Minimizing Communication in Numerical Linear Algebra

Grey Ballard
James Demmel
Olga Holtz
Oded Schwartz

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2011-15
http://www.eecs.berkeley.edu/Pubs/TechRpts/2011/EECS-2011-15.html

February 28, 2011

MINIMIZING COMMUNICATION IN NUMERICAL LINEAR ALGEBRA

GREY BALLARD [*], JAMES DEMMEL [†], OLGA HOLTZ [‡], AND ODED SCHWARTZ [§]

Abstract. In 1981 Hong and Kung proved a lower bound on the amount of communication (amount of data moved between a small, fast memory and large, slow memory) needed to perform dense, n-by-n matrix-multiplication using the conventional $O(n^3)$ algorithm, where the input matrices were too large to fit in the small, fast memory. In 2004 Irony, Toledo and Tiskin gave a new proof of this result and extended it to the parallel case (where communication means the amount of data moved between processors). In both cases the lower bound may be expressed as Ω(#arithmetic operations / $\sqrt{M}$), where M is the size of the fast memory (or local memory in the parallel case).

Here we generalize these results to a much wider variety of algorithms, including LU factorization, Cholesky factorization, LDL^T factorization, QR factorization, Gram–Schmidt algorithm, algorithms for eigenvalues and singular values, i.e., essentially all direct methods of linear algebra.

The proof works for dense or sparse matrices, and for sequential or parallel algorithms. In addition to lower bounds on the amount of data moved (bandwidth-cost), we get lower bounds on the number of messages required to move it (latency-cost).

We extend our lower bound technique to compositions of linear algebra operations (like computing powers of a matrix), to decide whether it is enough to call a sequence of simpler optimal algorithms (like matrix multiplication) to minimize communication, or if we can do better. We give examples of both. We also show how to extend our lower bounds to certain graph theoretic problems.

We point out recently designed algorithms that attain many of these lower bounds.

Key words. Linear algebra algorithms, bandwidth, latency, communication avoiding, lower bound.

1. Introduction.

Algorithms have two kinds of costs: arithmetic and communication. By communication we mean either moving data between levels of a memory hierarchy (in the sequential case) or over a network connecting processors (in the parallel case). There are two costs associated with communication: *bandwidth-cost* (proportional to the total number of words of data moved) and *latency-cost* (proportional to the number of messages in which these words are packed and sent). For example, we may model the cost of sending m words in a single message as $\alpha + \beta m$, where α is the latency (measured in seconds) and β is the reciprocal bandwidth (measured in seconds per word). Depending on the technology, either latency or bandwidth costs may be larger, often dominating the cost of arithmetic. So it is of interest to have algorithms minimizing both bandwidth-cost and latency-cost.

In this paper we prove a general lower bound on the amount of data moved (i.e., bandwidth-cost) by a general class of algorithms, including most dense and sparse linear algebra algorithms, as well as some graph theoretical algorithms. A similar model was discussed by Hong and Kung [HK81]. They show that to multiply two

[*]Computer Science Department, University of California, Berkeley, CA 94720. Research supported by Microsoft (Award #024263) and Intel (Award #024894) funding and by matching funding by U.C. Discovery (Award #DIG07-10227). (ballard@eecs.berkeley.edu).

[†]Mathematics Department and CS Division, University of California, Berkeley, CA 94720. This material is based on work supported by U.S. Department of Energy grants under Grant Numbers DE-SC0003959, DE-SC0004938, and DE-FC02-06-ER25786, as well as Lawrence Berkeley National Laboratory Contract DE-AC02-05CH11231. (demmel@cs.berkeley.edu).

[‡]Departments of Mathematics, University of California, Berkeley and Technische Universität Berlin. O. Holtz acknowledges support of the Sofja Kovalevskaja program of Alexander von Humboldt Foundation. (holtz@math.berkeley.edu).

