Write-Avoiding Algorithms

Erin Carson
James Demmel
Laura Grigori
Nick Knight
Penporn Koanantakool
Oded Schwartz
Harsha Vardhan Simhadri

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2015-163
http://www.eecs.berkeley.edu/Pubs/TechRpts/2015/EECS-2015-163.html

June 7, 2015

Write-Avoiding Algorithms

Erin Carson[*], James Demmel[†], Laura Grigori[‡], Nicholas Knight[§], Penporn Koanantakool[¶], Oded Schwartz[‖], Harsha Vardhan Simhadri[**]

June 7, 2015

Abstract

Communication, i.e., moving data, either between levels of a memory hierarchy or between processors over a network, is much more expensive (in time or energy) than arithmetic. So there has been much recent work designing algorithms that minimize communication, when possible attaining lower bounds on the total number of reads and writes.

However, most of this previous work does not distinguish between the costs of reads and writes. Writes can be much more expensive than reads in some current and emerging technologies. The first example is nonvolatile memory, such as Flash and Phase Change Memory. Second, in cloud computing frameworks like MapReduce, Hadoop, and Spark, intermediate results may be written to disk for reliability purposes, whereas read-only data may be kept in DRAM. Third, in a shared memory environment, writes may result in more coherency traffic over a shared bus than reads.

This motivates us to first ask whether there are lower bounds on the number of writes that certain algorithms must perform, and when these bounds are asymptotically smaller than bounds on the sum of reads and writes together. When these smaller lower bounds exist, we then ask when they are attainable; we call such algorithms "write-avoiding (WA)", to distinguish them from "communication-avoiding (CA)" algorithms, which only minimize the sum of reads and writes. We identify a number of cases in linear algebra and direct N-body methods where known CA algorithms are also WA (some are and some aren't). We also identify classes of algorithms, including Strassen's matrix multiplication, Cooley-Tukey FFT, and cache oblivious algorithms for classical linear algebra, where a WA algorithm cannot exist: the number of writes is unavoidably high, within a constant factor of the total number of reads and writes. We explore the interaction of WA algorithms with cache replacement policies and argue that the Least Recently Used (LRU) policy works well with the WA algorithms in this paper. We provide empirical hardware counter measurements from Intel's Nehalem-EX microarchitecture to validate our theory. In the parallel case, for classical linear algebra, we show that it is impossible to attain lower bounds both on interprocessor communication and on writes to local memory, but either one is attainable by itself. Finally, we discuss WA algorithms for sparse iterative linear algebra. We show that, for sequential communication-avoiding Krylov subspace methods, which can perform s iterations of the conventional algorithm for the communication cost of 1 classical iteration, it is possible to reduce the number of writes by a factor of $\Theta(s)$ by interleaving a matrix powers computation with orthogonalization operations in a blockwise fashion.

[*]Computer Science Div., Univ. of California, Berkeley, CA 94720 (ecc2z@eecs.berkeley.edu).

[†]Computer Science Div. and Mathematics Dept., Univ. of California, Berkeley, CA 94720 (demmel@berkeley.edu).

[‡]INRIA Paris-Rocquencourt, Alpines, and UPMC - Univ Paris 6, CNRS UMR 7598, Laboratoire Jacques-Louis Lions, France (laura.grigori@inria.fr).

[§]Computer Science Div., Univ. of California, Berkeley, CA 94720 (knight@cs.berkeley.edu).

[¶]Computer Science Div., Univ. of California, Berkeley, CA 94720 (penpornk@cs.berkeley.edu).

[‖]School of Engineering and Computer Science, Hebrew Univ. of Jerusalem, Israel (odedsc@cs.huji.ac.il).

[**]Computational Research Div., Lawrence Berkeley National Lab., Berkeley, CA 94704 (harshas@lbl.gov).

1 Introduction

The most expensive operation performed by current computers (measured in time or energy) is not arithmetic but communication, i.e., moving data, either between levels of a memory hierarchy or between processors over a network. Furthermore, technological trends are making the gap in costs between arithmetic and communication grow over time [43, 16]. With this motivation, there has been much recent work designing algorithms that communicate much less than their predecessors, ideally achieving lower bounds on the total number of loads and stores performed. We call these algorithms *communication-avoiding (CA)*.

Most of this prior work does not distinguish between loads and stores, i.e., between reads and writes to a particular memory unit. But in fact there are some current and emerging memory technologies and computing frameworks where writes can be much more expensive (in time and energy) than reads. One example is nonvolatile memory (NVM), such as Flash or Phase Change memory (PCM), where, for example, a read takes 12 ns but write throughput is just 1 MB/s [18]. Writes to NVM are also less reliable than reads, with a higher probability of failure. For example, work in [29, 12] (and references therein) attempts to reduce the number of writes to NVM. Another example is a cloud computing framework, where read-only data is kept in DRAM, but intermediate results are also written to disk for reliability purposes [41]. A third example is cache coherency traffic over a shared bus, which may be caused by writes but not reads [26].

This motivates us to first refine prior work on communication lower bounds, which did not distinguish between loads and stores, to derive new lower bounds on writes to different levels of a memory hierarchy. For example, in a 2-level memory model with a small, fast memory and a large, slow memory, we want to distinguish a load (which reads from slow memory and writes to fast memory) from a store (which reads from fast memory and writes to slow memory). When these new lower bounds on writes are asymptotically smaller than the previous bounds on the total number of loads and stores, we ask whether there are algorithms that attain them. We call such algorithms, that both minimize the total number of loads and stores (i.e., are CA), and also do asymptotically fewer writes than reads, *write-avoiding (WA)*[1]. This is in contrast with [12] wherein an algorithm that reduces writes by a constant factor without asymptotically increasing the number of reads is considered "write-efficient".

In this paper, we identify several classes of problems where either WA algorithms exist, or provably cannot, i.e., the numbers of reads and writes cannot differ by more than a constant factor. We summarize our results as follows. First we consider sequential algorithms with communication in a memory hierarchy, and then parallel algorithms with communication over a network.

Section 2 presents our two-level memory hierarchy model in more detail, and proves Theorem 1, which says that the number of writes to fast memory must be at least half as large as the total number of loads and stores. In other words, we can only hope to avoid writes to slow memory. Assuming the output needs to reside in slow memory at the end of computation, a simple lower bound on the number of writes to slow memory is just the size of the output. Thus, the best we can aim for is WA algorithms that only write the final output to slow memory.

Section 3 presents a negative result. We can describe an algorithm by its CDAG (computation directed acyclic graph), with a vertex for each input or computed result, and directed edges indicating dependencies. Theorem 2 proves that if the out-degree of this CDAG is bounded by some constant d, and the inputs are not reused too many times, then the number of writes to slow

[1]For certain computations it is conceivable that by allowing recomputation or by increasing the total communication count, one may reduce the number of writes, hence have a WA algorithm which is not CA. However, in all computations discussed in this manuscript, either avoiding writes is not possible, or it is doable without asymptotically much recomputation or increase of the total communication.

memory must be at least a constant fraction of the total number of loads and stores, i.e., a WA algorithm is impossible. The intuition is that d limits the reuse of any operand in the program. Two well-known examples of algorithms with bounded d are the Cooley-Tukey FFT and Strassen's matrix multiplication.

In contrast, Section 4 gives a number of WA algorithms for well-known problems, including classical ($O(n^3)$) matrix multiplication, triangular solve (TRSM), Cholesky factorization, and the direct N-body problem. All are special cases of known CA algorithms, which may or may not be WA depending on the order of loop nesting. All these algorithms use explicit blocking based on the fast memory size M, and extend naturally to multiple levels of memory hierarchy.

We note that a naive matrix multiplication algorithm for $C = A \cdot B$, three nested loops where the innermost loop computes the dot product of a row of A and column of B, can also minimize writes to slow memory. But since it maximizes reads of A and B (i.e., is not CA), we will not consider such algorithms further.

Dealing with multiple levels of memory hierarchy without needing to know the number of levels or their sizes would obviously be convenient, and many such *cache-oblivious (CO)* CA algorithms have been invented [23, 10]. So it is natural to ask if write-avoiding, cache-oblivious (WACO) algorithms exist. Theorem 3 and Corollary 4 in Section 5 prove a negative result: for a large class of problems, including most direct linear algebra for dense or sparse matrices, and some graph-theoretic algorithms, no WACO algorithm can exist, i.e., the number of writes to slow memory is proportional to the number of reads.

The WA algorithms in Section 4 explicitly control the movement of data between caches and memory. While this may be an appropriate model for the way many current architectures move data between DRAM and NVM, it is also of interest to consider hardware-controlled data movement between levels of the memory hierarchy. In this case, most architectures only allow the programmer to address data by virtual memory address and provide limited explicit control over caching. The cache replacement policy determines the mapping of virtual memory addresses to cache lines based on the ordering of instructions (and the data they access). We study this interaction in Section 6. We report hardware counter measurements on an Intel Xeon 7560 machine ("Nehalem-EX" microarchitecture) to demonstrate how the algorithms in previous sections perform in practice. We argue that the explicit movement of cache lines in the algorithms in Section 4 can be replaced with the Least Recently Used (LRU) replacement policy while preserving their write-avoiding properties (Propositions 6.1 and 6.2).

Next we consider the parallel case in Section 7, in particular a homogeneous distributed memory architecture with identical processors connected over a network. Here interprocessor communication involves a read from the sending processor's local memory and a write to the receiving processor's local memory, so the number of reads and writes caused by interprocessor communication are necessarily equal (up to a modest factor, depending on how collective communications are implemented). Thus, if we are only interested in counting "local memory accesses," then CA and WA are equivalent (to within a modest factor). Interesting questions arise when we consider the local memory hierarchies on each processor. We consider three scenarios.

In the first and simplest scenario (called Model 1 in Section 7) we suppose that the network reads from and writes to the lowest level of the memory hierarchy on each processor, say L_2. So a natural idea is to try to use a CA algorithm to minimize writes from the network, and a WA algorithm locally on each processor to minimize writes to L_2 from L_1, the highest level. Such algorithms exist for various problems like classical matrix multiplication, TRSM, Cholesky, and N-body. While this does minimize writes from the network, it does *not* attain the lower bound for writes to L_2 from L_1. For example, for n-by-n matrix multiplication the number of writes to L_2 from L_1 exceeds the lower bound $\Omega(n^2/P)$ by a factor $\Theta(\sqrt{P})$, where P is the number of processors.

But since the number of writes $O(n^2/\sqrt{P})$ equals the number of writes from the network, which are very likely to be more expensive, this cost is unlikely to dominate. One can in fact attain the $\Omega(n^2/P)$ lower bound, by using an L_2 that is $\Theta(\sqrt{P})$ times larger than the minimum required, but the presence of this much extra memory is likely not realistic.

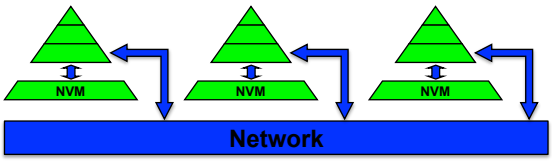


Figure 1: Distributed memory model with NVM disks on each node (see Models 2.1 and 2.2 in Section 7). Interprocessor communication occurs between second lowest level of the memories of each node.

In the second scenario (called Model 2.1 in Section 7) we again suppose that the network reads from and writes to the same level of memory hierarchy on each processor (say DRAM), but that there is another, lower level of memory on each processor, say NVM (see Figure 1). We additionally suppose that all the data fits in DRAM, so that we don't need to use NVM. Here the question becomes whether we can exploit this additional (slow) memory to go faster. There is a class of algorithms that may do this, including for linear algebra (see [20, 44, 36, 4] and the references therein), N-body problems [21, 38] and more general algorithms as well [15], that replicate the input data to avoid (much more) subsequent interprocessor communication. We do a detailed performance analysis of this possibility for the 2.5D matrix multiplication algorithm, which replicates the data $c \geq 1$ times in order to reduce the number of words transferred between processors by a factor $\Theta(c^{1/2})$. By using additional NVM one can increase the replication factor c for the additional cost of writing and reading NVM. Our analysis gives conditions on various algorithm and hardware parameters (e.g., the ratio of interprocessor communication bandwidth to NVM write bandwidth) that determine whether this is advantageous.

In the third scenario (called Model 2.2 in Section 7) we again assume the architecture in Figure 1, but now assume that the data does not fit in DRAM, so we need to use NVM. Now we have two communication lower bounds to try to attain, on interprocessor communication and on writes to NVM. In Theorem 4 we prove this is impossible, that any algorithm must asymptotically exceed at least one of these lower bounds. We then present two algorithms, each of which attains one of these lower bounds. Which one is faster will again depend on a detailed performance analysis using various algorithm and hardware parameters. Section 7.2 extends these algorithms to LU factorization without pivoting.

In Section 8, we consider Krylov subspace methods (KSMs), such as conjugate gradient (CG), for which a variety of CA versions exist (see [4] for a survey). These CA-KSMs are *s-step methods*, which means that they can take s steps of the conventional algorithm for the communication cost of 1 step. We show that it is indeed possible to reorganize them to reduce the number of writes by a factor of $\Theta(s)$, but at the cost of increasing both the number of reads and the number of arithmetic operations by a factor of at most 2.

Finally, Section 9 draws conclusions and lists open problems.

2 Memory Model and Lower Bounds

Here we both recall an earlier complexity model, which counted the total number of words moved between memory units by load and store operations, and present our new model which instead counts reads and writes separately. This will in turn require a more detailed execution model, which maps the presence of a particular word of data in memory to a sequence of load/store and then read/write operations.

The previous complexity model [7] assumed there were two levels of memory, fast and slow. Each memory operation either moved a block of (consecutive) words of data from slow to fast memory (a "load" operation), or from fast to slow memory (a "store" operation). It counted the total number W of words moved in either direction by all memory operations, as well as the total number S of memory operations. The cost model for all these operations was $\beta W + \alpha S$, where β was the reciprocal bandwidth (seconds per word moved), and α was the latency (seconds per memory operation).

Consider a model memory hierarchy with levels L_1, L_2, L_3, and DRAM, where we assume data does not bypass any level: e.g., for data to move from L_1 to DRAM it must first move to L_2 and then to L_3 before on to DRAM.

Fact 1 *For lower bound purposes, we can treat some upper levels, say L_1 and L_2, as fast memory, and the remaining lower levels, L_3 and DRAM, as slow memory. Thus we can model the data movement between any two consecutive levels of the memory hierarchy.*

This technique is well known (cf. [7]), and extends to translating write lower bounds from the two-level model to the memory hierarchy model. Note that converting a WA algorithm for the two-level model into one for the memory hierarchy model is not as straightforward. A similar, more subtle argument can be used to derive lower bounds for the distributed model, by observing the operation of one processor.

Much previous work on communication lower bounds and optimal algorithms explicitly or implicitly used these models. Since our goal is to bound the number of writes to a particular memory level, we refine this model as follows:

- A load operation, which moves data from slow to fast memory, consists of a read from slow memory and a write to fast memory.
- A store operation, which moves data from fast to slow memory, consists of a read from fast memory and a write to slow memory.
- An arithmetic operation can only cause reads and writes in fast memory.

We do not assume a minimum number of reads or writes per arithmetic operation. For example, in the scenario above with fast $= \{L_1, L_2\}$ and slow $= \{L_3, \text{DRAM}\}$ memories, arbitrarily many arithmetic operations can be performed on data moved from L_2 to L_1 without any additional L_2 traffic.

If we only have a lower bound on the total number of loads and stores, then we don't know enough to separately bound the number of writes to either fast or slow memory. And knowing how many arithmetic operations we perform also does not give us a lower bound on writes to fast memory. We need the following more detailed model of the entire duration with which a word in memory is associated with a particular "variable" of the computation. Of course a compiler may introduce various temporary variables that are not visible at the algorithmic or source code level. Ignoring these, and considering only, for example, the entries of matrices in matrix multiplication

$C = A \cdot B$, will still let us translate known lower bounds on loads and stores for matrix multiplication to a lower bound on writes. But when analyzing a particular algorithm, we can take temporary variables into account, typically by assuming that variables like loop indices can reside in a higher level of the memory hierarchy, and not cause data movement between the levels we are interested in.

We consider a variable *resident* in fast memory from the time it first appears to the time it is last used (read or written). It may be updated (written) multiple times during this time period, but it must be associated with a unique data item in the program, for instance a member of a data structure, like the matrix entry $A(i, j)$ in matrix multiplication. If it is a temporary accumulator, say first for $C(1, 1)$, then for $C(1, 2)$, then between each read/write we can still identify it with a unique entry of C. During the period of time in which a variable is resident, it is stored in a fixed fast memory location and identified with a unique data item in the program. We distinguish two ways a variable's residency can begin and can end. Borrowing notation from [7], a residency can begin when

R1: the location is loaded (read from slow memory and written to fast memory), or

R2: the location is computed and written to fast memory, without accessing slow memory; for example, an accumulator may be initialized to zero just by writing to fast memory.

At the end of residency, we determine another label as follows:

D1: the location is stored (read from fast memory and written to slow memory), or

D2: the location is discarded, i.e., not read or written again while associated with the same variable.

This lets us classify all residencies into one of 4 categories:

R1/D1: The location is initially loaded (read from slow and written to fast memory), and eventually stored (read from fast and written to slow).

R1/D2: The location is initially loaded (read from slow and written to fast memory), and eventually discarded.

R2/D1: The location is written to fast memory, and eventually stored (read from fast and written to slow memory).

R2/D2: The location is written to fast memory, and eventually discarded, without accessing slow memory.

In each category there is a write to fast memory, and possibly more, if the value in fast memory is updated. In particular, given all the loads and stores executed by a program, we can uniquely label them by the residencies they correspond to. Since each residency results in at least one write to fast memory, the number of writes to fast memory is at least half the total number of loads and stores (this lower bound corresponds to all residencies being R1/D1). This proves the following result:

Theorem 1 *Given the preceding memory model, the number of writes to fast memory is at least half the total number of loads and stores between fast and slow memory.*

Thus, the various existing communication lower bounds, which are lower bounds on the total number of loads and stores, immediately yield lower bounds on writes to fast memory. In contrast, if most of the residencies are R1/D2 or R2/D2, then we see that no corresponding lower bound on

writes to slow memory exists. In this case, if we additionally assume the final output must reside in slow memory at the end of execution, we can lower bound the number of writes to slow memory by the size of the output. For the rest of this paper, we will make this assumption, i.e., that the output must be written to slow memory at the end of the algorithm.

2.1 Bounds for 3 or more levels of memory

To be more specific, we consider communication lower bounds on the number W of loads and stores between fast and slow memory of the form $W = \Omega(\#flops/f(M))$, where $\#flops$ is the number of arithmetic operations performed, M is the size of the fast memory, and f is an increasing function. For example, $f(M) = M^{1/2}$ for classical linear algebra [7], $f(M) = M^{\log_2 7-1}$ for Strassen's fast matrix multiplication [8], $f(M) = M$ for the direct N-body problem [15, 21], $f(M) = \log M$ for the FFT [2, 28], and more generally $f(M) = M^e$ for some $e > 0$ for a large class of algorithms that access arrays with subscripts that are linear functions of the loop indices [15]. Thus, the lower bound is a decreasing function of fast memory size M.[2].

Corollary 1 *Consider a three level memory hierarchy with levels L_3, L_2, and L_1 of sizes $M_3 > M_2 > M_1$. Let W_{ij} be the number of words moved between L_i and L_j. Suppose that $W_{23} = \Omega(\#flops/f(M_2))$ and $W_{12} = \Omega(\#flops/f(M_1))$. Then the number of writes to L_2 is at least $W_{23}/2 = \Omega(\#flops/f(M_2))$.*

Proof: By Theorem 1 and Fact 1. □

Note that in Corollary 1 the write lower bound is the smaller of the two communication lower bounds. This will give us opportunities to do asymptotically fewer writes than reads to all intermediate levels of the memory hierarchy.

