Communication-Optimal Parallel Algorithm for Strassen's Matrix Multiplication

Grey Ballard[*]
EECS Department
UC Berkeley
Berkeley, CA 94720
ballard@cs.berkeley.edu

James Demmel[*†]
Mathematics Department
and CS Division, UC Berkeley
Berkeley, CA 94720
demmel@cs.berkeley.edu

Olga Holtz[‡]
Mathematics Departments
UC Berkeley and TU Berlin
Berkeley, CA 94720
holtz@math.berkeley.edu

Benjamin Lipshitz[*]
EECS Department
UC Berkeley
Berkeley, CA 94720
lipshitz@cs.berkeley.edu

Oded Schwartz[§]
EECS Department
UC Berkeley
Berkeley, CA 94720
odedsc@cs.berkeley.edu

ABSTRACT

Parallel matrix multiplication is one of the most studied fundamental problems in distributed and high performance computing. We obtain a new parallel algorithm that is based on Strassen's fast matrix multiplication and minimizes communication. The algorithm outperforms all known parallel matrix multiplication algorithms, classical and Strassen-based, both asymptotically and in practice.

A critical bottleneck in parallelizing Strassen's algorithm is the communication between the processors. Ballard, Demmel, Holtz, and Schwartz (SPAA '11) prove lower bounds on these communication costs, using expansion properties of the underlying computation graph. Our algorithm matches these lower bounds, and so is communication-optimal. It exhibits perfect strong scaling within the maximum possible range.

Benchmarking our implementation on a Cray XT4, we obtain speedups over classical and Strassen-based algorithms ranging from 24% to 184% for a fixed matrix dimension $n = 94080$, where the number of processors ranges from 49 to 7203.

Our parallelization approach generalizes to other fast matrix multiplication algorithms.

Categories and Subject Descriptors: F.2.1 [Analysis of Algorithms and Problem Complexity]: Numerical Algorithms and Problems: Computations on matrices
ACM General Terms: algorithms
Keywords: parallel algorithms, communication-avoiding algorithms, fast matrix multiplication

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.*, [10, 6, 12, 1, 26, 21, 36, 11, 24, 31, 4, 3]). 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.

We obtain a new parallel algorithm based on Strassen's fast matrix multiplication.[1] It is more efficient than any other parallel matrix multiplication algorithm of which we are aware, including those that are based on classical ($\Theta(n^3)$) multiplication, and those that are based on Strassen's and other Strassen-like matrix multiplications. We compare the efficiency of the new algorithm with previous algorithms, and provide both asymptotic analysis (Sections 3 and 4) and benchmarking data (Section 5).

1.1 The communication bottleneck

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 inter-processor communication costs are in many cases significantly higher than the computational costs. Moreover, hardware trends predict that more problems will become

[*]Research supported by Microsoft (Award #024263) and Intel (Award #024894) funding and by matching funding by U.C. Discovery (Award #DIG07-10227). Additional support comes from Par Lab affiliates National Instruments, Nokia, NVIDIA, Oracle, and Samsung.

[†]Research is also supported by DOE grants DE-SC0003959, DE- SC0004938, and DE-AC02-05CH11231.

[‡]Research supported by the Sofja Kovalevskaja programme of Alexander von Humboldt Foundation and by the National Science Foundation under agreement DMS-0635607, while visiting the Institute for Advanced Study.

[§]Research supported by U.S. Department of Energy grants under Grant Numbers DE-SC0003959.

SPAA'12, June 25–27, 2012, Pittsburgh, Pennsylvania, USA.

[1]Our actual implementation uses the Winograd variant [38]; see Section 2.3 for details.

communication-bound in the future [20, 19]. Even matrix multiplication becomes communication-bound when run on sufficiently many processors. Given the importance of communication costs, it is preferable to match the performance of an algorithm to a communication lower bound, obtaining a communication-optimal algorithm.

1.2 Communication costs of matrix multiplication

We consider a distributed-memory parallel machine model as described in Section 2.1. The communication costs are measured as a function of the number of processors P, the local memory size M in words, and the matrix dimension n. Irony, Toledo, and Tiskin [24] proved that in the distributed memory parallel model, the bandwidth cost of classical n-by-n matrix multiplication is bounded below by $\Omega\left(\frac{n^3}{PM^{1/2}}\right)$. Using their technique one can also deduce a memory-independent bandwidth cost lower bound of $\Omega\left(\frac{n^2}{P^{2/3}}\right)$ [3] and generalize it to other classes of algorithms [5]. For a shared-memory model similar bounds were shown in [2]. Until recently, parallel classical matrix multiplication algorithms (e.g., "2D" [10, 36], and "3D" [6, 1]) have minimized communication only for specific values of M. The first algorithm that minimizes the communication costs for the entire range of M has recently been obtained by Solomonik and Demmel [31]. See Section 4.1 for more details.

None of these techniques for obtaining lower bounds or parallel algorithms generalizes to fast matrix multiplication, such as [33, 27, 7, 30, 29, 14, 34, 15, 13, 37]. A communication cost lower bound for fast matrix multiplication algorithms has only recently been obtained [4]: Strassen's algorithm run on a distributed-memory parallel machine has bandwidth cost $\Omega\left(\left(\frac{n}{M^{1/2}}\right)^{\omega_0} \cdot \frac{M}{P}\right)$ and latency cost $\Omega\left(\left(\frac{n}{M^{1/2}}\right)^{\omega_0} \cdot \frac{1}{P}\right)$, where $\omega_0 = \log_2 7$ (see Section 2.5). These bounds generalize to other, but not all, fast matrix multiplication algorithms, with ω_0 being the exponent of the computational complexity.

