Communication-Avoiding Parallel Strassen: Implementation and Performance

Benjamin Lipshitz
Grey Ballard
Oded Schwartz
James Demmel

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2012-90
http://www.eecs.berkeley.edu/Pubs/TechRpts/2012/EECS-2012-90.html

May 11, 2012

Copyright © 2012, 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.

Acknowledgement

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.

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.

Communication-Avoiding Parallel Strassen: Implementation and Performance

Benjamin Lipshitz, Grey Ballard, James Demmel, and Oded Schwartz
Computer Science Division and Mathematics Department
UC Berkeley
Berkeley, California 94704
Email: {lipshitz,ballard,demmel,odedsc}@cs.berkeley.edu

***Abstract*—Matrix multiplication is a fundamental kernel of many high performance and scientific computing applications. Most parallel implementations use classical $O(n^3)$ matrix multiplication, even though there exist algorithms with lower arithmetic complexity. We recently obtained a new Communication-Avoiding Parallel Strassen algorithm (CAPS), based on Strassen's fast matrix multiplication, that minimizes communication (SPAA '12). It communicates asymptotically less than all classical and all previous Strassen-based algorithms, and it attains theoretical lower bounds.**

In this paper we show that CAPS is also faster in practice. We benchmark and compare its performance to previous algorithms on Hopper (Cray XE6), Intrepid (IBM BG/P), and Franklin (Cray XT4). We demonstrate significant speedups over previous algorithms both for large matrices and for small matrices on large numbers of processors. We model and analyze the performance of CAPS and predict its performance on future exascale platforms.

I. Introduction

Strassen's algorithm is well-known for performing matrix multiplication with asymptotically fewer floating point operations (*flops*) than the classical $O(n^3)$ algorithm. It is less well-known that Strassen's algorithm requires less data movement (*communication*) as well. A theoretical lower bound for the communication of Strassen's algorithm is given in [5], and the bound is asymptotically smaller than what is required for the classical algorithm [13]. This bound applies to both the sequential case, where communication is the movement of data between main memory and cache, and the parallel case, where communication is movement of data between processors.

In the sequential case, the lower bound is attained by the cache-oblivious recursive algorithm [5], [10]. However, in the parallel case, finding an algorithm that minimizes communication is not as straightforward. Several previous attempts to use Strassen for parallel multiplication did not reduce communication relative to classical algorithms and achieved only modest speedups. In [3], we present a new Communication-Avoiding Parallel Strassen algorithm (*CAPS*) which minimizes communication: it communicates asymptotically less than all classical and all previous Strassen-based algorithms, and it attains the lower bound for parallel Strassen algorithms. There we also show promising preliminary performance results.

In this paper, we demonstrate that CAPS achieves significantly better performance than tuned implementations of classical algorithms. We analyze the performance of our implementation, compare to theoretical predictions, and identify room for further performance improvement. Our main conclusion is that Strassen's algorithm should be used in many practical cases, and we believe it should be adopted into parallel libraries.

We show that by using Strassen, it is possible to achieve effective performance which exceeds the machine's peak for any classical implementation. We observe speedups over the best classical implementations of up to $2\times$. In Section III we benchmark and compare performance on three machines: Hopper (Cray XE6), Intrepid (IBM BG/P), and Franklin (Cray XT4). On all three machines we demonstrate significant speedups over previous parallel classical and Strassen-based algorithms for large matrices across the entire range of the number of nodes. We also show speedups for communication-bound multiplication of small matrices on large numbers of processors. These speedups are especially significant considering how much effort is devoted to tuning classical matrix multiplication to achieve speedups on the order of 10-20%.

In Section IV we model the performance of CAPS and compare it to our benchmarks. This comparison validates the qualitative theoretical conclusions drawn in [3] and also identifies opportunities for further optimizations. Not only does Strassen perform well on today's machines, we expect its advantage to grow in the future. Asymptotic analysis shows that bigger problem sizes, more parallelism, and limited local memory sizes all increase the communication savings of Strassen's compared with classical matrix multiplication. Bigger problem sizes also increases the computational savings. We discuss these predictions in more detail in Section IV-C.

However, there are drawbacks to Strassen's algorithm and to our CAPS implementation in particular. As we discuss in Section VI-B, Strassen does not map as readily to current hardware as does classical matrix multiplication. There are also differences in the numerical stability properties of Strassen and the classical algorithm. While the stability consequences of using Strassen are often exaggerated, these issues have been well understood [12], and we discuss them in Section VI-C.

II. Algorithm and Review

Let n, P, and M be the matrix dimension, number of processors, and memory per processor, respectively.

