Communication-Optimal Parallel Recursive Rectangular Matrix Multiplication

James Demmel
David Eliahu
Armando Fox
Shoaib Ashraf Kamil
Benjamin Lipshitz
Oded Schwartz
Omer Spillinger

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2012-205
http://www.eecs.berkeley.edu/Pubs/TechRpts/2012/EECS-2012-205.html

October 24, 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 and DE-SC0004938.
This research used resources of the National Energy Re- search Scientific Computing Center, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231.

Communication-Optimal Parallel Recursive Rectangular Matrix Multiplication

James Demmel*, David Eliahu†††, Armando Fox‡††, Shoaib Kamil§,
Benjamin Lipshitz¶††, Oded Schwartz‖††, and Omer Spillinger**††

***Abstract*—Communication-optimal algorithms are known for square matrix multiplication. Here, we obtain the first communication-optimal algorithm for all dimensions of rectangular matrices. Combining the dimension-splitting technique of Frigo, Leiserson, Prokop and Ramachandran (1999) with the recursive BFS/DFS approach of Ballard, Demmel, Holtz, Lipshitz and Schwartz (2012) allows for a communication-optimal as well as cache- and network-oblivious algorithm. Moreover, the implementation is simple: approximately 50 lines of code for the shared-memory version. Since the new algorithm minimizes communication across the network, between NUMA domains, and between levels of cache, it performs well in practice on both shared- and distributed-memory machines. We show significant speedups over existing parallel linear algebra libraries both on a 32-core shared-memory machine and on a distributed-memory supercomputer.**

I. INTRODUCTION

Matrix multiplication is a fundamental kernel of high-performance computing, scientific computing, and distributed computing. It is an inherently parallelizable task, and algorithms which take advantage of parallel architectures achieve much higher performance than serial implementations. When designing efficient parallel algorithms, it is important not only to load-balance the arithmetic operations (*flops*), but also to minimize the amount of data transferred between processors and between levels of the memory hierarchy (*communication*). Since communication is becoming more expensive relative to computation (both in terms of time and energy), finding communication-optimal algorithms has significant practical impact.

The most widely-used algorithm for parallel matrix multiplication is SUMMA [31], which perfectly load-balances the flops for any matrix dimension, but is only communication-optimal for certain matrix dimensions or if assuming no extra memory. For square matrix multiplication, communication cost lower bounds have been proved [22], [5], [2], suggesting that known 2D algorithms (such as SUMMA) and 3D algorithms [7], [1] are only optimal in certain memory ranges. These bounds led to "2.5D" algorithms [25], [28], [27] as well as a BFS/DFS-based algorithm [3], which are communication-optimal for all memory sizes and provide substantial speedups in practice. In this work, we prove new communication cost lower bounds in the case of rectangular matrix multiplication, and give the first parallel algorithm that minimizes communication for all matrix dimensions and memory sizes.

It is possible to perform rectangular matrix multiplication with fewer than the naïve number of operations by using fast algorithms. The arithmetic cost of such algorithms has been extensively studied (see [19], [9], [14], [24], [20], [21], [15] and further details in [10]). Additionally, their communication cost is asymptotically lower than that of the classical algorithm [4]. However, the constant factors in their costs are often large enough that they are not practically useful. An alternate approach is to split the rectangular matrix multiplication problem into several square matrix multiplications, and then solve each of those with a fast square algorithm (such as Strassen's algorithm [30], which has been shown to work well in parallel [6]). In this work we focus only on classical matrix multiplication, and do not consider algorithms that perform less arithmetic.

A. Cache-oblivious algorithms

Although we are primarily interested in the parallel case here, minimizing communication is also useful for sequential matrix multiplication. One technique is to block the algorithm into sizes that fit into cache, which attains the sequential lower bound [18].

An alternate approach is to use a recursive algorithm where the largest of the three dimensions is split at each recursive step. In [16] it is shown that the latter approach asymptotically minimizes the communication costs for any hierarchy of caches and any matrix dimension, without any machine-dependent parameters appearing in the algorithm, making the algorithm *cache-oblivious*. Cache-oblivious algorithms can get good performance on a wide variety of platforms with relatively little programmer effort. Although most high-performance linear algebra libraries are hand-tuned or auto-tuned for specific architectures, there have been a few attempts to write competitive cache-oblivious libraries [32], [33].

We next consider an analogous algorithmic choice in the parallel case.

*Mathematics Department and CS Division, UC Berkeley, Berkeley, CA 94720. demmel@cs.berkeley.edu
†deliahu@berkeley.edu
††EECS Department, UC Berkeley, Berkeley, CA 94720.
‡fox@cs.berkeley.edu
§CSAIL, MIT, Cambridge, MA 02139. skamil@mit.edu
¶lipshitz@cs.berkeley.edu
‖odedsc@cs.berkeley.edu
**omers88@berkeley.edu

B. BFS/DFS versus Grid-based algorithms

Two approaches have emerged for parallelizing dense linear algebra—one that requires tuning and one that does not.

The first is iterative and grid-based, where the processors are thought of as residing on a two- or three-dimensional grid. This class includes SUMMA and 2.5D, as well as the recently proposed 3D-SUMMA algorithms [26], which attempt to combine the communication-avoidance of 2.5D matrix multiplication with the generality of SUMMA. As our communication lower bounds show, 3D-SUMMA is communication-optimal for many, but not all, matrix dimensions (see Table I). Grid-based algorithms can provide very high performance, especially when matched to the grid or torus-based topologies of many modern supercomputers [29]. However they may not perform as well in more general topologies. Even when the global topology is a torus, on some supercomputers the allocation can be an arbitrary subset, eliminating the possibility of matching the algorithm to the network.

