Write-Avoiding Algorithms

Erin Carson
Courant Institute of Mathematical Sciences, New York University
erin.carson@nyu.edu

James Demmel
Dept. of Mathematics and Computer Science Div., Univ. of California, Berkeley
demmel@berkeley.edu

Laura Grigori
INRIA Paris-Rocquencourt, Alpines, and UPMC - Univ Paris 6, CNRS UMR 7598, Laboratoire Jacques-Louis Lions, France
laura.grigori@inria.fr

Nicholas Knight
Courant Institute of Mathematical Sciences, New York University
nknight@nyu.edu

Penporn Koanantakool
Computer Science Div., Univ. of California, Berkeley
penpornk@eecs.berkeley.edu

Oded Schwartz
School of Engineering and Computer Science, Hebrew Univ. of Jerusalem, Israel
odedsc@cs.huji.ac.il

Harsha Vardhan Simhadri
Computational Research Div., Lawrence Berkeley National Lab.
harshas@lbl.gov.

***Abstract*—Communication, i.e., moving data between levels of a memory hierarchy or between processors over a network, is much more expensive (in time or energy) than arithmetic. There has thus been a recent focus on designing algorithms that minimize communication and, when possible, attain lower bounds on the total number of reads and writes. However, most 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 storage devices such as nonvolatile memories.**

This motivates us to ask whether there are lower bounds on the number of writes that certain algorithms must perform, and whether 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 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 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.

I. 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 [2], [3]. With this motivation, there has been much recent work [4], [5] designing new 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)*. To be clear, while some CA algorithms are new, others are in fact new schedules—order of instructions—of known algorithms. With some overloading of notation, we sometimes also refer to these CA schedules as CA algorithms. There has also been considerable work on proving lower bounds on the amount of communication required for a problem (e.g., comparison sorting [6]) and, where such bounds are difficult to prove, the minimum amount of communication required by any valid schedule of an algorithm (e.g., FFT [7], numerical linear algebra [4], [8], [9], [10]). We refer to these as communication lower bounds.

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 nonvolatile memory technologies (NVM) [11] where writes can be much more expensive (in time and energy) than reads. NVM technologies are being considered for scientific applications on extreme scale computers [12] and for cluster computing platforms [13], in addition to commodity computers. One example of nonvolatile memory (NVM) is Phase Change Memory [11], where, e.g., a write is 15 times slower than a read both in terms of latency and bandwidth [14] and consumes 10 times as much energy [15]. Another technology called CBRAM uses significantly more energy for writes (1pJ) than reads (50fJ) [16], [17]. Writes to NVM can also be less reliable than reads, require multiple attempts for success, and can cause device wear out [18], [19]. Motivated by this, work in [20], [21]—and references therein—attempts to reduce the number of writes to NVM.

This motivates us to first refine prior work on communication lower bounds of algorithms which did not distinguish

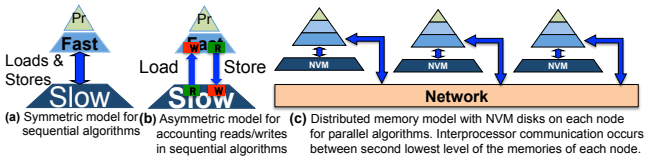

Figure 1: Memory models for sequential and parallel algorithms.

between loads and stores (Fig. 1(a)) 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 (Fig. 1(b)). 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)*. In this paper, we identify several classes of problems where either sequential or parallel WA algorithms exist, or provably cannot.

Contribution: We first consider sequential algorithms with communication within a memory hierarchy, and then parallel algorithms with communication over a network. To analyze sequential algorithms, we start with the widely used two-level memory hierarchy [6], [22], [23] with a "fast" memory closer to the processor and a "slow" memory further away, and refine the cost model to separate writes from reads. Whereas previous models analyze the total amount of data movement between the memory levels required by the sequence of instructions in the algorithm, the refinement we present in Section II counts, for each variable, the number of writes to the slow and fast memories (and to each level in a multi-level hierarchy, in the technical report [1]). Proposition 1 states that the number of writes to fast memory is at least half the number of loads and stores, thus ruling out the possibility of any algorithm (or schedule) avoiding writes to the fast memory.

To understand which algorithms have schedules that avoid writes to the slow memory, in Section III, we look to their representation as a CDAG — the computation directed acyclic graph with a vertex for each input or computed result and directed edges for dependencies. Theorem 1 states that if an algorithm has constant factor reuse, i.e., the outdegree of the CDAG is bounded by a constant, and the inputs are not reused too many times, then again the number of writes by any schedule of the algorithm (i.e., any traversal of the algorithm's CDAG) to slow memory is at least a constant factor of the total number of loads and stores. Thus, "fast algorithms" which have bounded reuse, e.g., Cooley-Tukey FFT and Strassen's matrix multiplication, cannot be WA.

This leaves us with algorithms with CDAGs that have a large degree of reuse such as those in classical direct linear algebra and N-body methods. In Section IV, we build upon the WA schedule for classical matrix multiplication **(MM)** in [20] to construct WA schedules for triangular solves and Cholesky factorization. It is known that on a two-level memory hierarchy with fast memory of size M, any schedule for multiplying two $n \times n$ matrices requires moving at least $\Omega(n^3/\sqrt{M})$ words between slow and fast memories [7]. The WA schedule in Algorithm 1 is a special case of a CA schedule that orders the execution of the blocks so that only n^2 words are written to the slow memory — no more than the size of the output. Similarly, the WA schedules we present for triangular solve, Cholesky factorization, and the direct N-body algorithm cause no more writes to the slow memory than the size of the output.