In the sequential case,[2] the lower bounds are attained by the natural recursive implementation [33] which is thus optimal. However, a communication-optimal parallel Strassen algorithm was not previously known. Previous parallel algorithms that use Strassen (*e.g.*, [21, 26, 18]) decrease the computational costs at the expense of higher communication costs. The factors by which these algorithms exceed the Strassen lower bounds are typically small powers of P and M, as discussed in Section 4. However both P and M can be large (*e.g.* on a modern supercomputer, one may have $P \sim 10^5$ and $M \sim 10^9$).

1.3 Parallelizing Strassen's matrix multiplication in a communication efficient way

The main impetus for this work was the observation of the asymptotic gap between the communication costs of existing parallel Strassen-based algorithms and the communication lower bounds for Strassen. Because of the attainability of the lower bounds in the sequential case, we hypothesized that the gap could be closed by finding a new algorithm rather than by tightening the lower bounds.

We made three observations from the lower bound results of [4] that led to the new algorithm. First, the lower bounds for Strassen are lower than those for classical matrix multiplication. This implies that in order to obtain an optimal parallel Strassen algorithm, the communication pattern cannot be that of a classical algorithm but must reflect the properties of Strassen's algorithm. Second, the factor $M^{\omega_0/2-1}$ that appears in the denominator of the communication cost lower bound implies that an optimal algorithm must use as much local memory as possible. That is, there is a tradeoff between memory usage and communication (the same is true in the classical case). Third, the proof of the lower bounds shows that in order to minimize communication costs relative to computation, it is necessary to perform each sub-matrix multiplication of size $\Theta(\sqrt{M}) \times \Theta(\sqrt{M})$ on a single processor.

With these observations and assisted by techniques from previous approaches to parallelizing Strassen, we developed a new parallel algorithm which achieves perfect load balance, minimizes communication costs, and in particular performs asymptotically less computation and communication than is possible using classical matrix multiplication.

1.4 Our contributions and paper organization

Our main contribution is a new algorithm we call Communication-Avoiding Parallel Strassen, or *CAPS*.

THEOREM 1.1. *CAPS asymptotically minimizes computational and bandwidth costs over all parallel Strassen algorithms. It also minimizes latency cost up to a logarithmic factor in the number of processors.*

CAPS performs asymptotically better than any previous previous classical or Strassen-based parallel algorithm. It also runs faster in practice. The algorithm and its computational and communication cost analyses are presented in Section 3. There we show it matches the communication lower bounds.

We provide a review and analysis of previous algorithms in Section 4. We also consider two natural combinations of previously known algorithms (Sections 4.4 and 4.5). One of these new algorithms that we call "2.5D-Strassen" communicates less than all previous algorithms, but still more than CAPS.

We discuss our implementations of the new algorithms and compare their performance with previous ones in Section 5 to show that our new CAPS algorithm outperforms previous algorithms not just asymptotically, but also in practice. Benchmarking our implementation on a Cray XT4, we obtain speedups over classical and Strassen-based algorithms ranging from 24% to 184% for a fixed matrix dimension $n = 94080$, where the number of (quad-core) processors ranges from 49 to 7203.

In Section 6 we show that our parallelization method applies to other fast matrix multiplication algorithms. It also applies to classical recursive matrix multiplication, thus obtaining a new optimal classical algorithm that matches the 2.5D algorithm of Solomonik and Demmel [31]. In Section 6, we also discuss its numerical stability, generalizations, and future work.

2. PRELIMINARIES

2.1 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

[2]See [4] for a discussion of the sequential memory model.

be communicated in time $\alpha + \beta w$. The bandwidth cost of the algorithm is given by the word count and denoted by $BW(\cdot)$, and the latency cost is given by the message count and denoted by $L(\cdot)$. Similarly the computational cost is given by the number of floating point operations and denoted by $F(\cdot)$. We call the time per floating point operation γ.[3]

We count the number of words, messages and floating point operations along the *critical path* as defined in [39]. 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. This metric is closely related to the total running time of the algorithm, which we model as

$$\alpha L(\cdot) + \beta BW(\cdot) + \gamma F(\cdot).$$

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.

2.2 Strassen's algorithm

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 [33]. 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 [38]. Further reduction in the number of additions is not possible [28, 9]. We review the Strassen-Winograd algorithm in Section 2.3.

In this paper, we use the term *parallel Strassen algorithm* for a parallel algorithm that performs exactly the same arithmetic operations as any variant[4] of Strassen's (sequential) algorithm. 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.