The second approach, named BFS/DFS, has recently been used to parallelize Strassen's algorithm in a communication-optimal way to obtain significant speedups (see [3], [6], and a similar algorithm in [25]). BFS/DFS algorithms are based on sequential recursive algorithms, and view the processor layout as a hierarchy rather than a grid. Breadth-first steps (BFS) and depth-first steps (DFS) are alternate ways to solve the subproblems. At a BFS, all of the subproblems are solved in parallel on independent subsets of the processors, whereas at a DFS all the processors work together on one subproblem at a time. In general, BFS steps reduce communication costs, but may require extra memory relative to DFS steps. With correct interleaving of BFS and DFS steps to stay within the available memory, it has been shown that BFS/DFS gives a communication-optimal algorithm both for square classical matrix multiplication and for Strassen's algorithm, for any memory size. Because of their recursive structure, BFS/DFS algorithms are cache-, processor-, and network-oblivious in the sense of [8], [12], [13]. Note that they are not oblivious to the global memory size, which is necessary to determine the optimal interleaving of BFS and DFS steps. Additionally, they are typically a good fit for hierarchical computers, which are becoming more common as individual nodes become more complicated.

C. CARMA

We apply the BFS/DFS approach to the dimension-splitting recursive algorithm to obtain a communication-avoiding recursive matrix multiplication algorithm, *CARMA*, which is asymptotically communication-optimal for any matrix dimensions, number of processors, and memory size, and is cache- and network-oblivious. CARMA is a simple algorithm. However, because it is optimal across the entire range of inputs, we find cases where it significantly outperforms existing, carefully-tuned libraries. A simplified version of its pseudocode appears as Algorithm 1 (for more details, see Algorithm 2). At each recursive step, the largest of the three dimensions is split in half, yielding two subproblems. Depending on the available memory, these subproblems are solved by either a BFS or a DFS.

Algorithm 1 CARMA, in brief

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

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

D. Contributions

Our primary contributions here are the CARMA algorithm, as well as its analysis and benchmarking. We also prove tight lower bounds on the communication costs of rectangular matrix multiplication in all cases. Some of these bounds have appeared previously in [22], and the new bounds use the same techniques (along with those of [2]). As illustrated in Figure 1, the communication costs naturally divide into three cases that we call *one large dimension*, *two large dimensions*, and *three large dimensions*. SUMMA matches the lower bounds in the case of two large dimensions, and 3D-SUMMA matches the lower bounds in cases of two or three large dimensions. CARMA is the only algorithm that matches the communication lower bounds in all three cases.

We implement CARMA and compare it to existing parallel linear algebra libraries on both a multi-node distributed-memory machine and a NUMA shared-memory machine. In the shared-memory case, we compare to Intel's Math Kernel Library (MKL) and show comparable performance for square matrices and matrices where the middle dimension is small relative to the other two, and speedups of up to $6.6\times$ on matrices where the middle dimension is large. In the multi-node case, we compare CARMA to ScaLAPACK running on Hopper at NERSC, and see speedups of up to $3\times$ for square multiplication, $2500\times$ for multiplication where the middle dimension is large, and $2\times$ for multiplication where the middle dimension is small. With minor tuning, ScaLAPACK performs much better and our $2500\times$ speedup drops to $141\times$. As expected, the biggest speedups come in the case of one large dimension, where CARMA communicates much less than previous algorithms.

E. Paper organization

We describe CARMA in detail in Section II. After a brief discussion of the communication model in Section II-A, we prove communication lower bounds for classical rectangular matrix multiplication in Section II-B, and derive the communication costs of CARMA in Section II-C. We summarize the bandwidth costs of these three algorithms in each case in Table I. We implement CARMA and compare it to existing parallel linear algebra libraries in Section III. Finally, in Section IV we discuss data layout requirements for attaining the communication lower bounds, complications involved in

incorporating CARMA into existing libraries, and opportunities for tuning CARMA.

II. ALGORITHM

Detailed pseudocode for CARMA is shown in Algorithm 2. The recursion cuts the largest dimension in half to give two smaller subproblems. At each level of the recursion in CARMA, a decision is made between making a depth-first step (DFS) or a breadth-first step (BFS) to solve the two subproblems. A BFS step consists of two disjoint subsets of processors working independently on the two subproblems in parallel. In contrast, a DFS step consists of all processors in the current subset working on each subproblem in sequence. A BFS step increases memory usage by a constant factor, but decreases future communication costs. On the other hand, a DFS step decreases future memory usage by a constant factor, but increases future communication costs. On P processors, with unlimited memory, we show that the algorithm only needs BFS steps and is communication-optimal. If the execution of only BFS steps causes the memory requirements to surpass the bounds of available memory, it is necessary to interleave DFS steps within the BFS steps to limit memory usage. We show that the resulting algorithm is still communication-optimal, provided the minimal number of DFS steps are taken. In our experiments, memory size is not a limiting factor, and we only perform BFS until we reach single-processor subsets.

Algorithm 2 CARMA(A,B,C,m,k,n,P)

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

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

A. Communication Model and Notation

We model a distributed-memory machine as having P processors, each with local memory size M. The only way to access data in another processor's local memory is by receiving a message. We count the number of words of data sent and received (*bandwidth cost*) and the number of messages sent and received (*latency cost*). We assume that each processor can only send or receive one message at a time, and we count the bandwidth and latency costs along the critical path of an algorithm.

Although designed for a distributed-memory machine, the model also applies to the case of a shared-memory system with non-uniform access. In this case, M is the amount of memory closest to any given processor. Even though a processor may access other memory without sending an explicit message, it is desirable to reduce both the volume and frequency of access to remote memory.

