_3}{d_2}$ BFS steps, the original largest dimension is split; then for the remaining $\log_2 \frac{Pd_2}{d_3}$ BFS steps, the two original largest dimensions are alternately split. Again, no DFS steps are required. The bandwidth cost of the first phase is

$$W_1 = O\left(\sum_{i=0}^{\log_2 \frac{d_3}{d_2}-1} \frac{d_1 d_2}{P} 2^i\right) = O\left(\frac{d_1 d_3}{P}\right).$$

The bandwidth cost of the second phase is

$$W_2 = O\left(\sum_{i=0}^{\frac{1}{2}\log_2 \frac{Pd_2}{d_3}} \frac{d_1 d_2}{Pd_2/d_3} 2^i\right) = O\left(\sqrt{\frac{d_1^2 d_2 d_3}{P}}\right),$$

since every two BFS steps increases the amount of data being transferred by a factor of 2.

The cost of the second phase dominates the cost of the first. Again, the memory use is the memory required to hold the input and output, plus the memory required to hold all the data received, so

$$M = O\left(\frac{d_1 d_2 + d_1 d_3 + d_2 d_3}{P} + \sqrt{\frac{d_1^2 d_2 d_3}{P}}\right) = O\left(\frac{d_2 d_3}{P}\right).$$

At most a constant factor of extra memory is required, justifying our use of BFS only. In fact, by the assumptions of this case, the algorithm never uses more than 3 times the amount of memory used to store the input and output.

There are a constant number of messages sent at each BFS step, so the latency cost is $L = O(\log P)$.

Three large dimensions

Finally, consider the case that $P > \frac{d_2 d_3}{d_1^2}$. The first phase consists of $\log_2 \frac{d_3}{d_2}$ BFS steps splitting the largest dimension, and is exactly as in the previous case. The second phase consists of $2\log_2 \frac{d_2}{d_1}$ BFS steps alternately splitting the two original largest dimensions. After this phase, there are $P_3 = \frac{Pd_1^2}{d_2 d_3}$ processors working on each subproblem, and the subproblems are multiplication where all three dimensions are within a factor of 2 of each other. CARMA

splits each of the dimensions once every three steps, alternating BFS and DFS to stay within memory bounds, until it gets down to one processor.

The cost of the first phase is exactly as in the previous case. The bandwidth cost of the second phase is

$$W_2 = O\left(\sum_{i=0}^{\log_2 \frac{d_2}{d_1}} \frac{d_1 d_2}{P d_2/d_3} (2)^i\right) = O\left(\sqrt{\frac{d_2 d_3}{P}}\right).$$

In the final phase, the cost is within a factor of 4 of the square case, which was discussed in Section 2.9, giving

$$\begin{aligned} W_3 &= O\left(\frac{d_1^3}{P_3\sqrt{M}} + \left(\frac{d_1^3}{P_3}\right)^{2/3}\right) \\ &= O\left(\frac{d_1 d_2 d_3}{P\sqrt{M}} + \left(\frac{d_1 d_2 d_3}{P}\right)^{2/3}\right), \end{aligned}$$

while remaining within memory size M. W_3 is the dominant term in the bandwidth cost.

The latency cost of the first two phases is $\log \frac{d_2 d_3}{d_1^2}$, and the latency cost of the third phase is

$$L_3 = O\left(\left(\frac{d_1 d_2 d_3}{P M^{3/2}} + 1\right) \log \frac{P d_1^2}{d_2 d_3}\right),$$

giving a total latency cost of

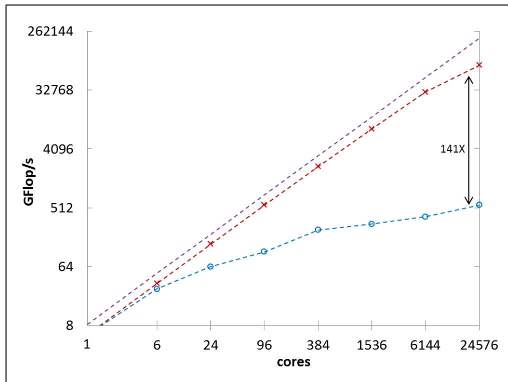
$$L = O\left(\frac{d_1 d_2 d_3}{P M^{3/2}} \log \frac{P d_1^2}{d_2 d_3} + \log P\right).$$

3.3 Performance Results

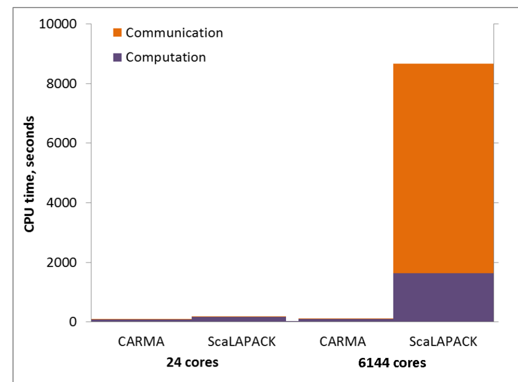
We have implemented CARMA in C++ with MPI. We benchmark on Hopper[1], a Cray XE6 at the National Energy Research Scientific Computing Center (NERSC). It consists of 6,384 compute nodes, each of which has 2 twelve-core AMD "Magny-Cours" 2.1 GHz processors and 32 GB of DRAM (384 of the nodes have 64 GB of DRAM). The 24 cores are divided between 4 NUMA regions.

CARMA gets the best performance when run as "flat MPI", with one MPI process per core. Local sequential matrix multiplications are performed by calls to Cray LibSci version 11.1.00. The distributed-memory version of CARMA supports splitting by arbitrary factors at each recursive step rather than just by a factor of 2. For each data point, several splitting factors and orders were explored and the one with the best performance is shown. It is possible that further performance improvements are possible by exploring the search space

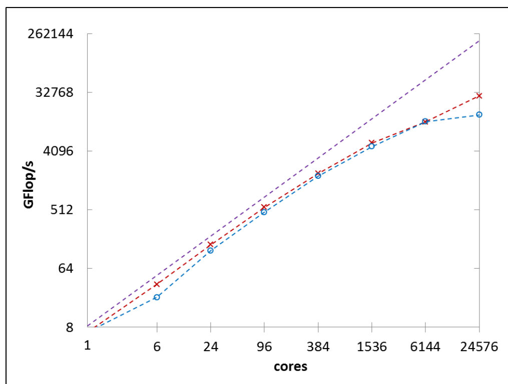
[1]For machine details, see `http://www.nersc.gov/users/computational-systems/hopper`



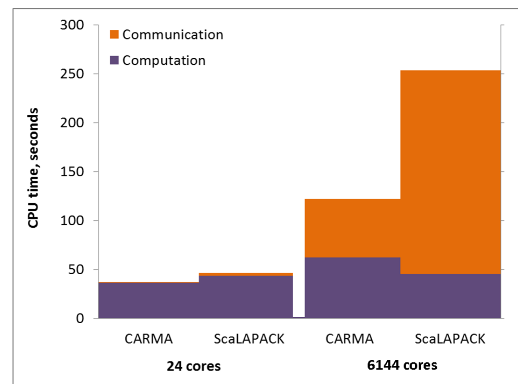
(a) Strong scaling, $192 \times 6291456 \times 192$.



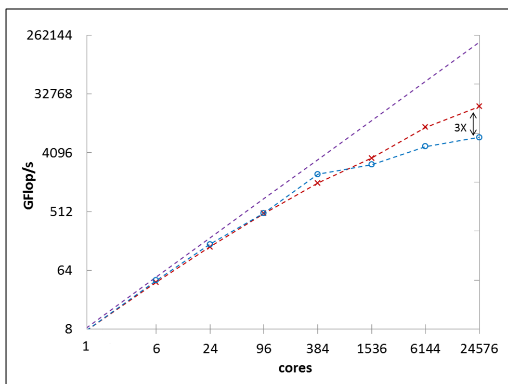
(b) Time breakdown, $192 \times 6291456 \times 192$.



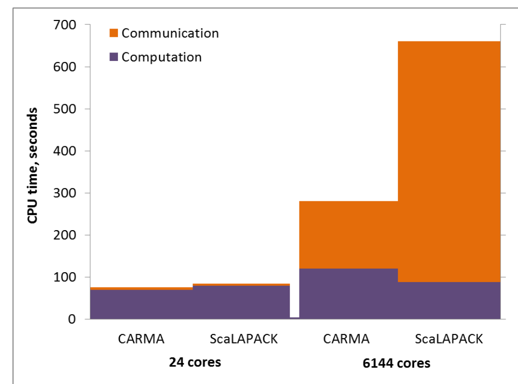
(c) Strong scaling, $12288 \times 192 \times 12288$.



(d) Time breakdown, $12288 \times 192 \times 12288$.



(e) Strong scaling, square $n = 6144$.



(f) Time breakdown, square $n = 6144$.

ScaLAPACK CARMA Floating Point Peak

Figure 3.3: CARMA compared to ScaLAPACK on Hopper. Left column: Strong scaling of performance. Right column: CPU-time breakdown summed over all cores (so perfect strong scaling would correspond to equal heights at 24 and 6144 cores).

more thoroughly. For a description of the data layout used by distributed CARMA, see Section 3.4.3.

We compare CARMA against ScaLAPACK version 1.8.0 as optimized by NERSC. ScaLAPACK also uses LibSci for local multiplications. For each data point we explore several possible executions and show the one with the highest performance. First, we try running with 1, 6, or 24 cores per MPI process. Parallelism between cores in a single process is provided by LibSci. Second, we explore all valid processor grids. We also try storing the input matrices as stated, or transposed. In some cases storing the transpose of one of the inputs increases ScaLAPACK's performance by more than a factor of 10.

The topology of the allocation of nodes on Hopper is outside the user's control, and, for communication-bound problems on many nodes, can affect the runtime by as much as a factor of 2. We do not attempt to measure this effect. Instead, for every data point shown, the CARMA and ScaLAPACK runs were performed during the same reservation and hence using the same allocation.

We benchmark three shapes of matrices corresponding to the three cases in the communication costs in Table 3.1. For the case of one large dimension, we benchmark $m = n = 192$, $k = 6291456$. The aspect ratio is very large so it is in the one large dimension case ($k/P > m, n$) even for our largest run on $P = 24576$ cores. In this case we see improvements of up to 140× over ScaLAPACK. This data is shown in Figure 3.3a. If ScaLAPACK is not allowed to transpose the input matrices, the improvement grows to 2500×.

For the case of two large dimensions, we benchmark $m = n = 24576$, $k = 192$. In this case both CARMA and ScaLAPACK (which uses SUMMA) are communication-optimal, so we do not expect a large performance difference. Indeed performance is close between the two except on very large numbers of processors (the right end of Figure 3.3c) where CARMA is nearly 2× faster.

Finally for the case of three large dimensions, we benchmark $m = n = k = 6144$. For small numbers of processors, the problem is compute-bound and both CARMA and ScaLAPACK perform comparably. For more than about 1000 cores, CARMA is faster, and on 24576 cores it is nearly 3× faster. See Figure 3.3e.

The right-hand column of Figure 3.3 shows the breakdown of time between computation and communication for CARMA and ScaLAPACK, for each of these matrix sizes, and for 24 cores (1 node) and 6144 cores (256 nodes). In the case of 1 large dimension on 6144 cores, CARMA is 16× faster at the computation, but more than 1000× faster at the communication. CARMA is faster at the computation because the local matrix multiplications are as close to square as possible allowing for more efficient use of the cache. For the other two sizes, the computation time is comparable between the two, but CARMA spends about 3.5× less time on communication on 6144 cores.

All tests are for multiplication of randomly generated double precision matrices. For each algorithm and size, one warm-up run was performed immediately before the benchmark.

3.4 Remarks

CARMA is the first distributed-memory parallel matrix multiplication algorithm to be communication-optimal for all dimensions of matrices and sizes of memory. We prove CARMA's communication optimality and compare it against ScaLAPACK. Despite its simple implementation, the algorithm minimizes communication, yielding performance improvements of 2× to 140×. As expected, our best improvement comes in ranges where CARMA achieves lower bounds on communication but previous algorithms do not.