A. Classical Parallel Algorithms

The most common algorithms for parallel matrix multiplication are Cannon's algorithm [8] and SUMMA [18]. Both of these algorithms are considered "2D" algorithms as the processor grid is organized in two dimensions and communication occurs among processors in the same row and in the same column of the processor grid. They load balance the $2n^3$ flops perfectly and communicate $\Theta\left(\frac{n^2}{\sqrt{P}}\right)$ words along the critical path of the algorithm, requiring $M = O\left(\frac{n^2}{P}\right)$ words of local memory.

The communication of classical parallel matrix multiplication can be reduced at the cost of extra local memory. The so-called "3D" algorithms [2], [7] incorporate a three-dimensional $\sqrt[3]{P} \times \sqrt[3]{P} \times \sqrt[3]{P}$ processor grid and reduce the bandwidth cost compared to 2D algorithms by a factor of $P^{1/6}$ at the expense of requiring a factor of $P^{1/3}$ more memory. A 3D algorithm can be executed only if $M = \Omega\left(\frac{n^2}{P^{2/3}}\right)$ words of local memory are available.

Recently, the "2.5D" matrix multiplication algorithm [17] was developed to interpolate between 2D and 3D algorithms. For any factor $1 \leq c \leq \sqrt[3]{P}$ such that c copies of the input and output matrices will fit into memory, the 2.5D algorithm reduces the bandwidth cost compared to 2D algorithms by a factor of $\sqrt{c}$. At the extremal values of c, the 2.5D algorithm reproduces the original 2D and 3D algorithms.

B. Previous Attempts to Parallelize Strassen

There are a couple of natural ways to parallelize Strassen's algorithm using the infrastructure of a classical parallel algorithm [11], [14]. We detail them below and note that neither of these parallelization schemes minimizes communication.

First, Strassen can be used "at the bottom" as the local matrix multiplication subroutine instead of a classical sequential algorithm. In this scheme, the interprocessor communication follows a classical algorithm exactly. If a 2D classical algorithm is used at the interprocessor level, we call this approach "2D-Strassen". The main limitations of this algorithm are: (a) the communication costs are the same as the classical algorithm used, even though Strassen offers the possibility of reduced communication, and (b) the reduction in flops is less pronounced because Strassen is used only on smaller subproblems.

Second, Strassen can be used "at the top" to generate seven subproblems of half the size (or using ℓ Strassen steps, 7^ℓ subproblems of dimension $\frac{n}{2^\ell}$), and a classical parallel algorithm can be used to evaluate the subproblems. If a 2D algorithm is used for the subproblems, we call this approach "Strassen-2D". The advantage of this approach over 2D-Strassen is that Strassen is used on larger problems and therefore further reduces the flop count. The main drawback of this approach is that it increases the communication costs relative to classical algorithms. For every Strassen step applied, the computation is reduced by a factor of roughly $\frac{7}{8}$ but the bandwidth cost is increased by a factor of $\frac{7}{4}$.

While the authors of [14] concluded that 2D-Strassen achieved higher performance than Strassen-2D due to the communication costs, the authors of [11] showed that with an efficient classical algorithm and a machine with higher network bandwidth, speedups over 2D-Strassen were still possible using Strassen-2D. However, they found that the optimal number of Strassen steps taken was never more than two due to the increasing communication costs. On modern machines, we have found that 2D-Strassen generally outperforms Strassen-2D.

C. 2.5D-Strassen and Strassen-2.5D

A natural extension of the Strassen parallelization approaches of [11], [14] is to replace the 2D classical algorithm with the 2.5D classical algorithm of [17]. Since the 2.5D algorithm reduces communication relative to 2D algorithms, 2.5D-Strassen and Strassen-2.5D communicate less than 2D-Strassen and Strassen-2D, respectively. As argued in [3], this combination still does not attain communication optimality.

D. The CAPS Algorithm

In [3] we introduced CAPS, a communication-optimal parallel algorithm for Strassen's matrix multiplication. As shown in Table I, the bandwidth cost of CAPS is asymptotically less than both the other algorithmic approaches to parallelizing Strassen and the 2.5D classical algorithm. That is, Strassen's algorithm allows for reducing both computation and communication costs compared to the classical algorithm, provided that it is parallelized in the right way.

The CAPS algorithm is based on a parallel traversal of the recursion tree, where at each level of the tree, either a depth-first step (DFS) or a breadth-first step (BFS) is taken. A DFS step consists of all processors working on each of the seven subproblems in sequence, and a BFS step consists of seven subsets of processors each working on one subproblem in parallel. BFS steps require extra memory but reduce communication costs overall. We show in [3] that choosing to do sufficiently many DFS steps (to control the memory footprint) followed by all BFS steps (to reduce the problem to all processors working independently) attains the communication lower bound. BFS and DFS steps may be interleaved; see Section V-B.