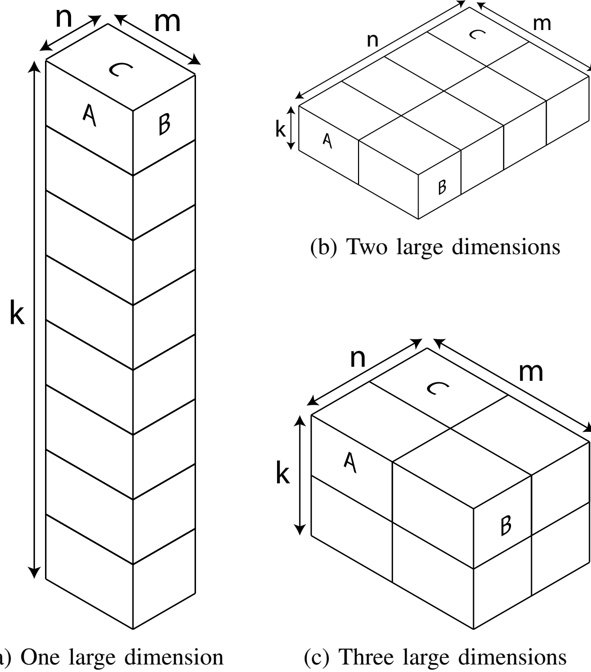
We consider the case of computing $C = AB$ where A is an $m \times k$ matrix, B is a $k \times n$ matrix, and C is an $m \times n$ matrix. In the next two subsections, it will be convenient to have an ordered notation for the three dimensions. Hence we define $d_1 = \min(m, n, k)$, $d_2 = \text{median}(m, n, k)$, and $d_3 = \max(m, n, k)$. Both the lower bounds and the communication costs of CARMA depend only on the values of the three dimensions, not on their order.

B. Communication Cost Lower Bounds

Following [22], the classical rectangular matrix multiplication algorithm requires mnk scalar multiplications, which may be arranged into a rectangular prism of size $m \times n \times k$. To perform a given multiplication, a given processor must have access to the entries of A, B, and C corresponding to the projections onto the $m \times k$, $n \times k$, and $m \times n$ faces of the prism, respectively. If these entries are not assigned to that processor by the initial or final data layout, these entries correspond to words that must be communicated. We assume that the input and output data layout, as well as the computation, are load-balanced. Hence we wish to partition the multiplications in such a way that each processor performs $\Theta(mnk/P)$ multiplications, and the sum of the three projections is minimized. The Loomis-Whitney inequality [23] guarantees that cubes achieve the minimum sum of projections. There are three cases to consider, depending on the aspect ratio of the matrices (see

	$P < \frac{d_3}{d_2}$ "1 large dimension"	$\frac{d_3}{d_2} < P < \frac{d_2 d_3}{d_1^2}$ "2 large dimensions"	$\frac{d_2 d_3}{d_1^2} < P$ "3 large dimensions"
Lower Bound [22][here]	$d_1 d_2$	$\sqrt{\frac{d_1^2 d_2 d_3}{P}}$	$\frac{d_1 d_2 d_3}{P\sqrt{M}} + \left(\frac{d_1 d_2 d_3}{P}\right)^{2/3}$
2D SUMMA [31]	$\sqrt{\frac{d_1^2 d_2 d_3}{P}}$	$\sqrt{\frac{d_1^2 d_2 d_3}{P}}$	$\sqrt{\frac{d_1^2 d_2 d_3}{P}}$
3D SUMMA [26]	$\sqrt{\frac{d_1^2 d_2 d_3}{P}}$	$\sqrt{\frac{d_1^2 d_2 d_3}{P}}$	$\frac{d_1 d_2 d_3}{P\sqrt{M}} + \left(\frac{d_1 d_2 d_3}{P}\right)^{2/3}$
CARMA [here]	$d_1 d_2$	$\sqrt{\frac{d_1^2 d_2 d_3}{P}}$	$\frac{d_1 d_2 d_3}{P\sqrt{M}} + \left(\frac{d_1 d_2 d_3}{P}\right)^{2/3}$

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



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

Fig. 1: Examples of the three cases of aspect ratios on 8 processors. The lowest communication cost is attained if an algorithm divides the $m \times n \times k$ prism of multiplications into 8 equal sub-prisms that are as close to cubical as possible. If that division involves dividing only one of the dimensions, we call that case *one large dimension*. If it involves dividing only two of the dimensions, we call that case *two large dimensions*. If all three dimensions are divided, we call that case *three large dimensions*. Note that in the first two cases, only one processor needs access to any given entry of the largest matrix, so with the correct data layout an algorithm only needs to transfer the smaller matrices.

Figure 1 for graphical representations of the cases).

1) Three large dimensions: If

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

or equivalently $P = \Omega\left(\frac{d_2 d_3}{d_1^2}\right)$, then each processor may compute a cubic sub-block of the prism of size $O(d_1)$, and the results of [22] and [2] apply, giving

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

This bound is attainable for any load-balanced data layout, since it is larger than the cost of redistributing the data.

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

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

2) Two large dimensions: Next consider the case that $P \neq \Omega\left(\frac{d_2 d_3}{d_1^2}\right)$, but $P = \Omega\left(\frac{d_3}{d_2}\right)$. This means that the partition of the prism computed by each processor may not be cubical, but is only limited by one small dimension. The optimal shape of the partitions is then $d_1 \times \sqrt{\frac{d_2 d_3}{P}} \times \sqrt{\frac{d_2 d_3}{P}}$ (which can be thought of as several adjacent cubes of size $d_1 \times d_1 \times d_1$). One processor will thus need access to at least $2\sqrt{\frac{d_1^2 d_2 d_3}{P}} + \frac{d_2 d_3}{P}$ words of A, B, and C. Load-balancing the initial and final data layout means that it owns at most $\frac{d_1 d_2 + d_1 d_3 + d_2 d_3}{P}$ words of A, B, and C. For the largest matrix, only one processor needs access to each entry, so it is possible that no communication is required. For the other two matrices, the data necessary is asymptotically larger than the input or output size, giving a bandwidth lower bound of

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

Note that this lower bound is only attainable if the largest matrix is distributed among the processors in large, asymptotically square blocks.