3.4.1 Opportunities for Tuning

The algorithm described in Section 3.2 always splits the largest dimension by a factor of 2. This can be generalized considerably. At each recursive step, the largest dimension could be split by any integer factor s, which could vary between steps. Increasing s from 2 decreases the bandwidth cost (by at most a small constant factor) while increasing the latency cost. The choice of split factors is also affected by the number of processors, since the product of all split factors at BFS steps must equal the number of processors. Additionally, when two dimensions are of similar size, either one could be split. As long as the s are bounded by a constant, and the dimension that is split at each step is within a constant factor of the largest dimension, a similar analysis to the one in Section 3.2.1 shows that CARMA is still asymptotically communication-optimal. Note that this means that CARMA can efficiently use any number of processors that does not have large prime factors, by choosing split factors s that factor the number of processors.

In practice, however, there is a large tuning space, and more performance improvements may be possible by exploring this space further. Our implementation allows the user to choose any dimension to split and any split factor at each recursive step (but the required data layout will vary; see Section 3.4.3). On Hopper, we have found that splitting 6 or 8 ways at each step typically performs better than splitting 2 ways, but we have not performed an exhaustive search of the tuning space.

3.4.2 Perfect Strong Scaling Range

Recall the definition of perfect strong scaling from Section 1.1. In the square case, the 2.5D algorithm and the square BFS/DFS algorithm exhibit perfect strong scaling in the range $P = \Omega(n^2/M)$ and $P = O(n^3/M^{3/2})$, which is the maximum possible range. Similarly, in the case of three large dimensions, defined by

$$P = \Omega\left(\frac{d_2 d_3}{d_1^2},\right),$$

both CARMA and 3D-SUMMA exhibit perfect strong scaling in the maximum possible range

$$P = \Omega\left(\frac{mn+mk+nk}{M}\right), \qquad P = O\left(\frac{mnk}{M^{3/2}}\right).$$

Note that in the plots shown in this paper, the entire problem fits on one node, so the range degenerates to just $P = 1$.

In the case of one or two large dimensions, the bandwidth lower bound does not decrease linearly with P (see Table 3.1). As a result, perfect strong scaling is not possible. Figure 3.3a shows very good strong scaling for CARMA in practice because, even though the bandwidth cost does not decrease with P in this case, it is small enough that it is not dominant up to 6144 cores (see Figure 3.3b).

3.4.3 Data Layout Requirements

Recall the three cases of the bandwidth cost lower bounds from Section 3.1. In the case of three large dimensions, the lower bound is higher than the size of the input and output data per processor: $\frac{mn+nk+mk}{P}$. This means it is possible to attain the bandwidth lower bound with any load balanced initial/final data layout, since the bandwidth cost of redistributing the data is sub-dominant.

However, in the case of one or two large dimensions, the bandwidth cost lower bound is lower than the size of the input and output data per processor. This means that a communication-optimal algorithm cannot afford to redistribute the largest matrix, which limits the data layouts that can be used. For example, in the case of one large dimension, where CARMA shows its greatest advantage, it is critical that only entries of the smallest matrix ever be communicated. As a result, it is necessary for corresponding entries of the two larger matrices to be on the same processor in the initial/final data layout.

CARMA only communicates one of the three matrices at each BFS step. It requires that each of the two halves of the other two matrices already resides entirely on the corresponding half of the processors. See Figure 3.4. This requirement applies recursively down to some block size, at which point CARMA uses a cyclic data layout (any load balanced layout would work for the base case). The recursive data layout that the distributed version of CARMA uses is different from any existing linear algebra library; hence CARMA cannot be directly incorporated into, for example, ScaLAPACK or Elemental. Changing the data layout before or after calling CARMA would defeat its advantage, which comes from not communicating the largest matrices at all.

In fact, even if a new library is designed for CARMA, there is a complication. If a matrix is used multiple times in a computation, sometimes as the largest and sometimes not the largest, the data layouts CARMA prefers will not be consistent. It should still be possible to asymptotically attain the communication lower bound for any sequence of multiplications by choosing the correct initial or final layout and possibly transforming the layout between certain multiplications. Doing so in a way that makes the library easy to use while remaining efficient is left as an open problem.

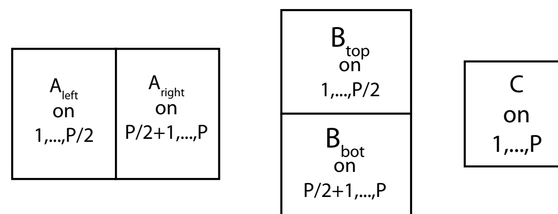


Figure 3.4: Data layout for a BFS step splitting dimension k. Before the BFS step, all three matrices are distributed on P processors. The distributed code assumes that A_{left} and B_{top} are distributed among the first $P/2$ processors, A_{right} and B_{bot} are distributed among the remaining $P/2$ processors, and C is distributed among all the processors. The layout applies recursively, following the execution pattern, and in the base case the layout is cyclic.

Chapter 4

Sparse Matrix Multiplication

A sparse matrix is one for which $o(n^2)$ entries are nonzero, and only the nonzero entries are explicitly stored. This means that, when applying the classical matrix multiplication algorithm, among the n^3 multiplications that are performed in the dense case, only those for which the corresponding entries of A and B are both nonzero must be performed. Here we only consider algorithms where every nonzero product is computed directly, and not fast sparse algorithms such as the one of Yuster and Zwick [67].

The results in this chapter are joint work with Grey Ballard, Aydın Buluç, James Demmel, Laura Grigori, Oded Schwartz and Sivan Toledo; the algorithms and analysis appear in [4].

Achieving scalability for parallel algorithms for sparse matrix problems is challenging because the computations tend not to have the potential for $\Theta(\sqrt{M})$ data re-use that is common in dense matrix problems. Further, the performance of sparse algorithms is often highly dependent on the sparsity structure of the input matrices. We show in this chapter that previous algorithms for sparse matrix-matrix multiplication are non optimal in their communication costs, and we obtain new algorithms which are communication optimal, communicating less than the previous algorithms and matching new lower bounds.

Our lower bounds require two important assumptions: (1) the sparsity of the input matrices is random, corresponding to Erdős-Rényi random graphs (see Definition 4.1), and (2) the algorithm is *sparsity-independent*, where the partitioning of the computation among the processors is independent of the sparsity structure of the input matrices (see Definition 4.4). The second assumption applies to nearly all existing algorithms for general sparse matrix-matrix multiplication. While *a priori* knowledge of sparsity structure can certainly reduce communication for many important classes of inputs, dynamically determining and efficiently exploiting the structure of general input matrices is a challenging problem. In fact, a common technique of current library implementations is to randomly permute rows and columns of the input matrices in an attempt to destroy their structure and improve computational load balance [17, 19]. Because the input matrices are random, our analyses are in terms of expected communication costs.

4.1 Preliminaries

For sparse matrix indexing, we use the colon notation, where $A(:, i)$ denotes the ith column, $A(i, :)$ denotes the ith row, and $A(i, j)$ denotes the element at the (i, j)th position of matrix A. We use $nnz(\cdot)$ to denote the number of nonzeros in a matrix or submatrix.

We consider the case where A and B are $n \times n$ ER(d) matrices:

Definition 4.1. *An ER(d) matrix is an adjacency matrix of an Erdős-Rényi graph with parameters n and d/n. That is, an ER(d) matrix is a square matrix of dimension n where each entry is nonzero with probability d/n. We assume $d \ll \sqrt{n}$.*

It is not important for our analysis to which semiring the matrix entries belong, though we assume algorithms do not short-circuit summations or exploit cancellation in the intermediate values or output entries. In this case, the following facts will be useful for our analysis.

Fact 4.2. *Let A and B be $n \times n$ ER(d) matrices. Then*

(a) the expected number of nonzeros in A and in B is dn,

(b) the expected number of scalar multiplications in $A \cdot B$ is d^2n, and

(c) the expected number of nonzeros in C is $d^2n(1 - o(1))$.

Proof. Since each entry of A and B is nonzero with probability d/n, the expected number of nonzeros in each matrix is $n^2(d/n) = dn$. For each of the possible n^3 scalar multiplications in $A \cdot B$, the computation is required only if both corresponding entries of A and B are nonzero, which are independent events. Thus the probability that any multiplication is required is d^2/n^2, and the expected number of scalar multiplications is d^2n. Finally, an entry of $C = A \cdot B$ is zero only if all n possible scalar multiplications corresponding to it are zero. Since the probability that a possible scalar multiplication is zero is $(1 - d^2/n^2)$ and the n possible scalar multiplications corresponding to a single output entry are independent, the probability that an entry of C is zero is $(1 - d^2/n^2)^n = 1 - d^2/n + O(d^4/n^2)$. Thus the expected number of nonzeros of C is $n^2(d^2/n - O(d^4/n^2) = d^2n(1 - o(1))$, since we assume $d \ll \sqrt{n}$. □

Recall from Section 1.4 that the n^3 scalar multiplications in dense matrix multiplication can be arranged into a cube $\mathcal{V}$ whose faces represent the input matrices.

Definition 4.3. *We say a voxel $(i, j, k) \in \mathcal{V}$ is* nonzero *if, for given input matrices A and B, both $A(i, k)$ and $B(k, j)$ are nonzero.*

Definition 4.4. *A* sparsity-independent *parallel algorithm for sparse matrix-matrix multiplication is one in which the assignment of entries of the input and output matrices to processors and the assignment of computation voxels to processors is independent of the sparsity pattern of the input (or output) matrices. If an assigned matrix entry is zero, the processor need*

not store it; if an assigned voxel is zero, the processor need not perform the computation corresponding to that voxel.

Our lower bound argument in Section 4.2 will apply to all sparsity-independent algorithms. However, we will analyze a more restricted class of algorithms in Section 4.3, those that assign contiguous brick-shaped sets of voxels to each processor.

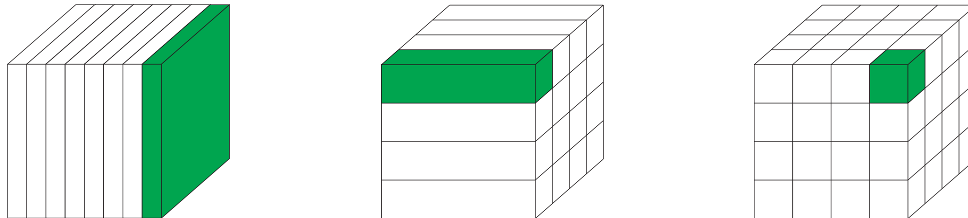


Figure 4.1: How the cube is partitioned in 1D (left), 2D (middle), and 3D (right) algorithms.

4.1.1 All-to-all Communication

Several of the algorithms we discuss make use of all-to-all communication. If each processor needs to send b different words to each other processor (so each processor needs to send a total of $b(P-1)$ words), the bandwidth lower bound is $W = \Omega(bP)$ and the latency lower bound is $S = \Omega(\log P)$. Each of these bounds is attainable, but it has been shown that they are not simultaneously attainable (see Theorem 2.9 of [15]). Depending on the relative costs of bandwidth and latency, one may wish to use the *point-to-point* algorithm (each processor sends data directly to each other processor) which incurs costs of $W = O(bP)$, $S = O(P)$ or the *bit-fixing* algorithm (each message of b words is sent by the bit-fixing routing algorithm) which incurs costs of $W = O(bP \log P)$, $S = O(\log P)$. Both of these are optimal, in the sense that neither the bandwidth cost nor the latency cost can be asymptotically improved without asymptotically increasing the other one.

4.2 Communication Lower Bounds

The general lower bounds for direct linear algebra [10] apply to our case and give

$$W = \Omega\left(\frac{d^2 n}{P\sqrt{M}}\right). \tag{4.1}$$

This bound is highest when M takes its minimum value d^2n/P, in which case it becomes $W = \Omega(\sqrt{d^2n/P})$. In this section we improve (increase) these lower bounds by a factor of $\sqrt{n}\cdot\min\{1, d/\sqrt{P}\}$. For larger values of M, the lower bound in Equation 4.1 becomes weaker, whereas our new bound does not, and the improvement factor increases to $\sqrt{M}\cdot\min\{1, \sqrt{P}/d\}$. The previous memory-independent lower bound [5] reduces to the trivial bound $W = \Omega(0)$.

Theorem 4.5. *Any sparsity-independent sparse matrix multiplication algorithm with load-balanced input and output has an expected communication cost lower bound of*

$$W = \Omega\left(\min\left\{\frac{dn}{\sqrt{P}}, \frac{d^2n}{P}\right\}\right)$$

when applied to $ER(d)$ input matrices on P processors.