To formalize the definition of WA for multiple levels of memory hierarchy, we let $L_r, L_{r-1}, \ldots, L_1$ be the levels of memory, with sizes $M_r > M_{r-1} > \cdots > M_1$. We assume all data fit in the largest level L_r. The lower bound on the total number of loads and stores between levels L_{s+1} and L_s is $\Omega(\#flops/f(M_s))$, which by Theorem 1 is also a lower bound on the number of writes to L_s, which is "fast memory" with respect to L_{s+1}. The lower bound on the total number of loads and stores between levels L_s and L_{s-1} is $\Omega(\#flops/f(M_{s-1}))$, but we know that this does not provide a lower bound on writes to L_s, which is "slow memory" with respect to L_{s-1}.

Thus a WA algorithm must perform $\Theta(\#flops/f(M_s))$ writes to L_s for $s < r$, but only $\Theta(output\_size)$ writes to L_r.

This is enough for a bandwidth lower bound, but not a latency lower bound, because the latter depends on the size of the largest message, i.e., the number of messages is bounded below by the number of words moved divided by the largest message size. If L_s is being written by data from L_{s+1}, which is larger, then messages are limited in size by at most M_s, i.e., a message from L_{s+1} to L_s cannot be larger than all of L_s. But if L_s is being written from L_{s-1}, which is smaller, the messages are limited in size by the size M_{s-1} of L_{s-1}. In the examples below, we will see that the number of writes to L_s from L_{s-1} and L_{s+1} (for $r > s > 1$) are within constant factors of one another, so the lower bound on messages written to L_s from L_{s+1} will be $\Theta(\#flops/(f(M_s)M_s))$, but the lower bound on messages written to L_s from L_{s-1} will be $\Theta(\#flops/(f(M_s)M_{s-1}))$, i.e., larger. We will use this latency analysis for the parallel WA algorithms in Section 7.

We note that the above analysis assumes that the data is so large that it only fits in the lowest, largest level of the memory hierarchy, L_r. When the data is smaller, asymptotic savings

[2]For some algorithms f may vary across memory levels, e.g., when switching between classical and Strassen-like algorithms according to matrix size.

are naturally possible. For example if all the input and output data fits in some level L_s for $s < r$, then one can read all the input data from L_r to L_s and then write the output to L_s with no writes to L_t for all $t > s$. We will refer to this possibility again when we discuss parallel WA algorithms in Section 7, where L_r's role is played by the memories of remote processors connected over a network.

2.2 Write-buffers

We should mention how to account for *write-buffers* (a.k.a. burst buffers) [26] in our model. A write-buffer is an extra layer of memory hierarchy in which items that have been written are temporarily stored after a write operation and eviction from cache, and from which they are eventually written to their final destination (typically DRAM). The purpose is to allow (faster) reads to continue and use the evicted location, and in distributed machines, to accommodate incoming and outgoing data that arrive faster than the network bandwidth or memory injection rate allow. In the best case, this could allow perfect overlap between all reads and writes, and so could decrease the total communication time by a factor of 2 for the sequential model. For the purpose of deriving lower bounds, we could also model it by treating a cache and its write-buffer as a single larger cache. Either way, this does not change any of our asymptotic analysis, or the need to do many fewer writes than reads if they are significantly slower than reads (also note that a write-buffer does not avoid the per-word energy cost of writing data).

3 Bounded Data Reuse Precludes Write-Avoiding

In this section we show that if each argument (input data or computed value) of a given computation is used only a constant number of times, then it cannot have a WA algorithm. Let us state this in terms of the algorithm's computation directed acyclic graph *(CDAG)*. Recall that for a given algorithm and input to that algorithm, its CDAG has a vertex for each input, intermediate and output argument, and edges according to direct dependencies. For example, the operation $x = y+z$ involves three vertices for x, y, and z, and directed edges (y, x) and (z, x). If two or more binary operations both update x, say $x = y+z, x = x+w$, then this is represented by 5 vertices w, x_1, x_2, y, z and four edges $(y, x_1), (z, x_1), (x_1, x_2), (w, x_2)$. Note that an input vertex has no ingoing edges, but an output vertex may have outgoing edges.

Theorem 2 (Bounded reuse precludes WA) *Let G be the CDAG of an algorithm A executed on input I on a sequential machine with a two-level memory hierarchy. Let G' be a subgraph of G. If all vertices of G', excluding the input vertices, have out-degree at most d, then*

1. *If the part of the execution corresponding to G' performs t loads, out of which N are loads of input arguments, then the algorithm must do at least $\lceil (t - N)/d \rceil$ writes to slow memory.*
2. *If the part of the execution corresponding to G' performs a total of W loads and stores, of which at most half are loads of input arguments, then the algorithm must do $\Omega(W/d)$ writes to slow memory.*

Proof: Of the t loads from slow memory, $t - N$ must be loads of intermediate results rather than inputs. These had to be previously written to slow memory. Since the maximum out-degree of any intermediate data vertex is d, at least $\lceil (t - N)/d \rceil$ distinct intermediate arguments have been written to slow memory. This proves (1).

If the execution corresponding to G' does at least $W/10d$ writes to the slow memory, then we are done. Otherwise, there are at least $t = \frac{10d-1}{10d}W$ loads. Applying (1) with $N \leq W/2$, we conclude that the number of writes to slow memory is at least $\lceil(\frac{10d-1}{10d} - \frac{1}{2})W/d\rceil = \Omega(W/d)$, proving (2). □

We next demonstrate the use of Theorem 2, applying it to Cooley-Tukey FFT and Strassen's (and "Strassen-like") matrix multiplication, thus showing they do not admit WA algorithms.

Corollary 2 (WA FFT is impossible) *Consider executing the Cooley-Tukey FFT on a vector of size n on a sequential machine with a two-level memory hierarchy whose fast memory has size $M \ll n$. Then the number of stores is asymptotically the same as the total number of loads and stores, namely $\Omega(n \log n / \log M)$.*

Proof: The Cooley-Tukey FFT has out-degree bounded by $d = 2$, input vertices included. By [28], the total number of loads and stores to slow memory performed by any implementation of the Cooley-Tukey FFT algorithm on an input of size n is $W = \Omega(n \log n / \log M)$. Since W is asymptotically larger than n, and so also larger than $N = 2n =$ the number of input loads, the result follows by applying Theorem 2 with $G' = G$. □

Corollary 3 (WA Strassen is impossible) *Consider executing Strassen's matrix multiplication on n-by-n matrices on a sequential machine with a two-level memory hierarchy whose fast memory has size $M \ll n$. Then the number of stores is asymptotically the same as the total number of loads and stores, namely $\Omega(n^{\omega_0}/M^{\omega_0/2-1})$, where $\omega_0 = \log_2 7$.*

Proof: We consider G' to be the induced subgraph of the CDAG that includes the vertices of the scalar multiplications and all their descendants, including the output vertices. As described in [8] (G' is denoted there by Dec_C), G' is connected, includes no input vertices ($N = 0$), and the maximum out-degree of any vertex in G' is $d = 4$. The lower bound from [8] on loads and stores for G', and so also for the entire algorithm, is $W = \Omega(n^{\omega_0}/M^{\omega_0/2-1})$. Plugging these into Theorem 2 the corollary follows. □

Corollary 3 extends to any Strassen-like algorithm (defined in [8]), with ω_0 replaced with the corresponding algorithm's exponent. Note that trying to apply the above to classical matrix multiplication does not work: G' is a disconnected graph, hence not satisfying the requirement in [8] for being Strassen-like. Indeed, WA algorithms for classical matrix multiplication do exist, as we later show. However, Corollary 3 does extend to Strassen-like rectangular matrix multiplication: they do not admit WA algorithms (see [5] for corresponding communication cost lower bounds that take into account the three possibly distinct matrix dimensions). And while G' may be disconnected for some Strassen-like rectangular matrix multiplications, this is taken into account in the lower bounds of [5].

4 Examples of WA Algorithms

In this section we give sequential WA algorithms for classical matrix multiplication $C = AB$, solving systems of linear equations $TX = B$ where T is triangular and B has multiple columns by successive substitution (TRSM), Cholesky factorization $A = LL^T$, and the direct N-body problem with multiple particle interactions. In all cases we give an explicit solution for a two-level memory hierarchy, and explain how to extend to multiple levels.

In all cases the WA algorithms are blocked explicitly using the fast memory size M. In fact they are known examples of CA algorithms, but we will see that not all explicitly blocked CA algorithms are WA: the nesting order of the loops must be chosen appropriately.

We also assume that the programmer has explicit control over all data motion, and so state this explicitly in the algorithms. Later in Section 6 we present measurements about how close to the minimum number of writes machines with hardware cache policies can get.

4.1 Classical Matrix Multiplication

We consider classical matrix multiplication, i.e., that performs mnl multiplications to compute $C^{m\times l} = A^{m\times n} * B^{n\times l}$. The lower bound on loads and stores for this operation in a two-level memory hierarchy with fast memory of size M is $\Omega(mnl/M^{1/2})$ [28, 36, 7].

Algorithm 1 is a well-known example of a CA matrix multiplication algorithm and its WA properties have been noted by Blelloch et al. [12]. We have inserted comments to indicate when data is loaded or stored. For simplicity we assume that all expressions like $\sqrt{M/3}$, m/b, etc., are integers.

Algorithm 1: Two-Level Blocked Classical Matrix Multiplication

Data: $C^{m\times l}, A^{m\times n}, B^{n\times l}$
Result: $C^{m\times l} = C^{m\times l} + A^{m\times n} * B^{n\times l}$

```
1 b = sqrt(M1/3)                          // block size for L1; assume n is a multiple of b
  // A(i,k), B(k,j), and C(i,j) refer to b-by-b blocks of A, B, and C
2 for i ← 1 to m/b do
3     for j ← 1 to l/b do
4         load C(i,j) from L2 to L1                          // #writes to L1 = b^2
                                          // total #writes to L1 = (m/b)(l/b)·b^2 = ml
5         for k ← 1 to n/b do
6             load A(i,k) from L2 to L1                      // #writes to L1 = b^2
                                      // total #writes to L1 = (m/b)(l/b)(n/b)·b^2 = mnl/b
7             load B(k,j) from L2 to L1                      // #writes to L1 = b^2
                                      // total #writes to L1 = (m/b)(l/b)(n/b)·b^2 = mnl/b
8             C(i,j) = C(i,j) + A(i,k) * B(k,j)          // b-by-b matrix multiplication
                                        // no communication between L1 and L2
9         store C(i,j) from L1 to L2                         // #writes to L2 = b^2
                                          // total #writes to L2 = (m/b)(l/b)·b^2 = ml
```

It is easy to see that the total number of loads to fast memory is $ml + 2mnl/b$, which attains the lower bound, and the number of stores to slow memory is ml, which also attains the lower bound (the size of the output). It is easy to see that this result depended on k being the innermost loop, otherwise each $C(i, j)$ would have been read and written n/b times, making the number of loads and stores within a constant factor of one another, rather than doing asymptotically fewer writes.

Notice that reducing the number of writes to slow memory also reduces the total number of loads to fast memory compared to CA algorithms that do not necessarily optimize for writes. Take, for example, the cache-oblivious matrix multiplication in [24] which requires $3mnl/b$ loads to fast memory. The number of loads to fast memory is about a third fewer in Algorithm 1 than in the cache-oblivious version when $l, m, n \gg b$. We will observe this in practice in experiments in Section 6.

To see how to deal with more than two levels of memory, we note that the basic problem being solved at each level of the memory hierarchy is $C = C + A * B$, for matrices of different sizes. This lets us repeat the above algorithm, with three more nested loops for each level of memory hierarchy,

in the same order as above. More formally, we use induction: Suppose we have a WA algorithm for r memory levels $L_r, \ldots, L_1$, and we add one more smaller one L_0 with memory size M_0. We need to show that adding three more innermost nested loops will

(1) not change the number of writes to $L_r, \ldots, L_2$,

(2) increase the number of writes to L_1, $O(mnl/\sqrt{M_1})$, by at most a constant factor, and

(3) do $O(mnl/M_0^{1/2})$ writes to L_0.

(1) and (3) follow immediately by the structure of the algorithm. To prove (2), we note that L_1 gets $mnl/(M_1/3)^{3/2}$ blocks of A, B, and C, each square of dimension $b_1 = (M_1/3)^{1/2}$, to multiply and add. For each such b_1-by-b_1 matrix multiplication, it will partition the matrices into blocks of dimension $b_0 = (M_0/3)^{1/2}$ and multiply them using Algorithm 1, resulting in a total of b_1^2 writes to L_1 from L_0. Since this happens $mnl/(M_1/3)^{3/2}$ times, the total number of writes to L_1 from L_0 is $mnl/(M_1/3)^{3/2} \cdot b_1^2 = mnl/(M_1/3)^{1/2}$ as desired.

4.2 Triangular Solve (TRSM)

After matrix multiplication, we consider solving a system of equations $TX = B$ for X, where T is n-by-n and upper triangular, and X and B are n-by-n matrices, using successive substitution. (The algorithms and analysis below are easily generalized to X and B being n-by-m, T being lower triangular, etc.) As with matrix multiplication, we will see that some explicitly blocked CA algorithms are also WA, and some are not.

Algorithm 2: 2-Level Blocked Triangular Solve (TRSM)

Data: T is $n \times n$ upper triangular, $B^{n \times n}$
Result: Solve $TX = B$ for $X^{n \times n}$ (X overwrites B)

```
1  b = sqrt(M1/3)                          // block size for L1; assume n is a multiple of b
   // T(i,k), X(k,j), and B(i,j) refer to b-by-b blocks of T, X, and B
2  for j ← 1 to n/b do
3      for i ← n/b downto 1 do
4          load B(i,j) from L2 to L1                          // #writes to L1 = b^2
                                                  // total #writes to L1 = (n/b)^2 · b^2 = n^2
5          for k ← i+1 to n/b do
6              load T(i,k) from L2 to L1                      // #writes to L1 = b^2
                                                  // total #writes to L1 ≈ .5(n/b)^3 · b^2 = .5n^3/b
7              load B(k,j) from L2 to L1                      // #writes to L1 = b^2
                                                  // total #writes to L1 ≈ .5(n/b)^3 · b^2 = .5n^3/b
8              B(i,j) = B(i,j) − T(i,k) * X(k,j)          // b-by-b matrix multiplication
                                                  // no communication between L1 and L2
9          load T(i,i) from L2 to L1  // about half as many writes as for B(i,j) as counted
           above
10         solve T(i,i) * Tmp = B(i,j) for Tmp; B(i,j) = Tmp          // b-by-b TRSM
                                                  // no communication between L1 and L2
11         store B(i,j) from L1 to L2                 // writes to L2 = b^2, total = (n/b)^2 · b^2 = n^2
```

Algorithm 2 presents an explicitly blocked WA two-level TRSM, using L_2 and L_1, which we then generalize to arbitrarily many levels of memory. The total number of writes to L_1 in Algorithm 2

is seen to be $n^3/b + 3n^2/2$, and the number of writes to L_2 is just n^2, the size of the output. So Algorithm 2 is WA for L_2. Again there is a (correct) CA version of this algorithm for any permutation of the three loops on i, j, and k, but the algorithm is only WA if k is the innermost loop, so that $B(i, j)$ may be updated many times without writing intermediate values to L_2. This is analogous to the analysis of Algorithm 1.

Now we generalize Algorithm 2 to multiple levels of memory. Analogously to Algorithm 1, Algorithm 2 calls itself on smaller problems at each level of the memory hierarchy, but also calls Algorithm 1. We again use induction, assuming the algorithm is WA with memory levels $L_r, \ldots, L_1$, and add one more smaller level L_0 of size M_0. We then replace line 8, $B(i, j) = B(i, j) - T(i, k) * X(k, j)$, by a call to Algorithm 1 but use a block size of $b_0 = (M_0/3)^{1/2}$, and replace line 10, that solves $T(i, i) * Tmp = B(i, j)$, with a call to Algorithm 2, again with block size b_0.

As with matrix multiplication, there are three things to prove in the induction step to show that this is WA. As before (1) follows since adding a level of memory does not change the number of writes to $L_r, \ldots, L_2$. Let $b_1 = (M_1/3)^{1/2}$. To prove (2), we note that by induction, $O(n^3/\sqrt{M_1})$ words are written to L_1 from L_2 in the form of b_1-by-b_1 matrices which are inputs to either matrix multiplication or TRSM. Thus the size of the outputs of each of these matrix multiplications or TRSMs is also b_1-by-b_1, and so also consists of a total of $O(n^3/\sqrt{M_1})$ words. Since both matrix multiplication and TRSM only write the output once to L_1 from L_0 for each matrix multiplication or TRSM, the total number of additional writes to L_1 from L_0 is $O(n^3/\sqrt{M_1})$, the same as the number of writes to L_1 from L_2, as desired. (3) follows by a similar argument.

4.3 Cholesky Factorization

Cholesky factorizes a real symmetric positive-definite matrix A into the product $A = LL^T$ of a lower triangular matrix L and its transpose L^T, and uses both matrix multiplication and TRSM as building blocks. We will once again see that some explicitly blocked CA algorithms are also WA (left-looking Cholesky), and some are not (right-looking). Based on the similar structure of other one-sided factorizations in linear algebra, we conjecture that similar conclusions hold for LU, QR, and related factorizations.

Algorithm 3 presents an explicitly blocked WA left-looking two-level Cholesky, using L_2 and L_1, which we will again use to describe how to write a version for arbitrarily many levels of memory.

As can be seen in Algorithm 3, the total number of writes to L_2 is about $n^2/2$, because the output (lower half of A) is stored just once, and the number of writes to L_1 is $\Theta(n^3/\sqrt{M_1})$.

By using WA versions of the b-by-b matrix multiplications, TRSMs, and Cholesky factorizations, this WA property can again be extended to multiple levels of memory, using an analogous induction argument.

This version of Cholesky is called left-looking because the innermost (k) loop starts from the original entries of $A(j, i)$ in block column i, and completely computes the corresponding entries of L by reading entries $A(i, k)$ and $A(j, k)$ to its left. In contrast, a right-looking algorithm would use block column i to immediately update all entries of A to its right, i.e., the Schur complement, leading to asymptotically more writes.