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 schedules and algorithms have been invented [23], [5]. It is natural to ask if write-avoiding, cache-oblivious (WACO) algorithms exist. Theorem 2 and Corollary 3 in Section V prove a negative result: for a large class of algorithms, including most direct linear algebra for dense or sparse matrices, and some graph-theoretic algorithms, no WACO schedule can exist, i.e., the number of writes to slow memory is proportional to the number of reads.

The WA schedules we present use explicit movement of data between the levels of the memory hierarchy. However in most cases, architectures only allow the programmer to address data by virtual memory address, and the hardware cache (replacement) policy determines the mapping of the address to physical locations and thus dictates reads and writes. In Section VI, we consider the interaction of the WA schedules in Section IV with cache replacement policies. Propositions 2 and 3 argue that the Least Recently Used (LRU) policy can replace the algorithms' explicit data movements, preserving WA properties. We also provide empirical hardware counter measurements of cache evictions and fills on an Intel Nehalem-EX machine (which uses an LRU-like policy) that match our claims.

Section VII discusses the parallel homogeneous case (Fig. 1(c)), particularly the effect of combining CA and WA algorithms, and the introduction of NVM as the largest level in the memory hierarchy. We consider three scenarios: without NVM (Model 1 in Section VII), with NVM and data fits in DRAM (Model 2.1 in Section VII), and with NVM but data does not fit in DRAM (Model 2.2 in Section VII). While the CA-WA algorithm alone seems unlikely to be beneficial in the first scenario, using NVM can be more advantageous in the second scenario, depending on algorithm and hardware parameters. In the third scenario where we must use NVM, we prove that it is impossible to attain both lower bounds on interprocessor communication and on writes to NVM. We then present two algorithms for matrix

multiplication, each of which attains one of these lower bounds. The technical report [1] also presents extensions of these algorithms to LU factorization without pivoting.

In Section VIII, we consider Krylov subspace methods (KSMs). We show that a known optimization for minimizing communication in KSMs is more generally applicable in the context of minimizing just writes. We also exhibit a family of machine and problem parameters where this optimization asymptotically reduces the number of writes at the cost of doubling the numbers of reads and arithmetic operations.

Related work: The most relevant prior work on this topic is that of Blelloch et al. [20] which proposes "write-efficient" algorithms in the asymmetric PRAM, External Memory, and Ideal Cache models, all designed to model machines with non-volatile storage. Most of their algorithms, except MM, exhibit a tradeoff between "reads" and "writes"; to decrease the number of writes, a greater number of reads than in the optimal algorithm must be performed. For example, in their asymmetric external memory model, they present a sorting algorithm that does $(k+1)\left\lceil\frac{n}{B}\right\rceil\left\lceil\log_{\frac{kM}{B}}\frac{n}{B}\right\rceil$ reads and $\left\lceil\frac{n}{B}\right\rceil\left\lceil\log_{\frac{kM}{B}}\frac{n}{B}\right\rceil$ writes with a fast memory of size $M + o(M)$ and block size B. The parameter k can be tuned based on the extent of asymmetry between the cost of reads and writes. Our algorithms do not exhibit such tradeoffs; we achieve asymptotic reduction in writes without asymptotically increasing the number of reads. Their cache-oblivious algorithms are oblivious to the cache size, but not the parameter k. Further, while they claim to present a cache-oblivious algorithm for matrix multiplication in Theorem 5.3, the number of writes is not optimal (i.e. more than $\Omega(n^2)$) and the bound is weaker for most input sizes except those that are of the form $k^i \cdot \sqrt{M}$ for some integer i. They also present a modified LRU cache replacement policy to support their asymmetric ideal cache model. The unmodified LRU policy works well for our algorithms, but may not necessarily extend to all algorithms.

There has been much work on adapting algorithms, data-structures, and file systems for Flash memories [24]. The considerations there are slightly different there than in the newer NVM technologies. Namely, writes in a NAND Flash device are done in large blocks, and smaller writes must be aggregated at the algorithm or OS level for performance and durability [25]. On the other hand, NVMs are byte-addressable or have small block sizes [12]. Wear-leveling for Flash memories is another well studied issue [26], [27], which we do not address as newer NVM technologies are more durable than Flash devices and hardware techniques for doing this have been proposed [28]. Memory-mapped persistent data structures that provide transactional semantics for fast byte-addressable memories like NVM have been studied and demonstrated to perform well [29], although this work does not analyze performance for specific algorithms.

II. Memory Model and Lower Bounds

We consider a two-level memory hierarchy with a fast memory of limited size close to the processor and a slow memory with no size limitations. [1] Since our goal is to bound the number of writes to a particular memory level, we refine this model as follows:

- **A load operation** consists of a read from slow memory and a write to fast memory.
- **A store operation** 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.

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.[2]

We consider a variable *resident* in fast memory from the time it first appears to the time it is last used (read/written). It may be written multiple times during this time period, but it must be associated with a unique data item in the program, for instance a matrix entry A_{ij}. If it is a temporary accumulator, say first for C_{11}, then for C_{12}, then between each read/write we can still identify it with a unique entry of C. A resident variable 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 [22], 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, R1/D2, R2/D1, and R2/D2. In each category there is a write to fast memory, and possibly more, if the value in fast memory is updated. Given all the loads and stores executed by a program, we can uniquely label them by the

[1] The technical report has an extension of the model and extends Proposition 1 to multi-level hierarchies.

[2] We assume compiler-generated variables such as loop indices can reside in fast memory and not cause data movement between the levels we are interested in.

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:

Proposition 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.