2.3 Strassen-Winograd Algorithm

The Strassen-Winograd algorithm is usually preferred to Strassen's algorithm in practice since it requires fewer additions. We use it for our implementation of CAPS. Divide the input matrices A, B and output matrix C into 4 submatrices:

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

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

$$\begin{array}{llll} T_0 = A_{11} & S_0 = B_{11} & Q_0 = T_0 \cdot S_0 & U_1 = Q_0 + Q_3 \\ T_1 = A_{12} & S_1 = B_{21} & Q_1 = T_1 \cdot S_1 & U_2 = U_1 + Q_4 \\ T_2 = A_{21} + A_{22} & S_2 = B_{12} + B_{11} & Q_2 = T_2 \cdot S_2 & U_3 = U_1 + Q_2 \\ T_3 = T_2 - A_{12} & S_3 = B_{22} - S_2 & Q_3 = T_3 \cdot S_3 & C_{11} = Q_0 + Q_1 \\ T_4 = A_{11} - A_{21} & S_4 = B_{22} - B_{12} & Q_4 = T_4 \cdot S_4 & C_{12} = U_3 + Q_5 \\ T_5 = A_{12} + T_3 & S_5 = B_{22} & Q_5 = T_5 \cdot S_5 & C_{21} = U_2 - Q_6 \\ T_6 = A_{22} & S_6 = S_3 - B_{21} & Q_6 = T_6 \cdot S_6 & C_{22} = U_2 + Q_2 \end{array}$$

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

$$\mathrm{F}(n) = c_s n^{\omega_0} - 5n^2. \tag{1}$$

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 $3.73n^{\omega_0} - 5n^2$. As with Strassen's algorithm, Equation (1) applies only if the ratio n/n_0 is a power of 2.

2.4 Previous work on parallel Strassen

In this section we briefly describe previous efforts to parallelize Strassen. More details, including communication analyses, are in Section 4. A summary appears in Table 1.

Luo and Drake [26] explored Strassen-based parallel algorithms that use the communication patterns known for classical matrix multiplication. They considered using a classical 2D parallel algorithm and using Strassen locally, which corresponds to what we call the "2D-Strassen" approach (see Section 4.2). They also consider using Strassen at the highest level and performing a classical parallel algorithm for each subproblem generated, which corresponds to what we call the "Strassen-2D" approach. The size of the subproblems depends on the number of Strassen steps taken (see Section 4.3). Luo and Drake also analyzed the communication costs for these two approaches.

Soon after, Grayson, Shah, and van de Geijn [21] improved on the Strassen-2D approach of [26] by using a better classical parallel matrix multiplication algorithm and running on a more communication-efficient machine. They obtained better performance results compared to a purely classical algorithm for up to three levels of Strassen's recursion.

Kumar, Huang, Johnson, and Sadayappan [25] implemented Strassen's algorithm on a shared-memory machine. They identified the tradeoff between available parallelism and total memory footprint by differentiating between "partial" and "complete" evaluation of the algorithm, which corresponds to what we call depth-first and breadth-first traversal of the recursion tree (see Section 3.1). They show that by using ℓ DFS steps before using BFS steps, the memory footprint is reduced by a factor of $(7/4)^{\ell}$ compared to using all

[3] This parallel model resembles the BSP model of Valiant [35], however we do not require the communication to be bulk synchronous. Note that both the CAPS algorithm and the lower bounds of [4, 3] hold under the more restrictive BSP model.

[4] A *variant* of Strassen's algorithm is any algorithm based on 2×2 matrix multiplication using 7 scalar multiplications.

BFS steps. They did not consider communication costs in their work.

Other parallel approaches [18, 23, 32] have used more complex parallel schemes and communication patterns. However, they restrict attention to only one or two steps of Strassen and obtain modest performance improvements over classical algorithms.

2.5 Strassen lower bounds

For parallel Strassen algorithms, the bandwidth cost lower bound has been proved using expansion arguments on the computation graph, and the latency cost lower bound is an immediate corollary.

THEOREM 2.1. *(Memory-dependent lower bound) [4] Consider a parallel Strassen algorithm running on P processors each with local memory size M. Let $BW(n, P, M)$ be the bandwidth cost and $L(n, P, M)$ be the latency cost of the algorithm. Assume that no intermediate values are computed twice. Then*

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

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

A memory-independent lower bound has recently been proved using the same expansion approach:

THEOREM 2.2. *(Memory-independent lower bound) [3] Consider a parallel Strassen algorithm running on P processors. Let $BW(n, P)$ be the bandwidth cost and $L(n, P)$ 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*

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

and the latency cost is $L(n, P) = \Omega(1)$.

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.

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 3.1. *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. Thus Theorem 1.1 follows. We prove Theorem 3.1 in Section 3.5.

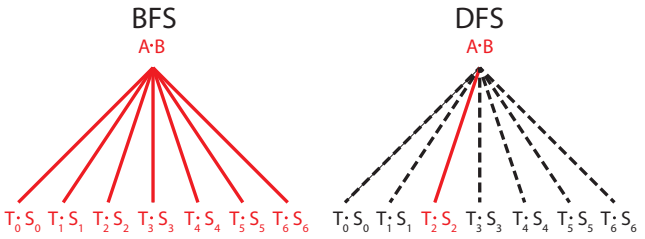

Figure 1: Representation of BFS and DFS steps. In a BFS step, all seven subproblems are computed at once, each on 1/7 of the processors. In a DFS step, the seven subproblems are computed in sequence, each using all the processors. The notation follows that of Section 2.3.

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 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. 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.

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. The second scheme, which we call the Limited Memory *(LM)* scheme 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.

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

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 2.3.

```
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
```

sub-matrices can be performed locally. This technique follows [26] 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.