[§] The Weizmann Institute of Science, Rehovot 76100, Israel. This work was done while at the Departments of Mathematics, Technische Universität Berlin, and while visiting University of California, Berkeley. (oded.schwartz@weizmann.ac.il).

dense n-by-n matrices, using the conventional $\Theta(n^3)$ algorithm, on a machine with a large slow memory (in which the matrices initially reside) and a small fast memory of size M (too small to store the matrices, but arithmetic may only be done on data in fast memory), $\Omega(n^3/\sqrt{M})$ words of data must be moved between fast and slow memory. This lower bound is attained by a variety of "blocked" algorithms. This lower bound may also be expressed as $\Omega(\#\text{arithmetic\_operations} / \sqrt{M})$ [1].

This result was proven differently by Irony, Toledo and Tiskin [ITT04] and generalized to the parallel case, where P processors multiply two n-by-n matrices. In the "memory-scalable" case, where each processor stores the minimal $M = O(n^2/P)$ words of data, they obtain the lower bound:
$\Omega(\#\text{arithmetic\_operations\_per\_processor} / \sqrt{\text{memory\_per\_processor}}) = \Omega(\frac{n^3/P}{\sqrt{n^2/P}}) = \Omega(\frac{n^2}{\sqrt{P}})$, which is attained by Cannon's algorithm [Can69] [Dem96, Lecture 11]. The paper [ITT04] also considers the so-called "3D" case, which does less communication by replicating the matrices and using $O(P^{1/3})$ times as much memory as the minimal possible.

Here we begin with the proof in [ITT04], which starts with the sum $C_{ij} = \sum_k A_{ik} \cdot B_{kj}$, and uses a geometric argument on the lattice of indices (i, j, k) to bound the number of updates $C_{ij} := C_{ij} + A_{ik} \cdot B_{kj}$ that can be performed when a subset of matrix entries are in fast memory. This proof generalizes in a number of ways: in particular it does not depend on the matrices being dense, or the output being distinct from the input. These observations let us state and prove a general Theorem 2.2 in Section 2, that a lower bound on the number of words moved into or out of a fast or local memory of size M is $\Omega(\#\text{arithmetic operations} / \sqrt{M})$. This applies to both the sequential case (where M is a fast memory) and the parallel case (where M is each processor's local memory); in the parallel case further assumptions about whether the algorithm is memory or load balanced (to estimate the effective M and #arithmetic operations) are needed to get a lower bound on the overall algorithm.

Corollary 2.3 of Theorem 2.2 provides a simple lower bound on latency-cost (just the lower bound on bandwidth-cost divided by the largest possible message size, namely the memory size M). Both bandwidth-cost and latency-cost lower bounds apply straightforwardly to a nested memory hierarchy with more than two layers, bounding from below the communication between any adjacent layers in the hierarchy [Sav95, BDHS10a].

In Section 3, we present simple corollaries applying Theorem 2.2 to conventional (non-Strassen-like) implementations of matrix multiplication and other BLAS operations [BDD$^+$02, BDD$^+$01] (dense or sparse), LU factorization, Cholesky factorization and LDL^T factorization, where D is either real diagonal matrix, or block-diagonal matrix, i.e., Bunch-Kaufman [BK77] type factorization. These factorizations may also be dense or sparse, with any kind of pivoting, and be exact or "incomplete", e.g., ILU [Saa96] (for dense matrices some of these results can be also obtained by suitable reductions from [HK81] or [ITT04], and we point these out). We also introduce a technique to extend these lower bounds to cases like computing $\|A \cdot B\|_F$, so the output is a single scalar, and where each $A(i, j)$ and $B(j, k)$ is given by an explicit formula, so there are no inputs to read from memory (we will require that each explicit formula

[1] The sequential communication model used here is sometimes called the *two-level I/O model* or *disk access machine (DAM)* model (see [AV88], [BBF$^+$07], [CR06]). Our model follows that of [HK81] and [ITT04] in that it assumes the block-transfer size is one word of data ($B = 1$ in the common notation).

is evaluated at most once).