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

3) One large dimension: Finally, consider the case of one very large dimension, so that $P \neq \Omega\left(\frac{d_3}{d_2}\right)$. The optimal partitioning of the multiplications is a 1D partitioning into prisms of dimension $d_1 \times d_2 \times \frac{d_3}{P}$. One processor will thus need access to at least $d_1 d_2 + \frac{d_1 d_3}{P} + \frac{d_2 d_3}{P}$ words of A, B, and C. Load-balancing the initial and final data layout means that it owns at most $\frac{d_1 d_2 + d_1 d_3 + d_2 d_3}{P}$ words of A, B, and C.

In this case, the largest two matrices might not need any communication, making the lower bound

$$W = \Omega\left(d_1 d_2\right).$$

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

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

C. Communication cost of CARMA

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

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

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

since $d_1 d_2 / P$ is the initial amount of data of the smallest matrix per processor, and it increases by a factor of 2 at each BFS step. In this case the BFS steps can be thought of as performing an all-gather on the smallest matrix.

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

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

At most a constant factor of extra memory is required.

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

2) Two large dimensions: Next consider the case that $\frac{d_3}{d_2} < P \leq \frac{d_2 d_3}{d_1^2}$. There will be two phases: for the first $\log_2 \frac{d_3}{d_2}$ BFS steps, the original largest dimension is split; then for the remaining $\log_2 \frac{P d_2}{d_3}$ BFS steps, the two original largest dimensions are alternately split. Again, no DFS steps are required. The bandwidth cost of the first phase is

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

The bandwidth cost of the second phase is

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

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

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

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

At most a constant factor of extra memory is required, justifying our use of BFS only.

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

3) Three large dimensions: Finally, consider the case that $P > \frac{d_2 d_3}{d_1^2}$. The first phase consists of $\log_2 \frac{d_3}{d_2}$ BFS steps splitting the largest dimension, and is exactly as in the previous case. The second phase consists of $2 \log_2 \frac{d_2}{d_1}$ BFS steps alternately splitting the two original largest dimensions. After this phase, there are $P_3 = \frac{P d_1^2}{d_2 d_3}$ processors working on each subproblem, and the subproblems are multiplication where all three dimensions are within a factor of 2 of each other. CARMA splits each of the dimensions once every three steps, alternating BFS and DFS to stay within memory bounds, until it gets down to one processor.

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

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

In the final phase, the cost is within a factor of 4 of the square case, which was discussed in Section 6.4 of [3], giving

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

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

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

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

giving a total latency cost of

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

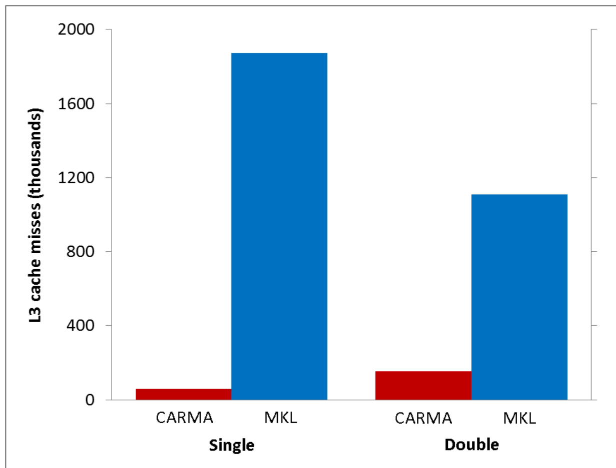


Fig. 3: Level 3 cache misses for CARMA versus MKL for $m = n = 64$, $k = 524288$. CARMA suffers fewer cache misses than MKL, and this reduction in data movement accounts for its speedup.

III. Experimental Results

We have implemented two versions of CARMA: a shared-memory version written using Intel Cilk Plus and a distributed-memory version written in C++ with MPI. Each is benchmarked on three shapes of matrices corresponding to the three cases in the communication costs in Table I.

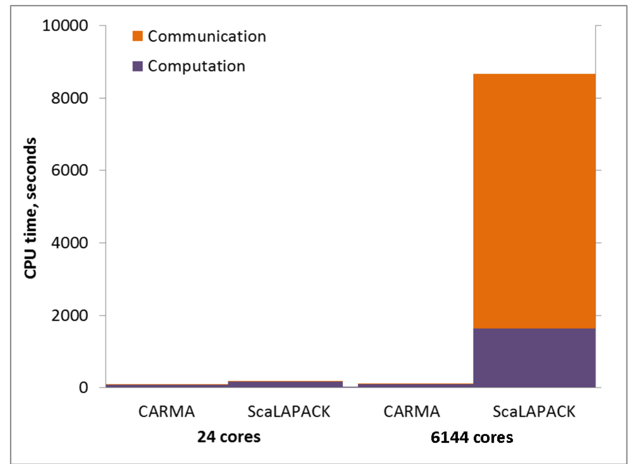
A. *Shared-Memory Version*

We benchmark the shared-memory version on Emerald, which has 4 Intel Xeon X7560 processors, for a total of 32 cores split between 4 NUMA regions. We compare CARMA's performance to Intel's Math Kernel Library (MKL) version 10.3.6. Since Emerald has $32 = 2^5$ cores, CARMA performs 5 BFS steps and then uses MKL serially for the base case multiplications. All tests are for multiplication of randomly generated matrices. In each case the cache is cleared immediately before the benchmark, and the values shown are the average of 50 trials.