4.4 Direct N-Body

Consider a system of particles (bodies) where each particle exerts physical forces on every other particle and moves according to the total force on it. The direct N-body problem simulates this by calculating all forces from all k-tuples of particles directly ($k = 1, 2, \ldots$). Letting P be an input array of N particles, F be an output array of accumulated forces on each particle in P, and Φ_k be

Algorithm 3: Two-Level Blocked Classical Cholesky $A^{n\times n} = LL^T$

Data: symmetric positive-definite $A^{n\times n}$ (only lower triangle of A is accessed)
Result: L such that $A = LL^T$ (L overwrites A)

```
b = sqrt(M_1/3)                                   // block size for L_1; assume n is a multiple of b
// A(i,k) refers to b-by-b block of A
for i ← 1 to n/b do
    load A(i,i) (just the lower half) from L_2 to L_1          // #writes to L_1 = .5b^2
                                                  // total #writes to L_1 = (n/b)·.5b^2 = .5nb
    for k ← 1 to i − 1 do
        load A(i,k) from L_2 to L_1                            // #writes to L_1 = b^2
                                                  // total #writes to L_1 ≈ .5(n/b)^2·b^2 = .5n^2
        A(i,i) = A(i,i) − A(i,k) * A(i,k)^T        // b-by-b SYRK (similar to matrix multiplication)
                                                  // no communication between L_1 and L_2
    overwrite A(i,i) by its Cholesky factor        // all done in L_1, no communication
    store A(i,i) (just the lower half) from L_1 to L_2         // #writes to L_2 = .5b^2
                                                  // total #writes to L_2 = (n/b)·.5b^2 = .5nb
    for j ← i + 1 to n/b do
        load A(j,i) from L_2 to L_1                            // #writes to L_1 = b^2
                                                  // total #writes to L_1 ≈ .5(n/b)^2·b^2 = .5n^2
        for k = 1 to i − 1 do
            load A(i,k) from L_2 to L_1                        // #writes to L_1 = b^2
                                                  // total #writes to L_1 ≈ (1/6)(n/b)^3·b^2 = n^3/(6b)
            load A(j,k) from L_2 to L_1                        // #writes to L_1 = b^2
                                                  // total # writes to L_1 ≈ (1/6)(n/b)^3·b^2 = n^3/(6b)
            A(j,i) = A(j,i) − A(j,k) * A(i,k)^T    // b-by-b matrix multiplication
                                                  // no communication between L_1 and L_2
        load A(i,i) (just the lower half) from L_2 to L_1      // #writes to L_1 = .5b^2
                                                  // total #writes to L_1 ≈ .5(n/b)^2·.5b^2 = .25n^2
        solve Tmp * A(i,i)^T = A(j,i) for Tmp; A(j,i) = Tmp    // b-by-b TRSM
                                                  // no communication between L_1 and L_2
        store A(j,i) from L_1 to L_2                           // #writes to L_2 = b^2
                                                  // total #writes to L_2 ≈ .5(n/b)^2·b^2 = .5n^2
```

a force function for a k-tuple of particles, the problem can be formulated as

$$F_i = \Phi_1(P_i) + \sum_{i\neq j} \Phi_2(P_i, P_j) + \sum_{i\neq j\neq m\neq i} \Phi_3(P_i, P_j, P_m) + \cdots$$

Typically, the term N-body refers to just pairwise interactions ($k = 2$) because, possibly except for $k = 1$, they have the most contribution to the total force and are much lower complexity, $O(n^2)$, as opposed to $O(n^k)$. However, there are cases where k-tuple interactions are needed, so we will consider $k > 2$ in this section as well. To avoid confusion, let (N, k)-body denote the problem of computing all k-tuple interactions. The lower bound on the number of reads and writes in a two-level memory model for the (N, k)-body problem is $O(n^k/M^{k-1})$ [38, 15]. This leads to lower bounds on writes for a multiple-level memory hierarchy as in previous sections.

Throughout this section, we will use the particle size as a unit for memory, i.e., L_1 and L_2 can store M_1 and M_2 particles, respectively. We assume that a force is of the same size as a particle.

First we consider the direct $(N, 2)$-body problem. The $\Omega(n^2/M)$ lower bound for this problem can be achieved with the explicitly-blocked Algorithm 4. Two nested loops are required for pairwise interactions. In order to use this code recursively, we express it as taking two input arrays of particles $P^{(1)}$ and $P^{(2)}$, which may be identical, and computing the output forces $F^{(1)}$ on the particles in $P^{(1)}$. For simplicity we assume $\Phi_2(x, x)$ immediately returns 0 for identical input arguments x.

Algorithm 4: Two-Level Blocked Direct $(N, 2)$-body

Data: Input arrays $P_i^{(1)}, P_j^{(2)}$ (possibly identical)

Result: $F_i^{(1)} = \sum_{1 \le j \le N} \Phi_2(P_i^{(1)}, P_j^{(2)})$, for each $1 \le i \le N$

```
1 b = M1/3                                   // block size for L1; assume N is a multiple of b
  // P(1)(i), P(2)(i), and F(1)(i) refer to length-b blocks of P(1), P(2), and F(1)
2 for i ← 1 to N/b do
3     load P(1)(i) from L2 to L1                                   // #writes to L1 = b
                                                 // total #writes to L1 = (N/b) · b = N
4     initialize F(1)(i) to 0 in L1                                // #writes to L1 = b
                                                 // total #writes to L1 = (N/b) · b = N
5     for j ← 1 to N/b do
6         load P(2)(j) from L2 to L1                               // #writes to L1 = b
                                             // total #writes to L1 = (N/b)^2 · b = N^2/b
7         update F(1)(i) with interactions between P(1)(i) and P(2)(j)   // b^2 interactions
                                             // no communication between L1 and L2
8     store F(1)(i) from L1 to L2                                  // #writes to L2 = b
                                                 // total #writes to L2 = (N/b) · b = N
```

As can be seen in Algorithm 4, the number of writes to L_1 attains the lower bound $N^2/b = \Omega(N^2/M_1)$, as does the number of writes to L_2, which is N, the size of the output. To extend to multiple levels of memory, we can replace the line "update $F^{(1)}(i)$" by a call to the same routine, with an appropriate fast memory size. As before, a simple induction shows that this attains the lower bound on writes to all levels of memory.

One classic time-saving technique for the $(N, 2)$-body problem is to utilize force symmetry, i.e., the force from P_i on P_j is equal to the negative of the force from P_j on P_i, which lets us halve the number of interactions. So it is natural to ask if it is possible to attain the lower bound on writes with such an algorithm. It is easy to see that this does not work, because every pass through the inner (j) loop would update forces on all N particles, i.e., N^2 updates altogether, which must generate $O(N^2/b)$ writes to slow memory.

Next we consider the (N, k)-body problem. Analogously to Algorithm 4, we take k arrays as inputs, some of which may be identical, and assume $\Phi_k(P_1, \ldots, P_k)$ returns 0 if any arguments are identical. We now have k nested loops from 1 to $N/b = N/(M/(k+1))$, with loop indices $i_1, \ldots, i_k$. We read a block of b words from $P^{(j)}$ at the beginning of the j-th nested loop, update $F^{(1)}(i_1)$ based on interactions among $P^{(1)}(i_1), \ldots, P^{(k)}(i_k)$ in the innermost loop, and store $F^{(1)}(i_1)$ at the end of the outermost loop. It is easy to see that this does

$$2N + N^2/b + \cdots + N^{k-1}/b^{k-2} + N^k/b^{k-1} = O(N^k/b^{k-1})$$

writes to L_1, and N writes to L_2, attaining the lower bounds on writes. Again, calling itself recursively extends its WA property to multiple levels of memory. Now there is a penalty of a factor of $k!$, in both arithmetic and number of reads, which can be quite large, for not taking advantage of symmetry in the arguments of Φ_k in order to minimize writes.

5 Cache-Oblivious Algorithms Cannot be Write-Avoiding

Following [23] and [11, Section 4], we define a *cache-oblivious (CO)* algorithm as one in which the sequence of instructions executed does not depend on the memory hierarchy of the machine; otherwise it is called *cache-aware*. Here we prove that sequential CO algorithms cannot be WA. This is in contrast to the existence of many CO algorithms that are CA.

As stated, our proof applies to any algorithm to which the lower bounds analysis of [7] applies, so most direct linear algebra algorithms like classical matrix multiplication, other BLAS routines, Cholesky, LU decomposition, etc., for sparse as well as dense matrices, and related algorithms like tensor contractions and Floyd-Warshall all-pairs shortest-paths in a graph. (At the end of the section we suggest extensions to other algorithms.)

For this class of algorithms, given a set S of triples of nonnegative integers (i, j, k), for all triples in S the algorithm reads two array locations $A(i, k)$ and $B(k, j)$ and updates array location $C(i, j)$; for simplicity we call this update operation an "inner loop iteration". This obviously includes dense matrix multiplication, for which $S = \{(i, j, k) : 1 \leq i, j, k \leq n\}$ and $C(i, j) = C(i, j) + A(i, k) * B(k, j)$, but also other algorithms like LU because the matrices A, B, and C may overlap or be identical.

A main tool we need to use is the Loomis-Whitney inequality [40]. Given a fixed number of different entries of A, B, and C that are available (say because they are in fast memory), the Loomis-Whitney inequality bounds the number of inner loop iterations that may be performed: $\#iterations \leq \sqrt{|A| \cdot |B| \cdot |C|}$, where $|A|$ is the number of available entries of A, etc.

Following the argument in [7], and introduced in [36], we consider a program to be a sequence of load (read from slow, write to fast memory), store (read from fast, write to slow memory), and arithmetic/logical instructions. Then assuming fast memory is of size M, we analyze the algorithm as follows:

1. Break the stream of instructions into *segments*, where each segment contains exactly M load and store instructions, as well as the intervening arithmetic/logical instructions. Assuming there are no R2/D2 residencies (see Section 2) then this means that the number of distinct entries available during the segment to perform arithmetic/logical instructions is at most $4M$ (see [7] for further explanation of where the bound $4M$ arises; this includes all the linear algebra and related algorithms mentioned above). We will also assume that no entries of C are discarded, i.e., none are D2.

2. Using Loomis-Whitney and the bound $4M$ on the number of entries of A, B, and C, we bound the maximum number of inner loop iterations that can be performed during a segment by $\sqrt{(4M)^3} = 8M^{3/2}$.

3. Denoting the total number of inner loop iterations by $|S|$, we can bound below the number of complete segments by $s = \lfloor |S|/(8M^{3/2}) \rfloor$.

4. Since each complete segment performs M load and store instructions, the total number of load and store instructions is at least $M \cdot s = M \cdot \lfloor |S|/(8M^{3/2}) \rfloor \geq |S|/(8M^{1/2}) - M$. When $|S| \gg M^{3/2}$, this is close to $|S|/(8M^{1/2}) = \Omega(|S|/M^{1/2})$. We note that the floor function

accommodates the situation where the inputs are all small enough to fit in fast memory at the beginning of the algorithm, and for the output to be left in fast memory at the end of the algorithm, and so no loads or stores are required.

Theorem 3 *Consider an algorithm that satisfies the assumptions presented above. First, suppose that for a particular input $\mathcal{I}$, the algorithm executes the same sequence of instructions, independent of the memory hierarchy. Second, suppose that for the same input $\mathcal{I}$ and fast memory size M, the algorithm is CA in the following sense: the total number of loads and stores it performs is bounded above by $c \cdot |S|/M^{1/2}$ for some constant $c \geq 1/8$. (Note that c cannot be less than $1/8$ by paragraph 4 above.) Then the algorithm cannot be WA in the following sense: When executed using a sufficiently smaller fast memory size $M' < M/(64c^2)$, the number of writes W_s to slow memory is at least*

$$W_s \geq \frac{\lfloor |S|/(8M^{3/2}) \rfloor}{16c-1} \cdot \left(\frac{M}{64c^2} - M' \right) = \Omega\left(\frac{|S|}{M^{1/2}} \right). \tag{1}$$

For example, consider n-by-n dense matrix multiplication, where $|S| = n^3$. A WA algorithm would perform $O(n^2)$ writes to slow memory, but a CO algorithm would perform at least $\Omega(n^3/M^{1/2})$ writes with a smaller cache size M'.

Proof: For a particular input, let s be the number of complete segments in the algorithm. Then by assumption $s \cdot M \leq c|S|/M^{1/2}$. This means the average number of inner loop iterations per segment, $A_{avg} = |S|/s$, is at least $A_{avg} \geq M^{3/2}/c$. By Loomis-Whitney, the maximum number of inner loop iterations per segment is $A_{max} \leq 8M^{3/2}$. Now write $s = s_1 + s_2$ where s_1 is the number of segments containing at least $A_{avg}/2$ inner loop iterations, and s_2 is the number of segments containing less than $A_{avg}/2$ inner loop iterations. Letting a_i be the number of inner loop iterations in segment i, we get

$$A_{avg} = \sum_{i=1}^{s} \frac{a_i}{s} = \frac{\sum_{i:a_i < A_{avg}/2} a_i + \sum_{i:a_i \geq A_{avg}/2} a_i}{s} \leq \frac{s_2 \cdot A_{avg}/2 + s_1 \cdot A_{max}}{s_1 + s_2},$$

or rearranging,

$$s_2 \leq 2\left(\frac{A_{max}}{A_{avg}} - 1 \right) s_1 \leq 2(8c-1)s_1,$$

so $s = s_1 + s_2 \leq (16c-1)s_1$, and we see that $s_1 \geq s/(16c-1)$ segments perform at least $A_{avg}/2 \geq M^{3/2}/(2c)$ inner loop iterations.

Next, since there are at most $4M$ entries of A and B available during any one of these s_1 segments, Loomis-Whitney tells us that the number $|C|$ of entries of C written to slow memory during any one of these segments must satisfy $\frac{M^{3/2}}{2c} \leq (4M \cdot 4M \cdot |C|)^{1/2}$ or $\frac{M}{64c^2} \leq |C|$.

Now consider running the algorithm with a smaller cache size $M' < M/(64c^2)$, and consider what happens during any one of these s_1 segments. Since at least $\frac{M}{64c^2}$ different entries of C are written and none are discarded (D2), at least $\frac{M}{64c^2} - M'$ entries must be written to slow memory during a segment. Thus the total number W_s of writes to slow memory satisfies

$$W_s \geq s_1 \cdot \left(\frac{M}{64c^2} - M' \right) \geq \frac{s}{16c-1} \cdot \left(\frac{M}{64c^2} - M' \right) \geq \frac{\lfloor |S|/(8M^{3/2}) \rfloor}{16c-1} \cdot \left(\frac{M}{64c^2} - M' \right) = \Omega\left(\frac{|S|}{M^{1/2}} \right),$$

as claimed. □

Corollary 4 *Suppose a CO algorithm (assuming the same hypotheses as Theorem 3) is CA for all inputs and fast memory sizes M, in the sense that it performs at most $c \cdot |S|/M^{1/2}$ loads and stores*

for some constant $c \geq 1/8$. Then it cannot be WA in the following sense: for all fast memory sizes M, the algorithm performs

$$W_s \geq \frac{\lfloor |S|/(8(128c^2M)^{3/2}) \rfloor}{16c-1} \cdot M = \Omega\left(\frac{|S|}{M^{1/2}}\right)$$

writes to slow memory.

Proof: For ease of notation, denote the M in the statement of the Corollary by $\hat{M}$. Then apply Theorem 3 with $M' = \hat{M}$, and $M = 128c^2\hat{M}$, so that the lower bound in (1) becomes

$$W_s \geq \frac{\lfloor |S|/(8(128c^2\hat{M})^{3/2}) \rfloor}{16c-1} \cdot \hat{M} = \Omega\left(\frac{|S|}{\hat{M}^{1/2}}\right).$$

□

We note that our proof technique applies more generally to the class of algorithms considered in [15], i.e., algorithms that can be expressed with a set S of tuples of integers, and where there can be arbitrarily many arrays with arbitrarily many subscripts in an inner loop iteration, as long as each subscript is an affine function of the integers in the tuple (pointers are also allowed). The formulation of Theorem 3 will change because the exponents 3/2 and 1/2 will vary depending on the algorithm, and which arrays are read and written.

6 Write-Avoidance in Practice: Hardware Counter Measurements and Cache Replacement Policies

The WA properties of the algorithms described in Section 4 depend on explicitly controlling data movement between caches. However, most instruction sets like x86-64 do not provide the user with an elaborate interface for explicitly managing data movement between caches. Typically, the user is allowed to specify the order of execution of the instructions in the algorithm addressing data by virtual memory address, while the mapping from virtual address to physical location in the caches is determined by hardware cache management (cache replacement and coherence policy). Therefore, an analysis of the behavior of an algorithm on real caches must account for the interaction between the execution order of the instructions and the hardware cache management. Further, the hardware management must be conducive to the properties desired by the user (e.g., minimizing data movement between caches, or in our case, write-backs to the memory).

In this section, we provide hardware counter measurements of caching events for several instruction orders in the classical matrix multiplication algorithm to measure this interaction with a view towards understanding the gap between practice and the theory in previous sections. The instruction orders we report include the cache-oblivious order from [24] and the WA orders for various cache levels from Section 4 and Intel's MKL `dgemm`. Drawing upon this data, we hypothesize and prove that the Least Recently Used (LRU) replacement policy is good at minimizing writes in the two-level write-avoiding algorithms in Section 4.

6.1 Hardware Counter Measurements

Machine and experimental setup. We choose for our measurements the Nehalem-EX microarchitecture based Intel Xeon 7560 processor since its performance counters are well documented (core counters in [34, Table 19-13] and Xeon 7500-series uncore counters in [31]). We program the hardware counters using a customized version of the Intel PCM 2.4 tool [32]. The machine we use has

an L1 cache of size 32KB, an L2 cache of size 256KB per physical core, and an L3 cache of size 24MB (sum of 8 L3 banks on a die) shared among eight cores. All caches are organized into 64-byte cache lines. We use the `hugectl` tool [39] for allocating "huge" virtual memory pages to avoid TLB misses. The Linux kernel version is 3.6.1. We use Intel MKL shipped with Composer XE 2013 (package 2013.3.163). All the experiments are started from a "cold cache" state, i.e., none of the matrix entries are cached. This makes the experiments with smaller instances more predictable and repeatable.

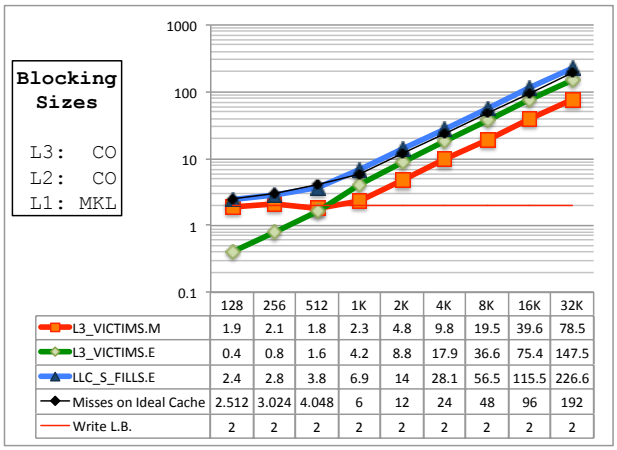
Coherence policy. This architecture uses the MESIF coherence policy [30]. Data read directly from DRAM is cached in the "Exclusive"(E) state. Data that has been modified (written by a processor) is cached locally by a processor in a "Modified" (M) state. All our experiments are single threaded and thus the "Shared" (S) and "Forward" (F) state are less relevant. When a modified cache line is evicted by the replacement policy, its contents are written back to a lower cache level [33, Section 2.1.4], or in the case of L3 cache on this machine, to DRAM.

Counters. To measure the number of writes to slow memory, we measure the evictions of modified cache lines from L3 cache as these are obligatory write-backs to DRAM. We use the C-box event `LLC_VICTIMS.M` [31] for this. We also measure the number of exclusive lines evicted from L3 (which are presumably forgotten) using the event `LLC_VICTIMS.E`. We measure the number of reads from DRAM into L3 cache necessitated by L3 cache misses with the performance event `LLC_S_FILLS.E`.

Replacement Policy. To the best of our knowledge, Intel does not officially state the cache replacement policies used in its microarchitectures. However, it has been speculated and informally acknowledged by Intel [22, 35, 27] that the L3 replacement policy on the Nehalem microarchitecture is the 3-bit LRU-like "clock algorithm" [17]. This algorithm attempts to mimic the LRU policy by maintaining a 3-bit marker to indicate recent usage within an associative set of cache lines. To evict a cache line, the hardware searches for, within the associative set in clockwise order, a line which has not been recently used (marked 000). If such a line is not found, the markers on all cache lines are decremented and the search for a recently unused line is performed again. Cache line hits increase the marker. It has also been speculated [27] that Intel has adopted a new cache replacement policy similar to RRIP [37] from the Ivy Bridge microarchitecture onwards.