E. Strong Scaling Range

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. Note that 2D classical algorithms do not exhibit perfect strong scaling. However, both 2.5D and CAPS strongly scale perfectly within the following ranges [4]:

$$\begin{aligned} P_{\min} \leq P \leq P_{\min}^{\omega_0/2} \quad & \text{for CAPS} \\ P_{\min} \leq P \leq P_{\min}^{3/2} \quad & \text{for 2.5D} \end{aligned}$$

	Flops	Bandwidth
2D [8], [18]	$\frac{n^3}{P}$	$\frac{n^2}{P^{1/2}}$
2.5D [17]	$\frac{n^3}{P}$	$\max\left\{\frac{n^3}{PM^{1/2}}, \frac{n^2}{P^{2/3}}\right\}$
2D-Strassen [14]	$\frac{n^{\omega_0}}{P^{(\omega_0-1)/2}}$	$\frac{n^2}{P^{1/2}}$
Strassen-2D [11]	$\left(\frac{7}{8}\right)^{\ell}\frac{n^3}{P}$	$\left(\frac{7}{4}\right)^{\ell}\frac{n^2}{P^{1/2}}$
CAPS [3]	$\frac{n^{\omega_0}}{P}$	$\max\left\{\frac{n^{\omega_0}}{PM^{\omega_0/2-1}}, \frac{n^2}{P^{2/\omega_0}}\right\}$

TABLE I: Comparison of asymptotic computational and communication costs of classical matrix multiplication algorithms and the Communication-Avoiding Parallel Strassen algorithm. Both the 2.5D classical algorithm and CAPS attain their relative communication lower bounds. Here $\omega_0 = \log_2 7 \approx 2.81$ is the exponent of Strassen; ℓ is the number of Strassen steps.

where $P_{\text{min}} \approx \frac{4n^2}{M}$ is the minimum number of processors required to store the input/output and the required intermediate values, and ω_0 is $\log_2 7$.

III. Performance: CAPS vs. Previous Algorithms

We have implemented CAPS using MPI on three supercomputers, a Cray XE6 (Hopper), an IBM BG/P (Intrepid), and a Cray XT4 (Franklin), 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 slightly 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.

For each of the three machines, we present two types of plots. First, in Figures 1a, 1c, and 1e, 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.

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.

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 II-E with a shaded region. 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.

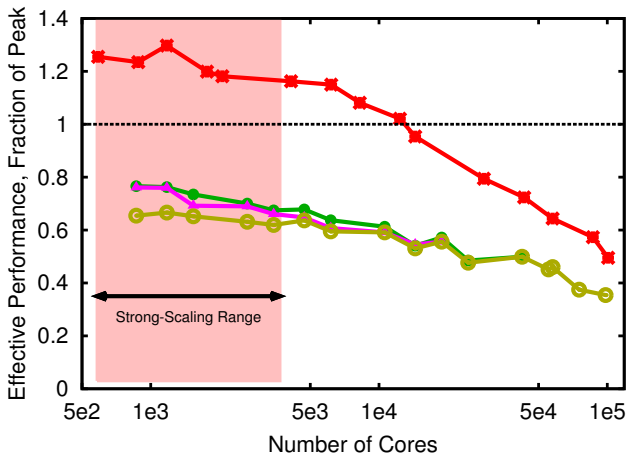
Second, we show execution time for fixed, small matrix dimension over an increasing number of processors. See Figures 1b, 1d, and 1f. For these problem sizes, the execution time is dominated by communication, and the speedup relative to classical algorithms is based primarily on decreases in communication. 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 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.

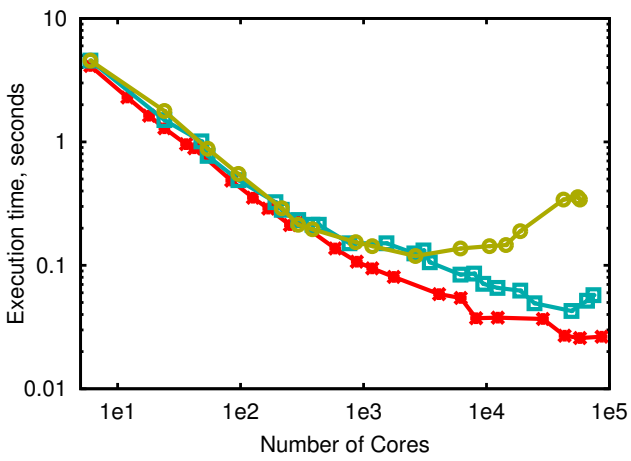
A. Cray XE6 Hopper

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. 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 is ranked number 8 on the TOP500 list [15], with a LINPACK score of 1.05 Tflop/s on a matrix of dimension about 4.5 million.