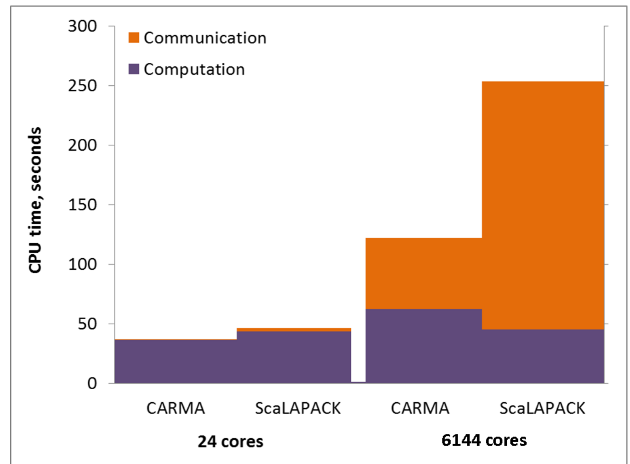
For the case of one large dimension, we benchmark $64 \times k \times 64$ multiplication for k ranging from 2^6 up to 2^{24}, shown in Figure 2a. MKL's performance is roughly flat at about 20 GFlop/s, whereas CARMA is able to use the extra parallelism of larger k values to realize up to a $6.6\times$ speedup for single-precision multiplication, and a $5\times$ speedup for double-precision multiplication.

For the case of two large dimensions, we benchmark $k = 64$ for $m = n$ ranging from 2^6 up to 2^{15}. This is one of the most common shapes since it is used, for example, to perform the updates in LU factorization. On this shape, as shown in Figure 2c, MKL and CARMA achieve similar performance; in most cases MKL is slightly faster.

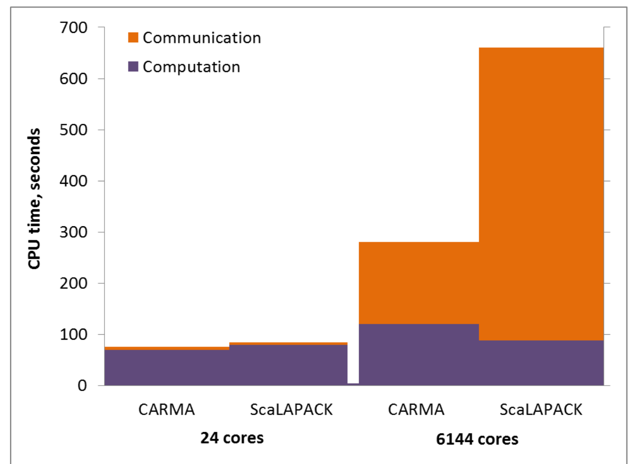
Finally, for the case of three large dimensions, we benchmark square matrices with dimension ranging from 2^6 up to 2^{15}. In this case CARMA is slightly faster than MKL (see Figure 2e). For square multiplication of dimension 8192,



(a) $m = n = 192$, $k = 6291456$

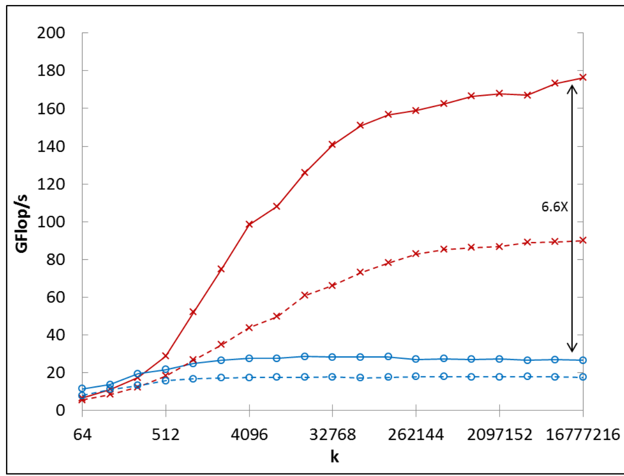


(b) $m = n = 12288$, $k = 192$

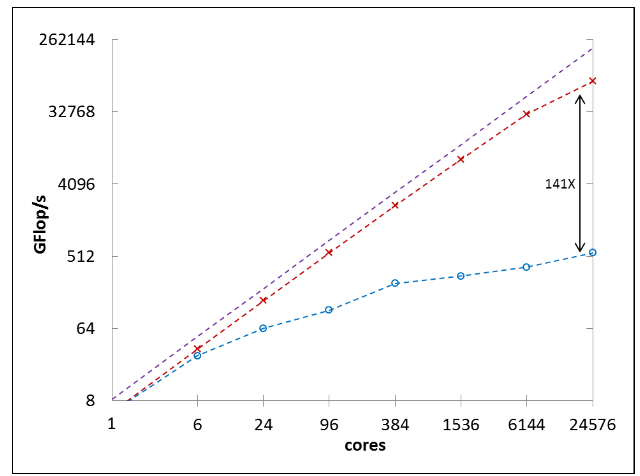


(c) $m = n = k = 6144$

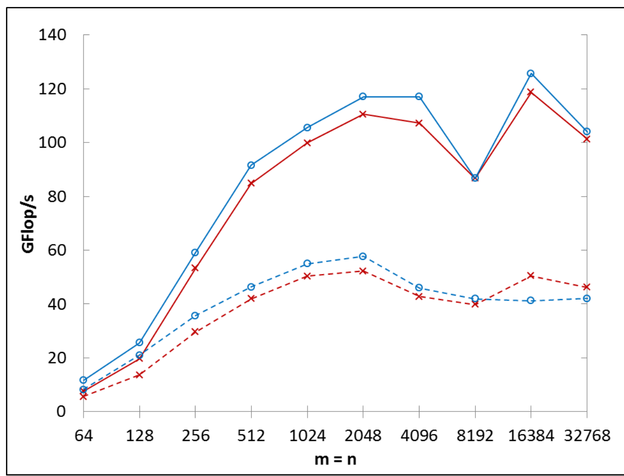
Fig. 4: Time breakdown between communication (MPI calls) and computation (everything else, including local data movement). Perfect strong scaling would correspond to equal total heights at 24 cores and 6144 cores.