Note that this bound can also be written as

$$W = \Omega\left(\frac{dn}{\sqrt{P}}\min\left\{1, \frac{d}{\sqrt{P}}\right\}\right),$$

and so which bound applies depends on the ratio $d/\sqrt{P}$.

Proof. Consider the n^3 voxels that correspond to potential scalar multiplications $A(i,k) \cdot B(k,j)$. A sparsity-independent algorithm gives a partitioning of these multiplications among the P processors. Let V be the largest set of voxels assigned to a processor, so $|V| \geq \frac{n^3}{P}$. For each i, j, let ℓ_{ij}^C be the number of values of k such that $(i,j,k) \in V$, see Figure 4.2. We count how many of the voxels in V correspond to $\ell_{ij}^C < \frac{n}{4}$ and divide into two cases.

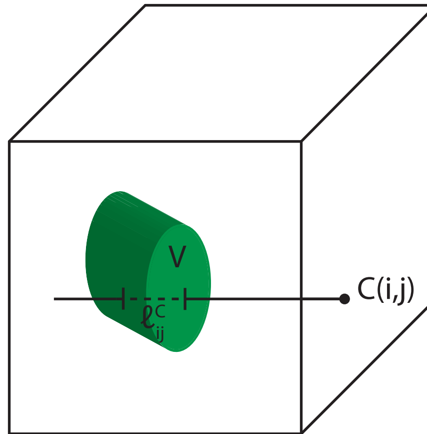


Figure 4.2: Graphical representation of V and ℓ_{ij}^C.

Case 1: At least $\frac{n^3}{2P}$ voxels of V correspond to $\ell_{ij}^C < \frac{n}{4}$. Let V' be these voxels, so $|V'| \geq \frac{n^3}{2P}$. We will analyze the communication cost corresponding to the computation of V' and get a bound on the number of products computed by this processor that must be sent to other processors. Since the output is load balanced and the algorithm is sparsity-independent, the processor that computes V' is allowed to store only a particular set of $\frac{n^2}{P}$ entries of C in the output data layout. Since every voxel in V' corresponds to an $\ell_{ij}^C < \frac{n}{4}$, the $\frac{n^2}{P}$ output elements stored by the processor correspond to at most $\frac{n^3}{4P}$ voxels in V', which is at most half of $|V'|$. All of the nonzero voxels in the remainder of V' contribute to entries of C that

must be sent to another processor. In expectation, this is at least $\frac{d^2n}{4P}$ nonzero voxels, since each voxel is nonzero with probability $\frac{d^2}{n^2}$. Moreover, from Lemma 4.2, only a small number of the nonzero entries of C have contributions from more than one voxel, so very few of the values can be summed before being communicated. The expected bandwidth cost is then bounded by $W = \Omega(d^2n/P)$.

Case 2: Fewer than $\frac{n^3}{2P}$ voxels of V correspond to $\ell^C_{ij} < \frac{n}{4}$. This means that at least $\frac{n^3}{2P}$ voxels of V correspond to $\ell^C_{ij} \geq \frac{n}{4}$. Let V'' be these voxels, so $|V''| \geq \frac{n^3}{2P}$. We will analyze the communication cost corresponding to the computation of V'' and get a lower bound on the amount of input data needed by this processor. For each i, k, let ℓ^A_{ik} be the number of values of j such that $(i, j, k) \in V''$. Similarly, for each j, k, let ℓ^B_{jk} be the number of values of i such that $(i, j, k) \in V''$. Partition V'' into three sets: V_0 is the set of voxels that correspond to $\ell^A_{ik} > \frac{n}{d}$ and $\ell^B_{jk} > \frac{n}{d}$; V_A is the set of voxels that correspond to $\ell^A_{ik} \leq \frac{n}{d}$; and V_B is the set of voxels that correspond to $\ell^B_{jk} \leq \frac{n}{d}$ and $\ell^A_{ik} > \frac{n}{d}$. At least one of these sets has at least $\frac{n^3}{6P}$ voxels, and we divide into three subcases.

Case 2a: $|V_0| \geq \frac{n^3}{6P}$. Let p_A, p_B, and p_C be the sizes of the projections of V_0 onto A, B, and C, respectively. Lemma 1.1 implies that $p_A p_B p_C \geq |V_0|^2 = \frac{n^6}{36P^2}$. The assumptions of Case 2 implies $p_C \leq \frac{|V_0|}{n/4}$. Thus $p_A p_B \geq \frac{n^4}{24P}$, or $\max\{p_A, p_B\} \geq \frac{n^2}{\sqrt{24P}}$. Since the situation is symmetric with respect to A and B, assume without loss of generality that A has the larger projection, so $p_A \geq \frac{n^2}{\sqrt{24P}}$. Since the density of A is $\frac{d}{n}$, this means that the expected number of nonzeros in the projection of V_0 onto A is at least $\frac{dn}{\sqrt{24P}}$. Since each of these nonzeros in A corresponds to a $\ell^A_{ik} > \frac{n}{d}$, it is needed to compute V_0 with probability at least $1 - (1 - d/n)^{n/d} > 1 - 1/e$. Thus in expectation a constant fraction of the nonzeros of A in the projection of V_0 are needed. The number of nonzeros the processor holds in the initial data layout is $\frac{dn}{P}$ in expectation, which is asymptotically less than the number needed for the computation. Thus we get a bandwidth lower bound of $W = \Omega(dn/\sqrt{P})$.

Case 2b: $|V_A| \geq \frac{n^3}{6P}$. Each voxel in V_A corresponds to $\ell^A_{ik} \leq \frac{n}{d}$. In this case we are able to bound the re-use of entries of mA to get a lower bound. Count how many entries of A correspond to each possible value of ℓ^A_{ik}, $1 \leq r \leq \frac{n}{d}$, and call this number N_r. Note that $\sum_{r=1}^{n/d} r \cdot N_r = |V_A|$. Suppose a given entry $A(i, k)$ corresponds to $\ell^A_{ik} = r$. We can bound the probability that $A(i, k)$ is needed by the processor to compute V_A as a function $f(r)$. The probability that $A(i, k)$ is needed is the probability that both $A(i, k)$ is nonzero and one of the r voxels corresponding to $A(i, k)$ in V_A is nonzero, so

$$f(r) = \frac{d}{n}\left(1 - \left(1 - \frac{d}{n}\right)^r\right) \geq \frac{rd^2}{2n^2},$$

since $r \leq \frac{n}{d}$. Thus the expected number of nonzeros of A that are needed by the processor is

$$\sum_{r=1}^{n/d} N_r f(r) \geq \frac{d^2}{2n^2} \sum_{r=1}^{n/d} r \cdot N_r \geq \frac{d^2 n}{12P}.$$

This is asymptotically larger than the number of nonzeros the processor holds at the beginning of the computation, so we get a bandwidth lower bound of $W = \Omega(d^2n/P)$.

Case 2c: $|V_B| \geq \frac{n^3}{6P}$. The analysis is identical to the previous case, except we look at the number of nonzeros of B that are required.

Since an algorithm may be in any of these cases, the overall lower bound is the minimum:

$$W = \Omega\left(\min\left\{\frac{dn}{\sqrt{P}}, \frac{d^2n}{P}\right\}\right).$$

□

4.3 Algorithms

In this section we consider algorithms which assign contiguous bricks of voxels to processors. We categorize these algorithms into 1D, 2D, and 3D algorithms, as shown in Figure 4.1. If we consider the dimensions of the brick of voxels assigned to each processor, 1D algorithms correspond to bricks with two dimensions of length n (and 1 shorter), 2D algorithms correspond to bricks with one dimension of length n (and 2 shorter), and 3D algorithms correspond to bricks with all 3 dimensions shorter than n. Table 4.1 provides a summary of the communication costs of the sparsity-independent algorithms we consider.

	Algorithm	**Bandwidth cost**	**Latency cost**
	Previous Lower Bound [10]	$\frac{d^2n}{P\sqrt{M}}$	0
	Lower Bound [4]	$\min\left\{\frac{dn}{\sqrt{P}}, \frac{d^2n}{P}\right\}$	1
1D	Naïve Block Row [18]	dn	P
1D	Improved Block Row* [22]	$\frac{d^2n}{P}$	$\min\{\log P, \frac{dn}{P}\}$
1D	Outer Product* [46]	$\frac{d^2n}{P}$	$\log P$
2D	SpSUMMA [18]	$\frac{dn}{\sqrt{P}}$	$\sqrt{P}$
3D	**Recursive** [4]	$\min\left\{\frac{dn}{\sqrt{P}}, \frac{d^2n}{P}\left\lceil\log\frac{P}{d^2}\right\rceil\right\}$	$\log P$

Table 4.1: Asymptotic expected communication costs of sparsity-independent algorithms and lower bounds. Algorithms marked with an asterisk make use of all-to-all communication. Depending on the algorithm used for the all-to-all, either the bandwidth or latency cost listed is attainable, but not both; see Section 4.1.1.

4.3.1 1D Algorithms

Naïve Block Row Algorithm

The naïve block row algorithm [18] distributes the input and output matrices to processors in a block row fashion. Then in order for processor i to compute the ith block row, it needs access to the ith block row of A (which it already owns), and potentially all of B. Thus, we can allow each processor to compute its block row of C by leaving A and C stationary and cyclically shifting block rows of B around a ring of the processors. This algorithm requires P stages, with each processor communicating with its two neighbors in the ring. The size of each message is the number of nonzeros in a block row of B, which is expected to be dn/P words. Thus, the bandwidth cost of the block row algorithm is dn and the latency cost is P. An analogous block column algorithm works by cyclically shifting block columns of A with identical communication costs.

Improved Block Row Algorithm

The communication costs of the block row algorithm can be reduced without changing the assignment of matrix entries or voxels to processors [22]. The key idea is for each processor to determine exactly which rows of B it needs to access in order to perform its computations. For example, if processor i owns the ith block row of A, A_i, and the jth subcolumn of A_i contains no nonzeros, then processor i doesn't need to access the jth row of B. Further, since the height of a subcolumn is n/P, the probability that the subcolumn is completely empty is

$$Pr\left[nnz(A_i(:,j)) = 0\right] = \left(1 - \frac{d}{n}\right)^{\frac{n}{P}} \approx 1 - \frac{d}{P},$$

assuming $d < P$. In this case, the expected number of subcolumns of A_i which have at least one nonzero is dn/P. Since processor i needs to access only those rows of B which correspond to nonzero subcolumns of A_i, and because the expected number of nonzeros in each row of B is d, the expected number of nonzeros of B that processor i needs to access is d^2n/P.

Note that the local memory of each processor must be of size $\Omega\left(d^2n/P\right)$ in order to store the output matrix C. Thus, it is possible for each processor to gather all of their required rows of B at once. The improved algorithm consists of each processor determining which rows it needs, requesting those rows from the appropriate processors, and then sending and receiving approximately d rows. While this can be implemented in various ways, the bandwidth cost of the algorithm is at least $\Omega\left(d^2n/P\right)$ and if point-to-point communication is used, the latency cost is at least $\Omega(\min\{P, dn/P\})$. The block column algorithm can be improved in the same manner.

Outer Product Algorithm

Another possible 1D algorithm is to partition A in block columns, and B in block rows [46]. Without communication, each processor locally generates an $n \times n$ sparse matrix of rank n/P,

and processors combine their results to produce the output C. Because each column of A and row of B have about d nonzeros, the expected number of nonzeros in the locally computed output is d^2n/P. By deciding the distribution of C to processors up front, each processor can determine where to send each of its computed nonzeros. The final communication pattern is realized with an all-to-all collective in which each processer sends and receives $O(d^2n/P)$ words. Note that assuming A and B are initially distributed to processors in different ways may be unrealistic; however, no matter how they are initially distributed, A and B can be transformed to block column and row layouts with all-to-all collectives for a communication cost which is dominated by the final communication phase.

To avoid the all-to-all, it is possible to compute the expected number of blocks of the output which actually contain nonzeros; the best distribution of C is 2D, in which case the expected number of blocks of C you need to communicate is $\min\{P, dn/\sqrt{P}\}$. Thus for $P > (dn)^{2/3}$, the outer product algorithm can have $W = O(d^2n/P)$ and $S = O(dn/\sqrt{P})$.