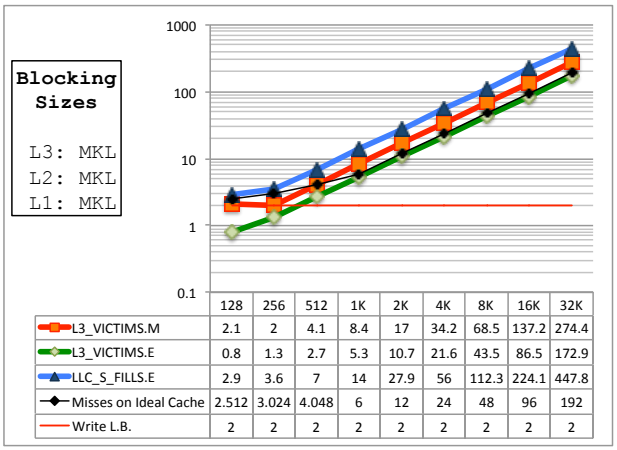
Experimental Data. The experiments in Figure 2 demonstrate the L3 behavior of different versions of the classical double precision matrix multiplication algorithm that computes $C = A * B$. Let the dimensions of A, B and C be l-by-m, m-by-n and l-by-n respectively. Each of the six plots in Figure 2 correspond to instances of dimensions 4000-by-m-by-4000 where m takes the values from the set $\{128, 256, \ldots, 32 \cdot 2^{10}\}$. In each of these instances, the size of the output array C is a constant $4000^2 \cdot 8\,\mathrm{B} = 122.07\,\mathrm{MB}$ which translates to 2.0 million cache lines. This is marked with a horizontal red line in each plot. The arrays A and B vary in size between the instances. All plots show trends in the hardware events `LLC_VICTIMS.M`, `LLC_VICTIMS.E`, and `LLC_S_FILLS.E`, and report measurements in millions of cache lines. In instances where $m \leq 512$, arrays A and B fit in the L3 cache, while in other instances ($m \geq 1024$) they overflow L3 cache.

The first version in Figure 2a is a recursive cache-oblivious version [24] that splits the problem into two along the largest of the three dimensions l, m, and n and recursively computes the two multiplications in sequence. The base case of the recursion fits in the L1 cache and makes a call to



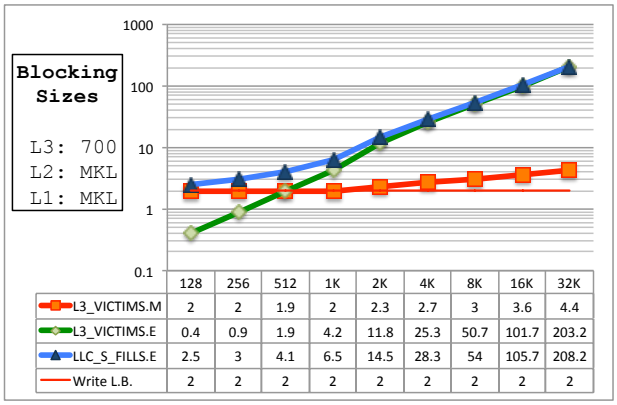
	128	256	512	1K	2K	4K	8K	16K	32K
L3_VICTIMS.M	1.9	2.1	1.8	2.3	4.8	9.8	19.5	39.6	78.5
L3_VICTIMS.E	0.4	0.8	1.6	4.2	8.8	17.9	36.6	75.4	147.5
LLC_S_FILLS.E	2.4	2.8	3.8	6.9	14	28.1	56.5	115.5	226.6
Misses on Ideal Cache	2.512	3.024	4.048	6	12	24	48	96	192
Write L.B.	2	2	2	2	2	2	2	2	2

(a) Cache-oblivious version. The black line indicates estimated number of cache misses on an ideal cache.



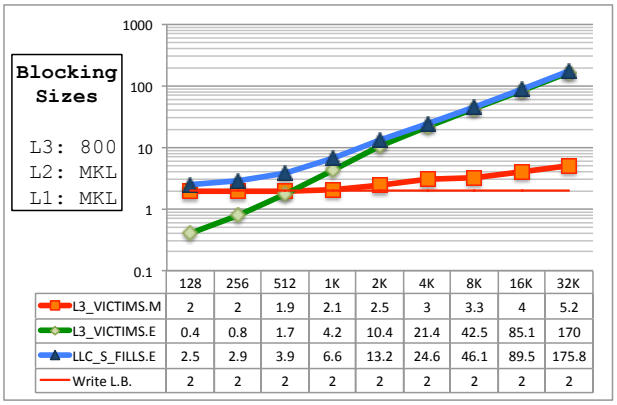
	128	256	512	1K	2K	4K	8K	16K	32K
L3_VICTIMS.M	2.1	2	4.1	8.4	17	34.2	68.5	137.2	274.4
L3_VICTIMS.E	0.8	1.3	2.7	5.3	10.7	21.6	43.5	86.5	172.9
LLC_S_FILLS.E	2.9	3.6	7	14	27.9	56	112.3	224.1	447.8
Misses on Ideal Cache	2.512	3.024	4.048	6	12	24	48	96	192
Write L.B.	2	2	2	2	2	2	2	2	2

(b) Direct call to Intel MKL `dgemm`. The black line is replicated from Figure 2a for comparison.



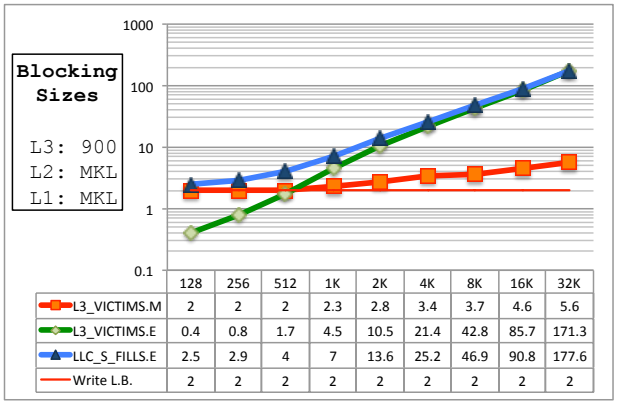
	128	256	512	1K	2K	4K	8K	16K	32K
L3_VICTIMS.M	2	2	1.9	2	2.3	2.7	3	3.6	4.4
L3_VICTIMS.E	0.4	0.9	1.9	4.2	11.8	25.3	50.7	101.7	203.2
LLC_S_FILLS.E	2.5	3	4.1	6.5	14.5	28.3	54	105.7	208.2
Write L.B.	2	2	2	2	2	2	2	2	2

(c) Two-level WA version, L3 blocking size 700.



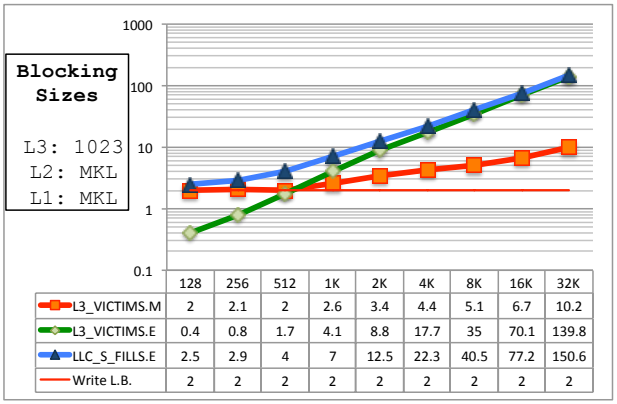
	128	256	512	1K	2K	4K	8K	16K	32K
L3_VICTIMS.M	2	2	1.9	2.1	2.5	3	3.3	4	5.2
L3_VICTIMS.E	0.4	0.8	1.7	4.2	10.4	21.4	42.5	85.1	170
LLC_S_FILLS.E	2.5	2.9	3.9	6.6	13.2	24.6	46.1	89.5	175.8
Write L.B.	2	2	2	2	2	2	2	2	2

(d) Two-level WA version, L3 blocking size 800.



	128	256	512	1K	2K	4K	8K	16K	32K
L3_VICTIMS.M	2	2	2	2.3	2.8	3.4	3.7	4.6	5.6
L3_VICTIMS.E	0.4	0.8	1.7	4.5	10.5	21.4	42.8	85.7	171.3
LLC_S_FILLS.E	2.5	2.9	4	7	13.6	25.2	46.9	90.8	177.6
Write L.B.	2	2	2	2	2	2	2	2	2

(e) Two-level WA version, L3 blocking size 900.



	128	256	512	1K	2K	4K	8K	16K	32K
L3_VICTIMS.M	2	2.1	2	2.6	3.4	4.4	5.1	6.7	10.2
L3_VICTIMS.E	0.4	0.8	1.7	4.1	8.8	17.7	35	70.1	139.8
LLC_S_FILLS.E	2.5	2.9	4	7	12.5	22.3	40.5	77.2	150.6
Write L.B.	2	2	2	2	2	2	2	2	2

(f) Two-level WA version, L3 blocking size 1023.

Figure 2: L3 cache counter measurements of various execution orders of classical matrix multiplication on Intel Xeon 7560. Each plot corresponds to a different variant. All experiments have fixed outer dimensions both equal to 4000. In each plot, the x-axis corresponds to the middle dimension. The y-axis corresponds to various cache events, measured in millions of cache lines. The bottom four plots correspond to variants that attempt to minimize write-backs from L3 to DRAM with various block sizes (hence the label "Two-level WA") but not between L1, L2, and L3.

Intel MKL `dgemm`. This algorithm incurs

$$\left(mn\left\lceil\frac{l}{(M/(3*\text{sz(double)}))^{1/2}}\right\rceil + ln\left\lceil\frac{m}{(M/(3*\text{sz(double)}))^{1/2}}\right\rceil + lm\left\lceil\frac{n}{(M/(3*\text{sz(double)}))^{1/2}}\right\rceil\right) \times \frac{\text{sz(double)}}{L}$$

cache misses (in terms of cache lines) on an ideal cache [24] with an optimal replacement policy, where M is the cache size and L is the cache line size. This is marked with a black line and diamond-shaped markers in the plots in Figures 2a and 2b. The actual number of cache fills (`LLC_S_FILLS.E`) matches the above expression very closely with M set to 24 MB and L set to 64 B. To make way for these, there should be an almost equal number of `LLC_VICTIMS`. Of these, we note that the victims in the E and M state are approximately in a 2:1 ratio when $m > 1024$. This can be explained by the fact that (a) each block that fits in L3 cache has twice as much "read data" (subarrays of A and B) as there is "write data" (subarray C), (b) accesses to A, B, and C are interleaved in a Z-order curve with no special preference for reads or writes, and (c) the replacement policy used on this machine presumably does not distinguish between coherence states. When $m < 1024$, the victims are dominated by "write data" as the output array C is larger than the arrays A and B. This experiment is in agreement with our claim that a cache-oblivious version of the classical matrix multiplication algorithm can not have WA properties.

Figure 2b is a direct call to the Intel MKL `dgemm`. Neither the number of cache misses (as measured by fills) nor the number of writes is close to the minimum possible. We note that MKL is optimized for runtime, not necessarily memory traffic, and indeed is the fastest algorithm of all the ones we tried (sometimes by just a few percent). Since reads and writes to DRAM are very similar in time, minimizing writes to DRAM (as opposed to NVM) is unlikely to result in speedups in the sequential case. Again, the point of these experiments is to evaluate algorithms running with hardware-controlled access to NVM.

Figures 2c-2f correspond to "two-level WA" versions that attempt to minimize the number of write-backs from L3 to DRAM but not between L1, L2, and L3. The problem is divided into blocks that fit into L3 cache (blocking size specified in the plot label) and all the blocks that correspond to a single block of C are executed first using calls to Intel MKL `dgemm` before moving to another block of C. This pattern is similar to the one described in Section 4 except that we are only concerned with minimizing writes from L3 cache to DRAM. It does not control blocking for L2 and L1 caches, leaving this to Intel MKL `dgemm`. With complete control over caching, it is possible to write each element of C to DRAM only once. This would need 2 million write-backs ($4000^2 \cdot \text{sz(double)}/L = 4000^2 \cdot 8/64$) as measured by `LLC_VICTIMS.M`, irrespective of the middle dimension m. We note that the replacement policy does a reasonably good, if not perfect, job of keeping the number of writes to DRAM close to 2 million across blocking sizes. The number of L3 fills `LLC_S_FILLS.E` is accounted for almost entirely by evictions in the E state `LLC_VICTIMS.E`. This is attributable to the fact that in the WA version, the cache lines corresponding to array C are repeatedly touched (specifically, written to) at closer intervals than the accesses to lines with elements from arrays A and B. While a fully associative cache with an ideal replacement policy would have precisely 2 million write-backs for the WA version with a block size of 1024, the LRU-like cache on this machine deviates from this behavior causing more write-backs than the minimum possible. This is attributable to the online replacement policy as well as limited associativity. It is also to be noted that the smaller the block size, the lesser the deviation of the number of write-backs from the lower bound. We well now closely analyze these observations.

6.2 Cache Replacement Policy and Cache Miss Analysis

Background. Most commonly, the cache replacement policy that determines the mapping of virtual memory addresses to cache lines seeks to minimize the number of cache misses incurred by the instructions in their execution order. While the optimal mapping [9] is not always possible to decide in an online setting, the online "least-recently used" (LRU) policy is competitive with the optimal offline policy [42] in terms of the number of cache misses. Sleator and Tarjan [42] show that for any sequence of instructions (memory accesses), the number of cache misses for the LRU policy on a fully associative cache with M cache lines each of size L is within a factor of $(M/(M - M' + 1))$ of that for the optimal offline policy on a fully associative cache with M' lines each of size L, when starting from an empty cache state. This means that a $2M$-size LRU-based cache incurs at most twice as many misses as a cache of size M with optimal replacement. This bound motivates the simplification of the analysis of algorithms on real caches (which is difficult) to an analysis on an "ideal cache model" which uses the optimal offline replacement policy and is fully associative [24]. Analysis of a stream of instructions on a single-level ideal cache model yields a theoretical upper bound on the number of cache misses that will be incurred on a multi-level cache with LRU replacement policy and limited associativity used in many architectures [24]. Therefore, it can be argued that the LRU policy and its theoretical guarantees greatly simplify algorithmic analysis.

LRU and write-avoidance. The LRU policy does not specifically prioritize the minimization of writes to memory. So it is natural to ask if LRU or LRU-like replacement policies can preserve the write-avoiding properties we are looking for. In recent work, Blelloch et al. [12] define "Asymmetric Ideal-Cache" and "Asymmetric External Memory" models which have different costs for cache evictions in the exclusive or modified states. They show [12, Lemma 2.1] that a simple modification of LRU, wherein one half of the cache lines are reserved for reads and the other half for writes, can be competitive with the asymmetric ideal-cache model. While this clean theoretical guarantee greatly simplifies algorithmic analysis, the reservation policy is conservative in terms of cache usage.

We argue that the unmodified LRU policy does in fact preserve WA properties for the algorithms in Section 4, if not for all algorithms, when an appropriate block size is chosen.

Proposition 6.1 *If the two-level WA classical matrix multiplication ($C^{m\times n} = A^{m\times l} * B^{l\times n}$) in Algorithm 1 is executed on a sequential machine with a two-level memory hierarchy, and the block size b is chosen so that five blocks of size b-by-b fit in the fast memory with at least one cache line remaining ($5b^2 * \text{sz(element)} + 1 \leq M$), the number of write-backs to the slow memory caused by the Least Recently Used (LRU) replacement policy running on a fully associative fast memory is mn irrespective of the order of instructions within the call to the multiplication of individual blocks (the call nested inside the loops).*

Proof: Consider a column of blocks corresponding to some C-block that are executed successively. At no point during the execution of this column can an element of this C-block be ranked lower than $5b^2$ in the LRU priority. To see this, consider the center diagram in Figure 3 and the block multiplications corresponding to C-block 7. Suppose that the block multiplications with respect to blocks $\{1, 2\}$ and $\{3, 4\}$ are completed in that order, and the block multiplication with respect to blocks $\{5, 6\}$ is currently being executed. Suppose, if possible, that an element x in the C-block 7 is ranked lower than $5b^2$ in the LRU order. Then, at least one element y from a block other than blocks 3, 4, 5, 6, and 7 must be ranked higher than $5b^2$, and thus, higher than element x. Suppose y is from block 1 or 2 (other cases are similar). The element x has been written to at some point during the multiplication with blocks $\{3, 4\}$, and this necessarily succeeds any reference to blocks 1

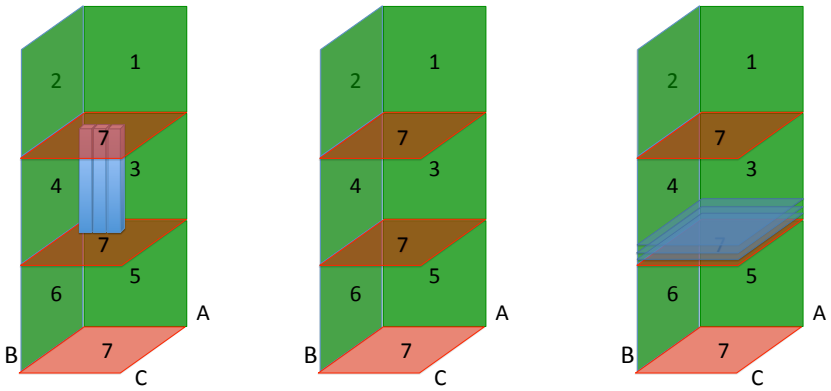


Figure 3: Execution of a column perpendicular to the C-block 7 in classical matrix multiplication. The left and the right diagram correspond to the code in Figures 4a and 4b, respectively. Their corresponding hardware measurements are in the left and right columns of Figure 5.

and 2 since block multiplication with respect to $\{1, 2\}$ is completed before the block multiplication with $\{3, 4\}$, which is a contradiction.

So, once a C-block is loaded into fast memory, it is never evicted until the column perpendicular to it is complete, at which point accesses corresponding to the next column induce an eviction causing a write-back to slow memory. Hence each element of C is written back to slow memory only once. □

This proposition suggests that the LRU policy does very well at avoiding writes for classical matrix multiplication. This claim is validated by plots in Figures 2 and 5 with L3 block size 700 (five blocks of size 793 fit in L3). The number of L3 evictions in the modified state is close to the lower bound for all orderings of instructions within the block multiplication that fits in L3 cache. We speculate that the small gap arises because the cache is not perfect LRU (it is a limited-state approximation described earlier) and not fully associative.

When fewer than five blocks fit in L3, the multi-level WA algorithm in Figure 4a does poorly in conjunction with LRU (see left column of Figure 5). This is because large parts of the C-block currently being used have very low LRU priority and get evicted repeatedly. To see this, consider the left diagram in Figure 3. The block multiplication corresponding to blocks $3, 4$ and 7 is ordered by subcolumns. As a result, at the end of this block multiplication, several subblocks of C-blocks have lower LRU priority than the A- and B-surfaces of recently executed subcolumns. When fewer than five blocks fit in L3, the block multiplication corresponding to input blocks $\{5, 6\}$ and output block 7 forces the eviction of low LRU-priority subblocks of C-block 7 to make space for blocks 5 and 6. The larger the block size, the greater the number of write-backs to DRAM. In fact, when the block size is such that just three blocks fit in L3 (1024 for this machine), a constant fraction of C-block is evicted for each block multiplication. This can been seen in the linear trend in `L3_VICTIMS.M` in the top-left plot in Figure 5. To make LRU work reasonably well at avoiding writes, we are forced to choose a smaller block size than the maximum possible, incurring more cache misses and fills in the exclusive state (notice that, all other parameters fixed, the number of `LLC_S_FILLS.E` and `L3_VICTIMS.E` events is higher for smaller block sizes in the left column of Figure 5).