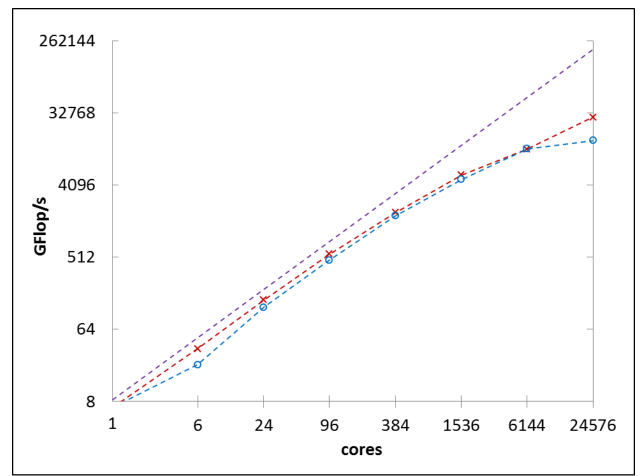
(a) Cilk Plus version, $64 \times k \times 64$.



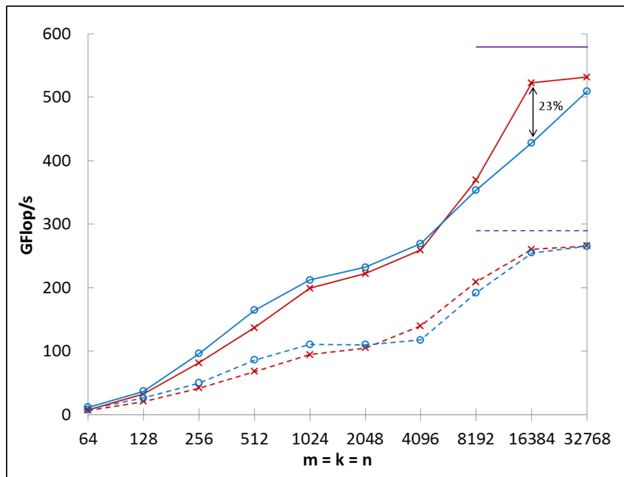
(b) MPI version, $192 \times 6291456 \times 192$



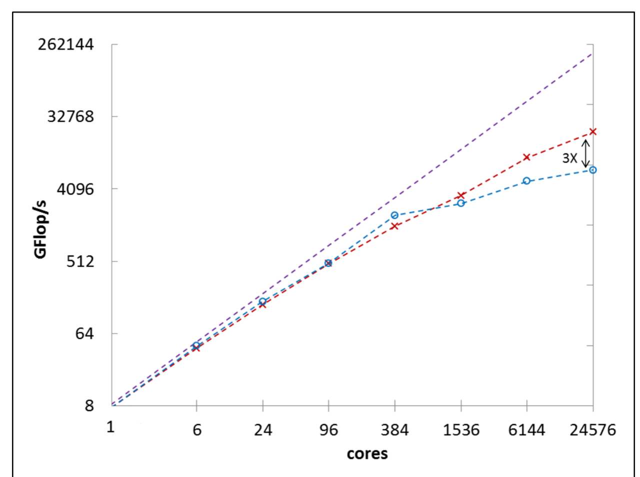
(c) Cilk Plus version, $m \times 64 \times n$, $m = n$.



(d) MPI version, $12288 \times 192 \times 12288$



(e) Cilk Plus version, square.



(f) MPI version, square.

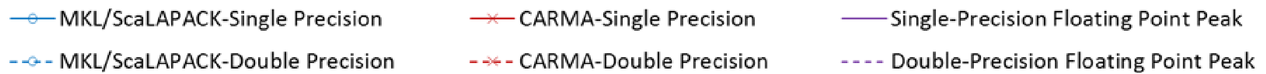


Fig. 2: Performance results for both versions. Left column: performance versus matrix size of the Cilk Plus version of CARMA compared to MKL on Emerald. Right column: strong scaling of the MPI version compared to ScaLAPACK on Hopper.

we see very inconsistent performance for both MKL and CARMA, for which we do not have a definitive explanation.

Figure 3 shows the number of level 3 cache misses for CARMA and for MKL on $m = n = 64$, $k = 524288$, a case where CARMA has a large speedup, obtained using the Performance Application Programming Interface (PAPI [17]). CARMA incurs only 14% as many cache misses in double precision, and only 3% as many in single precision, while performing essentially the same number of floating point operations. As expected, our speedup is due to communication being reduced.

B. Distributed-Memory Version

We benchmark the distributed-memory version on Hopper, a Cray XE6 at the National Energy Research Scientific Computing Center (NERSC). It consists of 6,384 compute nodes, each of which has 2 twelve-core AMD "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.

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

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

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

For the case of one large dimension, we benchmark $m = n = 192$, $k = 6291456$. The aspect ratio is very large so it is in the one large dimension case ($k/P > m, n$) even for our largest run on $P = 24576$ cores. In this case we see speedups of up to $140\times$ over ScaLAPACK. This data is shown in Figure 2b. If ScaLAPACK is not allowed to transpose the input matrices, the speedup grows to $2500\times$.