4.3.2 2D Algorithms

Sparse SUMMA

In the Sparse SUMMA algorithm [18], the brick of voxels assigned to a processor has its longest dimension (of length n) in the k dimension. For each output entry of C to which it is assigned, the processor computes all the nonzero voxels which contribute to that output entry. The algorithm has a bandwidth cost of $O(dn/\sqrt{P})$ and a latency cost of $O(\sqrt{P})$ [18].

Improved Sparse SUMMA

In order to reduce the latency cost of Sparse SUMMA, each processor can gather all the necessary input data up front. That is, each processor is computing a product of a block row of A with a block column of B, so if it gathers all the nonzeros in those regions of the input matrices, it can compute its block of C with no more communication. Since every row of A and column of B contain about d nonzeros, and the number of rows of A and columns of B in a block is $n/\sqrt{P}$, the number of nonzeros a processor must gather is $O(dn/\sqrt{P})$. If $d > \sqrt{P}$, then the memory requirements for this gather operation do not exceed the memory requirements for storing the block of the output matrix C, which is $\Omega(d^2n/P)$.

The global communication pattern for each processor to gather its necessary data consists of allgather collectives along processor columns and along processor grids. The bandwidth cost of these collectives is $O(dn/\sqrt{P})$, which is the same as the standard algorithm, and the latency cost is reduced to $O(\log P)$. To our knowledge, this improvement has not appeared in the literature before.

We might also consider applying the optimization that improved the 1D block row (or column) algorithm. Processor (i, j) would need to gather the indices of the nonzero subcolumns of A_i and the nonzero subrows of B_j. This requires receiving $\Omega(dn/\sqrt{P})$ words, and so it cannot reduce the communication cost of Sparse SUMMA.

As in the dense case, there are variants on the sparse SUMMA algorithm that leave one of the input matrices stationary, rather than leaving the output matrix C stationary [40]. When multiplying ER(d) matrices, stationary input matrix algorithms require more communication that the standard approach because the global data involved in communicating C is about d^2n, while the global data involved in communicating A and B is only dn.

4.3.3 3D Recursive Algorithm

Next we adapt the BFS/DFS approach to the sparse case. Although we have assumed that the input matrices are square, the recursive algorithm will use rectangular matrices for subproblems. Assume that P processors are solving a subproblem of size $m \times k \times m$, that is A is $m \times k$, and B is $k \times m$, and C is $m \times m$. We will split into four subproblems, and then solve each subproblem independently on a quarter of the processor. There are two natural ways to split the problem into four equal subproblems that respect the density similarity between A and B, see Figure 4.3.

1. Split m in half, creating four subproblems of shape $(m/2) \times k \times (m/2)$. In this case each of the four subproblems needs access to a different part of C, so no communication of C is needed. However one half of A and B is needed for each subproblem, and since each quarter of the processors holds only one quarter of each matrix, it will be necessary to replicate A and B. This can be done via allgather collectives among disjoint pairs of processors at the cost of $O\left(dmk/(nP)\right)$ words and $O(1)$ messages.

2. Split k in quarters, creating four subproblems of shape $m \times (k/4) \times m$. In this case each of the four subproblems needs access to a different part of A and B, so with the right data layout, no communication of A or B is needed. However each subproblem will compute nonzeros across all of C, so those entries need to be redistributed and combined if necessary. This can be done via all-to-all collective among disjoint sets of 4 processors at a cost of $O(d^2m^2/(nP))$ words and $O(1)$ messages.

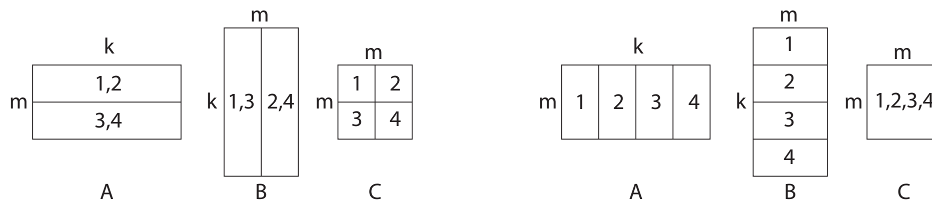


Figure 4.3: Two ways to split the matrix multiplication into four subproblems, with the parts of each matrix required by each subproblem labelled. On the left is split 1 and on the right is split 2.

At each recursive step, the algorithm chooses whichever split is cheapest in terms of communication cost. Initially, $m = k = n$ so split 1 costs $O(dn/P)$ words and is cheaper than split 2, which costs $O(d^2n/P)$ words. There are two cases to consider.

Case 1: If $P \leq d^2$, the algorithm reaches a single processor before split 1 becomes more expensive than split 2, so only split 1 is used. This case corresponds to a 2D algorithm, and the communication costs are

$$W = \sum_{i=0}^{\log_4 P-1} O\left(\frac{d(n/2^i)n}{P/4^i}\right) = O\left(\frac{dn}{\sqrt{P}}\right), \qquad S = O(\log P).$$

Case 2: If $P > d^2$, split 1 becomes more expensive than split 2 after $\log_2 d$ steps. After $\log_2 d$ steps, the subproblems have dimensions $(n/d) \times n \times (n/d)$ and there are P/d^2 processors working on each subproblem. The first $\log_2 d$ steps are split 1, and the rest are split 2, giving communication costs of

$$W = \sum_{i=0}^{\log_2 d-1} O\left(\frac{d(n/2^i)n}{P/4^i}\right) + \sum_{i=\log_2 d}^{\log_4 P} O\left(\frac{d^2 n}{P}\right) = O\left(\frac{d^2 n}{P}\left\lceil \log \frac{P}{d^2} \right\rceil\right), \qquad S = O(\log P).$$

This case corresponds to a 3D algorithm.

In both cases, the communication costs match the lower bound from Section 4.2 up to factors of at most $\log P$. Only layouts that are compatible with the recursive structure of the algorithm will allow these communication costs. One simple layout is to have A is block-column layout, B in block-row layout. Then C should have blocks of size $n/d \times n/d$, each distributed on a different $\lceil P/d^2 \rceil$ of the processors.

Note that since BFS only algorithm does not use more than a constant factor extra memory, there is no need for DFS steps.

4.4 Performance Results

We have implemented the block row algorithm, the improved block row algorithm, the outer product algorithm, sparse SUMMA, and the recursive algorithm in MPI and C++. The five implementations share the same local sparse matrix multiplication code. All our experiments are of Erdős-Rényi random matrices whose entries are random double precision values. They are stored in coordinate layout, and use either 32-bit or 64-bit integers for indexing, as appropriate for the matrix dimension.

We benchmarked on Titan[1], a Cray XK7 at Oak Ridge National Laboratory. It consists of 18,688 nodes connected by a Gemini interconnect. Each node has 32GB of RAM, a 16 core AMD Opteron 6274 processor, and an Nvidia K20 GPU. As of November 2012, it ranked first on the TOP500 list [53], with a LINPACK score of 17.59 Tflop/s. In our experiments, we only make use of the CPUs and not the GPUs. We use one core per process (16 MPI processes per node).

In general, the recursive algorithm outperforms all the previous algorithms, especially when run at large scale. Among the previous algorithms, when d is small relative to $\sqrt{P}$, the

[1]For machine details, see `http://www.olcf.ornl.gov/titan/`

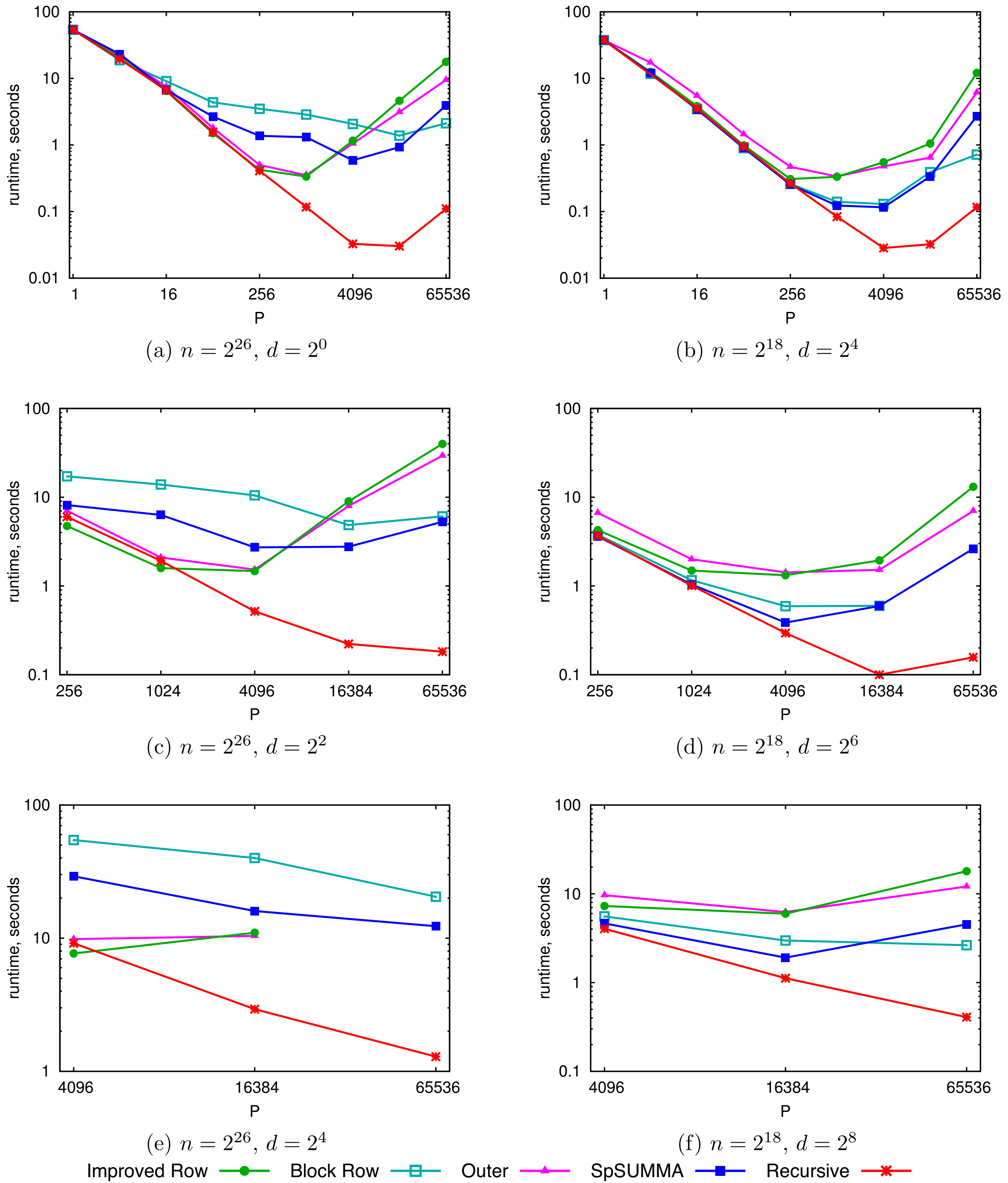


Figure 4.4: Strong scaling of sparse matrix multiplication.

outer product algorithm and the improved block row algorithm perform best; whereas when d is large relative to $\sqrt{P}$, sparse SUMMA performs best. This is as one would expect from the communication costs of each algorithm (see Table 4.1).

Figure 4.4 shows strong scaling of all five algorithms on a variety of problem sizes. In the top row, the output has 2^{26} expected nonzeros (the actual number of nonzeros in the experiments is within 0.1% of the expected number), so the input and output can be stored on one node. When $d = 1$, on up to 256 cores, the best performing algorithms are the recursive algorithm, the outer product algorithm, and the improved block row algorithm. Beyond that point, the recursive algorithm substantially outperforms the other two. For $d = 1$, every step of the recursive algorithm is of type 2, and it is essentially the same algorithm as the outer product algorithm. The difference in performance is because, when using many cores, the all-to-all on entries of C that is performed by our recursive code substantially outperforms the Alltoallv function of MPI. In this case, the recursive algorithm has a best time to solution of 0.0303 seconds, which makes it 11× faster than the next best algorithm: improved block row algorithm. When $d = 16$, sparse SUMMA is the closest competitor, and the recursive algorithm is about 4× faster.