On the other hand, if we use a WA approach only between L3 and DRAM, these issues can be avoided by executing block multiplications in slabs parallel to the C-block as in the code Figure 4b and illustrated in the right diagram in Figure 3. At the end of each block multiplication, this ordering leaves all elements of the C-blocks at a relatively high LRU priority. Therefore, even when

```
int
round_up (int range, int block_size) {
  if ((range/block_size)*block_size == range)
    return range/block_size;
  else
    return 1 + range/block_size;
}

template<class DT>
void WAMatMul (denseMat<DT> A, denseMat<DT> B, denseMat<DT> C,
               int num_levels, lluint* block_sizes) {
 if (num_levels==1) {
   mklMM(A, B, C, false, false, 1.0, 1.0);
 } else {
   for (int i=0; i<round_up(C.numrows,*block_sizes); ++i)
     for (int k=0; k<round_up(C.numcols,*block_sizes); ++k)
       for (int j=0; j<round_up(A.numcols,*block_sizes); ++j)
       WAMatMul<DT> (A.block(i,j,*block_sizes),
                     B.block(j,k,*block_sizes),
                     C.block(i,k,*block_sizes),
                     num_levels-1,block_sizes+1);
 }
}

lluint block_sizes[4] = {0,1023,100,32};
WAMatMul<double> (A, B, C, 4, block_sizes);
```

(a) Multi-level WA matrix multiplication. Works well when five blocks fit in L3.

```
template<class DT>
void ABMatMul (denseMat<DT> A, denseMat<DT> B, denseMat<DT> C,
               int num_levels, lluint* block_sizes) {
  if (num_levels==1) {
    mklMM(A, B, C, false, false, 1.0, 1.0);
  } else {
    for (int j=0; j<round_up(A.numcols,*block_sizes); ++j)
      for (int i=0; i<round_up(C.numrows,*block_sizes); ++i)
        for (int k=0; k<round_up(C.numcols,*block_sizes); ++k)
          ABMatMul<DT> (A.block(i,j,*block_sizes),
                        B.block(j,k,*block_sizes),
                        C.block(i,k,*block_sizes),
                        num_levels-1,block_sizes+1);
  }
}
template<class DT>
void WAMatMul (denseMat<DT> A, denseMat<DT> B, denseMat<DT> C,
               int num_levels, lluint* block_sizes) {
  for (int i=0; i<round_up(C.numrows,*block_sizes); ++i)
    for (int k=0; k<round_up(C.numcols,*block_sizes); ++k)
      for (int j=0; j<round_up(A.numcols,*block_sizes); ++j)
        ABMatMul<DT> (A.block(i,j,*block_sizes),
                      B.block(j,k,*block_sizes),
                      C.block(i,k,*block_sizes),
                      num_levels-1,block_sizes+1);
}
lluint block_sizes[4] = {0,1023,100,32};
WAMatMul<double> (A, B, C, 4, block_sizes);
```

(b) Two-level WA matrix multiplication. Works well when three blocks fit in L3.

Figure 4: The code to the left (resp., right) corresponds to the left (right) diagram in Figure 3 and the left (right) column of Figure 5 with varying L3 block sizes (set to 1023 in this listing).

the block size is large enough that just under three blocks fit in L3, the C-block is retained in the fast memory by LRU between successive block multiplications. This is validated by the plots in the right column of Figure 5. The number of write-backs is close to the lower bound for all blocking sizes. This allows us to choose a larger block size to minimize the number of cache misses in exclusive state.

This suggests that there is a trade-off between cache misses in the exclusive state and write-backs to DRAM when using LRU replacement policy with the multi-level WA schedule. This may be avoided by taking a hint from the two-level WA version and touching the C-block between successive block multiplications to bump up its LRU priority.

Similar observations apply to the interaction of LRU replacement policy with the WA TRSM, Cholesky factorization and direct N-body algorithms.

Proposition 6.2 *If the two-level WA TRSM (Algorithm 2 with n-by-n-by-m input size), Cholesky factorization (Algorithm 3 with n-by-n input size) and direct N-body algorithm (Algorithm 4 with N input size) are executed on a sequential machine with a two-level memory hierarchy, and the block size b is chosen so that five blocks of size b-by-b fit in fast memory with at least one cache line remaining ($5b^2 * \text{sz(elements)} + 1 \leq M$), the number of write-backs to slow memory caused by the LRU policy running on a fully associative fast memory are nm, $n^2/2$, and N, respectively, irrespective of the order of instructions within the call nested inside the loops.*

The proof follows along the same lines as the proof of Proposition 6.1. As is the case with classical matrix multiplication, certain instruction orders within the nested loop allow larger block sizes to minimize writes with LRU. We leave these details, as well as a study of instruction orders necessary for LRU to provide write-avoiding properties at multiple levels, for future work.

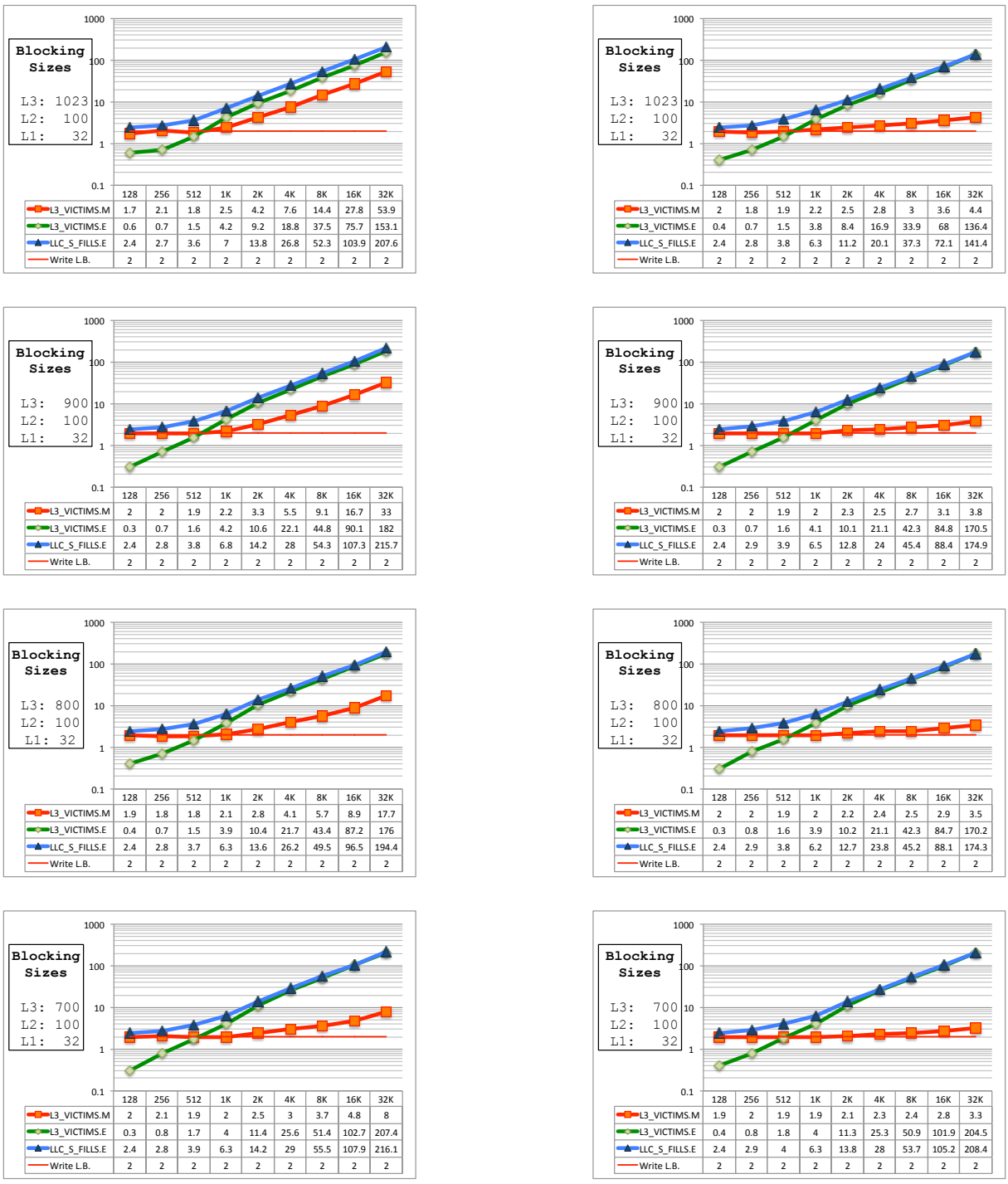


Figure 5: L3 cache counter measurements of various execution orders of classical matrix multiplication on Intel Xeon 7560. Plots on the left correspond to multi-level WA instruction orders with different L3 block sizes (the dimension perpendicular to C array is the outermost loop at each level of recursion). Plots on the right correspond to versions that are blocked to optimize L3 to DRAM write-backs but not L1 to L2 and L2 to L3 write-backs (the dimension perpendicular to C is the outermost loop only at the top level of recursion but not the lower two levels). All experiments have fixed outer dimensions of 4000 each. The x-axis corresponds to the middle dimension. The y-axis corresponds to cache events described in Section 6.1, measured in millions of cache lines. Each plot is labeled with the blocking sizes at the three levels of recursion. L1 and L2 block sizes are such that three blocks fit within the caches with about 5-10% space left as margin.

7 Parallel WA Algorithms

We consider distributed memory parallel algorithms. There is a large literature on CA distributed memory algorithms (see [4, 21, 15, 44] and the references therein). In the model used there, communication is measured by the number of words and messages moved into and out of individual processors' local memories along the critical path of the algorithm (under various assumptions like balanced load and/or balanced memory usage and/or starting with just one copy of the input data). So in this model, a read from one processor's local memory is necessarily a write in another processor's local memory. (The number of reads and writes may differ by a modest factor in the case of broadcasts and reductions, depending on how they are implemented.) In other words, if we are only interested in counting "local memory accesses" without further architectural details, CA and WA are equivalent to within a modest factor.

We may refine this simple architectural model in several ways. First, we assume that the processors are homogeneous, that each one has it own (identical) memory hierarchy. We define three models to analyze:

Model 1: Each processor has a two-level memory hierarchy, labeled L_2 (say DRAM) and L_1 (say cache). Interprocessor communication is between L_2s of different processors. Initially one copy of all input data is stored in a balanced way across the L_2s of all processors.

Model 2: Each processor has a three-level memory hierarchy, labeled L_3 (say NVM), L_2, and L_1. Interprocessor communication is again between L_2s of different processors.

- **Model 2.1:** Initially one copy of all input data is stored in a balanced way across the L_2s of all processors.
- **Model 2.2:** Initially one copy of all input data is stored in a balanced way across the L_3s of all processors. In particular, we assume the input data is too large to fit in all the L_2s.

We let M_1, M_2, and M_3 be the sizes of L_1, L_2, and L_3 on each processor, resp.

For all these models, our goal is to determine lower bounds on communication and identify or invent algorithms that attain them. In particular, we are most concerned with interprocessor communication and writes (and possibly reads) to the lowest level of memory on each processor, since these are the most expensive operations.

First consider Model 1, which is the simplest of the three. Referring back to Section 2, we also have a lower bound on writes to L_2/reads from L_1 equal to the size of the output; assuming memory balance, this is $1/P$-th of the total output for each L_2. The literature cited above includes a variety of lower bounds on interprocessor communication, which (to within a modest factor) means both reads and writes of L_2. Finally, Theorem 1 gives us a lower bound on the number of reads from L_2/writes to L_1.

In particular, consider classical dense linear algebra, including matrix multiplication, TRSM, Cholesky, and similar operations. The output size is $W_1 = n^2/P$, a lower bound on the number of writes to L_2 (on at least one processor in general, or all processors if memory-balanced). The results in [7] provide a lower bound on interprocessor communication of $\Omega(W_2)$ words moved, where $W_2 = n^2/\sqrt{Pc}$, and $1 \le c \le P^{1/3}$ is the number of copies of the input data that the algorithm can make (so also limited by $cn^2/P \le M_2$). This applies to all processors, assuming load balance. (Reference [15] extends this beyond linear algebra.) And [7] together with Theorem 1 tells us that there are $\Omega(W_3)$ reads from L_2/writes to L_1, where $W_3 = (n^3/P)/\sqrt{M_1}$. In general $W_1 \le W_2 \le W_3$, with asymptotically large gaps in their values (i.e., when $n \gg \sqrt{P} \gg 1$).

It is natural to ask whether it is possible to attain all three lower bounds, defined by W_1, W_2 and W_3, by (1) using a CA algorithm to minimize interprocessor communication, and so attaining W_2, and (2) using a WA algorithm for the computations on a single processor, to try to attain W_1 and W_3.

For example, the many CA algorithms for parallel matrix multiplication that have been developed over time, including Cannon's algorithm [13], SUMMA [1, 45], and 2.5D matrix multiplication [20], all perform local matrix multiplications on submatrices, so Algorithm 1 could be used on each processor.

In fact, this does not work. To see why, consider SUMMA: at each of $\sqrt{P}$ steps, $\frac{n}{\sqrt{P}}$-by-$\frac{n}{\sqrt{P}}$ submatrices of A and B are sent to each processor, which multiplies them and adds the product to a locally stored submatrix of C of the same size. Using Algorithm 1, this incurs only $\frac{n^2}{P}$ writes to L_2. But this is repeated $\sqrt{P}$ times, for a total of $\frac{n^2}{\sqrt{P}}$ writes to L_2, which equals W_2 ($c = 1$ for SUMMA), and so asymptotically exceeds the lower bound W_1.

Still, this is likely to be good enough in practice, because the cost (measured in bandwidth or latency) of the words written to L_2 over the network in general greatly exceeds the cost of the words written to L_2 from L_1. So if the number of words written to L_2 from L_1 and from the network are both (roughly) W_2, runtime will still be minimized.

It is actually possible to attain all three lower bounds, at the cost of needing much more memory (see the last paragraph of Section 2.1): One again runs SUMMA, but now one stores all submatrices of A and B sent to each processor in L_2 before doing any computation. In this case, the interprocessor communication W_2 is the same. This requires enough memory in L_2 to store an $\frac{n}{\sqrt{P}}$-by-n submatrix of A and an n-by-$\frac{n}{\sqrt{P}}$ submatrix of B, so $2n^2/\sqrt{P}$ words. Then one multiplies these submatrices by calling Algorithm 1 just once, attaining the lower bound W_1 on writes to L_2 from L_1. As stated in the previous paragraph, this increase in memory size (by a factor of $\sqrt{P}$) is unlikely to result in a significant speedup.

Now consider Model 2.1. We could potentially use the same algorithm that optimized communication in Model 1 and simply ignore L_3. But using L_3 could help, by letting us make more copies of the data (increase c), and so decrease interprocessor communication. But this will come at the cost of writing these copies to L_3. So whether this is worthwhile will depend on the relative costs of interprocessor communication and writing to L_3. In Section 7.1, we will also develop a detailed performance model of this second algorithm for matrix multiplication, called 2.5DMML3, and compare it to the algorithm that just uses L_2, called 2.5DMML2. For simplicity, we compare here just three of the dominant cost terms of these algorithms.

We let β_{NW} be the time per word to send data over the network (i.e., the reciprocal bandwidth), and similarly let β_{23} be the time per word to read data from L_2 and write it to L_3, and let β_{32} be the time per word to read from L_3 and write to L_2; thus we expect $\beta_{23} \gg \beta_{32}$. We also let $1 \leq c_2 < P^{1/3}$ be the number of copies of the data that we have space for in L_2, and $c_2 < c_3 \leq P^{1/3}$ be the (larger) number of copies that we can fit in L_3. With this notation we can say that the dominant bandwidth cost of 2.5DMML2 is

$$dom\beta cost(\text{2.5DMML2}) = \frac{2n^2}{\sqrt{Pc_2}}\beta_{NW}$$

and the dominant bandwidth cost of 2.5DMML3 is

$$dom\beta cost(\text{2.5DMML3}) = \frac{2n^2}{\sqrt{Pc_3}}(\beta_{NW} + 1.5\beta_{23} + \beta_{32})$$

The ratio of these costs is

$$\frac{dom\beta cost(\text{2.5DMML2})}{dom\beta cost(\text{2.5DMML3})} = \sqrt{\frac{c_3}{c_2}}\frac{\beta_{NW}}{\beta_{NW} + 1.5\beta_{23} + \beta_{32}}$$

which makes it simple to predict which is faster, given the algorithm and hardware parameters.

Finally, consider Model 2.2, where the matrices being multiplied are so large that they only fit in L_3 (again, with $1 \le c_3 \le P^{1/3}$ copies). We now determine a slightly different set of lower bounds against which to compare our algorithms. Now $W_1 = n^2/p$ is a lower bound on writes to L_3 (the size of the output, assuming it is balanced across processors). W_2 and W_3 are the same as before. Treating L_2 (and L_1) on one processor as "fast memory" and the local L_3 and all remote memories as "slow memory", [7] and Theorem 1 again give us a lower bound on writes to each L_2 of $\Omega(W_3')$, where $W_3' = (n^3/P)/\sqrt{M_2}$, which could come from L_3 or the network.

We claim that the lower bounds given by W_1 and W_2 cannot be simultaneously attained. presentation below of two algorithms below each of which attains just one of these lower bounds. Both algorithms will also attain both lower bounds W_3 (on reads from L_2/writes to L_1) and W_3' (on writes to L_2 from either L_3 or the network).

Theorem 4 *Assume $n \gg \sqrt{P} \gg 1$ and $n^2/P \gg M_2$. If the number of words written to L_2 from the network is a small fraction of $W_3' = (n^3/P)/\sqrt{M_2}$, in particular if the lower bound $\Omega(W_2)$, where $W_2 = n^2/\sqrt{Pc}$, is attained for some $1 \le c \le P^{1/3}$, then $\Omega(n^2/P^{2/3})$ words must be written to L_3 from L_2. In particular the lower bound $W_1 = n^2/P$ on writes to L_3 from L_2 cannot be attained.*

Proof: The assumptions $n \gg \sqrt{P} \gg 1$ and $n^2/P \gg M_2$ imply $W_1 \ll W_2 \ll W_3'$. If the number of words written to L_2 from the network is a small fraction of W_3', in particular if the W_2 bound is attained, then $\Omega(W_3')$ writes to L_2 must come from reading L_3. By the Loomis-Whitney inequality [40], the number of multiplications performed by a processor satisfies $n^3/P \le \sqrt{\#As \cdot \#Bs \cdot \#Cs}$, where $\#As$ is the number of distinct entries of A available in L_2 sometime during execution (and similarly for $\#Bs$ and $\#Cs$). Thus $n^3/P \le \max(\#As, \#Bs, \#Cs)^{3/2}$ or $n^2/P^{2/3} \le \max(\#As, \#Bs, \#Cs)$. $n^2/P^{2/3}$ is asymptotically larger the amount of data $O(n^2/P)$ originally stored in L_3, so $\Omega(n^2/P^{2/3})$ words must be written to L_3. □

Now we briefly describe the two algorithms that attain all but one of the lower bounds: 2.5DMML3ooL2 ("out of L2") will attain lower bounds given by W_2, W_3, and W_3', and SUMMAL3ooL2 will attain lower bounds given by W_1, W_3, and W_3'. 2.5DMML3ooL2 will basically implement 2.5DMM, moving all the transmitted submatrices to L_3 as they arrive (in sub-submatrices of size M_2). SUMMAL3ooL2 will perform the SUMMA algorithm, computing each $\sqrt{M_2}$-by-$\sqrt{M_2}$ submatrix of C completely in L_2 before writing it to L_3. Using the same notation as above, we can compute the dominant bandwidth costs of these two algorithms as

$$dom\beta cost(\text{2.5DMML3ooL2}) = \frac{\beta_{NW}n^2}{\sqrt{Pc_3}} + \frac{\beta_{23}n^2}{\sqrt{Pc_3}} + \frac{\beta_{32}n^3}{P\sqrt{M_2}} \tag{2}$$

$$dom\beta cost(\text{SUMMAL3ooL2}) = \frac{\beta_{NW}n^3}{P\sqrt{M_2}} + \frac{\beta_{23}n^2}{P} + \frac{\beta_{32}n^3}{P\sqrt{M_2}} \tag{3}$$

which may again be easily compared given the algorithm and hardware parameters.