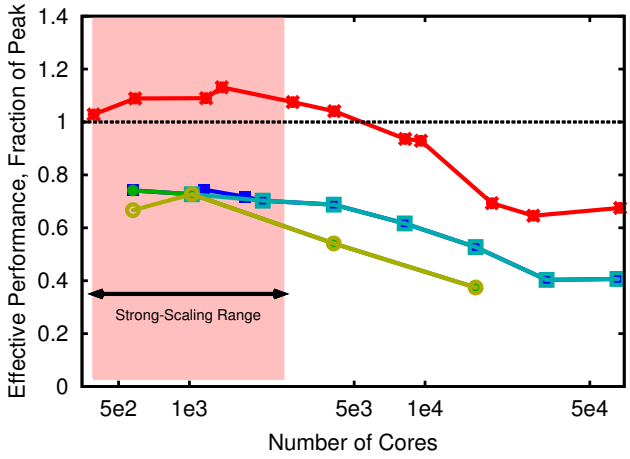
CAPS outperforms all of the previous algorithms. For the large problem ($n = 131712$), it attains performance as high as 30% above the peak for classical matrix multiplication, 83% above 2D, and 75% above Strassen-2D. Note that there does not exist tuned 2.5D code for Hopper, so we did not compare against that algorithm (which should theoretically outperform 2D). On this machine, we benchmark ScaLAPACK/PBLAS 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. 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.



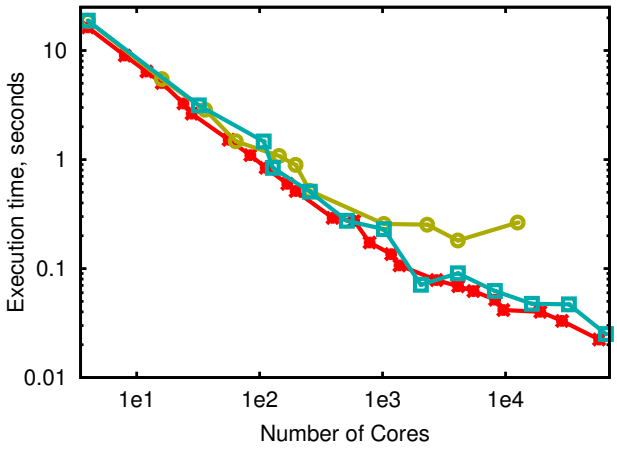
(a) Hopper (Cray XE6), $n = 131712$



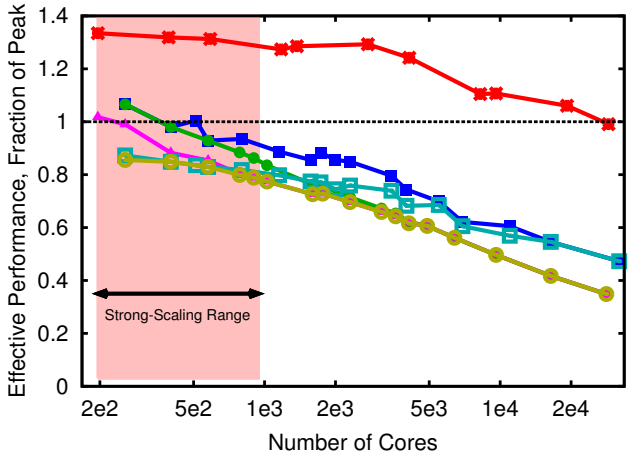
(b) Hopper (Cray XE6), $n = 4704$



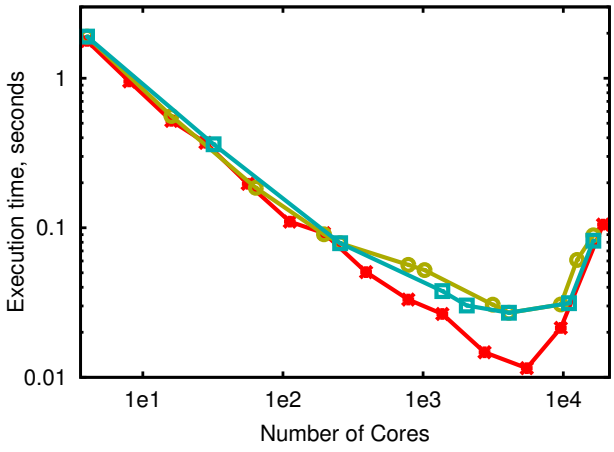
(c) Intrepid (IBM BG/P), $n = 65856$



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



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



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

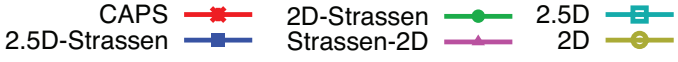


Fig. 1: Strong scaling results for large problems (left) and small problems (right) on three machines. Left column: effective performance on large matrices. Right column: execution time on small matrices.