In the second row of Figure 4.4, the output has 2^{30} expected nonzeros. When $d = 4$, the recursive algorithm's minimum time to solution is about 8.1× better than the improved block row algorithm, which is the closest competitor. When $d = 16$, sparse SUMMA is the closest competitor, and the recursive algorithm is 3.9× faster.

In the third row of Figure 4.4, the output has 2^{34} expected nonzeros. When $d = 16$, the recursive algorithm beats all the other algorithms by at least 6×. Note that among the other algorithms, the best performance is by the improved block row algorithm using 4096 cores, which is near the minimum to hold the input and output; only the recursive algorithm was able to use more parallelism to run faster than this. When $d = 256$, sparse SUMMA is again the closest competitor, and the recursive algorithm is 4.7× faster. Only the recursive algorithm shows significant performance gains when going from 16k cores to 64k cores.

Figure 4.5 shows a breakdown of the time between interprocessor communication (MPI calls), local computation (everything else), and time imbalance between the processors using 4096 cores with 2^{30} nonzeros in the output. All of the previous algorithms are communication bound, whereas the recursive algorithm is able to reduce the communication to be faster than the local computation (which is expected to itself be communication bound in the memory hierarchy). Load imbalance is not a significant problem for any of the algorithms.

In Figure 4.6, we explore the possibilities for interleaving type 1 and type 2 recursive steps in the recursive algorithm on $P = 1024$ cores. Our implementation allows for arbitrary interleavings, which makes for $2^{\log_4 P} = \sqrt{P}$ possible executions. For large values of P, this is too many to realistically explore, so in Figures 4.4 and 4.5 we only explored "simple" interleavings: those for which all of the type 1 steps are taken before any of the type 2 steps. These are shown in green in Figure 4.6. In red in Figure 4.6 is the case of $\log_2 d - 1$ type 1 steps followed by $\log_4 P - \log_2 d + 1$ type 2 steps. Taking into account the fact that two matrices must be communicated at a type 1 step (A and B), whereas only one must be communicated at a type 2 step (C), this is what the analysis of Section 4.3.3 suggests will

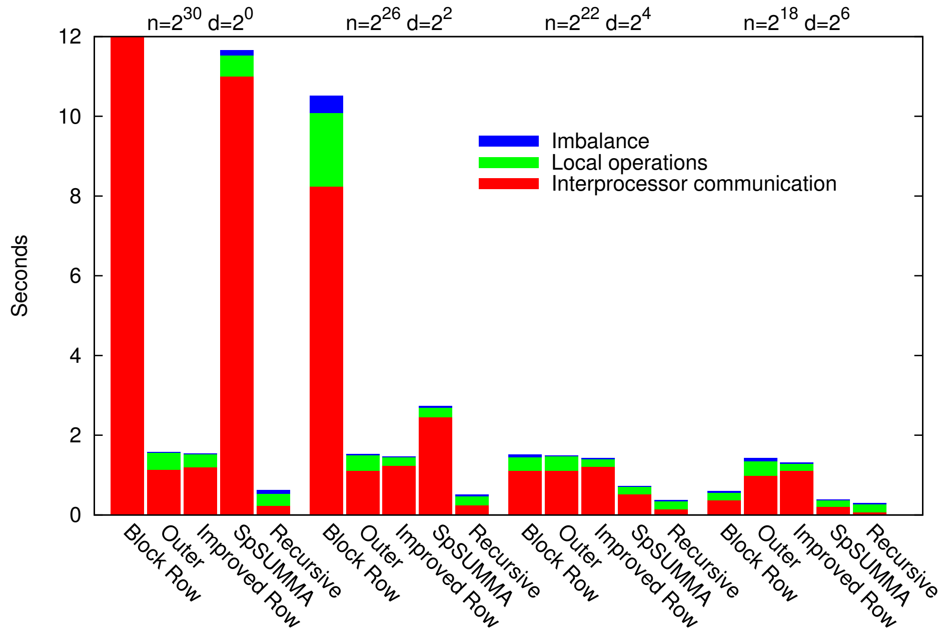
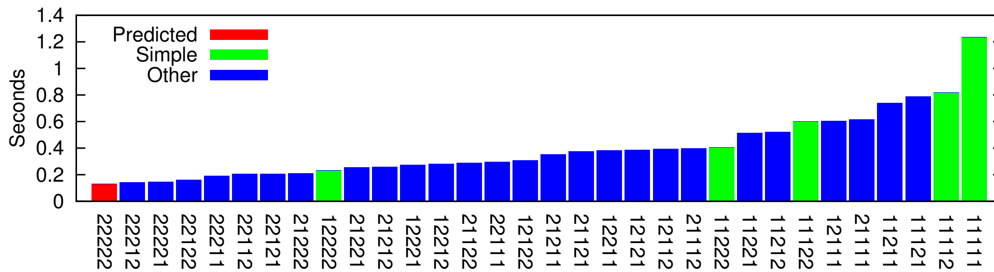
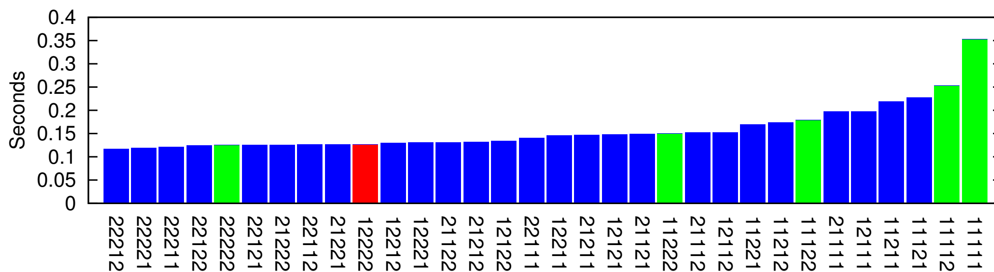


Figure 4.5: Time breakdown on 4096 cores. In each case, the expected number of nonzeros in the output is $d^2 n = 2^{30}$. For $n = 2^{30}$, $d = 1$, the block row algorithm spent 39.3 seconds on communication, 11.9 seconds on local computation, and 0.985 seconds were due to load imbalance, but the plot is cut off at 12 seconds.

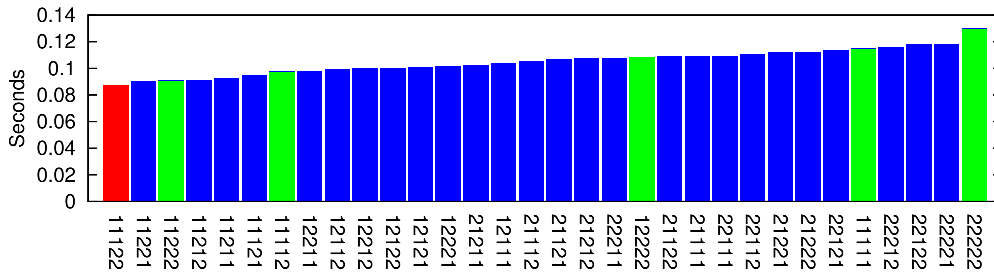
perform best. In three of the four cases this is the fastest option, and in the fourth it is only 8.3% slower than the best interleaving. This suggests that following that rule will give very close to optimal performance, and tuning is probably not necessary.



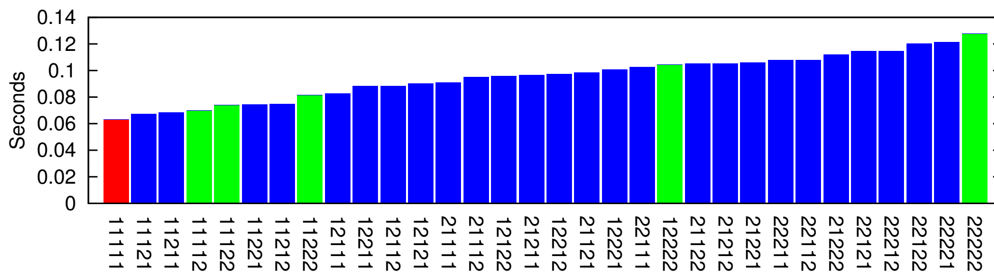
(a) $n = 2^{26}$, $d = 2^0$



(b) $n = 2^{22}$, $d = 2^2$



(c) $n = 2^{18}$, $d = 2^4$



(d) $n = 2^{14}$, $d = 2^6$

Figure 4.6: Possible interleavings of the recursive algorithm on 1024 processors. The red bar corresponds to taking $\log_2 d - 1$ steps of type 1 followed by the remaining steps of type 2. Green bars correspond to taking all type 1 steps before the first type 2 step. Blue bars correspond to other interleavings.

Chapter 5

Beyond Matrix Multiplication

In this chapter we show how to use the BFS/DFS approach to parallelize various recursive algorithms beyond matrix multiplication. We start with naïve n-body interaction, for which the recursive calls are independent. We then examine several algorithms with dependencies between the recursive calls.

5.1 Naïve n-body Interaction

For an n-body simulation, at each timestep we want to compute the force on each particle due to each other particle. This can be computed using the naïve iterative algorithm as:

```
for i in 1 : n do
    for j in 1 : n do
        F_i = force(i, j)
```

where force(i, j) computes the force on particle i due to particle j.

To recast this as a recursive algorithm, generalize it slightly to computing the forces on n particle in a list A due to n particles in a (possibly different) list B. Then the recursive algorithm is:

```
function RecNbody(A,B,n)
    if n = 1 then
        force(A(1), B(1))
    else
        RecNbody(A(1 : n/2), B(1 : n/2))
        RecNbody(A(1 : n/2), B(n/2 + 1 : n))
        RecNbody(A(n/2 + 1 : n), B(1 : n/2))
        RecNbody(A(n/2 + 1 : n), B(n/2 + 1 : n))
```

The function makes four recursive calls to itself. The four calls are not quite independent: two of them update each particle in A. If we instead make two copies A for the two different calls, and then combine the contributions at the end, they become independent at the cost

of $O(n)$ work after the recursive calls. The parallelization is now very similar to the case of square matrix multiplication.

5.1.1 Unlimited Memory

Assume, for now, that memory is not a limiting factor. Then we can perform the calculation by doing k BFS steps, followed by local computation. For each BFS step, the communication consists of 3 all-to-all's between disjoint sets of 4 processors: one to redistribute A, one to redistribute B, and one to collect the forces. Assume that a single particle or force can be represented by $O(1)$ words. The number of words in each message is all the data a processor has about half of the particles (or forces). Thus at each step, each processor sends and receives $O(1)$ messages of length $O(n/P)$. The recurrences for bandwidth and latency are thus

$$W(n,P) = W(n/2, P/4) + O(n/P) = O(n/\sqrt{P}),$$
$$S(n,P) = S(n/2, P/4) + O(1) = O(\log P).$$

The memory usage is a geometric sum, since each BFS step needs twice as much memory as the previous, and is dominated by the last term:

$$M(n,P) = O(2^k n/P) = O(n/\sqrt{P}).$$

5.1.2 Limited Memory

If there is not enough memory to perform the unlimited memory scheme above, it is possible to perform DFS steps, each of which reduces the future memory requirements by a factor of 2. If there is M local memory available, take

$$\ell = \log_2 \frac{n}{M\sqrt{P}} + c,$$

for some constant c, DFS steps followed by k BFS steps. This fits inside the available local memory M, and consists of 4^ℓ calls to the unlimited memory case with size $n/2^\ell$. Thus its costs are

$$W(n,P) = 4^\ell O\left(\frac{n}{2^\ell\sqrt{P}}\right) = O\left(\frac{n^2}{MP}\right),$$
$$S(n,P) = 4^\ell O(\log P) = O\left(\frac{n^2}{M^2P}\log P\right).$$

These asymptotically match the communication costs of the 1.5D iterative algorithm and the lower bounds presented in [33].

5.2 Dealing with Dependencies

So far, all of the algorithm we have considered have had independent recursive calls, possibly with some work at the beginning and end. For many recursive algorithms, however, this is not the case. To obtain communication-efficient parallelizations in such cases, we modify BFS/DFS approach by allowing the algorithm to discard some fraction of the processors at each recursive step. The reason is to avoid having to solve many small base cases on all P processors, which would incur a high latency cost. Discarding processors will always increase the arithmetic and bandwidth cost, but if we do not discard too many at each recursive step, we can keep these from increasing by more than constant factors.