Section 4 considers lower bounds for algorithms that involve orthogonal factorizations. This class includes the QR factorization, the standard algorithms for eigenvalues and eigenvectors, and the singular value decomposition (SVD). After dealing with the easier case of Gram-Schmidt in Section 4.1, Section 4.2 considers the harder case of algorithms that apply sequences of orthogonal transformations. For reasons explained there, the counting techniques of [HK81] and [ITT04] do not directly apply, so we need a different but related lower bound argument. Our proofs require some technical assumptions that we conjecture could be removed. Finally, Section 4.3 extends the lower bounds to eigenvalue and singular value problems.

Section 5 shows how to extend our lower bounds to more general computations where we compose a sequence of simpler linear algebra operations (like matrix multiplication, LU decomposition, etc.), so the outputs of one operation may be inputs to later ones. If these intermediate results do not need to be saved in slow memory, or if some inputs are given by formulas (like $A(i,j) = 1/(i+j)$) and so do not need to be fetched from memory, or if the final output is just a scalar (the norm or determinant of a matrix), then it is natural to ask whether there is a better algorithm than just using optimized versions of each operation in the sequence. We give examples where this simple approach is optimal and when it is not. We also exploit the natural correspondence between matrices and graphs to derive communication lower bounds for certain graph algorithms, like All-Pairs-Shortest-Path.

Finally, Section 6 discusses attainability of these lower bounds, and open problems. Briefly, in the dense case all the lower bounds are attainable (in the parallel case, this is modulo polylog$P$ factors, and assuming the minimal $O(n^2/P)$ storage per processor); see Tables 6.1 and 6.2 (some of these algorithms are also pointed out in sections 3 and 4.2). The optimal algorithms for square matrix multiplication are well known, as mentioned above. Optimal algorithms for dense LU, Cholesky, QR, eigenvalue problems and the SVD are more recent and not part of standard libraries like LAPACK [ABB$^+$92] and ScaLAPACK [BCC$^+$97]. Several of these references describe prototypes of the new algorithms that attain large speedups over standard libraries. Beyond the BLAS, only in the case of Cholesky do we know of a sequential algorithm that does as few flops as the conventional algorithm (modulo lower order terms) as well as achieving both minimal bandwidth-cost and latency-cost across arbitrary levels of memory hierarchy. Beyond Cholesky [BDHS10a, DDGP10] and the BLAS, no optimal algorithm is known for architectures mixing parallelism and multiple memory hierarchies, i.e., most real architectures (but some lower bounds for specific architecture/algorithm combinations do exist, see for example [Saa86]). "3D" algorithms, that use multiple copies of the data in order to communicate less than "2D" algorithms using minimal total memory, were obtained in [IT02, Ash91, Ash93, SD11], and are discussed in Section 6. Communication optimal algorithms for sparse matrices are known only for sparse Cholesky [DDGP10]. For highly rectangular dense matrices (e.g., matrix-vector multiplication) or for sufficiently sparse matrices, our new lower bound is sometimes lower than the trivial lower bound (#inputs + #outputs) and therefore not always attainable.

2. First Lower Bound.

We first define our model of computation formally, and illustrate it on the simplest case of dense matrix multiplication.

We work with $n \times n$ matrices, so we define $V = \{1, 2, ...n\}$ to be the index set for the rows and columns. Let $S_a \subseteq V \times V$ be the subset of entries of the indices of the input matrix A that are read by the algorithm (e.g., the indices of the non-zeros entries of a sparse matrix). Let $a : S_a \mapsto \mathcal{M}$ be a mapping from the matrix entries to locations in memory (on a parallel machine $\mathcal{M}$ refers to a location in some processor's memory; the processor number is implicit). The map is one-to-one. Similarly define S_b, S_c and $b(\cdot,\cdot), c(\cdot,\cdot)$ for the matrices B and C. Note that the ranges of a, b and c are not necessarily disjoint. The value of a memory location l is denoted by $Mem(l)$.