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.

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 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 $(n/2, P, s-1)$.

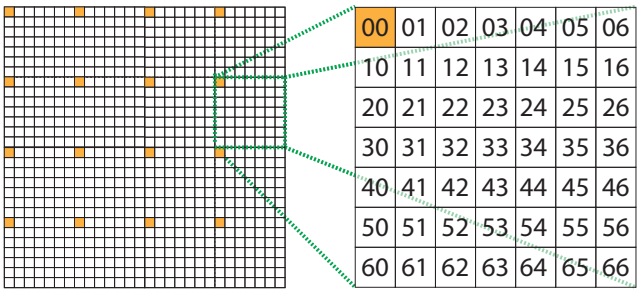

Figure 2: An example matrix layout for CAPS. Each of the 16 submatrices as shown on the left has exactly the same layout. The colored blocks are the ones owned by processor 00. On the right is a zoomed-in view of one submatrix, showing which processor, numbered base 7, owns each block. This is block-cyclic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

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 receieve a message from each of the other 6). To see this, consider the numbering of the processors base-7. On the m^{th} BFS step, the communication is between the seven processors whose numbers agree on all digits except the m^{th} (counting from the right). After the m^{th} 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 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.

3.3.1 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.[5] Thus, the total local memory requirement for taking k BFS steps is given by

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

3.3.2 Computational costs

The computation required at a given BFS step is that of the local additions and subtractions associated with computing

[5] 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.

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: $C = A \cdot B$

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

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

procedure BFS(A, B, C, k, rank)
  ▷ Do $C = A \cdot B$ by k BFS steps, then local Strassen
  if $k == 0$ **then call** localStrassen(A, B, C); **return**
  for $i = 1 \ldots 7$ **do**
    locally compute S_i and T_i from A and B
      ▷ following Strassen's algorithm
  for $i = 1 \ldots 7$ **do**
    target = rank
    target[k] = i
    send S_i to target
    receive into L
    ▷ One part of L comes from each of 7 processors
    send T_i to target
    receive into R
    ▷ One part of R comes from each of 7 processors
  call BFS(L, R, V, $k - 1$, rank)
  for $i = 1 \ldots 7$ **do**
    target = rank
    target[k] = i
    send i^{th} part of V to target
    receive from target into Q_i
  locally compute C from Q_i
    ▷ following Strassen's algorithm

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 2.3 for more details. The solution to this recurrence is

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

3.3.3 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.

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:

$$BW_{\text{UM}}(n, P) = 36\frac{n^2}{4P} + BW_{\text{UM}}\left(\frac{n}{2}, \frac{P}{7}\right)$$
$$L_{\text{UM}}(n, P) = 36 + L_{\text{UM}}\left(\frac{n}{2}, \frac{P}{7}\right)$$

with base case $L_{\text{UM}}(n, 1) = BW_{\text{UM}}(n, 1) = 0$. Thus

$$BW_{\text{UM}}(n, P) = \frac{12n^2}{P^{2/\omega_0}} - \frac{12n^2}{P} = \Theta\left(\frac{n^2}{P^{2/\omega_0}}\right)$$
$$L_{\text{UM}}(n, P) = 36 \log_7 P = \Theta(\log P). \quad (2)$$

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 i^{th} 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:

$$\frac{3n^2}{P} \leq \frac{1}{3}M. \quad (3)$$

In the LM scheme, we set

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

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

3.4.1 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

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

where the last line follows from (4) and (3). Thus, the limited memory scheme does not exceed the available memory.

3.4.2 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

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

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

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

and

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

3.4.3 Communication costs

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

$$\begin{aligned}BW_{\mathrm{LM}}(n,P) &= 7\cdot BW_{\mathrm{LM}}\left(\frac{n}{2},P\right)\\ L_{\mathrm{LM}}(n,P) &= 7\cdot L_{\mathrm{LM}}\left(\frac{n}{2},P\right)\end{aligned}$$

with base cases

$$\begin{aligned}BW_{\mathrm{LM}}\left(\frac{n}{2^\ell},P\right) &= BW_{\mathrm{UM}}\left(\frac{n}{2^\ell},P\right)\\ L_{\mathrm{LM}}\left(\frac{n}{2^\ell},P\right) &= L_{\mathrm{UM}}\left(\frac{n}{2^\ell},P\right).\end{aligned}$$

Thus the total communication costs are given by

$$\begin{aligned}BW_{\mathrm{LM}}(n,P) &= 7^\ell\cdot BW_{\mathrm{UM}}\left(\frac{n}{2^\ell},P\right)\\ &\le \frac{12\cdot 4^{\omega_0-2}n^{\omega_0}}{PM^{\omega_0/2-1}}\\ &= \Theta\left(\frac{n^{\omega_0}}{PM^{\omega_0/2-1}}\right).\\ L_{\mathrm{LM}}(n,P) &= 7^\ell\cdot L_{\mathrm{UM}}\left(\frac{n}{2^\ell},P\right)\\ &\le \frac{(4n)^{\omega_0}}{PM^{\omega_0/2}}36\log_7 P\\ &= \Theta\left(\frac{n^{\omega_0}}{PM^{\omega_0/2}}\log P\right). \qquad (5)\end{aligned}$$

3.5 Communication optimality

Proof. (*of Theorem 3.1*). In the case that $M \ge \mathrm{Mem}_{\mathrm{UM}}(n,P) = \Omega\left(\frac{n^2}{P^{2/\omega_0}}\right)$ the UM scheme is possible. Then the communication costs are given by (2) which matches the lower bound of Theorem 2.2. 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 3.2). For smaller values of M, the LM scheme must be used. Then the communication costs are given by (5) and match the lower bound of Theorem 2.1, 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 $\frac{1}{P}$, 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 [3] for details.