5.3 All-Pairs Shortest Paths

Consider the Floyd-Warshall algorithm for computing all-pairs shortest paths on a graph (the minimum distance between every pair of vertices in a graph). We can write the problem using three nested loops so that it looks like matrix multiplication, except with addition and multiplication replaced by taking minima and addition, respectively. Additionally, unlike matrix multiplication, there are dependencies between the iterations in the inner loop, so they cannot be arbitrarily re-ordered and parallelized.

Like matrix multiplication, the Floyd-Warshall algorithm can be recast as a recursive algorithm, where each index is split in half and there are 8 subproblems to solve [55]. Because of the dependencies, all 8 subproblems must be solved in order. Fortunately, however, 6 of the 8 subproblems have no internal dependencies, and thus have exactly the same asymptotic costs as matrix multiplication, while the remaining 2 have the same dependencies as the original problem.

Consider starting with the problem of size n on P processors, using only P/a of those processors to solve the two subproblems. The cost recurrence is then $\texttt{FW}(n,P) = 2\texttt{FW}(n/2,P/a) + 6\texttt{GEMM}(n/2,P) + \beta n^2/P + \alpha$. Here $\texttt{FW}(n,P)$ is the cost (arithmetic, bandiwdth, or latency) of the recursive Floyd-Warshall algorithm, $\texttt{GEMM}(n,P)$ is the cost of square $n \times n$ matrix multiplication on P processors, and $\beta n^2/P + \alpha$ is the communication cost of redistributing the input and output at each recursive step. Assuming $a > 1$, the base case computation is performed on one processor, without incurring any communication costs, after $\log_a P$ recursive steps. In terms of flops, this gives

$$F(n,P) = 2F\left(\frac{n}{2},\frac{P}{a}\right) + O\left(\frac{n^3}{P}\right) = \sum_{i=1}^{\log_a P} O\left(\frac{n^3}{P}\left(\frac{a}{4}\right)^i\right) = \begin{cases} O\left(\frac{n^3}{P}\right) & a < 4 \\ O\left(\frac{n^3}{P^{\log_4 a}}\right) & a > 4 \end{cases}.$$

Thus as long as $a < 4$ (at least 1/4 of the processors are kept at each recursive step), the computation is asymptotically load balanced, although it will suffer a constant factor increase in the flop cost. Taking $a > 4$ turns out to be asymptotically equivalent to discarding some

of the processors at the beginning and then using a smaller value of a, so we do not consider that case further.

In terms of bandwidth cost, it gives

$$W(n,P) = 2W\left(\frac{n}{2},\frac{P}{a}\right) + O\left(\frac{n^2}{P^{2/3}} + \frac{n^3}{P\sqrt{M}}\right) = \begin{cases} O\left(\frac{n^2}{P^{2/3}} + \frac{n^3}{P\sqrt{M}}\right) & a < 2^{3/2} \\ O\left(\frac{n^2}{P^{\log_a 2}} + \frac{n^3}{P\sqrt{M}}\right) & 2^{3/2} < a < 4 \end{cases}.$$

The communication lower bounds of [10, 5] apply to Floyd-Warshall, so the recursive algorithm is bandwidth-optimal for small values of a. The value of a between $a = 2^{3/2}$ and $a = 4$ at which it becomes not bandwidth-optimal depends on how much memory is available.

The latency cost is

$$S(n,P) = 2S\left(\frac{n}{2},\frac{P}{a}\right) + O\left(1 + \frac{n^3}{PM^{3/2}}\right) = O\left(P^{\log_a 2} + \frac{n^3}{PM^{3/2}}\right),$$

as long as $a < 4$. Increasing a decreases the latency cost, because it avoids having many subproblems to solve, each using many processors. Because of the dependencies, the latency is always higher than $\log P$, unlike for matrix multiplication. For $a > 2^{3/2}$ (assuming there is enough memory), the latency and bandwidth match the conjectured tradeoff lower bound of [59]. The asymptotic costs of their iterative algorithm are very similar to the costs of the recursive algorithm shown above: their case of $c = 1$ corresponds here to $a = 4$, and their case of $c = P^{1/3}$ corresponds to $a = 2^{3/2}$. A similar algorithm was also presented in [63]. The recursive algorithm requires no more than a constant factor extra memory for any $a \leq 4$, although if more is available the subproblems that look like matrix multiplication benefit from it.

5.4 Triangular Solve with Multiple Righthand Sides

Next consider solving a triangular system $LX = Y$, where L is an $n \times n$ lower triangular matrix, and X and Y are $n \times m$ matrices; the inputs are L and Y and the output is X.

First consider the square case that $m = n$. A recursive algorithm is given in [8] that splits both dimensions of all three matrices in half, and has four recursive calls to triangular solves, and two matrix multiplications. The critical path consists of one triangular solve, then one matrix multiplication, then another triangular solve; meanwhile the other half of the computation can be done in parallel. This leads to a cost recurrence of $\texttt{TRSM}(n,n,P) = 2\texttt{TRSM}(n/2,n/2,P/(2a)) + \texttt{GEMM}(n/2,P/2) + \beta n^2/P + \alpha$. The base case is local computation on one processor after $\log_{2a} P$ recursive steps. As in the case of all pairs shortest path, we analyze the flop, bandwidth, and latency costs as a function of a. The flop cost is

$$F(n,P) = 2F\left(\frac{n}{2},\frac{P}{2a}\right) + O\left(\frac{n^3}{P}\right) = O\left(\frac{n^3}{P}\right),$$

as long as $a < 2$. The bandwidth cost is

$$W(n,P) = 2W\left(\frac{n}{2},\frac{P}{2a}\right) + O\left(\frac{n^2}{P^{2/3}} + \frac{n^3}{P\sqrt{M}}\right) = \begin{cases} O\left(\frac{n^2}{P^{2/3}} + \frac{n^3}{P\sqrt{M}}\right) & a < \sqrt{2} \\ O\left(\frac{n^2}{P^{\log_{2a} 2}} + \frac{n^3}{P\sqrt{M}}\right) & \sqrt{2} < a < 2 \end{cases}.$$

The latency cost is

$$S(n,P) = 2S\left(\frac{n}{2},\frac{P}{2a}\right) + O\left(1 + \frac{n^3}{PM^{3/2}}\right) = O\left(P^{\log_{2a} 2} + \frac{n^3}{PM^{3/2}}\right),$$

as long as $a < 2$. Increasing a decreases latency cost at the expense of higher bandwidth and flop cost. Since we are limited to $a < 2$, the latency cost cannot be reduced below $O(P^{1/2})$. For small values of a, the bandwidth cost asymptotically matches the lower bounds of [10, 5].

We can improve the latency cost, even for the square case, by allowing the two dimensions to be split separately. In [11], it is shown how to break the recursive algorithm of [8] into two recursive steps. One step splits m, creating two triangular solves on matrices of half as many columns that can be performed in parallel. The other step splits n, creating two triangular solves with half as many rows, and a matrix multiplication, all three of which must be performed in order. Consider an algorithm that combines one step of splitting n with k steps of splitting m (so the algorithm considered above is the case $k = 1$). This leads to a recurrence of $\texttt{TRSM}(n,m,P) = 2\texttt{TRSM}(n/2, m/2^k, P/(2^k a)) + O(\texttt{GEMM}(n,n,m,P)) + (mn+n^2)/P\beta$, where the last term is the bandwidth cost of redistribution at each step. The flop cost is

$$F(n,m,P) = 2F\left(\frac{n}{2},\frac{m}{2^k},\frac{P}{2^k a}\right) + O\left(\frac{n^3}{P}\right) = O\left(\frac{n^2 m}{P}\right),$$

as long as $a < 2$. The bandwidth cost due to the matrix multiplications is

$$\begin{aligned} W_{MM}(n,m,P) &= 2W_{MM}\left(\frac{n}{2},\frac{m}{2^k}\frac{P}{2^k a}\right) + O\left(\frac{(mn^2)^{2/3}}{P^{2/3}} + \frac{mn^2}{P\sqrt{M}}\right) \\ &= \begin{cases} O\left(\frac{(mn^2)^{2/3}}{P^{2/3}} + \frac{mn^2}{P\sqrt{M}}\right) & a < \sqrt{2} \\ O\left(\frac{(mn^2)^{2/3}}{P^{\log_{2^k a} 2^{2k/3+1/3}}} + \frac{mn^2}{P\sqrt{M}}\right) & \sqrt{2} < a < 2 \end{cases}. \end{aligned}$$

The bandwidth cost due to redistribution is

$$W_R(n,P) = 2W_R\left(\frac{n}{2},\frac{m}{2^k}\frac{P}{2^k a}\right) + O\left(\frac{n^2}{P} + \frac{mn}{P}\right) = O\left(\frac{mn}{P^{\log_{2^k a} 2^k}}\right) + \begin{cases} O\left(\frac{n^2}{P}\right) & 2^k a < 2 \\ O\left(\frac{n^2}{P^{\log_{2^k a} 2}}\right) & 2^k a > 2 \end{cases}.$$

The total bandwidth cost is the sum of these two contributions. The latency cost is

$$S(n,P) = 2S\left(\frac{n}{2},\frac{P}{2a}\right) + O\left(1 + \frac{n^3}{PM^{3/2}}\right) = O\left(P^{\log_{2^k a} 2} + \frac{n^3}{PM^{3/2}}\right),$$

as long as $a < 2$. In general, one should choose k and a so that $W_R = W_{MM}$; there will still be a remaining degree of freedom, and increasing k or a will decrease latency at the expense of higher bandwidth.

A simple example is the square case $m = n$, with $a = 1$ and $k \geq 3/2$. Then the above costs simplify to

$$F = O\left(\frac{n^3}{P}\right), \qquad W = O\left(\frac{n^2}{P^{1/k}} + \frac{n^3}{P\sqrt{M}}\right), \qquad S = O\left(P^{1/k} + \frac{n^3}{PM^{3/2}}\right).$$

Note that by taking large values of k, the latency is substantially reduced relative to the previous algorithm, at to cost of further increase to the bandwidth term. It is only possible to run this algorithm if $m \geq P$, since with $a = 1$ the only reduction in number of processors comes from splitting m. If we instead take $\sqrt{2} < a < 2$ and $k \geq 5/2$ and define $b = k + \log_2 a$, the asymptotic costs are

$$F = O\left(\frac{n^3}{P}\right), \qquad W = O\left(\frac{n^2}{P^{1/b}} + \frac{n^3}{P\sqrt{M}}\right), \qquad S = O\left(P^{1/b} + \frac{n^3}{PM^{3/2}}\right),$$

and the algorithm works as long as $m \geq P^{k/(k+\log_2 a)}$.

The latency costs are substantially higher than the lower bounds of [10]. We conjecture that taking into account dependencies will give tighter lower bounds.

5.5 Cholesky Decomposition

Sequential Cholesky decomposition of an $n \times n$ matrix can be done recursively by making two calls to Cholesky decomposition on an $\frac{n}{2} \times \frac{n}{2}$ matrix, one call to a square triangular solve, and one call to matrix multiplication [3, 8]. All four of those calls must be done in order. Thus our parallelization gives the recurrence $\texttt{CHOL}(n, P) = 2\texttt{CHOL}(n/2, P/a) + O(\texttt{GEMM}(n, n, n, P) + \texttt{TRSM}(n, n, P) + n^2/P\beta)$. For the costs of the triangular solve, use the $k = 3/2$, $a = 1$ algorithm from the previous section; further reductions to the latency cost of the triangular solve do not reduce the latency cost of the Cholesky decomposition.