III. Bounded Data Reuse Precludes Write-Avoiding

Using the *computation directed acyclic graph (CDAG)* representation of an algorithm, we show that if each argument (input data or computed value) of a given computation is used only a constant number of times, then no execution order of the algorithm can decrease the number of writes asymptotically, i.e., it cannot have a WA schedule. 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 operations $x = y + z$, $x = x + w$ are represented by five 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 1 (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 $\geq \lceil (\frac{10d-1}{10d} - \frac{1}{2}) \frac{W}{d} \rceil = \Omega(\frac{W}{d})$, proving (2). ∎

This Theorem allows us to show that the certain "fast algorithms" have no WA schedules.

Corollary 1 (Cooley-Tukey FFT cannot be WA): Consider executing the n-point Cooley-Tukey FFT algorithm 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: $\Omega(\frac{n \log n}{\log M})$.

Proof: The Cooley-Tukey FFT has out-degree bounded by $d = 2$, input vertices included. By [7], the total number of loads and stores to fast memory performed by any schedule 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 1 with $G' = G$. ∎

Corollary 2 (Strassen's agorithm cannot be WA): Consider executing Strassen's matrix multiplication algorithm on n-by-n matrices on the machine described in Corollary 1. 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: Let G' 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 $W = \Omega(n^{\omega_0}/M^{\omega_0/2-1})$. Apply Theorem 1. ∎

Corollary 2 extends to any Strassen-like fast matrix multiplication algorithm (defined in [8]), with ω_0 replaced with the corresponding algorithm's exponent. It also applies to Strassen-like rectangular matrix multiplication (see [30]).

IV. Examples of WA Algorithms

Consider classical matrix multiplication (**for** $i \in [m], k \in [n], j \in [l]$ **do** $C_{ij} += A_{ik} \cdot B_{kj}$) which performs mnl multiplications to compute $C^{m \times l} = A^{m \times n} \cdot B^{n \times l}$. The lower bound on loads and stores for this algorithm in a two-level memory hierarchy with fast memory of size M is $\Omega(mnl/M^{1/2})$ [22], [7], [10]. Algorithm 1 is a well-known example of a CA schedule that matches the communication lower bound when the block size parameter b is set to $\sqrt{M/3}$. In addition it is also WA as noted by Blelloch et al. [20] — it requires just ml writes to the slow memory.

For simplicity we assume that all expressions like $\sqrt{M/3}$, m/b, etc., are integers. For a schedule to match the com-

Algorithm 1: 2-Level Blocked MM

Data: $C^{m\times l}, A^{m\times n}, B^{n\times l}$; **Result**: $C^{m\times l} += A^{m\times n} \cdot B^{n\times l}$;
// L_1 & L_2 denote fast & slow memories.
$b = \sqrt{M_1/3}$ // block size for L_1; assume b divides n.
// $A(i,k), B(k,j), C(i,j)$ refer to $b \times b$ blocks.
for $i \leftarrow 1$ **to** m/b **do**
  for $j \leftarrow 1$ **to** l/b **do**
    load $C(i,j)$ from L_2 to L_1 **for** $k \leftarrow 1$ **to** n/b **do**
      load $A(i,k)$ and $B(k,j)$ from L_2 to L_1
      $C(i,j) = C(i,j) + A(i,k) * B(k,j)$
    store $C(i,j)$ from L_1 to L_2

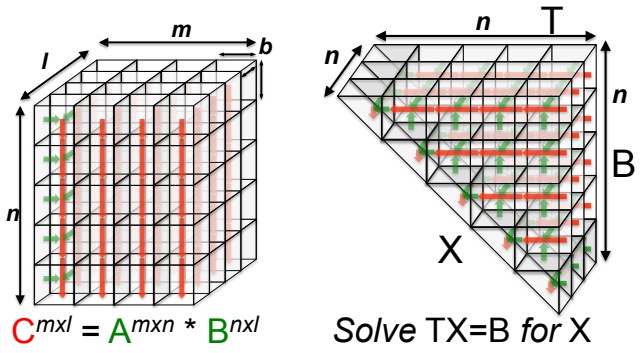

Figure 2: Geometric picture of WA matrix multiplication (left); WA TRSM (right).

munication lower bound, it must break the iteration space, $\{i \in [m], k \in [n], j \in [l]\}$, into blocks of size $b \times b \times b$, and execute each block contiguously (this is represented by lmn points in the cube in Fig. 2(left) divided into smaller b^3-size cubes). For this, the necessary data – $b \times b$ blocks of arrays A, B and C – must be loaded into the fast memory. In Fig. 2 these correspond to projections of the subcube onto the left face, representing array A, the front face (B) and the bottom face (C). Thus for the price of loading $3b^2$ data, we are able to cover b^3 points in the iteration space, or equivalently, perform b^3 arithmetic operations, thus making the schedule CA. While any order of execution of the blocks has CA properties, Algorithm 1 groups the execution of all blocks of the iteration space corresponding to a block of C (columns of subcubes marked by red arrow in Fig. 2). Since each variable in C belongs to exactly one block of C, all updates to it are accumulated before writing it back to the slow memory just once, making the schedule write-avoiding.

Algorithm 2 presents WA triangular solve (solve $TX = B$ for $X^{n\times n}$) which requires only as many writes to the slow memory as the size of the output X. All loads or stores of data are annotated with the write costs incurred in one iteration and across all iterations. The total number of reads from slow memory is an asymptotically optimal $\Theta(n^3/M^{1/2})$. Fig. 2(right) illustrates the schedule. Algorithm 3 presents WA Cholesky factorization which does $n^2/2$ writes (size of the output) and $\Theta(n^3/M^{1/2})$ reads to the slow memory. The technical report [1] contains direct N-body methods, and discusses how to adapt all the schedules in this section to multi-level hierarchies.