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 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.

4.1 Classical Algorithms

Any classical algorithm must communicate asymptotically more than an optimal parallel Strassen algorithm. To compare the lower bounds, 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. This analysis is detailed in Appendix A. Briefly, the factor by which the classical bandwidth cost exceeds the Strassen bandwidth cost is P^a where a ranges from $\frac{2}{\omega_0}-\frac{2}{3}\approx 0.046$ to $\frac{3-\omega_0}{2}\approx 0.10$ depending on the relative problem size. The same sort of analysis is used throughout Section 4 to compare each algorithm with the parallel Strassen lower bounds.

Various parallel classical matrix multiplication algorithms minimize communication relative to the classical lower bounds for certain amounts of local memory M. For example, Cannon's algorithm [10] minimizes communication for $M = O(n^2/P)$. Several more practical algorithms exist (such as SUMMA [36]) which use the same amount of local memory and have the same asymptotic communication costs. We call this class of algorithms "2D" because the communication patterns follow a two-dimensional processor grid.

		Flops	Bandwidth	Latency
Classical	Lower Bound [3, 24]	$\frac{n^3}{P}$	$\max\left\{\frac{n^3}{PM^{1/2}}, \frac{n^2}{P^{2/3}}\right\}$	$\max\left\{\frac{n^3}{PM^{3/2}}, 1\right\}$
	2D [10, 36]	$\frac{n^3}{P}$	$\frac{n^2}{P^{1/2}}$	$P^{1/2}$
	3D [1, 6]	$\frac{n^3}{P}$	$\frac{n^2}{P^{2/3}}$	$\log P$
	2.5D [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 [3, 4]	$\frac{n^{\omega_0}}{P}$	$\max\left\{\frac{n^{\omega_0}}{PM^{\omega_0/2-1}}, \frac{n^2}{P^{2/\omega_0}}\right\}$	$\max\left\{\frac{n^{\omega_0}}{PM^{\omega_0/2}}, 1\right\}$
	2D-Strassen [26]	$\frac{n^{\omega_0}}{P^{(\omega_0-1)/2}}$	$\frac{n^2}{P^{1/2}}$	$P^{1/2}$
	Strassen-2D [21, 26]	$\left(\frac{7}{8}\right)^\ell \frac{n^3}{P}$	$\left(\frac{7}{4}\right)^\ell \frac{n^2}{P^{1/2}}$	$7^\ell P^{1/2}$
	2.5D-Strassen	$\max\left\{\frac{n^3}{PM^{3/2-\omega_0/2}}, \frac{n^{\omega_0}}{P^{\omega_0/3}}\right\}$	$\max\left\{\frac{n^3}{PM^{1/2}}, \frac{n^2}{P^{2/3}}\right\}$	$\frac{n^3}{PM^{3/2}} + \log P$
	Strassen-2.5D	$\left(\frac{7}{8}\right)^\ell \frac{n^3}{P}$	$\max\left\{\left(\frac{7}{8}\right)^\ell \frac{n^3}{PM^{1/2}}, \left(\frac{7}{4}\right)^\ell \frac{n^2}{P^{2/3}}\right\}$	$\left(\frac{7}{8}\right)^\ell \frac{n^3}{PM^{3/2}} + 7^\ell \log P$
	CAPS	$\frac{n^{\omega_0}}{P}$	$\max\left\{\frac{n^{\omega_0}}{PM^{\omega_0/2-1}}, \frac{n^2}{P^{2/\omega_0}}\right\}$	$\max\left\{\frac{n^{\omega_0}}{PM^{\omega_0/2}} \log P, \log P\right\}$

Table 1: Asymptotic computational and communication costs of 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 2.5, and thus is optimal. All of the other Strassen-based algorithms have asymptotically higher communication costs; see Section 4 for details.

Another class of algorithms, known as "3D" [6, 1] because the communication pattern maps to a three-dimensional processor grid, uses more local memory and reduces communication relative to 2D algorithms. This class of algorithms minimizes communication relative to the classical lower bounds for $M = \Omega(n^2/P^{2/3})$. As shown in [3], it is not possible to use more memory than $M = \Theta(n^2/P^{2/3})$ to reduce communication.

Recently, a more general algorithm has been developed which minimizes communication in all cases. Because it reduces to a 2D and 3D for the extreme values of M but interpolates for the values between, it is known as the "2.5D" algorithm [31].

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 [26]. 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 provides 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 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 $1/2 = .5$.

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 3.2, it is possible to do the additions and subtractions for Strassen's algorithm without any communication [26]. 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.

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.

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 comptutational 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.

5. PERFORMANCE RESULTS

We have implemented CAPS using MPI on a Cray XT4 ("Franklin" at the National Energy Research Scientific Computing Center), and compared it to various previous classical and Strassen-based algorithms. The benchmarking data is shown in Figure 3.