Section 7.1 describes and analyzes in more detail all the algorithms presented above.

Section 7.2 presents algorithms for LU factorization (without pivoting) whose complexity analyses in Model 2.2 are very similar to 2.5DMML3ooL2 and SUMMAL3ooL2. Since Theorem 4 may be extended to "three-nested loop" algorithms like LU [7], we again do not expect to attain all communication lower bounds simultaneously.

Extending the above analyses to parallel shared memory algorithms is future work.

7.1 Detailed Analysis of Parallel WA Matrix Multiplication Algorithms

Now we consider the costs of WA matrix multiplication to compute $C = A*B$ with n-by-n matrices using P processors in more detail. We will compare the costs of three algorithms which apply to Model 2.1 presented above, i.e., all the data can fit in L_2:

2DMML2: 2D matrix multiplication, so using one copy of all the data, using only L_2;

2.5DMML2: 2.5D matrix multiplication, replicating the data $c_2 > 1$ times, using only L_2;

2.5DMML3: 2.5D matrix multiplication, replicating the data $c_3 > c_2$ times, using L_3.

In all cases, we assume an initial 2D data layout, with one copy of the data stored in L_2, i.e., each processor's L_2 contains an $\frac{n}{\sqrt{P}}$-by-$\frac{n}{\sqrt{P}}$ submatrix of A and B, so $2n^2/P$ words altogether. (For simplicity we assume all quantities like $\sqrt{P}$, $\sqrt{c}$, $\sqrt{P/c}$, $n/\sqrt{P}$, $n/\sqrt{P/c}$, etc., are integers.) This means that 2.5DMML2 and 2.5DMML3 will need an initial step to transform the input data layout to the one assumed by 2.5D matrix multiplication [20].

The 2.5D algorithms then proceed in 4 steps. When we use the notation c in a formula, we will later specialize it to $c = c_2$ for 2.5DMML2 and to $c = c_3$ for 2.5DMML3. The limit on c is $c \leq P^{1/3}$ [6].

1. The processors on the top layer of the $\sqrt{P/c}$-by-$\sqrt{P/c}$-by-c 2.5D processor grid gather all c $\frac{n}{\sqrt{P}}$-by-$\frac{n}{\sqrt{P}}$ submatrices of A and B (stored in L_2s) to build their own $\frac{n}{\sqrt{P/c}}$-by-$\frac{n}{\sqrt{P/c}}$ submatrices of A and B, stored in their L_2s (for 2.5DMML2) or L_3s (for 2.5DMML3). Each gather consists of c messages of size $2n^2/P$, so a total of $2n^2c/P$ words and c messages; we assume that the network topology permits all these gathers to occur simultaneously. Each message has a cost as described above, and so this step costs

$$c_2\alpha_{NW} + \frac{2n^2c_2}{P}\beta_{NW} \tag{4}$$

for 2.5DMML2, and

$$c_3(\alpha_{NW} + \alpha_{23}) + \frac{2n^2c_3}{P}(\beta_{NW} + \beta_{23}) \tag{5}$$

for 2.5DMML3.

At the end of this step, the data is in the format assumed by the 2.5D matrix multiplication algorithm in [20], and stored in L_2 for 2.5DMML2 and in L_3 for 2.5DMML3.

2. We perform step 1 of the 2.5D matrix multiplication algorithm, broadcasting the $2n^2c/P$ words owned by each processor on the top layer of the 2.5D processor grid to the other $c-1$ layers, thus replicating the input data c times. This cannot be done by 2.5DMML3 in a single broadcast because the message size is limited, so we instead do c_3/c_2 broadcasts of size $2n^2c_2/P$, for the same number of words moved, $2n^2c_3/P$, but about c_3/c_2 times the number of messages, $2c_3\log_2(c_3)$. This raises the cost from

$$2\log_2(c_2)\left(\alpha_{NW} + \frac{2n^2c_2}{P}\beta_{NW}\right) \tag{6}$$

for 2.5DMML2 to

$$2\frac{c_3}{c_2}\log_2(c_3)\left(\alpha_{32} + \alpha_{NW} + \alpha_{23} + \frac{2n^2c_2}{P}(\beta_{32} + \beta_{NW} + \beta_{23})\right) \tag{7}$$

for 2.5DMML3. (More efficient broadcast algorithms are known that can reduce some constant or $\log_2 c$ factors, but we consider just the simplest algorithm for clarity.)

3. We perform step 2 of the 2.5D matrix multiplication algorithm, performing $1/c$-th of the steps of SUMMA or Cannon on each of the c layers of the 2.5D processor grid; for simplicity of modeling we consider Cannon. The number of words moved between processors is the same as the original analysis, $2n^2/\sqrt{cP}$. 2.5DMML2 will also send $\sqrt{P/c_2^3}$ messages as in the original analysis, but as above, 2.5DMML3 will send c_3/c_2 times as many messages since the message size is limited by L_2. This means the horizontal communication cost is

$$2\sqrt{\frac{P}{c_2^3}}\alpha_{NW} + \frac{2n^2}{\sqrt{Pc_2}}\beta_{NW} \tag{8}$$

for 2.5DMML2 and

$$2\sqrt{\frac{P}{c_3c_2^2}}(\alpha_{32} + \alpha_{NW} + \alpha_{23}) + \frac{2n^2}{\sqrt{Pc_3}}(\beta_{32} + \beta_{NW} + \beta_{23}) \tag{9}$$

for 2.5DMML3. By using Algorithm 1 for the local matrix multiplications at each step, this raises the vertical communication cost from

$$O\left(\frac{n^3}{P}\left(\frac{\alpha_{21}}{M_1^{3/2}} + \frac{\beta_{21}}{M_1^{1/2}}\right) + \frac{n^2}{\sqrt{Pc_2}}\left(\frac{\alpha_{12}}{M_1} + \beta_{12}\right)\right) \tag{10}$$

for 2.5DMML2 to

$$O\left(\frac{n^3}{P}\left(\frac{\alpha_{21}}{M_1^{3/2}} + \frac{\beta_{21}}{M_1^{1/2}} + \frac{\alpha_{12}}{M_2^{1/2}M_1} + \frac{\beta_{12}}{M_2^{1/2}} + \frac{\alpha_{32}}{M_2^{3/2}} + \frac{\beta_{32}}{M_2^{1/2}}\right) + \frac{n^2}{\sqrt{Pc_3}}\left(\frac{\alpha_{23}}{M_2} + \beta_{23}\right)\right) \tag{11}$$

for 2.5DMML3.

4. Finally, the last step of 2.5D matrix multiplication does a reduction of the c partial sums of C computed on each layer of the 2.5D processor grid, for the same communication costs as the broadcast in step 2 above.

Our goal is to compare the total communication costs of 2DMML2, 2.5DMML2 and 2.5DMML3. The cost of 2DMML2 is given by adding formulas (8) and (10), substituting $c_2 = 1$. The cost of 2.5DMML2 is obtained by adding formulas (4) (twice), (6), (8), and (10). Similarly, the cost of 2.5DMML3 is gotten by adding formulas (5) (twice), (7), (9), and (11).

To simplify the comparison, we will collect all terms in these (long) formulas in Table 1 algorithm, and rows showing the term that is proportional to each hardware parameter (or quotient of hardware parameters). To simplify further, when all the terms in a row have another common factor depending on n and P, that factor is placed in the column labeled "Common Factor".

For example, to get the total cost of 2.5DMML3, for every row without an "NA" in column 5, one takes the product of the factors in columns 2, 3, and 5, and sums these products over all these rows.

Whether 2.5DMML2 or 2.5DMML3 is the better algorithm of course depends on values of all the architectural and problem size parameters and the sum of the many terms in Table 1. some insight as to which algorithm is likely to be faster, Table 1 lets us more easily compare similar terms appearing in the total cost. We note that some formulas have explicit constant factors, and some use asymptotic notation, so some comparisons are best interpreted asymptotically.

$L_2 \rightarrow L_1$ **costs:** These are easily seen to be identical for all three algorithms.

$L_1 \rightarrow L_2$ **costs:** For 2.5DMML2, these costs are asymptotically smaller than the corresponding $L_2 \rightarrow L_1$ costs because of the WA property and also because L_2 is the bottom of the memory hierarchy (and also assuming $\alpha_{12} \approx \alpha_{21}$ and $\beta_{12} \approx \beta_{21}$). For 2.5DMML3, the bandwidth and latency costs are lower than the $L_2 \rightarrow L_1$ costs by a factor $(M_1/M_2)^{1/2}$ (again assuming $\alpha_{12} \approx \alpha_{21}$ and $\beta_{12} \approx \beta_{21}$).

To summarize, $L_2 \rightarrow L_1$ costs are likely to be roughly the same as, or dominate, $L_1 \rightarrow L_2$ costs.

Interprocessor costs: Comparing the first terms in columns 4 and 5 (without a $P^{1/2}$ in the denominator), we see that both latency and bandwidth costs are a factor of $(c_2/c_3)^{1/2}$ smaller for 2.5DMML3 than 2.5DMML2. This is the source of the major potential speedup of 2.5DMML3 over 2.5DMML2.

Comparing the second terms (with $P^{1/2}$) we see that both latency and bandwidth costs are roughly a factor of c_3/c_2 higher for 2.5DMML3 than 2.5DMML2. As long as $c_2 < c_3 \ll P$, we expect the first terms to dominate the second terms.

$L_3 \rightarrow L_2$ **costs:** Consider the first two rows of these costs. As shown, they differ from the network costs only by factors of α_{32}/α_{NW} and β_{32}/β_{NW}, resp. So, if reading L_3 is faster than interprocessor communication, these costs are dominated by network costs.

Now consider the second two rows. They are most easily compared to the corresponding two rows of $L_2 \rightarrow L_1$ costs, and differ by the ratios $\frac{\alpha_{21}/M_1^{3/2}}{\alpha_{32}/M_2^{3/2}}$ and $\frac{\beta_{21}/M_1^{1/2}}{\beta_{32}/M_2^{1/2}}$ resp. So, whether the additional cost of using L_3 is worthwhile may depend on these ratios.

$L_2 \rightarrow L_3$ **costs:** This represents possibly the most expensive communication, writing L_3. The first two rows again differ from the networking costs by factors of roughly α_{23}/α_{NW} and β_{23}/β_{NW}, resp. So if writing L_3 is faster than interprocessor communication, these costs are dominated by network costs.

Now consider the last row. It is most readily compared to the first row of $L_1 \rightarrow L_2$ costs, and differs by the ratio $\frac{\alpha_{23}/(M_2 c_3^{1/2})}{\alpha_{12}/(M_1 c_2^{1/2})}$. On the other hand, the 2.5DMML2 cost in the denominator of this ratio can be much smaller than the first row of $L_2 \rightarrow L_1$ costs. So again, whether the additional cost of using L_3 is worthwhile may depend on these ratios.

Table 1: Communication Costs of Parallel Matrix Multiplication When Data Fits in L_2					
Data Movement	Hardware Parameter	Common Factor	2DMML2 Cost	2.5DMML2 Cost	2.5DMML3 Cost
$L_2 \rightarrow L_1$	$\frac{\alpha_{21}}{M_1^{3/2}}$	$\frac{n^3}{P}$	1	1	1
	$\frac{\beta_{21}}{M_1^{1/2}}$	$\frac{n^3}{P}$	1	1	1
$L_1 \rightarrow L_2$	$\frac{\alpha_{12}}{M_1}$	$\frac{n^2}{P^{1/2}}$	1	$\frac{1}{c_2^{1/2}}$	NA
	β_{12}	$\frac{n^2}{P^{1/2}}$	1	$\frac{1}{c_2^{1/2}}$	NA
	$\frac{\alpha_{12}}{M_2^{1/2} M_1}$	$\frac{n^3}{P}$	NA	NA	1
	$\frac{\beta_{12}}{M_2^{1/2}}$	$\frac{n^3}{P}$	NA	NA	1
Interprocessor	α_{NW}	$2P^{1/2}$	1	$\frac{1}{c_2^{3/2}} + \frac{c_2 + \log c_2}{P^{1/2}}$	$\frac{1}{c_3^{1/2} c_2} + \frac{c_3(1+(\log c_3)/c_2))}{P^{1/2}}$
	β_{NW}	$\frac{2n^2}{P^{1/2}}$	1	$\frac{1}{c_2^{1/2}} + \frac{2c_2(1+\log c_2)}{P^{1/2}}$	$\frac{1}{c_3^{1/2}} + \frac{2c_3(1+\log c_3)}{P^{1/2}}$
$L_3 \rightarrow L_2$	α_{32}	$2P^{1/2}$	NA	NA	same as for $\alpha_{NW} - \frac{c_3}{P^{1/2}}$
	β_{32}	$\frac{2n^2}{P^{1/2}}$	NA	NA	same as for $\beta_{NW} - \frac{2c_3}{P^{1/2}}$
	$\frac{\alpha_{32}}{M_2^{3/2}}$	$\frac{n^3}{P}$	NA	NA	1
	$\frac{\beta_{32}}{M_2^{1/2}}$	$\frac{n^3}{P}$	NA	NA	1
$L_2 \rightarrow L_3$	α_{23}	$2P^{1/2}$	NA	NA	same as for α_{NW}
	β_{23}	$\frac{2n^2}{P^{1/2}}$	NA	NA	same as for $\beta_{NW} + \frac{.5}{c_3^{1/2}}$
	$\frac{\alpha_{23}}{M_2}$	$\frac{n^2}{P^{1/2}}$	NA	NA	$\frac{1}{c_3^{1/2}}$

Next, we consider the case where the data is initially stored in the union of all the L_3s, and is too large to fit in the L_2s. We will see that there are two kinds of "optimal" algorithms: (1) 2.5DMML3ooL2 (short for "out of L_2"), which is very similar to 2.5DMML3 and minimizes interprocessor communication but not writes to L_3, and (2) SUMMAL3ooL2, which minimizes writes to L_3 (i.e., n^2/P) but not interprocessor communication.

2.5DMML3ooL2 performs the same 4 steps as 2.5DMML2 and 2.5DMML3:

1. The processors on the top layer of a $\sqrt{P/c_3}$-by-$\sqrt{P/c_3}$-by-c_3 2.5D processor grid gather all c_3 $\frac{n}{\sqrt{P}}$-by-$\frac{n}{\sqrt{P}}$ submatrices of A and B (stored in L_3s) to build their own $\frac{n}{\sqrt{P/c_3}}$-by-$\frac{n}{\sqrt{P/c_3}}$ submatrices of A and B, stored in L_3. Analogously to the previous analysis, the cost is

$$\frac{2n^2 c_3}{P}\left(\beta_{32} + \beta_{NW} + \beta_{23} + \frac{\alpha_{32}}{M_2} + \frac{\alpha_{NW}}{M_2} + \frac{\alpha_{23}}{M_2}\right) \tag{12}$$

2. We perform step 1 of the 2.5D matrix multiplication algorithm, broadcasting the $2n^2c_3/P$ words owned by each processor on the top layer of the 2.5D processor grid to the other $c_3 - 1$ layers. The cost is

$$\frac{4n^2 c_3 \log_2(c_3)}{P}\left(\beta_{32} + \beta_{NW} + \beta_{23} + \frac{\alpha_{32}}{M_2} + \frac{\alpha_{NW}}{M_2} + \frac{\alpha_{23}}{M_2}\right) \tag{13}$$

3. We perform step 2 of the 2.5D matrix multiplication algorithm, performing $1/c_3$-th of the steps of SUMMA or Cannon on each of the c_3 layers of the 2.5D processor grid. The horizontal communication costs are

$$\frac{2n^2}{\sqrt{Pc_3}}\left(\beta_{32}+\beta_{NW}+\beta_{23}+\frac{\alpha_{32}}{M_2}+\frac{\alpha_{NW}}{M_2}+\frac{\alpha_{23}}{M_2}\right) \tag{14}$$

and the vertical communication costs are the same as in (11),

$$O\left(\frac{n^3}{P}\left(\frac{\alpha_{21}}{M_1^{3/2}}+\frac{\beta_{21}}{M_1^{1/2}}+\frac{\alpha_{12}}{M_2^{1/2}M_1}+\frac{\beta_{12}}{M_2^{1/2}}+\frac{\alpha_{32}}{M_2^{3/2}}+\frac{\beta_{32}}{M_2^{1/2}}\right)+\frac{n^2}{\sqrt{Pc_3}}\left(\frac{\alpha_{23}}{M_2}+\beta_{23}\right)\right). \tag{15}$$

4. Finally, the last step of 2.5D matrix multiplication does a reduction of the c_3 partial sums of C computed on each layer of the 2.5D processor grid, for the same communication cost as the broadcast in step 2 above.

Thus the total cost of 2.5DMML3ooL2 is the sum of the costs in expressions (12), (13) (twice), (14) and (15).

Finally, we present and analyze SUMMAL3ooL2. whose goal is to minimize writes to L_3 at the cost of more interprocessor communication. We do this by two levels of blocking of the output matrix C: each processor will store a $\frac{n}{\sqrt{P}}$-by-$\frac{n}{\sqrt{P}}$ block of C. Each such block will in turn be blocked into $\sqrt{M_2/3}$-by-$\sqrt{M_2/3}$ subblocks; let $C_p(i,j)$ denote the (i,j)-th $\sqrt{M_2/3}$-by-$\sqrt{M_2/3}$ subblock owned by processor p. For each (i,j), with $1 \le i,j \le n\sqrt{3/(PM_2)}$, all $C_p(i,j)$ will be computed by using SUMMA to multiply a $\sqrt{PM_2/3}$-by-n submatrix of A by a n-by-$\sqrt{PM_2/3}$ submatrix of B. The block size used by SUMMA will be $\sqrt{M_2/3}$, so that all three $\sqrt{M_2/3}$-by-$\sqrt{M_2/3}$ submatrices of A, B, and C accessed by SUMMA fit in L_2. After $C_p(i,j)$ is computed, it is written to L_3 just once, minimizing writes to L_3. The cost of one step of SUMMA, i.e., to read $\sqrt{M_2/3}$-by-$\sqrt{M_2/3}$ subblocks of A and B from L_3 to L_2, to broadcast them along rows and columns of $\sqrt{P}$ processors, and multiply them, is

$$M_2\beta_{32}+\alpha_{32}+M_2\beta_{NW}+\log_2(P)\alpha_{NW}+\left(\frac{M_2}{3}\right)^{3/2}\left(\frac{\beta_{21}}{M_1^{1/2}}+\frac{\alpha_{21}}{M_1^{3/2}}\right)+\frac{M_2}{3}\left(\beta_{12}+\frac{\alpha_{12}}{M_1}\right). \tag{16}$$

Each call to SUMMA takes $n/\sqrt{M_2/3}$ such steps, and there are $3n^2/(PM_2)$ calls to SUMMA. Multiplying these factors by the cost in (16) and adding the cost of writing C to L_3 yields the total cost,

$$\begin{aligned}\frac{n^3}{P}\frac{3^{3/2}}{M_2^{1/2}}&\left(\beta_{32}+\beta_{NW}+\frac{\alpha_{32}}{M_2}+\frac{\log_2(P)\alpha_{NW}}{M_2}\right)\\ &+\frac{n^3}{P}\left(\frac{\beta_{21}}{M_1^{1/2}}+\frac{\alpha_{21}}{M_1^{3/2}}+\frac{\beta_{12}}{(M_2/3)^{1/2}}+\frac{\alpha_{12}}{(M_2/3)^{1/2}M_1}\right)+\frac{n^2}{P}\left(\beta_{23}+\frac{\alpha_{23}}{M_2}\right).\end{aligned} \tag{17}$$

To compare the costs of 2.5DMML3ooL2 and SUMMAL3ooL2, we again collect all the costs in Table 2, in the same format as Table 1. Since some formulas have explicit constant factors and some use asymptotic notation, we omit all constant factors and interpret comparisons asymptotically.