In terms of flops, we have

$$F(n, P) = 2F\left(\frac{n}{2}, \frac{P}{a}\right) + O\left(\frac{n^3}{P}\right) = O\left(\frac{n^3}{P}\right),$$

for $a < 4$. The bandwidth cost is

$$W(n, P) = 2W\left(\frac{n}{2}, \frac{P}{a}\right) + O\left(\frac{n^2}{P^{2/3}} + \frac{n^3}{P\sqrt{M}}\right) = \begin{cases} O\left(\frac{n^2}{P^{2/3}} + \frac{n^3}{P\sqrt{M}}\right) & a < 2^{3/2} \\ O\left(\frac{n^2}{P^{\log_a 2}} + \frac{n^3}{P\sqrt{M}}\right) & 2^{3/2} < a < 4 \end{cases}.$$

The latency cost is

$$S(n, P) = 2S\left(\frac{n}{2}, \frac{P}{a}\right) + O\left(P^{2/3} + \frac{n^3}{PM^{3/2}}\right) = O\left(P^{\log_a 2} + \frac{n^3}{PM^{3/2}}\right),$$

as long as $a < 4$. Note that increasing a from 1 to $2^{3/2}$ decreases the latency cost without any asymptotic affect on the bandwidth cost. The interesting range is $2^{3/2} < a < 4$, for which the costs are

$$F = O\left(\frac{n^3}{P}\right), \qquad W = O\left(\frac{n^2}{P^{\log_a 2}} + \frac{n^3}{P\sqrt{M}}\right), \qquad S = O\left(P^{\log_a 2} + \frac{n^3}{PM^{3/2}}\right).$$

This algorithm is similar to the pivot-free generic Gaussian elimination algorithm of [64], although that algorithm requires as many all-to-all communication phases as ours requires messages, and so it may be less efficient in practice. The costs are also equivalent to the iterative 2.5D Cholesky algorithm [37]; $c = 1$ there corresponds to $a = 4$, and $c = P^{1/3}$ corresponds to $a = 2^{3/2}$.

Chapter 6

Discussion and Open Questions

The original impetus for this work was the search for a communication-optimal parallel algorithm for Strassen's matrix multiplication. Following the communication-cost lower bounds of [9], we were unable to find a matching algorithm in the parallel case in the literature (although we later found an algorithm quite similar to our CAPS algorithm in [52]). Since, unlike most linear algebra algorithms, Strassen's algorithm has no simple iterative definition, we found a method of parallelization that respects the recursive structure of the algorithm: by traversing the recursion tree using BFS and DFS steps.

We call this algorithm Communication-Avoiding Parallel Strassen (or CAPS), which appears in Chapter 2, but the idea easily applies more broadly than just to Strassen's algorithm. Many other fast matrix multiplication algorithms, including those with exponents much smaller than Strassen's, have similar recursive formulations. However several practical issues prevent their use, and we are not aware of any practical algorithm with a substantially better exponent than Strassen's algorithm.

Question 1. *Is there a matrix multiplication algorithm that is faster than Strassen's algorithm in practice? If so, it may have a much higher branching factor than Strassen's algorithm (for example, the algorithms in [47] have thousands of recursive calls or more per step). Would the BFS/DFS approach work well in practice given the high branching factor?*

One other practical example (though with higher exponent than Strassen's algorithm) is the recursive formulation of classical matrix multiplication. Splitting each of the three dimensions in half gives 8 subproblems instead of the 7 subproblems in the case of Strassen's algorithm. Applying the BFS/DFS approach to this algorithm gives a recursive communication-optimal parallel algorithm with asymptotically identical costs to the 2.5D algorithm [52, 61].

In fact, the classical algorithm has more implementation freedom than Strassen's algorithm. Instead of splitting all three dimensions at once, one may split only one at a time. For rectangular matrices, it is natural to expect that splitting the largest dimension, to create closer to square subproblems, would save on communication (as in the sequential case [34]). When we first applied the BFS/DFS approach to this idea, it did indeed outperform previous algorithms for rectangular matrices. But when we proved lower bounds for classical

rectangular matrix multiplication, in some cases they were lower than the costs of our initial algorithm. To match the lower bounds, we realized that, with the right data layout, only the smallest matrix needs to be redistributed at each recursive step. This improvement gives the CARMA algorithm as presented in Chapter 3. Unfortunately, to communicate only one matrix at each recursive step seems to require that the initial data layout of a matrix depends on the other input matrix. There does not seem to be one standard layout that works well in all cases; rather the best layout depends on the context.

Question 2. *Can a library be developed that reaches the communication lower bounds of Section 3.1 for any sequence of matrix multiplications, possibly by changing data layouts when appropriate?*

When input matrices are sparse, the problem becomes more complicated. Even for the square case that we analyzed in Chapter 4, the number of nonzeros in the input and output matrices may be very different. Thus, as in the rectangular case, a natural algorithm is to communicate the smallest matrix (measured by number of nonzeros) and split the other dimension at each recursive step. For matrices with non-uniform distribution, the correct split may not be in the middle, and the split that balances data size may be different from the one that balances work. For the special case of square, random matrices with sparse output we showed that this algorithm is communication-optimal among sparsity-independent algorithms.

Question 3. *Using an efficient way to estimate the number of nonzeros in the output matrix, one could generalize the recursive algorithm to arbitrary sparse matrices. What can be proved about the communication costs of such an algorithm?*

Moving beyond matrix multiplication, many recursive algorithms have dependencies between the recursive calls. We analyzed several of these algorithms in Chapter 5 (all pairs shortest path, triangular solve with multiple righthand sides, and Cholesky decomposition), and presented bandwidth optimal algorithms based on the BFS/DFS approach. However the algorithms with dependencies do not match existing latency cost lower bounds. There are several open questions:

Question 4. *Can stronger latency lower bounds be proved for algorithms with dependencies? In other words, is the bandwidth-latency tradeoff exhibited by the best existing algorithms necessary?*

Question 5. *Would implementations of the algorithms in Chapter 5 perform well in practice? In particular, is the technique of discarding processors at each recursive step, to control latency costs, effective?*

Question 6. *Among the class of cache-oblivious sequential communication-optimal algorithms, which ones give communication-optimal parallel algorithms using the BFS/DFS approach? To what extent can the parallelization of recursive algorithms be automated?*

Acknowledgments

We acknowledge funding from Microsoft (award #024263) and Intel (award #024894), and matching funding by UC Discovery (award #DIG07-10227), with additional support from ParLab affiliates National Instruments, Nokia, NVIDIA, Oracle, and Samsung, and support from MathWorks. We also acknowledge the support of the US DOE (grants DE-SC0003959, DE-SC0004938, DE-SC0005136, DE-SC0008700, DE-AC02-05CH11231, DE-FC02-06ER25753, and DE-FC02-07ER25799), DARPA (award #HR0011-12-2-0016),

This research used resources of the Argonne Leadership Computing Facility at Argonne National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under contract DE-AC02-06CH11357, resources of the National Energy Research Scientific Computing Center, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231, and resources of the Oak Ridge Leadership Facility at the Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under contract No. DE-AC05-00OR22725.

Bibliography

[1] M. D. Adams and D. S. Wise. "Seven at one stroke: results from a cache-oblivious paradigm for scalable matrix algorithms". In: *MSPC '06: Proceedings of the 2006 Workshop on Memory System Performance and Correctness*. New York, NY, USA: ACM, 2006, pp. 41–50. ISBN: 1-59593-578-9. DOI: 10.1145/1178597.1178604.

[2] R. C. Agarwal, S. M. Balle, F. G. Gustavson, M. Joshi, and P. Palkar. "A three-dimensional approach to parallel matrix multiplication". In: *IBM Journal of Research and Development* 39 (1995), pp. 39–5.

[3] N. Ahmed and K. Pingali. "Automatic generation of block-recursive codes". In: *Euro-Par '00: Proceedings from the 6th International Euro-Par Conference on Parallel Processing*. London, UK: Springer-Verlag, 2000, pp. 368–378. ISBN: 3-540-67956-1.

[4] G. Ballard, A. Buluç, J. Demmel, L. Grigori, B. Lipshitz, O. Schwartz, and S. Toledo. "Communication optimal parallel multiplication of sparse random matrices". In: *Proceedings of the 25th ACM Symposium on Parallelism in Algorithms and Architectures*. SPAA '13. 2013.

[5] G. Ballard, J. Demmel, O. Holtz, B. Lipshitz, and O. Schwartz. "Brief announcement: strong scaling of matrix multiplication algorithms and memory-independent communication lower bounds". In: *Proceedings of the 24th ACM Symposium on Parallelism in Algorithms and Architectures*. SPAA '12. New York, NY, 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.

[6] G. Ballard, J. Demmel, O. Holtz, B. Lipshitz, and O. Schwartz. "Communication-optimal parallel algorithm for Strassen's matrix multiplication". In: *Proceedings of the 24th ACM Symposium on Parallelism in Algorithms and Architectures*. SPAA '12. New York, NY, USA: ACM, 2012, pp. 193–204. ISBN: 978-1-4503-1213-4. DOI: 10.1145/2312005.2312044. URL: http://doi.acm.org/10.1145/2312005.2312044.

[7] G. Ballard, J. Demmel, O. Holtz, B. Lipshitz, and O. Schwartz. "Graph expansion analysis for communication costs of fast rectangular matrix multiplication". In: *Proceedings of the Mediterranean Conference on Algorithms*. 2012.

[8] G. Ballard, J. Demmel, O. Holtz, and O. Schwartz. "Communication-optimal parallel and sequential Cholesky decomposition". In: *SIAM Journal on Scientific Computing* 32.6 (2010), pp. 3495–3523. DOI: 10.1137/090760969. URL: http://link.aip.org/link/?SCE/32/3495/1.

[9] G. Ballard, J. Demmel, O. Holtz, and O. Schwartz. "Graph expansion and communication costs of fast matrix multiplication: regular submission". In: *Proceedings of the 23rd ACM Symposium on Parallelism in Algorithms and Architectures*. SPAA '11. San Jose, California, USA: ACM, 2011, pp. 1–12. ISBN: 978-1-4503-0743-7. DOI: 10.1145/1989493.1989495. URL: http://doi.acm.org/10.1145/1989493.1989495.

[10] G. Ballard, J. Demmel, O. Holtz, and O. Schwartz. "Minimizing communication in numerical linear algebra". In: *SIAM Journal on Matrix Analysis and Applications* 32.3 (2011), pp. 866–901. DOI: 10.1137/090769156. URL: http://link.aip.org/link/?SML/32/866/1.

[11] G. Ballard, J. Demmel, B. Lipshitz, O. Schwartz, and S. Toledo. "Communication efficient Gaussian elimination with partial pivoting using a shape morphing data layout". In: *Proceedings of the 25th ACM Symposium on Parallelism in Algorithms and Architectures*. SPAA '13. 2013.

[12] J. Berntsen. "Communication efficient matrix multiplication on hypercubes". In: *Parallel Computing* 12.3 (1989), pp. 335 –342. ISSN: 0167-8191. DOI: 10.1016/0167-8191(89)90091-4. URL: http://www.sciencedirect.com/science/article/pii/0167819189900914.

[13] G. Bilardi, A. Pietracaprina, G. Pucci, and F. Silvestri. "Network-oblivious algorithms". In: *Proceedings of 21st International Parallel and Distributed Processing Symposium*. 2007.

[14] D. Bini, M. Capovani, F. Romani, and G. Lotti. "$O(n^{2.7799})$ complexity for $n \times n$ approximate matrix multiplication". In: *Information Processing Letters* 8.5 (1979), pp. 234 –235. ISSN: 0020-0190. DOI: DOI:10.1016/0020-0190(79)90113-3. URL: http://www.sciencedirect.com/science/article/B6V0F-45FCWS8-2J/2/5fe4a4c2f661f7d574409b8999cb3914.

[15] J. Bruck, C.-T. Ho, S. Kipnis, and D. Weathersby. "Efficient algorithms for all-to-all communications in multi-port message-passing systems". In: *Proceedings of the sixth annual ACM symposium on Parallel algorithms and architectures*. SPAA '94. New York, NY, USA: ACM, 1994, pp. 298–309.

[16] N. H. Bshouty. "On the additive complexity of 2×2 matrix multiplication". In: *Information processing letters* 56.6 (1995), pp. 329 –335. ISSN: 0020-0190. DOI: 10.1016/0020-0190(95)00176-X. URL: http://www.sciencedirect.com/science/article/pii/002001909500176X.

[17] A. Buluç and J. Gilbert. "The Combinatorial BLAS: design, implementation, and applications". In: *Int. J. High Perform. Comput. Appl.* 25.4 (Nov. 2011), pp. 496–509.

[18] A. Buluç and J. R. Gilbert. "Challenges and advances in parallel sparse matrix-matrix multiplication". In: *ICPP'08: Proc. of the Intl. Conf. on Parallel Processing*. Portland, Oregon, USA: IEEE Computer Society, 2008, pp. 503–510.

[19] A. Buluç and J. R. Gilbert. "Parallel sparse matrix-matrix multiplication and indexing: implementation and experiments". In: *SIAM Journal of Scientific Computing (SISC)* 34.4 (2012), pp. 170 –191.