5.1 Experimental setup

Each node of the Cray XT4 has 8GB of memory and a quad-core AMD "Budapest" processor running at 2.3GHz. We treat the entire node as a single processor, and when we use the classical algorithm we call the optimized threaded BLAS in Cray's LibSci to provide parallelism between the four cores in a node. The peak flop rate is 9.2 GFLOPS per core, or 36.8 GFLOPS per node. The machine consists of 9,572 nodes. All the data in Figure 3 is for multiplying two square matrices with $n = 94080$.

5.2 Performance

Note that the vertical scale of Figure 3 is "effective GFLOPS", which is a useful measure for comparing classical and fast matrix multiplication algorithms. It is calculated as

$$\text{Effective GFLOPS} = \frac{2n^3}{(\text{Execution time in seconds})10^9}. \quad (6)$$

For classical algorithms, which perform $2n^3$ floating point operations, this gives the actual GFLOPS. For fast matrix multiplication algorithms it gives the performance relative to classical algorithms, but does not accurately represent the number of floating point operations performed.

Our algorithm outperforms all previous algorithms, and attains performance as high as 49.1 effective GFLOPS/node, which is 33% above the theoretical maximum for all classical algorithms. Compared with the best classical implementation, our speedup ranges from 51% for small values of P up to 94% when using most of the machine. Compared with the best previous parallel Strassen algorithms, our speedup ranges from 24% up to 184%. Unlike previous Strassen algorithms, we are able to attain substantial speedups over the entire range of processors.

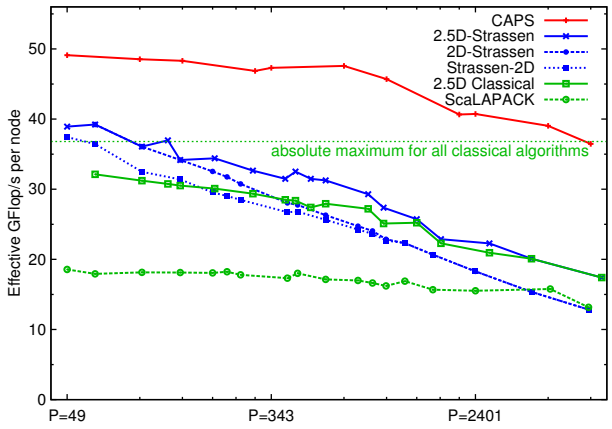

Figure 3: Strong scaling performance of various matrix multiplication algorithms on Cray XT4 for fixed problem size $n = 94080$. The top line is CAPS as described in Section 3, and substantially outperforms all the other classical and Strassen-based algorithms. The horizontal axis is the number of nodes in log-scale. The vertical axis is effective GFLOPS, which are a performance measure rather than a flop rate, as discussed in Section 5.2. See Section 5.4 for a description of each implementation.

5.3 Strong scaling

Figure 3 is a strong scaling plot: the problem size is fixed and each algorithm is run with P ranging from the minimum that provides enough memory up to the largest allowed value of P smaller than the size of the machine. Perfect strong scaling corresponds to a horizontal line in the plot. As the communication analysis predicts, CAPS exhibits better

strong scaling than any of the other algorithms (with the exception of ScaLAPACK, which obtains very good strong scaling by having poor performance for small values of P).

5.4 Details of the implementations

5.4.1 CAPS

This implementation is the CAPS algorithm, with a few modifications from the presentation in Section 3. First, when computing locally it switches to classical matrix multiplication below some size n_0. Second, it is generalized to run on $P = c7^k$ processors for $c \in \{1, 2, 3, 6\}$ rather than just 7^k processors. As a result, the base-case classical matrix multiplication is done on c processors rather than 1. Finally, the implementation uses the Winograd variant of Strassen; see Section 2.3 for more details. Every point in the plot is tuned to use the best interleaving pattern of BFS and DFS steps, and the best total number of Strassen steps. For points in the figure, the optimal total number of Strassen steps is always 5 or 6.

5.4.2 ScaLAPACK

We use ScaLAPACK [8] as optimized by Cray in LibSci. This is an implementation of the SUMMA algorithm, and can run on an arbitrary number of processors. It should give the best performance if P is a perfect square so the processors can be placed on a square 2D grid. All the runs shown in Figure 3 are with P a perfect square.

5.4.3 2.5D classical

This is the code of [31]. It places the P processors in a grid of size $\sqrt{P/c} \times \sqrt{P/c} \times c$, and requires that $\sqrt{P/c}$ and c are integers with $1 \leq c \leq P^{1/3}$, and c divides $\sqrt{P/c}$. Additionally, it gets the best performance if c is as large as possible, given the constraint that c copies of the input and output matrices fit in memory. In the case that $c = 1$ this code is an optimized implementation of SUMMA. The values of P and c for the runs in Figure 3 are chosen to get the best performance.

5.4.4 Strassen-2D

Following the algorithm of [21, 26], this implementation uses the DFS code from the implementation of CAPS at the top level, and then uses the optimized SUMMA code from the 2.5D implementation with $c = 1$. Since the classical code requires that P is a perfect square, this requirement applies here as well. The number of Strassen steps taken is tuned to give the best performance for each P value, and the optimal number varies from 0 to 2.