As before, whether 2.5DMML2ooL2 or SUMMAL3ooL2 is faster depends on the values of all architectural and problem size parameters, but Table 2 lets us compare similar cost terms.

$L_2 \to L_1$ **costs:** These are identical to one another, and to the algorithms in Table 1.

$L_1 \to L_2$ **costs:** These are identical to one another, and to 2.5DMML3.

Interprocessor costs: The costs for SUMMAL3ooL2 are higher by a factor of $n(c_3/(PM_2))^{1/2}$ than for 2.5DMML3ooL2, which attains the lower bound.

$L_3 \to L_2$ **costs:** The dominant costs terms are identical, and attain the lower bound.

$L_2 \to L_3$ **costs:** The costs for 2.5DMML3ooL2 are higher by a factor of $(P/c_3)^{1/2}$ than for SUMMAL3ooL2, which attains the lower bound.

Table 2: Communication Costs of Parallel Matrix Multiplication When Data Does Not Fit in L_2

Data Movement	Hardware Parameter	Common Factor	2.5DMML3ooL2 Cost	SUMMAL3ooL2 Cost
$L_2 \to L_1$	$\frac{\alpha_{21}}{M_1^{3/2}}$	$\frac{n^3}{P}$	1	1
	$\frac{\beta_{21}}{M_1^{1/2}}$	$\frac{n^3}{P}$	1	1
$L_1 \to L_2$	$\frac{\alpha_{12}}{M_2^{1/2}M_1}$	$\frac{n^3}{P}$	1	1
	$\frac{\beta_{12}}{M_2^{1/2}}$	$\frac{n^3}{P}$	1	1
Interprocessor	$\frac{\alpha_{NW}}{M_2}$	$\frac{n^2}{P^{1/2}}$	$\frac{1}{c_3^{1/2}} + \frac{c_3(1+\log c_3)}{P^{1/2}}$	$\frac{n \log P}{(PM_2)^{1/2}}$
	β_{NW}	$\frac{n^2}{P^{1/2}}$	$\frac{1}{c_3^{1/2}} + \frac{c_3(1+\log c_3)}{P^{1/2}}$	$\frac{n}{(PM_2)^{1/2}}$
$L_3 \to L_2$	$\frac{\alpha_{32}}{M_2}$	$\frac{n^2}{P^{1/2}}$	$\frac{n}{(PM_2)^{1/2}} + \frac{1}{c_3^{1/2}} + \frac{c_3(1+\log c_3)}{P^{1/2}}$	$\frac{n}{(PM_2)^{1/2}}$
	β_{32}	$\frac{n^2}{P^{1/2}}$	$\frac{n}{(PM_2)^{1/2}} + \frac{1}{c_3^{1/2}} + \frac{c_3(1+\log c_3)}{P^{1/2}}$	$\frac{n}{(PM_2)^{1/2}}$
$L_2 \to L_3$	$\frac{\alpha_{23}}{M_2}$	$\frac{n^2}{P}$	$(\frac{P}{c_3})^{1/2} + c_3(1 + \log c_3)$	1
	β_{23}	$\frac{n^2}{P}$	$(\frac{P}{c_3})^{1/2} + c_3(1 + \log c_3)$	1

7.2 Parallel WA LU Factorization

In this section we present a parallel WA algorithm for the LU factorization, without pivoting, of a dense matrix of size n-by-n. We use Model 2.2 from the beginning of Section 7: given P processors, we consider the case in which each processor has three levels of memory, L_3 (e.g., NVM), L_2, and L_1, and interprocessor communication is between L_2s of different processors. The sizes of these memories are M_3, M_2, and M_1, respectively. We focus on the case in which the size of L_2 is smaller than n^2/P, and we assume that the data is distributed evenly over the L_3s of the P processors.

We discuss two algorithms. The first algorithm (LL-LUNP, Algorithm 5) uses a left-looking approach and allows us to minimize the number of writes to L_3, but increases the interprocessor communication. The second algorithm (RL-LUNP, described later in this section) is based on a right-looking approach and corresponds to CALU [25], and allows us to minimize the interprocessor communication, but does not attain the lower bound on number of writes to L_3. We focus here on LU without pivoting. However, the same approach can be used for Cholesky, LU with tournament pivoting, or QR with a TSQR-based panel factorization; more details on tournament pivoting or TSQR can be found in [25, 19]. Here we consider only 2D algorithms, in which only one copy of the data is stored in L_3; however, we expect that the algorithms can be extended to 2.5D

or 3D approaches, in which c copies of the data are stored in L_3. Using the same notation as previously in Section 7, and given that $n^2/P \gg M_2$ ($n^2/\sqrt{P}$ is a lower-order term with respect to $(n^3 \log^2 P)/(PM_2^{1/2})$), the dominant bandwidth costs of the two algorithms are

$$\begin{aligned} dom\beta cost(\text{LL-LUNP}) &= O\left(\frac{n^3}{PM_2^{1/2}} \log_2^2 P\right) \beta_{NW} + O\left(\frac{n^2}{P}\right) \beta_{23} + O\left(\frac{n^3}{PM_2^{1/2}} \log_2^2 P\right) \beta_{32}, \\ dom\beta cost(\text{RL-LUNP}) &= O\left(\frac{n^2}{\sqrt{P}} \log_2 \sqrt{P}\right) \beta_{NW} + O\left(\frac{n^2}{\sqrt{P}} \log_2^2 P\right) \beta_{23} + O\left(\frac{n^3}{PM_2^{1/2}}\right) \beta_{32}. \end{aligned}$$

Note that these formulas are very similar to $dom\beta cost$(SUMMAL3ooL2) in (3) and $dom\beta cost$(2.5DMML3ooL2) in (2), modulo $\log_2 P$ factors.

LL-LUNP (Algorithm 5) computes in parallel the LU factorization (without pivoting) of a matrix A of size n-by-n by using a left-looking approach. For ease of analysis, we consider square matrices distributed block-cyclically over a $\sqrt{P}$-by-$\sqrt{P}$ grid of processors. The matrix A is partitioned as

$$A = \begin{pmatrix} A_{11} & \dots & A_{1,n/B} \\ \vdots & \ddots & \dots \\ A_{n/B,1} & \dots & A_{n/B,n/B} \end{pmatrix},$$

where each block $A_{I,J}$ of size B-by-B is distributed over the square grid of processors by using blocks of size b-by-b, $B = b\sqrt{P}$. The block size b is chosen such that $M_2 = sb^2$, where M_2 is the size of DRAM. The choice of s will be explained later. Given a B-by-B block $A_{I,J}$, we refer to the b-by-b subblock stored on a given processor p as $A_{I,J}(r,c)$, where (r,c) are the coordinates of p in the process grid.

Algorithm 5 computes the LU factorization of the matrix A by iterating over blocks of B columns. For each block column I, it first updates this block column with (already computed) data from all the block columns to its left. Then it computes the LU factorization of the diagonal block A_{II}, followed by the computation of the off-diagonal blocks of the I-th block column of L.

We detail now the cost of each of these steps, by analyzing first the number of floating point operations ($\#flops$), and then the interprocessor communication in terms of both communication volume and number of messages.

All the processors participate in the update of each block column I by the block columns to its left, and this is where the bulk of the computation occurs. This update is performed by iterating over the blocks of block column I, and each block $A_{J,I}$ is updated by all the blocks $L_{J,1:\min(I,J)-1}$ to its left and the corresponding blocks $U_{1:\min(I,J)-1,I}$. The overall cost of these updates is

$$\#flops = \sum_{I=1}^{n/B} \left(\sum_{J=1}^{I-1} \sum_{K=1}^{J-1} 2b^3\sqrt{P} + \sum_{J=I}^{n/B} \sum_{K=1}^{I-1} 2b^3\sqrt{P} + 3b^3\sqrt{P}(I-1) \right) = \frac{2}{3}\frac{n^3}{P} + \frac{3}{2}\frac{n^2 b}{\sqrt{P}} \tag{18}$$

and the interprocessor communication is

$$\frac{n^3\sqrt{s}}{PM_2^{3/2}}\alpha_{NW} + \left(\frac{n^3 s^{1/2}}{PM_2^{1/2}} + \frac{n^2}{\sqrt{P}}\right)\beta_{NW}. \tag{19}$$

The cost of the factorization of each block column of the L factor is

$$\#flops = \sum_{I=1}^{n/B} 3b^3\sqrt{P}(\frac{n}{B} - I) + \sum_{I=1}^{n/B} \sqrt{P}\left(\frac{2}{3}b^3 + b^3 + 2b^3\right) = \frac{3}{2}\frac{n^2 b}{\sqrt{P}} + \frac{11}{3}nb^2 \tag{20}$$

Algorithm 5: Parallel Left-Looking LU without pivoting (LL-LUNP)

Data: $A^{n\times n}$

Result: L and U such that $A^{n\times n} = L^{n\times n} * U^{n\times n}$

// Assume: $B = b\sqrt{P}$, $M_2 = (2s+1)b^2$, n is a multiple of Bs

for $I = 1$ **to** n/B **do**

  for $J = 1$ **to** n/B **do**

    /* All processors compute $A_{J,I} = A_{J,I} - \sum_{K=1}^{\min(I,J)-1} L_{J,K} * U_{K,I}$ */

    for $K = 1$ **to** $\min(I,J) - 1$ *step* s **do**

      for $g = 1$ **to** $\sqrt{P}$ **do**

        Processors in the g-th column broadcast $L_{J,K:K+s}(:, g)$ along rows

        Processors in the g-th row broadcast $U_{K:K+s,I}(g, :)$ along columns

        All processors compute $A_{J,I}(r,c) = A_{J,I}(r,c) - \sum_{K=1}^{min(I,J)-1} L_{J,K}(r,g) * U_{K,I}(c,g)$, $r, c = 1 : \sqrt{P}$

    if $J < I$ **then**

      /* Compute $U_{J,I} = L_{J,J}^{-1} * A_{J,I}$ */

      for $g = 1$ **to** $\sqrt{P}$ **do**

        Processors in the g-th column broadcast $L_{J,J}(:, g)$ along rows

        Processors in the g-th row compute $U_{J,I}(g,c) = L_{J,J}^{-1}(g,g) * A_{J,I}(g,c)$, $c = 1 : \sqrt{P}$, and broadcast $U_{J,I}(g,c)$, $c = 1 : \sqrt{P}$, along columns

        Processors in rows $r = g+1 : \sqrt{P}$ and columns $c = 1 : \sqrt{P}$ compute $A_{J,I}(r,c) = A_{J,I}(r,c) - L_{J,J}(r,g) * A_{J,I}(g,c)$, $r = g+1 : \sqrt{P}, c = 1 : \sqrt{P}$

  /* All processors compute $L_{I,I}$, the diagonal block of the L factor */

  for $g = 1$**to** $\sqrt{P}$ **do**

    Processor with coordinates (g,g) computes $A_{I,I}(g,g) = L_{I,I}(g,g) * U(I,I)(g,g)$

    Processor with coordinates (g,g) broadcasts $U(I,I)(g,g)$ along column g and $L(I,I)(g,g)$ along row g

    Processors in the g-th column compute $L_{I,I}(r,g) = A_{I,I}(r,g) * U_{I,I}(g,g)^{-1}$, $r = g+1 : \sqrt{P}$, and broadcast $L_{I,I}(r,g)$, $r = g+1 : \sqrt{P}$, along rows

    Processors in the g-th row compute $U_{I,I}(g,c) = L_{I,I}(g,g)^{-1} * A_{I,I}(g,c)$, $c = g+1 : \sqrt{P}$, and broadcast $U_{I,I}(g,c)$, $c = g+1 : \sqrt{P}$ along columns

    Processors in rows and columns $g+1 : \sqrt{P}$ compute $A_{I,I}(r,c) = A_{I,I}(r,c) - L_{I,I}(r,g) * U_{I,I}(g,c)$, $r, c = g+1 : \sqrt{P}$

  /* All processors compute $L_{I+1:n/B,I}$, the off-diagonal blocks of the L factor */

  for $J = I+1$ **to** n/B *step* s **do**

    for $g = 1$ **to** $\sqrt{P}$ **do**

      Processors in the g-th column compute $L_{J:J+s,I}(r,g) = A_{J:J+s,I}(r,g) * U_{I,I}(g,g)^{-1}$, $r = 1 : \sqrt{P}$, and broadcast $L_{J:J+s,I}(r,g)$, $r = 1 : \sqrt{P}$, along rows

      Processors in row g and columns $c = g+1 : \sqrt{P}$ broadcast $U_{I,I}(g,c)$ along columns

      Processors in rows/columns $g+1 : \sqrt{P}$ compute $A_{J:J+s,I}(r,c) = A_{J:J+s,I}(r,c) - L_{J:J+s,I}(r,g) * U_{I,I}(g,c)$, $r, c = g+1 : \sqrt{P}$

while the interprocessor communication is (ignoring lower-order terms)

$$\left(\frac{4n^2\log_2\sqrt{P}}{M_2\sqrt{P}}+4\frac{n}{b}\log_2\sqrt{P}\right)\alpha_{NW}+\frac{n^2}{\sqrt{P}}\beta_{NW}. \tag{21}$$

The overall cost in terms of flops of the parallel left looking factorization presented in Algorithm 5 is the following (in which we ignore several lower-order terms):

$$\#flops \;=\; \frac{2}{3}\frac{n^3}{P}+3\frac{n^2b}{\sqrt{P}}+\frac{11}{3}nb^2. \tag{22}$$

To keep $3\frac{n^2b}{\sqrt{P}}+\frac{11}{3}nb^2$ as lower-order terms in equation (22), b needs to be smaller than $n/(\sqrt{P}\cdot o(\log_2 P))$, for simplicity we impose $b < n/(\sqrt{P}\cdot\log_2^2 P)$. Hence, if $M_2/3 \le n^2/(P\cdot\log_2^4 P)$, then $s=1$ and $b=(M_2/3)^{1/2}$. Otherwise, $b=n/(\sqrt{P}\cdot\log_2^2 P))$, and $s=(M_2-b^2)/2$. Hence $s<\log_2^4 P$. We obtain the following overall interprocessor communication for LL-LUNP,

$$\left(\frac{n^3}{PM_2^{3/2}}\log_2^2 P+4\frac{n}{b}\log_2\sqrt{P}+\frac{4n^2\log_2\sqrt{P}}{M_2\sqrt{P}}\right)\alpha_{NW}+\left(\frac{n^3}{PM_2^{1/2}}\log_2^2 P+\frac{3}{2}\frac{n^2}{\sqrt{P}}\right)\beta_{NW}. \tag{23}$$

In terms of vertical data movement, the number of reads from L_3 to L_2 is the same as the volume of interprocessor communication given in (23). Each block of size b-by-b is written at most twice back to L_3. We obtain

$$\frac{2n^2}{P}\beta_{23}+\left(\frac{n^3}{PM_2^{1/2}}\log_2^2 P+\frac{3}{2}\frac{n^2}{\sqrt{P}}\right)\beta_{32}. \tag{24}$$

We discuss now a parallel right-looking algorithm (RL-LUNP) for computing the LU factorization (without pivoting) of a dense n-by-n matrix. The matrix is distributed over a $\sqrt{P}$-by-$\sqrt{P}$ grid of processors, using a block-cyclic distribution with blocks of size b-by-b. The algorithm iterates over panels of b columns and at each step i, it computes the LU factorization of the i-th panel and then it updates the trailing matrix. A more detailed description can be found in [25] for the case in which tournament pivoting is used. The interprocessor communication volume of this algorithm is $O\left(nb\log_2\sqrt{P}+\frac{n^2}{\sqrt{P}}\log_2\sqrt{P}\right)\beta_{NW}$.

We let $b=\frac{n}{\sqrt{P}\log_2^2 P}$ and the right-looking algorithm iterates over $\sqrt{P}\log_2^2 P$ block columns. As noted in [25], with this choice of b and by ignoring lower-order terms, RL-LUNP performs $2n^3/(3P)$ flops. At each iteration, a subset of processors from one row and one column in the process grid compute the factorization of the current block column of L and block row of U. The communication involved during this step is at most equal to the communication performed during the following steps and we ignore it here. After this step, the processors need to send $nb/\sqrt{P}=n^2/(P\log_2^2 P)$ words to processors in the same row or column of the process grid. We assume that the matrix is stored by using blocks of size $\sqrt{M_2}$-by-$\sqrt{M_2}$. Since $n^2/P \gg M_2$, multiple messages, each of size M_2, are used by each processor to send $n^2/(P\log_2^2 P)$ words. For each message, a processor reads from L_3 to L_2 a block of size $\sqrt{M_2}$-by-$\sqrt{M_2}$, it broadcasts this block to other processors, and then the receiving processors write from L_2 to L_3 the block. By summing over all iterations, the number of messages and the volume of interprocessor communication is

$$O\left(\frac{n^2}{\sqrt{P}M_2}\log_2\sqrt{P}\right)\alpha_{NW}+O\left(\frac{n^2}{\sqrt{P}}\log_2\sqrt{P}\right)\beta_{NW}, \tag{25}$$

and so RL-LUNP attains the lower bounds (modulo logarithmic factors) on interprocessor communication.

After this communication, each processor updates the trailing matrix by multiplying, in sequence, matrices of size b-by-b which are stored in L_3. By using the WA version of matrix multiplication from Section 4, which minimizes the number of writes to L_3, the update of each block of size b-by-b leads to $O(b^3/M_2^{1/2})$ reads from L_3 to L_2 and b^2 writes from L_2 to L_3. There are at most $(n/(b\sqrt{P}))^2$ blocks to be updated at each iteration, and there are $\sqrt{P}\log_2^2 P$ iterations. Overall, the data movement between L_2 and L_3 which occurs either during interprocessor communication or computation on local matrices, is bounded above by

$$O\left(\frac{n^2}{b^2P}b^2\sqrt{P}\log_2^2 P\right)\beta_{23}+O\left(\frac{n^2}{b^2P}\frac{b^3}{M_2^{1/2}}\sqrt{P}\log_2^2 P\right)\beta_{32} = O\left(\frac{n^2}{\sqrt{P}}\log_2^2 P\right)\beta_{23}+O\left(\frac{n^3}{PM_2^{1/2}}\right)\beta_{32}. \tag{26}$$

8 Krylov Subspace Methods

Krylov subspace methods (KSMs) are a family of algorithms to solve linear systems and eigenproblems. One key feature of many KSMs is a small memory footprint. For example, the conjugate gradient (CG) method for solving a symmetric positive-definite (SPD) linear system $Ax = b$, shown in Algorithm 6, can be implemented with four n-vectors of storage (x, p, r, and w) in addition to

Algorithm 6: Conjugate Gradient (CG)

Data: $n \times n$ SPD matrix A, right-hand side b, initial guess x_0
Result: Approximate solution x to $Ax = b$

$p = r = b - Ax_0$, $\delta_{\text{prv}} = r^T r$
for $j = 1, 2, \ldots$ *until convergence,* **do**
  $\alpha = \delta_{\text{prv}}/(p^T w)$
  $x = x + \alpha p$
  $r = r - \alpha w$
  $\delta_{\text{cur}} = r^T r$
  $\beta = \delta_{\text{cur}}/\delta_{\text{prv}}$
  $p = r + \beta p$

the system matrix and a few scalars. In this case, each iteration j incurs at least $4n - M_1$ writes to L_2; if n is sufficiently large with respect to M_1 and N iterations are performed, $W_{12} = \Omega(N \cdot n)$.