B. 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. As of November 2011, it is ranked number 23 on the TOP500 list [15], 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 ScaLAPACK, Strassen-2D, and Strassen-2.5D for the large problem size, so we omit those algorithms in the performance plots. 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%.

C. 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. As of November 2011, it is ranked number 38 on the TOP500 list [15], 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 1e. 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. We omit ScaLAPACK data because it is always outperformed by the other classical 2D and 2.5D algorithms on Franklin.

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; increasing the matrix size would only increase the performance of CAPS. 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.

D. CAPS vs. Strassen-based algorithms

Figure 2 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,

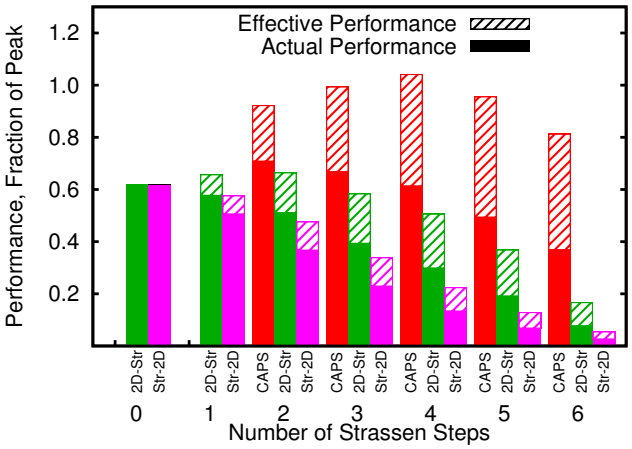


Fig. 2: Efficiency at various numbers of Strassen steps, $n = 21952$, on 49 nodes (196 cores) of Intrepid.

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, two Strassen steps is optimal, but 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.

IV. 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 when repeating an experiment, versus 10-20% on Hopper) and also because we believe there is opportunity for topology-aware optimizations, which we discuss in Section VI-D.

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

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

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_{DAXPY} is the time to read two values, compute one addition, and write one value.

The parameters of our single node model (in seconds) are:

$$T_{\text{DGEMM}}(n) = 2.04{\cdot}10^{-10}n^3 + 2.14{\cdot}10^{-8}n^2 - 4.18{\cdot}10^{-6}n + 2.11{\cdot}10^{-3}$$

$$T_{\text{DAXPY}} = 3.66 \cdot 10^{-9}$$

We present actual and modeled performance of both classical and Strassen performance on a single node in Figure 3. Note that the classical model is nearly indistinguishable from the data in the plot because it is a curve of best fit. The model from Equation (1) correctly chooses the optimal cutoff point to switch to the classical algorithm, and the performance of Strassen matches the classical algorithm below that point.

In Figure 4 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.

B. 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 [6], [13], [18]. 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

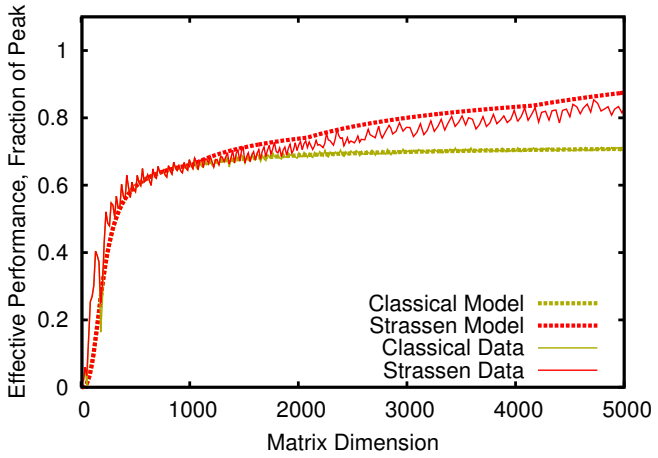


Fig. 3: Comparison of the sequential model to the actual performance of classical and Strassen matrix multiplication on four cores (one node) of Intrepid.