V. Cache-Oblivious Matrix Multiplication Cannot be Write-Avoiding

Define a *cache-oblivious (CO)* algorithm [23] as one in which the sequence of load (read from slow, write to fast memory), store (read from fast, write to slow memory), and arithmetic/logical instructions executed does not depend on the memory hierarchy of the machine. Depending on the cache policy, a load or a store instruction may or may not result in actual data movement, but arithmetic operations must be performed on the specified data in the order specified.

We show that even with such general cache policies, certain *cache oblivious algorithms cannot be write-avoiding*, in fact they must do at least a constant factor times as many writes as a CA algorithm does reads. This is in contrast to the existence of many CO algorithms that are CA. Our proof applies to any algorithm that has the following property:

Triply nested with two dimensional projections: The algorithm is a set S of triples of nonnegative integers (i,j,k), reads two array locations $A(i,k)$ and $B(k,j)$ and updates array location $C(i,j)$ for all triples in S; for simplicity we call this update operation an "inner loop iteration".

This class of algorithms includes direct linear algebra (including algorithms in Sec. IV), tensor contractions, and Floyd-Warshall all-pairs shortest-paths in a graph. We will assume that there are no R2/D2 residencies (see Sec. II).

Lemma 1 (Markov's inequality): If $s_i \geq 0$ for $i = 1, ..., N$, $A = \sum_{i=1}^{N} s_i/N$ is the average, and $m > 1$, then $N_< \equiv |\{i : \ s_i \leq mA\}|$ satisfies $N_< \geq \frac{m-1}{m} N$.

Theorem 2: Consider an algorithm that is triply nested with two dimensional projections. 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 a fixed fast memory size M, the algorithm is CA in the following sense: the total number of loads and stores is at most $c \cdot |S|/M^{1/2}$ for some constant $c > 0$. Then independent of the cache policy, when running with cache size $M' \leq \frac{0.5M}{(1+4c)^2}$, the number of writes to slow memory is at least

$$W_s \geq \frac{M}{4(1+4c)^2} \left\lfloor \frac{|S|}{2M^{3/2}} \right\rfloor = \Omega\left(\frac{|S|}{M^{1/2}}\right) \qquad (1)$$

Proof: Divide the stream of instructions executed by the program into contiguous *arithmetic segments*, each of which contains exactly $2M^{3/2}$ arithmetic operations accessing only fast memory, as well as the intervening load and store operations. Thus the number of complete arithmetic segments is $n_{as} \equiv \lfloor |S|/(2M^{3/2}) \rfloor$. Assume, w.l.o.g., $n_{as} \geq 1$. Let $n_{M,i}$ be the actual number of loads and stores performed during arithmetic segment i. By assumption $\sum_{i=1}^{n_{as}} n_{M,i} \leq c|S|/M^{1/2}$, so we can bound the average value $\frac{1}{n_{as}} \sum_{i=1}^{n_{as}} n_{M,i} \leq \frac{(c|S|/M^{1/2})}{\lfloor |S|/(2M^{3/2}) \rfloor} \leq 4cM$.

By Lemma 1 (with $m = 2$) we get $n_< \equiv |\{i : n_{M,i} \leq 8cM\}| \geq 0.5 n_{as}$. For each of these $n_<$ arithmetic segments

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)

```
b = sqrt(M1/3)                                   // block size for L1; assume n is a multiple of b
for j ← 1 to n/b do                              // T(i,k), X(k,j), and B(i,j) refer to b-by-b blocks of T, X, and B
    for i ← n/b downto 1 do                      //  | #writes to L1 | total #writes to L1 |
        load B(i,j) from L2 to L1                //  |      b^2      |  (n/b)^2 · b^2 = n^2 |
        for k ← i+1 to n/b do
            load T(i,k) and X(k,j) from L2 to L1 //  |    2 × b^2    | 2 × .5(n/b)^3 · b^2 = n^3/b |
            B(i,j) = B(i,j) − T(i,k) * X(k,j)    //  |       -       |              -              |
        load T(i,i) from L2 to L1                // half as many writes as for B(i,j) as counted above
        solve T(i,i) * Tmp = B(i,j) for Tmp; B(i,j) = Tmp   // b-by-b TRSM
        store B(i,j) from L1 to L2               // #writes to L2 = b^2, total #writes to L2 = (n/b)^2 · b^2 = n^2
```