We now show how to use an s-step CG variant called CA-CG to obtain $W_{12} = O(N \cdot n/s)$, where the integer parameter s is bounded by a function of n, M_1, and the nonzero structure of A. The trick to exploiting temporal locality is to break the cyclic dependence $p \to w \to r \to p$ (see Carson et al. [14] for more background). Suppose $\rho = (\rho_0, \rho_1, \ldots)$ is a sequence of polynomials satisfying an m-term recurrence for some $m \in \{1, 2, \ldots\}$ where ρ_j has degree j for each $j \in \{0, 1, \ldots\}$. Define

$$K_m(A, \rho, y) = [\rho_0(A)y, \ldots, \rho_{m-1}(A)y],$$

and note that there exists an $(m+1)$-by-m upper Hessenberg matrix H, whose entries are given by the recurrence coefficients defining ρ, such that

$$A \cdot K_{m-1}(A, \rho, y) = K_m(A, \rho, y) \cdot H.$$

Constructing such a recurrence every s iterations allows us to compute the coordinates of the CG vectors in a different basis. This leads to the CA-CG method, shown in Algorithm 7.

Algorithm 7: Communication-Avoiding Conjugate Gradient (CA-CG)

Data: $n \times n$ SPD matrix A, right-hand side b, initial guess x_0, $s \in \{1, 2, \ldots\}$, and sequence ρ of polynomials satisfying an s-term recurrence.

Result: Approximate solution x to $Ax = b$

1 $p = r = b - Ax_0$, $\delta_{\text{prv}} = r^T r$

2 **for** $k = 1, 2, \ldots$ *until convergence,* **do**

3 $\quad [P, R] = [K_{s+1}(A, \rho, p), K_s(A, \rho, r)]$

4 $\quad G = [P, R]^T [P, R]$

5 $\quad$ Initialize data structures:

6 $\quad H$ is square, satisfies $A[P(:, 1:s), R(:, 1:(s-1))] = [P, R]H(:, [1:s, (s+2):(2s-1)])$

7 $\quad [\hat{x}, \hat{p}, \hat{r}] = [0_{2s+1,1}, [1, 0_{1,2s}]^T, [0_{1,s+1}, 1, 0_{1,s-1}]^T]$

8 $\quad$ **for** $j = 1, 2, \ldots, s$ **do**

9 $\quad\quad \hat{w} = H\hat{p}$

10 $\quad\quad \alpha = \delta_{\text{prv}} / (\hat{p}^T G \hat{w})$

11 $\quad\quad \hat{x} = \hat{x} + \alpha \hat{p}$

12 $\quad\quad \hat{r} = \hat{r} - \alpha \hat{w}$

13 $\quad\quad \delta_{\text{cur}} = \hat{r}^T G \hat{r}$

14 $\quad\quad \beta = \delta_{\text{cur}} / \delta_{\text{prv}}$

15 $\quad\quad \hat{p} = \hat{r} + \beta \hat{p}$

16 $\quad [p, r, x] = [P, R][\hat{p}, \hat{r}, \hat{x}] + [0_{n,2}, x]$

In exact arithmetic, CA-CG produces the same iterates as CG (until termination). However, CA-CG performs more operations, requires more storage, and in finite-precision arithmetic may suffer worse rounding errors (which can be alleviated by the choice of ρ). The chief advantage to CA-CG is that it exposes temporal locality in the computations of $[P, R]$ and G, enabling a reduction in data movement. In particular, P and R can be computed in a blocked manner using a "matrix powers" optimization and G can be computed by matrix multiplication [14]. While these optimizations reduce communication, they do not reduce W_{12}: assuming n is sufficiently large with respect to M_1, N/s outer iterations cost $W_{12} = O(N \cdot n)$, attaining the lower bound given above for N iterations of CG.

However, writes can be avoided by exploiting a "streaming matrix powers" optimization [14, Section 6.3], at the cost of computing P and R twice. The idea is to interleave a blockwise computation of G (line 4 in Algorithm 7) and of $[p, r, x]$ (line 16 in Algorithm 7) with blockwise computations of $[P, R]$ (line 3 in Algorithm 7), each time discarding the entries of $[P, R]$ from fast memory after they have been multiplied and accumulated into G or $[p, r, x]$. (All $O(s)$-by-$O(s)$ matrices are assumed to fit in fast memory.) If the matrix powers optimization reduces the number of L_2 reads of vector entries by a factor of $f(s)$, then the streaming matrix powers optimization reduces the number of L_2 writes by $\Theta(f(s))$, at the cost of doubling the number of operations to compute P and R. In cases where $f(s) = \Theta(s)$, like for a $(2b+1)^d$-point stencil on a sufficiently large d-dimensional Cartesian mesh when $s = \Theta(M_1^{1/d}/b)$, we thus have that $W_{12} = O(N \cdot n/s)$ over N/s outer iterations of CA-CG, assuming n is sufficiently large with respect to M_1, an asymptotic reduction in the number of writes compared to N iterations of CG.

The streaming matrix powers optimization extends to other KSMs (more precisely, their s-

step variants). For Arnoldi-based KSMs, the computation of G is replaced by a tall-skinny QR factorization, which can be interleaved with the matrix powers computation in a similar manner.

9 Conclusions and Future Work

Conclusions. Motivated by the fact that writes to some levels of memory can be much more expensive than reads (measured in time or energy), for example in the case of nonvolatile memory, we have investigated algorithms that minimize the number of writes. First, we established new lower bounds on the number of writes needed to execute a variety of algorithms. In some cases (e.g., classical linear algebra), these lower bounds are asymptotically smaller than the lower bound on the number of reads, suggesting large savings are possible. We call algorithms that minimize the number of reads and writes (i.e., are communication-avoiding (CA)), and also attain asymptotically smaller lower bounds on writes, *Write-Avoiding (WA)* (see Section 2).

In some cases, e.g., the Cooley-Tukey FFT, Strassen's matrix multiplication algorithm, and, more generally, algorithms whose CDAGs satisfy certain simple conditions on their vertex degrees, the lower bounds on reads and writes differ by at most a modest constant. This implies that WA reorderings of these algorithms cannot exist (see Section 3).

When WA algorithms can exist, we investigated both sequential and distributed memory parallel algorithms. In the sequential case, we provide WA algorithms for matrix multiplication, triangular solve, Cholesky factorization, and the direct N-body algorithm. All these algorithms use explicit blocking based on the fast memory size M, and extend to multiple levels of memory hierarchy (see Section 4).

It is natural to ask whether cache-oblivious algorithms can attain these lower bounds, but we show this is impossible for direct linear algebra. In other words, a WA algorithm cannot be cache-oblivious (see Section 5).

We observed that the explicit movement of cache lines in WA algorithms can be relaxed and replaced with an online LRU replacement policy. We measured variants of classical matrix multiplication with hardware counters and related the empirical data to theory. These experiments support the case for the LRU replacement policy (see Section 6).

In the parallel case, we assumed homogeneous machines, with each processor having an identical local memory hierarchy. We analyzed three scenarios (see Section 7).

In the first scenario (called Model 1), the network connects the lowest level of memory on each processor. In this case, the natural approach is to use a CA algorithm to minimize interprocessor communication and a WA algorithm locally on each processor to try to minimize writes to the lowest level of memory. While this does reduce the number of local writes to the number of writes from the network (which is probably good enough in practice), it does not attain the lower bound without using a great deal of extra memory.

In the second scenario (Model 2.1), the network connects the second-lowest level of memory (say DRAM) on each processor, there is another lower, larger level (say NVM), and all the data fits in DRAM, so that it is not necessary to use NVM. Here it may still be advantageous to use NVM to replicate more copies of the data than possible using DRAM alone, in order to reduce interprocessor communication. We provide a detailed performance model for matrix multiplication, which depends on numerous algorithmic and architectural parameters, that predicts whether using NVM is helpful.

The third scenario (Model 2.2) differs from the second scenario in that the data is too large to fit in DRAM, and so needs to use NVM. Now we have a lower bound on interprocessor communication and a lower bound on writes to NVM: we prove that it is impossible to attain both for matrix

multiplication, i.e., any algorithm must do asymptotically more of one or the other. We also present two algorithms, each of which attains one of the lower bounds, but not the other. Detailed performance models are provided that can predict which one is faster. We present analogous algorithms for LU factorization without pivoting.

Finally, we analyzed iterative linear algebra algorithms (see Section 8) for solving linear systems $Ax = b$. Prior work on CA Krylov subspace methods (CA-KSMs) showed that under certain conditions on the nonzero structure of A, one could take s steps of the method (requiring s matrix-vector multiplications) for the communication cost of one step, i.e., reading A from DRAM to cache once. We show that it is also possible to reduce the number of writes by a factor of s, but at the cost of increasing both the number of reads and the number of arithmetic operations by a factor of at most 2.

Future Work. These results suggest a variety of open problems and future work. In particular, it is natural to try to extend many of the results from Sections 4 through 7 to other linear algebra algorithms, the direct N-body problem, and more generally algorithms accessing arrays as analyzed in [15]. We conjecture that any WA algorithm for the direct N-body problem (with 2-body interactions) needs to perform twice as many flops as a non-WA algorithm.

In section 7 we showed that no algorithm in Model 2.2 can simultaneously attain both lower bounds on the number of words communicated over the network, and the number of words written to L_3 from L_2, but that there are matmul and LU algorithms that can attain one but not the other bound. It is natural to ask whether there is a more general lower bound that shows how these two bandwidth quantities tradeoff against one another, and whether there is a family of algorithms that exhibits this tradeoff.

It is also of interest to investigate WA schedules for shared memory parallel machines. Blelloch et al. [12] suggest that previously known thread schedulers that are provably good for various cache configurations are good at minimizing writes in low-depth algorithms [10]. While this covers algorithms with polylogarithmic depth such as sorting, the FFT, and classical matrix multiplication, we note that many linear algebra algorithms (e.g., TRSM and Cholesky) have linear depth. Designing a WA SMP thread scheduler for these algorithms remains an open problem.

We proved that several algorithms can not be reordered to be write-avoiding. It seems that no WA algorithms exist for certain problems. Specifically, we conjecture that, on a machine with a two-level memory hierarchy with fast memory size M, no algorithm for the Discrete Fourier Transform problem or the sorting problem can simultaneously perform $o(n \log_M n)$ writes to slow memory and $O(n \log_M n)$ reads from slow memory (recall that $\Omega(n \log_M n)$ is a lower bound on the sum of reads and writes for these two problems [3, 28]). Asymptotically fewer writes for these problems seem to require an asymptotically greater number of reads.

Finally, if access to NVM is controlled by hardware instead of the programmer, extending the analysis of Section 6 to the predict the impact of real cache policies on WA algorithms will be important to their success in practice.

10 Acknowledgments

We thank Yuan Tang for pointing out the role of write-buffers. We thank U.S. DOE Office of Science, Office of Advanced Scientific Computing Research, Applied Mathematics Program, grants DE-SC0010200, DE-SC-0008700, and AC02-05CH11231, for financial support, along with DARPA grant HR0011-12-2-0016, ASPIRE Lab industrial sponsors and affiliates Intel, Google, Huawei, LG, NVIDIA, Oracle, and Samsung, and MathWorks. Research is supported by grants 1878/14,

and 1901/14 from the Israel Science Foundation (founded by the Israel Academy of Sciences and Humanities) and grant 3-10891 from the Ministry of Science and Technology, Israel. Research is also supported by the Einstein Foundation and the Minerva Foundation. We thank Guy Blelloch and Phillip Gibbons for access to the machine on which we ran our experiments.

References

[1] R. Agarwal, F. Gustavson, and M. Zubair. A high-performance matrix-multiplication algorithm on a distributed-memory parallel computer, using overlapped communication. *IBM J. of Research and Development*, 38(6):673–681, 1994.

[2] A. Aggarwal, A. K. Chandra, and M. Snir. Communication complexity of PRAMs. *Theoretical Computer Science*, 71(1), 1990.

[3] Alok Aggarwal and S. Vitter, Jeffrey. The input/output complexity of sorting and related problems. *Commun. ACM*, 31(9):1116–1127, September 1988.

[4] G. Ballard, E. Carson, J. Demmel, M. Hoemmen, N. Knight, and O. Schwartz. *Acta Numerica*, volume 23, chapter Communication lower bounds and optimal algorithms for numerical linear algebra. Cambridge University Press, 2014.

[5] G. Ballard, J. Demmel, O. Holtz, B. Lipshitz, and O. Schwartz. Graph expansion analysis for communication costs of fast rectangular matrix multiplication. In *Proc. First Mediterranean Conference on Algorithms (MedAlg)*, pages 13–36, 2012.

[6] G. Ballard, J. Demmel, O. Holtz, B. Lipshitz, and O. Schwartz. Strong scaling of matrix multiplication algorithms and memory-independent communication lower bounds (brief announcement). In *24th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA'12)*, pages 77–79, 2012.

[7] G. Ballard, J. Demmel, O. Holtz, and O. Schwartz. Minimizing communication in numerical linear algebra. *SIAM J. Mat. Anal. Appl.*, 32(3):866–901, 2011.

[8] G. Ballard, J. Demmel, O. Holtz, and O. Schwartz. Graph expansion and communication costs of fast matrix multiplication. *JACM*, 59(6), Dec 2012.

[9] L. A. Belady. A study of replacement algorithms for a virtual-storage computer. *IBM Syst. J.*, 5(2):78–101, June 1966.

[10] G. Blelloch, P. Gibbons, and H. Simhadri. Low depth cache-oblivious algorithms. In *22rd ACM Symposium on Parallelism in Algorithms and Architectures (SPAA 2010)*, 2010.

[11] G. Blelloch and R. Harper. Cache and I/O efficient functional algorithms. In *40th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, (POPL 2013)*, 2013.

[12] Guy Blelloch, Jeremy Fineman, Phillip Gibbons, Yan Gu, and Julian Shun. Sorting with asymmetric read and write costs. In ACM Symposium on Parallel Algorithms and Architectures (SPAA 2015), June 2015.

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

[14] E. Carson, N. Knight, and J. Demmel. Avoiding communication in nonsymmetric Lanczos-based Krylov subspace methods. *SIAM Journal on Scientific Computing*, 35(5):S42–S61, 2013.

[15] M. Christ, J. Demmel, N. Knight, T. Scanlon, and K. Yelick. Communication lower bounds and optimal algorithms for programs that reference arrays - part 1. Tech Report UCB/EECS-2013-61, UC Berkeley Computer Science Division, May 2013.

[16] National Research Council Committee on Sustaining Growth in Computing Performance. *The Future of Computing Performance: Game Over or Next Level?* National Academies Press, 2011. 200 pages, http://www.nap.edu.

[17] F.J. Corbató. *A Paging Experiment with the Multics System.* Project MAC. Defense Technical Information Center, 1968.

[18] G. De Sandre, L. Bettini, A. Pirola, L. Marmonier, M. Pasotti, M. Borghi, P. Mattavelli, P. Zuliani, L. Scotti, G. Mastracchio, F. Bedeschi, R. Gastaldi, and R. Bez. A 4 Mb LV MOS-Selected Embedded Phase Change Memory in 90 nm Standard CMOS Technology. *IEEE J. Solid-State Circuits*, 46(1), Jan 2011.

[19] J. Demmel, L. Grigori, M. Hoemmen, and J. Langou. Communication-optimal parallel and sequential QR and LU factorizations. *SIAM J. Sci. Comp.*, 34(1), Feb 2012.

[20] J. Demmel and E. Solomonik. Communication-optimal parallel 2.5D matrix multiplication and LU factorization algorithms. In *EuroPar'11*, 2011. report UCB/EECS-2011-72, Jun 2011.

[21] M. Driscoll, E. Georganas, P. Koanantakool, E. Solomonik, and K. Yelick. A communication-optimal n-body algorithm for direct interactions, 2013.

[22] David Eklov, Nikos Nikoleris, David Black-Schaffer, and Erik Hagersten. Cache pirating: Measuring the curse of the shared cache. In *Proceedings of the 2011 International Conference on Parallel Processing*, ICPP '11, pages 165–175, Washington, DC, USA, 2011. IEEE Computer Society.

[23] M. Frigo, C. Leiserson, H. Prokop, and S. Ramachandran. Cache-oblivious algorithms. In *Proceedings of the 40th IEEE Symposium on Foundations of Computer Science (FOCS 99)*, pages 285–297, 1999.

[24] Matteo Frigo, Charles E. Leiserson, Harald Prokop, and Sridhar Ramachandran. Cache-oblivious algorithms. In *FOCS*, 1999.

[25] L. Grigori, J. Demmel, and H. Xiang. CALU: a communication optimal LU factorization algorithm. *SIAM Journal on Matrix Analysis and Applications*, 32:1317–1350, 2011.

[26] J. Hennessy and D. Patterson. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann Publishers, 4th edition, 2007.

[27] Henry. Cache replacement policy for nehalem/snb/ib? `http://blog.stuffedcow.net/2013/01/ivb-cache-replacement/`, January 2013.

[28] X. Hong and H. T. Kung. I/O complexity: the red blue pebble game. In *Proceedings of the 13th Symposium on the Theory of Computing*, pages 326–334. ACM, 1981.

[29] J. Hu, Q. Zhuge, C. J. Xue, W.-C. Tseng, S. Gu, and E. Sha. Scheduling to optimize cache utilization for non-volatile main memories. *IEEE Trans. Computers*, 63(8), Aug 2014.

[30] Intel. Intel(R) Xeon(R) processor 7500 series datasheet, volume 2. `http://www.intel.com/content/dam/www/public/us/en/documents/datasheets/xeon-processor-7500-series-vol-2-datasheet.pdf`, March 2010.

[31] Intel. Intel(R) Xeon(R) processor 7500 series uncore programming guide. `http://www.intel.com/Assets/en_US/PDF/designguide/323535.pdf`, March 2010.

[32] Intel. Performance Counter Monitor (PCM). http://www.intel.com/software/pcm, 2013. Version 2.4.

[33] Intel. Intel(R) 64 and IA-32 architectures optimization reference manual. http://www.intel.com/content/www/us/en/architecture-and-technology/64-ia-32-architectures-optimization-manual.html, September 2014.

[34] Intel. Intel(R) 64 and IA-32 architectures software developer's manual. http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html, April 2015.

[35] Intel_Karla. Cache replacement policy for nehalem/snb/ib? https://communities.intel.com/thread/32695, November 2012.

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

[37] Aamer Jaleel, Kevin B. Theobald, Simon C. Steely, Jr., and Joel Emer. High performance cache replacement using re-reference interval prediction (rrip). In *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ISCA '10, pages 60–71, New York, NY, USA, 2010. ACM.

[38] P. Koanantakool and K. Yelick. A computation- and communication-optimal parallel direct 3-body algorithm. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '14, pages 363–374, Piscataway, NJ, USA, 2014. IEEE Press.

[39] Adam Litke, Eric Mundon, and Nishanth Aravamudan. libhugetlbfs. http://libhugetlbfs.sourceforge.net, 2006.

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

[41] A. Rasmussen, M. Conley, R. Kapoor, V. T. Lam, G. Porter, and A. Vahdat. Themis: An I/O Efficient MapReduce. In *Proc. ACM Symp. on Cloud Computing (SoCC 2012)*, San Jose, CA, Oct 2012.

[42] Daniel D. Sleator and Robert E. Tarjan. Amortized efficiency of list update and paging rules. *CACM*, 28(2), 1985.

[43] M. Snir and S. Graham, editors. *Getting up to speed: The Future of Supercomputing*. National Research Council, 2004. 227 pages.

[44] A. Tiskin. Communication-efficient parallel generic pairwise elimination. *Future Generation Computer Systems*, 23(2):179–188, 2007.

[45] R. van de Geijn and J. Watts. SUMMA: Scalable Universal Matrix Multiplication Algorithm. *Concurrency - Practice and Experience*, 9(4):255–274, 1997.