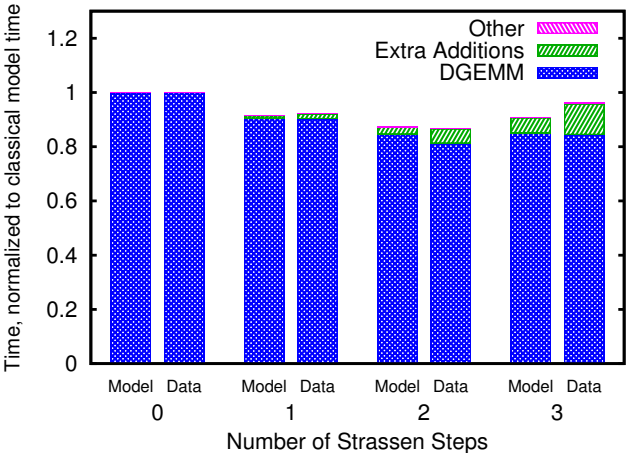


Fig. 4: Time breakdown comparison between the sequential model and the data for $n = 4097$. Both model and data times are normalized to the modeled classical algorithm time.

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).

For a more accurate model, 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_{DAXPY} for the time cost of each scalar addition during the parallel Strassen steps. Then the time spent on computation is

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

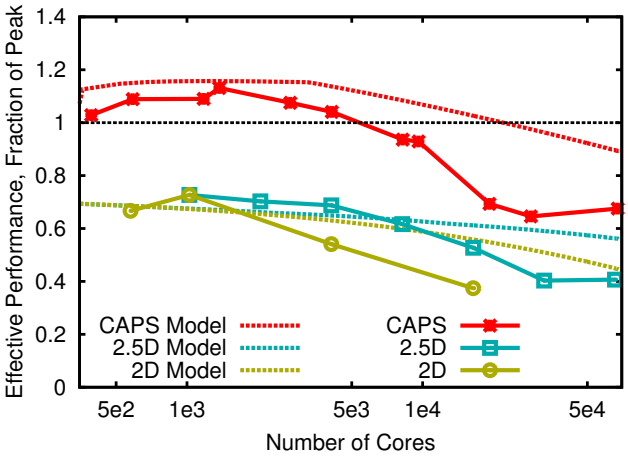


Fig. 5: Comparison of the parallel model with the algorithms in strong scaling of $n = 65856$ on Intrepid.

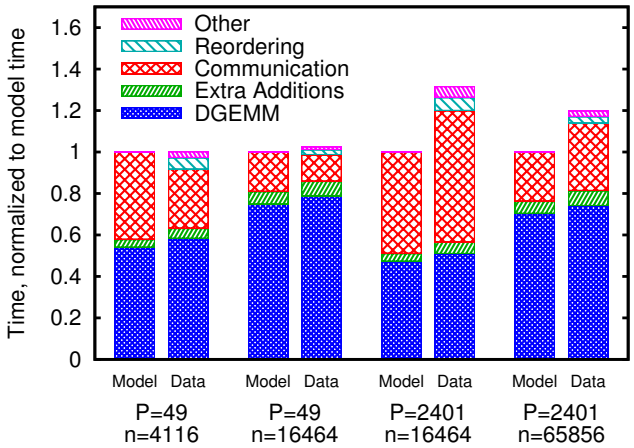


Fig. 6: Time breakdown comparison between the parallel model and the data. In each case the entire modeled execution time is normalized to 1.

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 in the model we allow k and ℓ to be non-integer valued to give a continuous function, even though the algorithm only makes sense for integer values of k and ℓ.

The number of words and messages are exactly as in [3]:

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

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 5. 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 VI-D.

The model also allows us to breakdown the time into communication time (the α and β terms), time spent in calls to `DGEMM` (the first term in Equation 1), and time spent in additions (the second term in Equation 1). We compare these times to the actual time breakdown in Figure 6. 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 only counts bandwidth along one direction on one of the six links to a given node, and ignores communication hiding.

C. 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 [16]:

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

The projected speedups of CAPS over 2.5D and 2D are shown in Figure 7. 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 $5.45\times$ 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 much more for small problems, there are substantial speedups of $5.45\times$ in the top right, and $5.27\times$ in the communication-bound regime at the bottom right.

V. Implementation Details

The implementation of CAPS follows the algorithm presented in [3]. This section will fill in several of the details that were left out of that discussion.