Algorithm 3: 2-Level Blocked 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(M1/3)    // block size for L1; assume b divides n.
// 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 L2 to L1
    for k ← 1 to i − 1 do
        load A(i,k) from L2 to L1
        A(i,i) = A(i,i) − A(i,k) * A(i,k)^T
    overwrite A(i,i) by its Cholesky factor
    store A(i,i) (just the lower half) from L1 to L2
    for j ← i + 1 to n/b do
        load A(j,i) from L2 to L1
        for k = 1 to i − 1 do
            load A(i,k) and A(j,k) from L2 to L1
            A(j,i) = A(j,i) − A(j,k) * A(i,k)^T
        load A(i,i) (just the lower half) from L2 to L1
        solve Tmp * A(i,i)^T = A(j,i) for Tmp; A(j,i) = Tmp
        store A(j,i) from L1 to L2
```

we apply the *Loomis-Whitney inequality* [31] to get a lower bound on the product of the number of entries $|A|$, $|B|$ and $|C|$ of the three matrices that are accessed in the arithmetic segment: $(|A|\cdot|B|\cdot|C|)^{1/2} \geq 2M^{3/2}$. The C matrix is being written by the algorithm, so we would like a lower bound on $|C|$. Since there are no R2/D2 arguments, we can bound $|A|$ and $|B|$ by bounding the total number of R1 and D1 arguments: The first is at most M plus #loads, the latter is at most M plus #stores. The total number of R1 and D1 arguments is at most $2M + \#\text{loads} + \#\text{stores} = 2M + n_{M,i} \leq (2 + 8c)M$. Thus $|C| \geq \frac{4M^3}{|A|\cdot|B|} \geq \frac{4M^3}{((2+8c)M)^2} = \frac{M}{(1+4c)^2}$. Now suppose we run the algorithm with a cache of size $M' \leq 0.5M/(1+4c)^2$. Each of these $n_{\leq}$ arithmetic segments will update $M/(1+4c)^2$ entries of C. This will cause at least $0.5M/(1+4c)^2$ writes to slow memory, for a total number of writes of at least

$$n_{\leq} \cdot \frac{0.5M}{(1+4c)^2} \geq \frac{n_{as}}{2} \cdot \frac{0.5M}{(1+4c)^2}$$
$$= \frac{0.25}{(1+4c)^2} M\lfloor |S|/(2M^{3/2})\rfloor = \Omega\left(\frac{|S|}{M^{1/2}}\right).$$

■

Corollary 3: Any CO algorithm satisfying the hypothesis in Theorem 2 cannot be WA in the following sense: For all fast memory sizes M, the number of writes to slow memory is at least

$$W_s \geq \frac{M}{2}\lfloor |S|/(4\sqrt{2}(1+4c)^3 M^{3/2})\rfloor = \Omega(\frac{|S|}{M^{1/2}}), \quad (2)$$

which is asymptotically the same as the number of reads it performs.

Proof: Denote the M in Corollary 3 by $\hat{M}$, and then apply Theorem 2 with $M' = \hat{M}$ and $M = 2(1+4c)^2\hat{M}$, so the lower bound in (1) becomes the one in (2). ■

Our proof technique applies more generally to the **broad class of algorithms – nested loops that access arrays – considered in** [9]. The formulation of Theorem 2 will change because the exponents $3/2$ and $1/2$ will vary depending on the algorithm, and which arrays are read and written.

VI. Cache Replacement Policy and Hardware Counter Measurements

While the algorithms describe in Section IV explicitly control movement of data between slow and fast memories, most hardware platforms do not expose such low level control to the programmer. Instead, variables are mapped to virtual memory addresses, which are in turn mapped to hardware locations by a cache replacement policy. The Least Recently Used (LRU) policy is a popular candidate because of the robust theoretical guarantees on its performance in an online setting [32], and its suitability for multi-level memory hierarchies [23]. On a memory (or cache) of size M, it maintains a list of M distinct virtual memory addresses most recently accessed by a sequence of instructions and keeps a copy of them in the memory, ready for future instructions in the sequence. If the next instruction accesses an address not on the list, the required location is brought in, and to make space, the copy corresponding to the last address on the LRU list is discarded, which if modified, is written back to the slow memory.

In Algorithms 1 and 2, three blocks of size $M/3$ each are explicitly moved to the fast memory and retained there while instructions involving them are completed. What happens

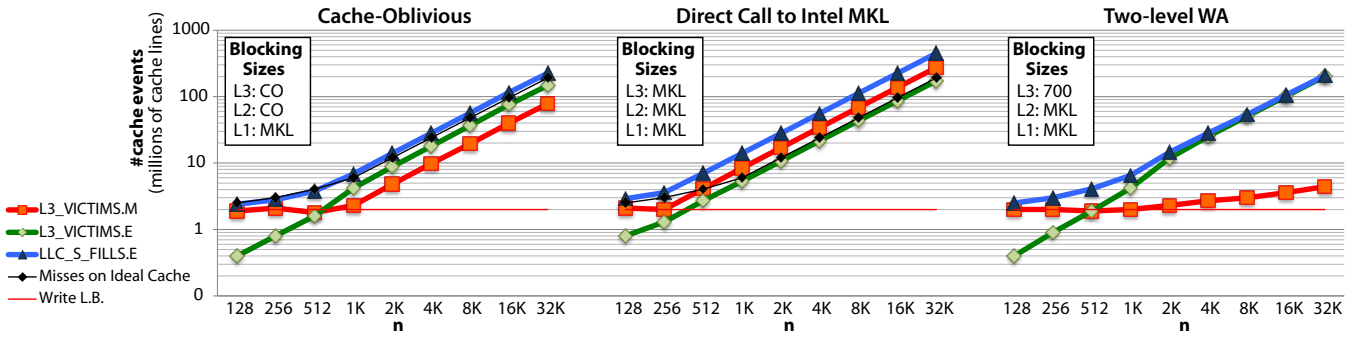

Figure 3: L3 cache counter measurements of various schedules of double precision MM $C^{m\times l} += A^{m\times n} \cdot B^{n\times l}$ on Intel Xeon 7560 with $m = l = 4000$. The x-axis represents n. The y-axis represents cache events in millions of cache lines, each 64 Bytes in size. "Two-level WA" attempts to minimize write-backs from L3 to DRAM but not between L1, L2, and L3.

if the same sequence of instructions is executed with the addresses mapped to fast memory with an LRU policy under the covers? The WA properties can be preserved by slightly adjusting the block size.

Proposition 2: Suppose that the two-level WA schedule for MM ($C^{m\times l} = A^{m\times n} \cdot B^{n\times l}$) in Algorithm 1 is executed on a sequential machine with a two-level memory hierarchy whose fast memory has size $M \ll ml$, uses the LRU replacement policy, and is fully associative. If 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 is ml, irrespective of the order of instructions within the block multiplication (line 7 of Alg. 1).

This proposition supposes no change to the LRU policy, in contrast to the modified LRU policy required to support the the Asymmetric Ideal-Cache model in [20].

The idea behind the proof is that while a column corresponding to a block of C is being executed, the C-block is frequently accessed keeping it high in LRU priority. No variable in the C-block falls below LRU priority $5b^2$ and, once loaded, remains in fast memory until the column is completely executed. This is because in each block of the iteration space, one $b \times b$ block of variables each from arrays A, B and C are used, and in the next block of the iteration space, new blocks of A and B are used while reusing the C-block (see Figure 4). In the technical report, we design block matrix multiplication (line 7 in Alg. 1) so that LRU avoids writes when b is such that three $b \times b$ blocks fit in the fast memory.