5.4.5 2D-Strassen

Following the algorithm of [26], the 2D-Strassen implementation is analagous to the Strassen-2D implementation, but with the classical algorithm run before taking local Strassen steps. Similarly the same code is used for local Strassen steps here and in our implementation of CAPS. This code also requires that P is a perfect square. The number of Strassen steps is tuned for each P value, and the optimal number varies from 0 to 3.

5.4.6 2.5D-Strassen

This implementation uses the 2.5D implementation to reduce the problem to one processor, then takes several Strassen steps. The processor requirements are the same as for the 2.5D implementation. The number of Strassen steps is tuned for each number of processors, and the optimal number varies from 0 to 3. We also tested the Strassen-2.5D algorithm, but its performance was always lower than 2.5D-Strassen in our experiments.

6. DISCUSSION AND FUTURE WORK

6.1 Stability of fast matrix multiplication

CAPS has the same stability properties as sequential versions of Strassen. For a complete discussion of the stability of fast matrix multiplication algorithms, see [22, 17]. 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 [22]. Accuracy can be improved with the use of diagonal scaling matrices: $D_1 C D_3 = D_1 A D_2 \cdot D_2^{-1} B D_3$. 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 [16]. Thus using Strassen for the matrix multiplications in a larger computation will often not harm the stability at all.

6.2 Testing on various architectures

We have implemented and benchmarked CAPS on only one architecture, a Cray XT4. It remains to check that it outperforms other matrix multiplication algorithms on a variety of architectures. On some architectures it may be more important to consider the topology of the network and redesign the algorithm to minimize contention, which we have not done.

6.3 Improvements to the algorithm

CAPS as presented in Section 3 requires P to be a power of seven. To be practically useful, it is important to generalize the number of processors on which CAPS can run. Of course, CAPS can then be run on any number of processors by simply ignoring no more than $\frac{6}{7}$ of them and incurring a constant factor overhead. Thus we can run on arbitrary P and attain the communication and computation lower bounds up to a constant factor. However the computation time is still dominant in most cases, and it is preferable to attain the computation lower bound exactly. It is an open question whether any algorithm can run on arbitrary P, attain the computation lower bound exactly, and attain the communication lower bound up to a constant factor.

Moreover, although the communication costs of this algorithm match the lower bound up to a constant factor in bandwidth, and up to a $\log P$ factor in latency, it is an open question to determine the optimal constant in the lower bound and perhaps provide a new algorithm that matches it exactly. Note that in the analysis of CAPS in Section 3, the constants could be slightly improved.

6.4 Parallelizing other algorithms

6.4.1 Another optimal classical algorithm

We can apply our parallelization approach to recursive classical matrix multiplication to obtain a communication-optimal algorithm. This algorithm has the same asymptotic communication costs as the 2.5D algorithm [31]. We observed comparable performance to the 2.5D algorithm on our experimental platform. As with CAPS, this algorithm has not been optimized for contention, whereas the 2.5D algorithm is very well optimized for contention on torus networks.

6.4.2 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 $\omega_0 = \log_{n_0} q$.

Using the techniques from Theorem 3.3 of [17], one can convert any fast 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.

7. REFERENCES

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

[2] Aggarwal, A., Chandra, A. K., and Snir, M. Communication complexity of PRAMs. *Theor. Comput. Sci. 71* (March 1990), 3–28.

[3] Ballard, G., Demmel, J., Holtz, O., Lipshitz, B., and Schwartz, O. Strong scaling of matrix multiplication algorithms and memory-independent communication lower bounds, 2012. Manuscript, submitted to SPAA 2012.

[4] Ballard, G., Demmel, J., Holtz, O., and Schwartz, O. Graph expansion and communication costs of fast matrix multiplication. In *SPAA '11: Proceedings of the 23rd annual symposium on parallelism in algorithms and architectures* (New York, NY, USA, 2011), ACM, pp. 1–12.

[5] Ballard, G., Demmel, J., Holtz, O., and Schwartz, O. Minimizing communication in numerical linear algebra. *SIAM J. Matrix Analysis Applications 32*, 3 (2011), 866–901.

[6] Berntsen, J. Communication efficient matrix multiplication on hypercubes. *Parallel Computing 12*, 3 (1989), 335 – 342.

[7] Bini, D. Relations between exact and approximate bilinear algorithms. applications. *Calcolo 17* (1980), 87–97. 10.1007/BF02575865.

[8] Blackford, L. S., Choi, J., Cleary, A., D'Azevedo, E., Demmel, J., Dhillon, I., Dongarra, J., Hammarling, S., Henry, G., Petitet, A., Stanley, K., Walker, D., and Whaley, R. C. *ScaLAPACK Users' Guide*. SIAM, Philadelphia, PA, USA, May 1997. Also available from http://www.netlib.org/scalapack/.

[9] Bshouty, N. H. On the additive complexity of 2×2 matrix multiplication. *Information Processing Letters 56*, 6 (1995), 329 – 335.

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

[11] Choi, J. A new parallel matrix multiplication algorithm on distributed-memory concurrent computers. *Concurrency: Practice and Experience 10*, 8 (1998), 655–670.

[12] Choi, J., Walker, D. W., and Dongarra, J. J. PUMMA: Parallel universal matrix multiplication algorithms on distributed memory concurrent computers. *Concurrency: Practice and Experience 6*, 7 (1994), 543–570.