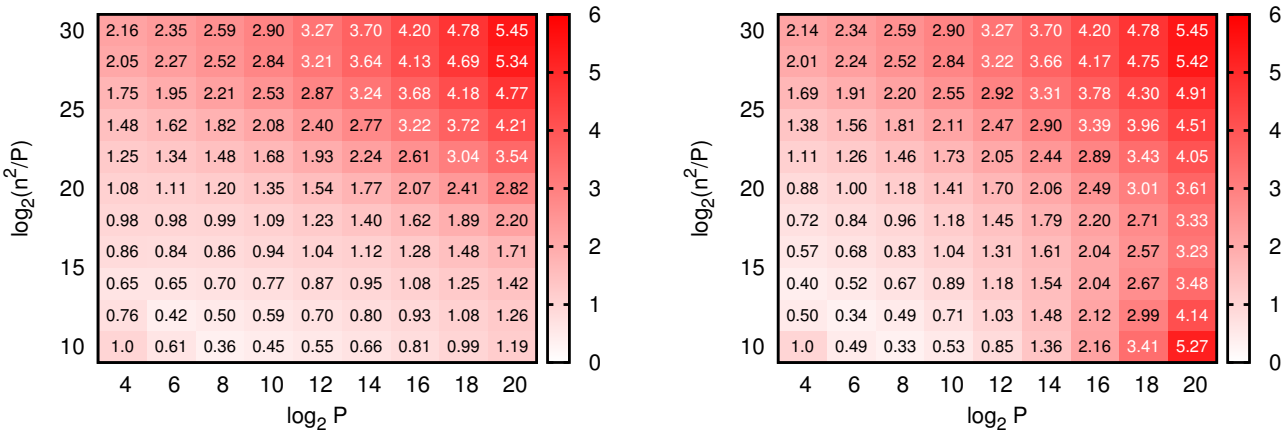


Fig. 7: Predicted speedups of CAPS over 2.5D (left) and 2D (right) on an exascale machine.

A. 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, 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 [3]. 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 Morton ordering.

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 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.

B. Interleaving BFS and DFS steps

As argued in [3], 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 significant reduction in the bandwidth costs.

For example, when running on $16807 = 7^5$ processors, it is sometimes possible to reduce the volume of communication by nearly 25% by choosing the optimal interleaving, depending on the amount of available memory; see Figure 8.

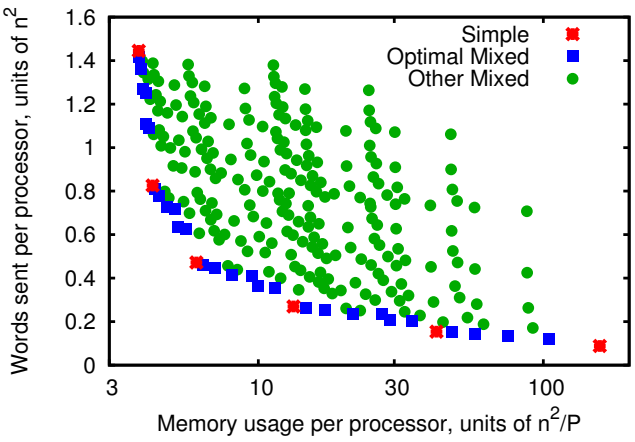


Fig. 8: The memory and communication costs of all 252 possible interleavings of BFS and DFS steps for multiplying matrices of size $n = 351232$ on $P = 16807$ processors using 10 Strassen steps. The optimal ones show the trade-off between memory size and communication cost. Simple interleavings are those for which all k BFS steps are performed as a block.

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

In [3], we assume 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.

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 VI-E), but the extra communication from DFS steps overlaps more easily with the calls to `DGEMM` (see Section V-D).

D. 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 product Q's 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.

VI. Discussion and Conclusions

A. Hardware scaling

Although Strassen performs asymptotically less computation and communication than classical matrix multiplication, it is more demanding on the hardware. That is, if one wants to run matrix multiplication near the peak CPU speed, Strassen is more demanding of the memory size and communication bandwidth. This is because the ratio of computational cost to bandwidth cost is lower for Strassen than for classical matrix multiplication. The asymptotic ratio of computational cost to bandwidth cost is $M^{\omega_0/2-1}$ for Strassen-based algorithms, versus $M^{1/2}$ for classical algorithms. In terms of the machine parameters β and γ introduced in Section IV, the condition to be able to be computation-bound is $\gamma M^{1/2} \geq c\beta$ for classical matrix multiplication and $\gamma M^{\omega_0/2-1} \geq c'\beta$ for Strassen. Here c and c' are constants that depend on the constants in the costs of classical and Strassen-based matrix multiplication.

The above inequalities may guide hardware design as long as classical and Strassen matrix multiplication are considered important computations. They apply both to the distributed case, where M is the local memory size and β is the inverse network bandwidth, and to the sequential case where M is the cache size and β is the inverse memory-cache bandwidth.

B. Mapping Strassen to current hardware

Current hardware designs are a much better fit to classical matrix multiplication algorithms than to Strassen. In this section we highlight two examples of such hardware choices. Classical matrix multiplication is balanced between additions and multiplications, whereas Strassen performs some extra additions. As long as the cutoff between Strassen and the classical algorithm is large, the time for these additions is not significant. However if one uses Strassen for small matrices, the imbalance is significant. Since most modern floating point units can operate at peak efficiency only if the additions and multiplications are balanced, this hardware choice puts Strassen and other fast algorithms at a disadvantage.