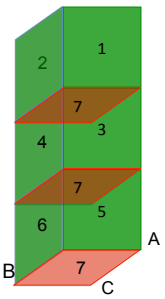

Figure 4: A column perpendicular to the C-block 7.

We support these claims with hardware measurements of cache event counters (Figure 3) for three MM schedules – cache-oblivious [23], Intel MKL `dgemm`, and the WA schedule in Alg. 1 – on an Intel Nehalem-EX 7560 processor. We track the traffic between L3 (fast memory) and DRAM (slow memory), counting the number of writebacks to slow memory using `L3_VICTIMS.M`, the number of writes from slow to fast memory with `LLC_S_FILLS.E` and the number of evictions from fast memory using `L3_VICTIMS.E` [33]. We fix the output at $4K \times 4K$ (2×10^6 cache lines on this machine), and vary the other dimension n between 2^7 and 2^{12}. Predictably, the number of loads increases linearly with n in all three schedules. In the first two schedules, writebacks to slow memory also increase proportionally, whereas in the WA schedule, they remain very close to the lower bound, i.e., the size of the output. We speculate that the small gap between the writebacks in WA schedule and the lower bound arises because the machine implements an approximation to the LRU policy [34], [35] and has limited associativity [36].

Finally, LRU replacement policy also works for other algorithms in this paper.

Proposition 3: If the two-level WA TRSM (Algorithm 2 with $n \times n \times m$ input size), Cholesky factorization (Algorithm 3 with $n \times n$ input size) and direct N-body algorithm (with N input size; see technical report) 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 are smaller than the fast memory ($5b^2 * \text{sz(elements)} + 1 \leq M$), 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.

VII. Parallel WA Algorithms

There is a large literature on CA distributed memory parallel algorithms (see [4], [9], [37], [38] and the references therein), and in this section we focus on classical dense linear algebra, including MM, LU factorization and similar operations. 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). So in this model, a read from one processor's local memory is necessarily a write in another processor's local memory.

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. We assume that the processors are homogeneous, that each one has its own (identical) memory hierarchy and we define three models:

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 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.

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, respectively. 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, the simplest. The output size is $W_1 = n^2/P$, a lower bound on the number of writes to L_2. The results in [22] 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 $cn^2/P \le M_2$). And [22] together with Proposition 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. 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. 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.

Now consider Model 2.1. Here the question becomes whether we can exploit the additional (slow) memory NVM to go faster. There is a class of algorithms that may do this, including for linear algebra (see [4], [10], [39], [38] and the references therein), that replicate the input data to avoid (much more) subsequent interprocessor communication. In the case of matrix multiplication, the 2.5D algorithm replicates the data $c \ge 1$ 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 accessing NVM. We consider two algorithms. The first one, referred to as 2.5DMML2, replicates the data $1 \le c_2 < P^{1/3}$ times, using only L_2. The second one, referred to as 2.5DMML3, replicates the data c_3 times, using L_3, where $c_2 < c_3 \le P^{1/3}$. A detailed analysis of these algorithms can be found in the technical report. Let β_{NW} be the time per word to send data over the network, 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 $\beta_{23} \gg \beta_{32}$. In summary, with this notation we can say that the ratio of the dominant bandwidth costs of these algorithms 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.

In the third scenario, Model 2.2, we assume that the data does not fit in DRAM, so we need to use NVM. We have two communication lower bounds to try to attain, on interprocessor communication and on writes to NVM. In Theorem 3 we prove this is impossible, that any algorithm must asymptotically exceed at least one of these lower bounds. A lower bound on writes to L_3 (NVM) is $W_1 = n^2/P$ (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", [22] and Proposition 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.

Theorem 3: Assume $n \gg \sqrt{P} \gg 1$ and $n^2/P \gg M_2$. For the MM algorithm, 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 [31], the number of multiplications performed by a processor satisfies $n^3/P \le \sqrt{|A| \cdot |B| \cdot |C|}$, where $|A|$ is the number of distinct entries of A available in L_2 sometime during execution (and similarly for $|B|$ and

$|C|)$. Thus $n^3/P \leq \max(|A|,|B|,|C|)^{3/2}$ or $n^2/P^{2/3} \leq \max(|A|,|B|,|C|)$. $n^2/P^{2/3}$ is asymptotically larger than 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. But this asymptotically exceeds W_1. ∎