[13] Cohn, H., Kleinberg, R. D., Szegedy, B., and Umans, C. Group-theoretic algorithms for matrix multiplication. In *FOCS* (2005), pp. 379–388.

[14] Coppersmith, D., and Winograd, S. On the asymptotic complexity of matrix multiplication. *SIAM Journal on Computing 11*, 3 (1982), 472–492.

[15] Coppersmith, D., and Winograd, S. Matrix multiplication via arithmetic progressions. In *Proceedings of the Nineteenth Annual ACM Symposium on Theory of Computing* (New York, NY, USA, 1987), STOC '87, ACM, pp. 1–6.

[16] Demmel, J., Dumitriu, I., and Holtz, O. Fast linear algebra is stable. *Numerische Mathematik 108*, 1 (2007), 59–91.

[17] Demmel, J., Dumitriu, I., Holtz, O., and Kleinberg, R. Fast matrix multiplication is stable. *Numerische Mathematik 106*, 2 (2007), 199–224.

[18] Desprez, F., and Suter, F. Impact of mixed-parallelism on parallel implementations of the Strassen and Winograd matrix multiplication algorithms: Research articles. *Concurr. Comput. : Pract. Exper. 16* (July 2004), 771–797.

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

[20] Graham, S. L., Snir, M., and Patterson, C. A., Eds. *Getting up to Speed: The Future of Supercomputing*. Report of National Research Council of the National Academies Sciences. The National

Academies Press, Washington, D.C., 2004. 289 pages, http://www.nap.edu.

[21] Grayson, B., Shah, A., and van de Geijn, R. A high performance parallel Strassen implementation. In *Parallel Processing Letters* (1995), vol. 6, pp. 3–12.

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

[23] Hunold, S., Rauber, T., and Rünger, G. Combining building blocks for parallel multi-level matrix multiplication. *Parallel Computing 34* (July 2008), 411–426.

[24] Irony, D., Toledo, S., and Tiskin, A. Communication lower bounds for distributed-memory matrix multiplication. *J. Parallel Distrib. Comput. 64*, 9 (2004), 1017–1026.

[25] Kumar, B., Huang, C.-H., Johnson, R., and Sadayappan, P. A tensor product formulation of Strassen's matrix multiplication algorithm with memory reduction. In *Proceedings of Seventh International Parallel Processing Symposium* (apr 1993), pp. 582 –588.

[26] Luo, Q., and Drake, J. A scalable parallel Strassen's matrix multiplication algorithm for distributed-memory computers. In *Proceedings of the 1995 ACM symposium on Applied computing* (New York, NY, USA, 1995), SAC '95, ACM, pp. 221–226.

[27] Pan, V. Y. New fast algorithms for matrix operations. *SIAM Journal on Computing 9*, 2 (1980), 321–342.

[28] Probert, R. L. On the additive complexity of matrix multiplication. *SIAM Journal on Computing 5*, 2 (1976), 187–203.

[29] Romani, F. Some properties of disjoint sums of tensors related to matrix multiplication. *SIAM Journal on Computing 11*, 2 (1982), 263–267.

[30] Schönhage, A. Partial and total matrix multiplication. *SIAM Journal on Computing 10*, 3 (1981), 434–455.

[31] Solomonik, E., and Demmel, J. Communication-optimal parallel 2.5D matrix multiplication and LU factorization algorithms. In *Euro-Par '11: Proceedings of the 17th International European Conference on Parallel and Distributed Computing* (2011), Springer.

[32] Song, F., Dongarra, J., and Moore, S. Experiments with Strassen's algorithm: From sequential to parallel. In *Proceedings of Parallel and Distributed Computing and Systems (PDCS)* (Nov. 2006), ACTA.

[33] Strassen, V. Gaussian elimination is not optimal. *Numer. Math. 13* (1969), 354–356.

[34] Strassen, V. Relative bilinear complexity and matrix multiplication. *Journal fűr die reine und angewandte Mathematik (Crelles Journal) 1987*, 375–376 (1987), 406–443.

[35] Valiant, L. G. A bridging model for parallel computation. *Commun. ACM 33*, 8 (Aug. 1990), 103–111.

[36] van de Geijn, R. A., and Watts, J. SUMMA: scalable universal matrix multiplication algorithm. *Concurrency - Practice and Experience 9*, 4 (1997), 255–274.

[37] Williams, V. V. Breaking the Coppersmith-Winograd barrier, 2012. Manuscript.

[38] Winograd, S. On the multiplication of 2 × 2 matrices. *Linear Algebra Appl. 4*, 4 (October 1971), 381–388.

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

APPENDIX

A. COMMUNICATION-COST RATIOS

In this section we derive the ratio R of classical to parallel Strassen bandwidth cost lower bounds that appear in the beginning of Section 4. Note that both classical and Strassen lower bounds are attained by optimal algorithms. Similar derivations apply to the other ratios quoted in that section. Because the bandwidth cost lower bounds are different in the memory-dependent and the memory-independent cases, and the threshold between these is different for the classical and Strassen bounds, it is necessary to consider three cases.

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. $M = O(P^{2/3})$. 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 between $\Theta(P^{2/\omega_0-2/3})$ and $\Theta(P^{(3-\omega_0)/2})$.