More relevant to the CAPS algorithm is the number of processors in a system. CAPS runs fastest when it is given a power of 7 processors. On machines where arbitrary sized partitions are allowed, this is not a problem. However on a BG/P such as Intrepid, one must use a power of two nodes, whereas CAPS would prefer using a power of 7. Ignoring the extra nodes will, in many cases, nullify the speedup of CAPS. The same problem exists on a shared-memory machine, where the number of cores or NUMA regions is typically a power of 2 and is almost never a multiple of 7.

C. Stability

Strassen's algorithm (and all other fast matrix multiplication algorithms) satisfy weaker error bounds than the classical algorithm. While the classical algorithm satisfies a component-wise bound, Strassen satisfies only norm-wise bounds. Further, the constant in the norm-wise bounds for Strassen is larger than for the classical algorithm. However, using fewer than $\log_2 n$ Strassen steps improves the theoretical constant. More precisely, using s Strassen steps, the error bound for Strassen-Winograd given in [12] is

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

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

However, as illustrated in [12], this theoretical bound is too pessimistic. In Figure 9, 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 random from $[-1, 1]$, varying the number of Strassen steps taken. For the "exact" answer

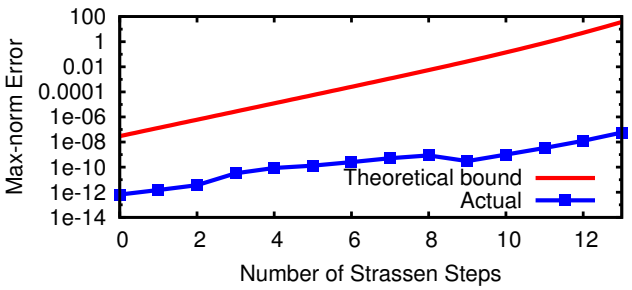


Fig. 9: Stability test: theoretical error bound versus actual error for n=16384. Zero Strassen steps corresponds to the classical algorithm. Double precision machine epsilon is $2.2 \cdot 10^{-16}$.

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.

As mentioned in [3] and [9], diagonal scaling can be used to improve the error bounds of Strassen so that they become comparable to other dense linear algebra algorithms, including LU and QR decomposition. In the context of many larger computations, this implies that the stability loss due to using Strassen instead of classical matrix multiplication is no worse than errors made in the rest of the computation.

D. 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 14%).

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. In most cases we got 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. 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 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.

E. 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 V-C) 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 [3], we need to consider only the case that $P = m$.

Claim 1: 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 [5], 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 $M = O(n^2)$. ■

ACKNOWLEDGMENT

We would like to thank Edgar Solomonik for access to his 2.5D code and helping us to get it running efficiently.

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.

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.

REFERENCES

[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*, pages 41–50, New York, NY, USA, 2006. ACM.

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

[3] G. Ballard, J. Demmel, O. Holtz, B. Lipshitz, and O. Schwartz. Communication-optimal parallel algorithm for Strassen's matrix multiplication. Technical Report EECS-2012-32, UC Berkeley, March 2012. To appear in SPAA 2012.

[4] G. Ballard, J. Demmel, O. Holtz, B. Lipshitz, and O. Schwartz. Strong scaling of matrix multiplication algorithms and memory-independent communication lower bounds. Technical Report EECS-2012-31, UC Berkeley, March 2012. To appear in SPAA 2012.

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

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

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

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

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

[10] 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*, page 285, Washington, DC, USA, 1999. IEEE Computer Society.

[11] B. Grayson, A. Shah, and R. van de Geijn. A high performance parallel Strassen implementation. In *Parallel Processing Letters*, volume 6, pages 3–12, 1995.

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

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

[14] 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, pages 221–226, New York, NY, USA, 1995. ACM.

[15] H. Meuer, E. Strohmaier, J. Dongarra, and H. Simon. Top500 supercomputer sites, 2011. www.top500.org.

[16] J. Shalf, S. S. Dosanjh, and J. Morrison. Exascale computing technology challenges. In J. M. L. M. Palma, M. J. Daydé, O. Marques, and J. C. Lopes, editors, *High Performance Computing for Computational Science - VECPAR 2010 - 9th International conference, Berkeley, CA, USA, June 22-25, 2010, Revised Selected Papers*, volume 6449 of *Lecture Notes in Computer Science*, pages 1–25. Springer, 2010.

[17] E. Solomonik and J. Demmel. 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*. Springer, 2011.

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