In the technical report, we present two algorithms for matrix multiplication (as well as LU factorization without pivoting), each of which attains one of these 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 $dom\beta cost(\text{2.5DMML3ooL2}) = \frac{\beta_{NW} n^2}{\sqrt{P} c_3} + \frac{\beta_{23} n^2}{\sqrt{P} c_3} + \frac{\beta_{32} n^3}{P\sqrt{M_2}}$, $dom\beta cost(\text{SUMMAL3ooL2}) = \frac{\beta_{NW} n^3}{P\sqrt{M_2}} + \frac{\beta_{23} n^2}{P} + \frac{\beta_{32} n^3}{P\sqrt{M_2}}$, which may again be easily compared given the algorithm and hardware parameters.

VIII. Krylov Subspace Methods

Krylov subspace methods (KSMs) are a family of algorithms to solve linear systems and eigenproblems. Communication-avoiding, or s-step Krylov subspace methods (CA-KSMs) are variants of classical KSMs that produce the same iterates in exact arithmetic, but can reduce data movement in the computation of basis vectors and orthogonalization operations; see, e.g., [40].

The main transformation in CA-KSMs is splitting the KSM iteration loop into an outer loop, which constructs $O(s)$-dimensional Krylov bases and performs block orthogonalization of the basis vectors, and an inner loop, which performs s iterations, updating the coordinates of the vectors in the generated bases. The bases can be computed in a blocked manner using a "matrix powers" optimization which exploits temporal locality, and orthogonalizations can be computed by matrix multiplication [40]. While these optimizations reduce communication, they do not reduce L_2 writes: if the matrix dimension n is sufficiently large with respect to M_1, N/s outer iterations incur $O(Nn)$ L_2 writes, attaining the lower bound for N iterations.

However, writes can be avoided by exploiting a "streaming matrix powers" optimization. The idea is to interleave blockwise orthogonalization with blockwise matrix powers computations, each time discarding the entries of the basis vectors from fast memory after they have been multiplied and accumulated. If the matrix powers optimization reduces L_2 reads of vector entries by a factor of $f(s)$, then the streaming matrix powers optimization reduces L_2 writes by $\Theta(f(s))$, at the cost of doubling the number of operations to compute the bases. In cases where $f(s) = \Theta(s)$, like for a $(2b+1)^d$-point stencil on a sufficiently large d-dimensional Cartesian mesh when $s = \Theta(M_1^{1/d}/b)$, still assuming n is sufficiently large with respect to M_1, N/s outer iterations of a CA-KSM incur $O(Nn/s)$ L_2 writes, an s-fold reduction compared to N iterations of a classical KSM.

The preceding loop transformation applies to both Lanczos-based and Arnoldi-based KSMs. The streaming matrix powers optimization was proposed earlier in [40, Sec. 6.3] and discussed again in [4, Sec. 7.2.4]; these works considered minimizing both reads and writes and only suggested this optimization when the matrix's memory footprint (e.g., number of nonzeros) is sufficiently small with respect to n and s. The contribution here is recognizing that the matrix's memory footprint is irrelevant when minimizing just writes, thus this optimization applies more generally.

IX. 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 showed that this not the case for some "fast" algorithms. For some classical linear algebra versions and Krylov subspace methods, we designed sequential versions that minimize the number of reads and writes on two-level memories. We also presented parallel versions of these algorithms for different memory and network configurations in parallel machines that represent different points in the tradeoff between writes to nonvolatile memory and communication over the network. We proved that cache-oblivious versions of some algorithms cannot minimize writes.

X. Acknowledgments

We thank Edgar Solomonik for critical comments on the statement and proof of Theorem 2. We thank Yuan Tang for pointing out the role of write-buffers. We thank Guy Blelloch and Phillip Gibbons for access to the machine on which we ran our experiments. 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, the Minerva Foundation, and the Fulbright Program. This research was supported in part by the Intel Collaborative Research Institute for Computational Intelligence (ICRI-CI). This research was supported by a grant from the United States-Israel Binational Science Foundation (BSF), Jerusalem, Israel.

REFERENCES

[1] E. Carson, J. Demmel, L. Grigori, N. Knight, P. Koanantakool, O. Schwartz, and H. V. Simhadri, "Write-avoiding algorithms," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2015-163, Jun 2015. [Online]. Available: http://www.eecs.berkeley.edu/Pubs/TechRpts/2015/EECS-2015-163.html

[2] N. R. C. 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.

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

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

[5] G. Blelloch, P. Gibbons, and H. Simhadri, "Low depth cache-oblivious algorithms," in *22nd ACM SPAA*, 2010.

[6] A. Aggarwal and S. Vitter, Jeffrey, "The input/output complexity of sorting and related problems," *Commun. ACM*, vol. 31, no. 9, pp. 1116–1127, Sep. 1988.

[7] X. Hong and H. T. Kung, "I/O complexity: the red blue pebble game," in *Proc. 13th Symp. Theor. Comput.*. ACM, 1981, pp. 326–334.

[8] G. Ballard, J. Demmel, O. Holtz, and O. Schwartz, "Graph expansion and communication costs of fast matrix multiplication," *JACM*, vol. 59, no. 6, Dec 2012.

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

[10] D. Irony, S. Toledo, and A. Tiskin, "Communication lower bounds for distributed-memory matrix multiplication," *J. Parallel Distrib. Comput.*, vol. 64, no. 9, pp. 1017–1026, 2004.