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

Finally for the case of three large dimensions, we benchmark $m = n = k = 6144$. For small numbers of processors, the problem is compute-bound and both CARMA and ScaLAPACK perform comparably. For more than about 1000 cores, CARMA is faster, and on 24576 cores it attains a speedup of nearly $3\times$. See Figure 2f.

Figure 4 shows the breakdown of time between computation and communication for CARMA and ScaLAPACK, for each of these matrix sizes, and for 24 cores (1 node) and 6144 cores (256 nodes). In the case of 1 large dimension on 6144 cores, CARMA is $16\times$ faster at the computation, but more than $1000\times$ faster at the communication. CARMA is faster at the computation because the local matrix multiplications are as close to square as possible allowing for more efficient use of the cache. For the other two sizes, the computation time is comparable between the two, but CARMA spends about $3.5\times$ less time on communication on 6144 cores.

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

IV. Conclusions and Open Problems

CARMA is the first parallel matrix multiplication algorithm to be communication-optimal for all dimensions of matrices and sizes of memory. We prove CARMA's communication optimality and compare it against ScaLAPACK and MKL. Despite its simple implementation, the algorithm minimizes communication on both distributed- and shared-memory machines, yielding performance improvements of up to $140\times$ and $5\times$, respectively. As expected, our best improvement comes in ranges where CARMA achieves lower bounds on communication but previous algorithms do not. We next discuss future research directions for parallel matrix multiplication.

A. Opportunities for Tuning

The algorithm described in Section II always splits the largest dimension by a factor of 2. This can be generalized considerably. At each recursive step, the largest dimension could be split by any integer factor s, which could vary between steps. Increasing s from 2 decreases the bandwidth cost (by at most a small constant factor) while increasing the latency cost. The choice of split factors is also affected by the number of processors, since the product of all split factors at BFS steps must equal the number of processors. Additionally, when two dimensions are of similar size, either one could be split. As long as the s are bounded by a constant, and the dimension that is split at each step is within a constant factor of the largest dimension, a similar analysis to the one in Section II-C shows that CARMA is still asymptotically

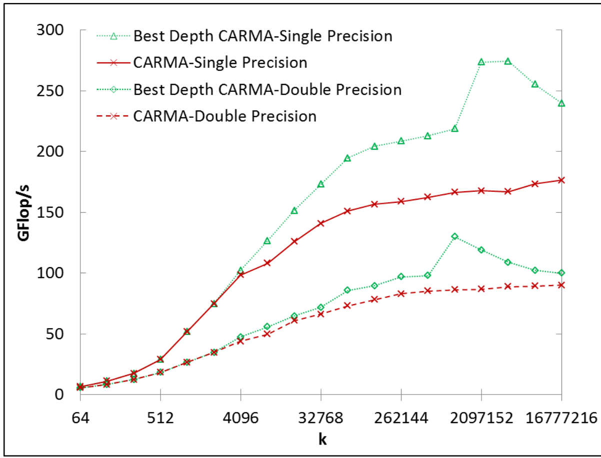


Fig. 5: Choosing the optimal number of recursive steps can give CARMA up to a 60% additional speedup over using the minimum number. The number of recursive steps is one of the parameters in CARMA that can be tuned to search for additional performance.

communication-optimal. Note that this means that CARMA can efficiently use any number of processors that does not have large prime factors, by choosing split factors s that factor the number of processors.

In practice, however, there is a large tuning space, and there may be room for further speedups by exploring more of this space. Our MPI implementation allows the user to choose any dimension to split and any split factor at each recursive step (but the required data layout will vary; see Section IV-D). On Hopper, we have found that splitting 6 or 8 ways at each step typically performs better than splitting 2 ways. We have not performed an exhaustive search of the tuning space, and further tuning CARMA to the specific machine may yield additional speedups.

For the shared-memory version, the data presented in Section II is without any tuning; we benchmarked the simplest version of CARMA, as outlined in Algorithm 2. By changing the depth of recursion above the minimum value of 5 for running on 32 cores, we obtain even larger speedups in some cases, see Figure 5.

B. Comparing Grid-Based and BFS/DFS Algorithms

In Section I-B, we discussed the relative merits of grid-based tuned algorithms with BFS/DFS algorithms that are resource-oblivious. However, we have not performed a thorough experimental comparison of the two. Such a comparison would involve testing both types of algorithms on computers like Hopper, where the allocation is arbitrary and there is an on-node hierarchy, and on computers like IBM Blue Gene, where allocations are guaranteed to be grid topologies and the machine can be thought of as "flat".

It may be possible to use the BFS/DFS approach for many other sequential recursive algorithms. Work is in progress to write a BFS/DFS based SEJITS (Selective Embedded JIT Specialization [11]) specializer to provide automatic resource-oblivious parallelization for recursive algorithms.

C. Perfect Strong Scaling Range

We say that an algorithm exhibits perfect strong scaling if its computation and communication costs decrease linearly with P. In the square case, the 2.5D algorithm and the square BFS/DFS algorithm exhibit perfect strong scaling in the range $P = \Omega(n^2/M)$ and $P = O(n^3/M^{3/2})$, which is the maximum possible range. Similarly, in the case of three large dimensions, defined by

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

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

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

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

In the case of one or two large dimensions, the bandwidth lower bound does not decrease linearly with P (see Table I). As a result, perfect strong scaling is not possible. Figure 2b shows very good strong scaling for CARMA in practice because, even though the bandwidth cost does not decrease with P in this case, it is small enough that it is not dominant up to 6144 cores (see Figure 4a).

D. Data Layout Requirements

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

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

The MPI version of CARMA only communicates one of the three matrices at each BFS step. It requires that each of the two halves of the other two matrices already resides entirely on the corresponding half of the processors. This requirement applies recursively down to some block size, at which point CARMA uses a cyclic data layout (any load-balanced layout would work for the base case). The recursive data layout that the distributed version of CARMA uses is different from

any existing linear algebra library; hence CARMA cannot be directly incorporated into, for example, ScaLAPACK.