[20] P. Bű́rgisser, M. Clausen, and M. A. Shokrollahi. *Algebraic Complexity Theory*. Grundlehren der mathematischen Wissenschaften 315. Springer Verlag, 1997. ISBN: 3-540-60582-7.

[21] L. Cannon. "A cellular computer to implement the Kalman filter algorithm". PhD thesis. Bozeman, MN: Montana State University, 1969.

[22] M. Challacombe. "A general parallel sparse-blocked matrix multiply for linear scaling SCF theory". In: *Computer Physics Communications* 128.1-2 (2000), pp. 93–107.

[23] J. Choi. "A new parallel matrix multiplication algorithm on distributed-memory concurrent computers". In: *Concurrency: practice and experience* 10.8 (1998), pp. 655–670. ISSN: 1096-9128. DOI: 10.1002/(SICI)1096-9128(199807)10:8<655::AID-CPE369>3.0.CO;2-O. URL: http://dx.doi.org/10.1002/(SICI)1096-9128(199807)10:8<655::AID-CPE369>3.0.CO;2-O.

[24] J. Choi, D. W. Walker, and J. J. Dongarra. "PUMMA: parallel universal matrix multiplication algorithms on distributed memory concurrent computers". In: *Concurrency: Practice and Experience* 6.7 (1994), pp. 543–570. ISSN: 1096-9128. DOI: 10.1002/cpe.4330060702. URL: http://dx.doi.org/10.1002/cpe.4330060702.

[25] R. A. Chowdhury, F. Silvestri, B. Blakeley, and V. Ramachandran. "Oblivious algorithms for multicores and network of processors". In: *IPDPS*. 2010, pp. 1–12.

[26] R. Cole and V. Ramachandran. "Resource oblivious sorting on multicores". In: *Proceedings of the 37th International Colloquium Conference on Automata, Languages and Programming*. ICALP'10. Berlin, Heidelberg: Springer-Verlag, 2010, pp. 226–237. ISBN: 3-642-14164-1, 978-3-642-14164-5. URL: http://dl.acm.org/citation.cfm?id=1880918.1880944.

[27] D. Coppersmith. "Rapid multiplication of rectangular matrices". In: *SIAM Journal on Computing* 11.3 (1982), pp. 467–471. DOI: 10.1137/0211037. URL: http://link.aip.org/link/?SMJ/11/467/1.

[28] D. Coppersmith. "Rectangular matrix multiplication revisited". In: *J. complex.* 13 (1 Mar. 1997), pp. 42–49. ISSN: 0885-064X. DOI: 10.1006/jcom.1997.0438. URL: http://portal.acm.org/citation.cfm?id=270488.270504.

[29] J. Demmel, I. Dumitriu, and O. Holtz. "Fast linear algebra is stable". In: *Numerische Mathematik* 108 (1 2007). 10.1007/s00211-007-0114-x, pp. 59–91. ISSN: 0029-599X. URL: http://dx.doi.org/10.1007/s00211-007-0114-x.

[30] J. Demmel, I. Dumitriu, O. Holtz, and R. Kleinberg. "Fast matrix multiplication is stable". In: *Numerische Mathematik* 106.2 (2007), pp. 199–224.

[31] J. Demmel, A. Fox, S. Kamil, B. Lipshitz, O. Schwartz, and O. Spillinger. "Communication-optimal parallel recursive rectangular matrix multiplication". In: *IPDPS*. 2013.

[32] J. Demmel, A. Gearhart, B. Lipshitz, and O. Schwartz. "Perfect strong scaling using no additional energy". In: *IPDPS*. 2013.

[33] M. Driscoll, P. Koanantakool, E. Georganas, E . Solomonik, and K. Yelick. "A Communication-Optimal N-Body Algorithm for Short-Range, Direct Interactions." In: *IPDPS*. 2013.

[34] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran. "Cache-oblivious algorithms". In: *FOCS '99: Proceedings of the 40th Annual Symposium on Foundations of Computer Science*. Washington, DC, USA: IEEE Computer Society, 1999, p. 285. ISBN: 0-7695-0409-4.

[35] S. H. Fuller and L. I. Millett, eds. *The Future of Computing Performance: Game Over or Next Level?* 200 pages, http://www.nap.edu. Washington, D.C.: The National Academies Press, 2011.

[36] R. A. van de Geijn and J. Watts. "SUMMA: scalable universal matrix multiplication algorithm". In: *Concurrency - Practice and Experience* 9.4 (1997), pp. 255–274.

[37] E. Georganas, J. González-Domínguez, E. Solomonik, Y. Zheng, J. Tourino, and K. Yelick. "Communication avoiding and overlapping for numerical linear algebra". In: *High Performance Computing, Networking, Storage and Analysis (SC), 2012 International Conference for*. IEEE. 2012, pp. 1–11.

[38] S. L. Graham, M. Snir, and C. A. Patterson, eds. *Getting up to Speed: The Future of Supercomputing*. Report of National Research Council of the National Academies Sciences. 289 pages, http://www.nap.edu. Washington, D.C.: The National Academies Press, 2004.

[39] B. Grayson, A. Shah, and R. van de Geijn. "A high performance parallel Strassen implementation". In: *Parallel Processing Letters*. Vol. 6. 1995, pp. 3–12.

[40] J. Gunnels, C. Lin, G. Morrow, and R. Van De Geijn. "A flexible class of parallel matrix multiplication algorithms". In: *Parallel Processing Symposium, 1998. IPPS/SPDP 1998. Proceedings of the First Merged International... and Symposium on Parallel and Distributed Processing 1998*. IEEE. 1998, pp. 110–116.

[41] N. J. Higham. *Accuracy and Stability of Numerical Algorithms*. 2nd. Philadelphia, PA: SIAM, 2002.

[42] J. E. Hopcroft and L. R. Kerr. "On minimizing the number of multiplications necessary for matrix multiplication". English. In: *SIAM Journal on Applied Mathematics* 20.1 (1971), pp. 30–36. ISSN: 00361399.

[43] X. Huang and V. Y. Pan. "Fast rectangular matrix multiplication and applications". In: *J. Complex.* 14 (2 June 1998), pp. 257–299. ISSN: 0885-064X. DOI: 10.1006/jcom.1998.0476. URL: http://portal.acm.org/citation.cfm?id=299029.299038.

[44] X. Huang and V. Y. Pan. "Fast rectangular matrix multiplications and improving parallel matrix computations". In: *Proceedings of the Second International Symposium on Parallel Symbolic Computation*. PASCO '97. New York, NY, USA: ACM, 1997, pp. 11–23. ISBN: 0-89791-951-3. DOI: http://doi.acm.org/10.1145/266670.266679. URL: http://doi.acm.org/10.1145/266670.266679.

[45] 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.

[46] C. P. Kruskal, L. Rudolph, and M. Snir. "Techniques for parallel manipulation of sparse matrices". In: *Theor. Comput. Sci.* 64.2 (1989), pp. 135–157.

[47] J. Laderman, V. Pan, and X.-H. Sha. "On practical algorithms for accelerated matrix multiplication". In: *Linear Algebra and Its Applications* 162 (1992), pp. 557–588.

[48] B. Lipshitz, G. Ballard, J. Demmel, and O. Schwartz. "Communication-avoiding parallel Strassen: implementation and performance". In: *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. IEEE Computer Society Press. 2012, p. 101.

[49] L. H. Loomis and H. Whitney. "An inequality related to the isoperimetric inequality". In: *Bulletin of the AMS* 55 (1949), pp. 961–962.

[50] G. Lotti and F. Romani. "On the asymptotic complexity of rectangular matrix multiplication". In: *Theoretical Computer Science* 23.2 (1983), pp. 171 –185. ISSN: 0304-3975. DOI: DOI:10.1016/0304-3975(83)90054-3. URL: http://www.sciencedirect.com/science/article/B6V1G-45F5WM7-1J/2/f5d51de21cdc48808f5e31f8788cf651.

[51] Q. Luo and J. Drake. "A scalable parallel Strassen's matrix multiplication algorithm for distributed-memory computers". In: *Proceedings of the 1995 ACM Symposium on Applied Computing*. SAC '95. Nashville, Tennessee, United States: ACM, 1995, pp. 221–226. ISBN: 0-89791-658-1. DOI: http://doi.acm.org/10.1145/315891.315965. URL: http://doi.acm.org/10.1145/315891.315965.

[52] W. F. McColl and A. Tiskin. "Memory-efficient matrix multiplication in the BSP model". In: *Algorithmica* 24 (3 1999), pp. 287–297. ISSN: 0178-4617.

[53] H. Meuer, E. Strohmaier, J. Dongarra, and H. Simon. *Top500 Supercomputer Sites*. www.top500.org. 2011.

[54] G. Morton. *A Computer Oriented Geodetic Data Base and a New Technique in File Sequencing*. International Business Machines Company, 1966.

[55] J.-S. Park, M. Penner, and V. K Prasanna. "Optimizing graph algorithms for improved cache performance". In: *Parallel and Distributed Systems, IEEE Transactions on* 15.9 (2004), pp. 769–782.

[56] R. L. Probert. "On the additive complexity of matrix multiplication". In: *SIAM Journal on Computing* 5.2 (1976), pp. 187–203.

[57] M. D. Schatz, J. Poulson, and R. A. van de Geijn. "Scalable universal matrix multiplication algorithms: 2d and 3d variations on a theme". In: *submitted to ACM Transactions on Mathematical Software* (2013).

[58] J. Shalf, S. S. Dosanjh, and J. Morrison. "Exascale computing technology challenges". In: *High Performance Computing for Computational Science - VECPAR 2010 - 9th International conference, Berkeley, CA, USA, June 22-25, 2010, Revised Selected Papers*. Ed. by J. M. L. M. Palma, M. J. Daydé, O. Marques, and J. C. Lopes. Vol. 6449. Lecture Notes in Computer Science. Springer, 2010, pp. 1–25. ISBN: 978-3-642-19327-9. DOI: http://dx.doi.org/10.1007/978-3-642-19328-6_1.

[59] E. Solomonik, A. Buluç, and J. Demmel. "Minimizing communication in all-pairs shortest paths". In: *IPDPS*. 2013.

[60] E. Solomonik and J. Demmel. *Communication-optimal parallel 2.5D matrix multiplication and LU factorization algorithms*. Tech. rep. UCB/EECS-2011-10. EECS Department, University of California, Berkeley, Feb. 2011. URL: http://www.eecs.berkeley.edu/Pubs/TechRpts/2011/EECS-2011-10.html.

[61] E. Solomonik and J. Demmel. "Communication-optimal parallel 2.5D matrix multiplication and LU factorization algorithms". In: *Euro-Par 2011 Parallel Processing*. Ed. by Emmanuel Jeannot, Raymond Namyst, and Jean Roman. Vol. 6853. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2011, pp. 90–109. ISBN: 978-3-642-23396-8. URL: http://dx.doi.org/10.1007/978-3-642-23397-5-10.

[62] V. Strassen. "Gaussian elimination is not optimal". In: *Numerische Mathematik* 13 (4 1969). 10.1007/BF02165411, pp. 354–356. ISSN: 0029-599X. URL: http://dx.doi.org/10.1007/BF02165411.

[63] A. Tiskin. "All-pairs shortest paths computation in the bsp model". In: *Proceedings of the 28th International Colloquium on Automata, Languages and Programming,* ICALP '01. London, UK, UK: Springer-Verlag, 2001, pp. 178–189. ISBN: 3-540-42287-0. URL: http://dl.acm.org/citation.cfm?id=646254.684237.

[64] A. Tiskin. "Communication-efficient parallel generic pairwise elimination". In: *Future Generation Computer Systems* 23.2 (2007), pp. 179 –188. ISSN: 0167-739X. DOI: 10.1016/j.future.2006.04.017. URL: http://www.sciencedirect.com/science/article/pii/S0167739X06000835.

[65] S. Winograd. "On the multiplication of 2×2 matrices". In: *Linear Algebra Appl.* 4.4 (Oct. 1971), pp. 381–388.

[66] 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*. June 1988, pp. 366–373. DOI: `10.1109/DCS.1988.12538`.

[67] R. Yuster and U. Zwick. “Fast sparse matrix multiplication”. In: *ACM Trans. Algorithms* 1.1 (2005), pp. 2–13. ISSN: 1549-6325. DOI: `http://doi.acm.org/10.1145/1077464.1077466`.