[11] G. W. Burr, M. J. Breitwisch, M. Franceschini, D. Garetto, K. Gopalakrishnan, B. Jackson, B. Kurdi, C. Lam, L. A. Lastras, A. Padilla, B. Rajendran, S. Raoux, and R. S. Shenoy, "Phase change memory technology," *J. Vacuum Science and Technology B*, vol. 28, no. 2, pp. 223–262, 2010.

[12] D. Li, J. S. Vetter, G. Marin, C. McCurdy, C. Cira, Z. Liu, and W. Yu, "Identifying opportunities for byte-addressable non-volatile memory in extreme-scale scientific applications," in *Proceedings of the 2012 IEEE 26th Internat. Parallel Distrib. Process. Symp.*, ser. IPDPS '12. Washington, DC, USA: IEEE Computer Society, 2012, pp. 945–956.

[13] K. Asanović, "Firebox: A hardware building block for 2020 warehouse-scale computers," https://www.usenix.org/sites/default/files/conference/protected-files/fast14_asanovic.pdf, 2014, keynote address at the 12th USENIX Conference on File and Storage Technologies (FAST'14).

[14] I. Koltsidas, P. Mueller, R. Pletka, T. Weigold, E. Eleftheriou, M. Varsamou, A. Ntalla, E. Bougioukou, A. Palli, and T. Antonakopoulos, "PSS: A prototype storage subsystem based on PCM", 2014, 5th Non-Volatile Memories Workshop.

[15] B. C. Lee, E. Ipek, O. Mutlu, and D. Burger, "Architecting phase change memory as a scalable dram alternative," in *Proc. 36th Annu. Internat. Symp. Comput. Architecture*, ser. ISCA '09. New York, NY, USA: ACM, 2009, pp. 2–13.

[16] N. Gilbert, Y. Zhang, J. Dinh, B. Calhoun, and S. Hollmer, "A 0.6v 8 pj/write non-volatile cbram macro embedded in a body sensor node for ultra low energy applications," in *2013 Symposium on VLSI Circuits*, June 2013, pp. C204–C205.

[17] A. Technologies, "How RRAM is changing the landscape of wearable and IOT applications," http://www.flashmemorysummit.com/English/Collaterals/Proceedings/2014/20140806_203A_Naveh.pdf, 2014, Flash Memory Summit.

[18] B. Rajendran, http://www.itrs.net/itwg/beyond_cmos/2010Memory_April/Proponent/Nanowire\%20PCRAM.pdf, 2010, iTRS Workshop Maturity Evaluation for Selected Emerging Research Memory Technologies.

[19] F. Xia, D.-J. Jiang, J. Xiong, and N.-H. Sun, "A survey of phase change memory systems," *J. Comput. Sci. Tech.*, vol. 30, no. 1, pp. 121–144, 2015.

[20] G. Blelloch, J. Fineman, P. Gibbons, Y. Gu, and J. Shun, "Sorting with asymmetric read and write costs," in ACM SPAA, June 2015.

[21] 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. Comput.*, vol. 63, no. 8, Aug 2014.

[22] G. Ballard, J. Demmel, O. Holtz, and O. Schwartz, "Minimizing communication in numerical linear algebra," *SIAM J. Mat. Anal. Appl.*, vol. 32, no. 3, pp. 866–901, 2011.

[23] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran, "Cache-oblivious algorithms," in *Proc. 40th IEEE Symp. Found. Comput. Sci. (FOCS 99)*, 1999, pp. 285–297.

[24] E. Gal and S. Toledo, "Algorithms and data structures for flash memories," *ACM Comput. Surv.*, vol. 37, no. 2, pp. 138–163, Jun. 2005.

[25] D. Ajwani, I. Malinger, U. Meyer, and S. Toledo, "Characterizing the performance of flash memory storage devices and its impact on algorithm design," in *Proc. 7th Internat. Conf. Exp. Algorithms*, ser. WEA'08. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 208–219.

[26] Micron Technologies, Inc., "Wear-leveling techniques in NAND flash devices," Tech. Rep. TN-29-42, 2008.

[27] A. Ben-Aroya and S. Toledo, "Competitive analysis of flash memory algorithms," *ACM Trans. Algorithms*, vol. 7, no. 2, pp. 23:1–23:37, Mar. 2011.

[28] M. K. Qureshi, J. Karidis, M. Franceschini, V. Srinivasan, L. Lastras, and B. Abali, "Enhancing lifetime and security of PCM-based main memory with start-gap wear leveling," in *Proc. 42nd Annu. IEEE/ACM Internat. Symp. on Microarchitecture*, ser. MICRO 42. New York, NY, USA: ACM, 2009, pp. 14–23.

[29] J. Coburn, A. M. Caulfield, A. Akel, L. M. Grupp, R. K. Gupta, R. Jhala, and S. Swanson, "Nv-heaps: Making persistent objects fast and safe with next-generation, non-volatile memories," in *Proc. 16th Internat. Conf. on Architectural Support for Programming Lang. and Operating*

Syst., ser. ASPLOS XVI. New York, NY, USA: ACM, 2011, pp. 105–118.

[30] G. Ballard, J. Demmel, O. Holtz, B. Lipshitz, and O. Schwartz, "Graph expansion analysis for communication costs of fast rectangular matrix multiplication," in *Proc. 1st Mediterranean Conference on Algorithms (MedAlg)*, 2012, pp. 13–36.

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

[32] D. D. Sleator and R. E. Tarjan, "Amortized efficiency of list update and paging rules," *CACM*, vol. 28, no. 2, 1985.

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

[34] D. Eklov, N. Nikoleris, D. Black-Schaffer, and E. Hagersten, "Cache pirating: Measuring the curse of the shared cache," in *Proc. 2011 Internat. Conf. Parallel Process.*, ser. ICPP '11, pp. 165–175.

[35] Intel_Karla, "Cache replacement policy for Nehalem/SNB/IB?" https://communities.intel.com/thread/32695, November 2012.

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

[37] M. Driscoll, E. Georganas, P. Koanantakool, E. Solomonik, and K. Yelick, "A communication-optimal n-body algorithm for direct interactions," in *Proc. 27th Internat. IEEE Symp. Parallel Distrib. Proc. (IPDPS)*, 2013, pp. 1075–1084.

[38] A. Tiskin, "Communication-efficient parallel generic pairwise elimination," *Future Generation Comput. Syst.*, vol. 23, no. 2, pp. 179–188, 2007.

[39] E. Solomonik and J. Demmel, "Communication-optimal parallel 2.5D matrix multiplication and LU factorization algorithms," in *Euro-Par 2011 Parallel Process.*, ser. Lecture Notes in Computer Science, E. Jeannot, R. Namyst, and J. Roman, Eds. Springer Berlin Heidelberg, 2011, vol. 6853, pp. 90–109.

[40] E. Carson, N. Knight, and J. Demmel, "Avoiding communication in nonsymmetric Lanczos-based Krylov subspace methods," *SIAM J. Sci. Comput.*, vol. 35, no. 5, pp. S42–S61, 2013.