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

This limitation does not affect the shared-memory version of CARMA, whose speedups are primarily due to the local matrix multiplications being close to square. Hence the Cilk Plus version of CARMA can be used as a drop-in replacement for the matrix multiplication routine of any shared-memory linear algebra library like MKL.

ACKNOWLEDGMENTS

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 and DE-SC0004938.

This research used resources of the National Energy Research Scientific Computing Center, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231.

REFERENCES

[1] 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.

[2] G. Ballard, J. Demmel, O. Holtz, B. Lipshitz, and O. Schwartz. Brief announcement: Strong scaling of matrix multiplication algorithms and memory-independent communication lower bounds. In *Proceedings of the 24th ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '12, pages 77–79, New York, NY, USA, 2012. ACM.

[3] G. Ballard, J. Demmel, O. Holtz, B. Lipshitz, and O. Schwartz. Communication-optimal parallel algorithm for Strassen's matrix multiplication. In *Proceedings of the 24th ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '12, pages 193–204, New York, NY, USA, 2012. ACM.

[4] G. Ballard, J. Demmel, O. Holtz, B. Lipshitz, and O. Schwartz. Graph expansion analysis for communication costs of fast rectangular matrix multiplication. In *Proceedings of The 1st Mediterranean Conference on Algorithms*, MedAlg '12. Springer, 2012.

[5] 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.

[6] G. Ballard, J. Demmel, B. Lipshitz, and O. Schwartz. Communication-avoiding parallel Strassen: Implementation and performance. In *Proceedings of 2012 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '12, New York, NY, USA, 2012. ACM.

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

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

[9] D. Bini, M. Capovani, F. Romani, and G. Lotti. $O(n^{2.7799})$ complexity for $n \times n$ approximate matrix multiplication. *Information Processing Letters*, 8(5):234 – 235, 1979.

[10] P. Bűrgisser, M. Clausen, and M. A. Shokrollahi. *Algebraic Complexity Theory*. Number 315 in Grundlehren der mathematischen Wissenschaften. Springer Verlag, 1997.

[11] B. Catanzaro, S. A. Kamil, Y. Lee, K. Asanovi, J. Demmel, K. Keutzer, J. Shalf, K. A. Yelick, and A. Fox. Sejits: Getting productivity and performance with selective embedded jit specialization. Technical Report UCB/EECS-2010-23, EECS Department, University of California, Berkeley, Mar 2010.

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

[13] R. Cole and V. Ramachandran. Resource oblivious sorting on multicores. In *Proceedings of the 37th international colloquium conference on Automata, languages and programming*, ICALP'10, pages 226–237, Berlin, Heidelberg, 2010. Springer-Verlag.

[14] D. Coppersmith. Rapid multiplication of rectangular matrices. *SIAM Journal on Computing*, 11(3):467–471, 1982.

[15] D. Coppersmith. Rectangular matrix multiplication revisited. *J. Complex.*, 13:42–49, March 1997.

[16] 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.

[17] B. D. Garner, S. Browne, J. Dongarra, N. Garner, G. Ho, and P. Mucci. A portable programming interface for performance evaluation on modern processors. *The International Journal of High Performance Computing Applications*, 14:189–204, 2000.

[18] J. W. Hong and H. T. Kung. I/O complexity: The red-blue pebble game. In *STOC '81: Proceedings of the thirteenth annual ACM symposium on Theory of computing*, pages 326–333, New York, NY, USA, 1981. ACM.

[19] J. E. Hopcroft and L. R. Kerr. On minimizing the number of multiplications necessary for matrix multiplication. *SIAM Journal on Applied Mathematics*, 20(1):pp. 30–36, 1971.

[20] X. Huang and V. Y. Pan. Fast rectangular matrix multiplications and improving parallel matrix computations. In *Proceedings of the second international symposium on Parallel symbolic computation*, PASCO '97, pages 11–23, New York, NY, USA, 1997. ACM.

[21] X. Huang and V. Y. Pan. Fast rectangular matrix multiplication and applications. *J. Complex.*, 14:257–299, June 1998.

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

[23] L. H. Loomis and H. Whitney. An inequality related to the isoperimetric inequality. *Bulletin of the AMS*, 55:961–962, 1949.

[24] G. Lotti and F. Romani. On the asymptotic complexity of rectangular matrix multiplication. *Theoretical Computer Science*, 23(2):171 – 185, 1983.

[25] W. F. McColl and A. Tiskin. Memory-efficient matrix multiplication in the BSP model. *Algorithmica*, 24:287–297, 1999. 10.1007/PL00008264.

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

[27] E. Solomonik, A. Bhatele, and J. Demmel. Improving communication performance in dense linear algebra via topology aware collectives. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pages 77:1–77:11, New York, NY, USA, 2011. ACM.

[28] 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.

[29] E. Solomonik and J. Demmel. Matrix multiplication on multidimensional torus networks. Technical Report UCB/EECS-2012-28, EECS Department, University of California, Berkeley, Feb 2012.

[30] V. Strassen. Gaussian elimination is not optimal. *Numer. Math.*, 13:354–356, 1969.

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

[32] K. Yotov, X. Li, G. Ren, M. Garzaran, D. Padua, K. Pingali, and P. Stodghill. Is search really necessary to generate high-performance BLAS? *Proceedings of the IEEE*, 93(2):358 –386, Feb 2005.

[33] K. Yotov, T. Roeder, K. Pingali, J. Gunnels, and F. Gustavson. An experimental comparison of cache-oblivious and cache-conscious programs. In *Proceedings of the Nineteenth Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '07, pages 93–104, New York, NY, USA, 2007. ACM.