Now let f_{ij} and g_{ijk} be "nontrivial" functions in a sense we make clear below. The computation we want to perform is for all $(i,j) \in S_c$:

$$Mem(c(i,j)) = f_{ij}(g_{ijk}(Mem(a(i,k)), Mem(b(k,j))) \text{ for } k \in S_{ij}, \text{any other arguments}) \tag{2.1}$$

Here f_{ij} depends nontrivially on its arguments $g_{ijk}(\cdot,\cdot)$ which in turn depend nontrivially on their arguments $Mem(a(i,k))$ and $Mem(b(k,j))$, in the following sense: we need at least one word of space to compute f_{ij} (which may or may not be $Mem(c(i,j))$) to act as "accumulator" of the value of f_{ij}, and we need the values $Mem(a(i,k))$ and $Mem(b(k,j))$ in fast memory before evaluating g_{ijk}. Note also that we may not know until after the computation what S_C, f_{ij}, S_{ij}, g_{ijk} or "any other arguments" were, since they may be determined on the fly (e.g., pivot order).

Now we illustrate the model in Equation (2.1) by applying it to sequential dense n-by-n matrix multiplication $C = A \cdot B$, where A, B and C are stored column-wise in memory: We take S_c as all pairs (i,j) with $0 \le i,j < n$ with $C(i,j)$ stored in location $c(i,j) = i + j \cdot n$. $A(i,k)$ is analogously stored at location $a(i,k) = i + k \cdot n$ and $B(k,j)$ is stored at location $b(k,j) = k + j \cdot n$. The set $S_{ij} = \{0, 1, ..., n-1\}$ for all (i,j). Operation g_{ijk} is scalar multiplication, and f_{ij} computes the sum of its n arguments.

The question is how many slow memory references are required to perform this computation, when all we are allowed to do is compute the g_{ijk} in a different order, and compute and store the f_{ij} is a different order. This appears to restrict possible reorderings to those where f_{ij} is computed correctly, since we are not assuming it is an associative or commutative function, or those reorderings that avoid races because some $c(i,j)$ may be used later as inputs. But there is no need for such restrictions: the lower bound applies to all reorderings, correct or incorrect, yielding the same bound in both cases.

Using only structural information, e.g., about the sparsity patterns of the matrices, we can sometimes deduce that the computed result $f_{ij}(\cdot)$ is exactly zero, to possibly avoid a memory reference to store the result at $c(i,j)$. Section 3.2.1 discusses this possibility more carefully, and shows how to carefully count operations to preserve the validity of our lower bounds.

The argument, following [ITT04], is:

- Break the stream of instructions executed into segments, where each segment contains exactly M load and store instructions (i.e., that cause communication), where M is the fast (or local) memory size.
- Bound from above the number of evaluations of functions g_{ijk} that can be performed during any segment, calling this upper bound F.
- Bound from below the number of (complete) segments by the total number of evaluations of g_{ijk} (call it G) divided by F, i.e., $\lfloor G/F \rfloor$.
- Bound from below the total number of loads and stores, by M (load/stores per segment) times the minimum number of complete segments, $\lfloor G/F \rfloor$, so it is at least $M \cdot \lfloor G/F \rfloor$.

Now we compute the upper bound F using a geometric theorem of Loomis and Whitney [LW49, BZ88]. We need only the simplest version of their result here[2]

LEMMA 2.1. *[LW49, BZ88]. Let V be a finite set of lattice points in $\mathbf{R}^3$, i.e., points (x, y, z) with integer coordinates. Let V_x be the projection of V in the x-direction, i.e., all points (y, z) such that there exists an x so that $(x, y, z) \in V$. Define V_y and V_z similarly. Let $|\cdot|$ denote the cardinality of a set. Then $|V| \leq \sqrt{|V_x| \times |V_y| \times |V_z|}$.*

To see the relationship of this geometric result to our model in Equation (2.1), see Figure 2.1, shown for the special case of n-by-n matrix multiplication, for $n = 3$. We model the computation as an n-by-n-by-n set of lattice points, drawn as a set of n^3 1-by-1-by-1 cubes for easier labeling: each 1-by-1-by-1 cube represents the lattice point at its bottom front right corner. The cubes (or lattice points) are indexed from corner $(i, j, k) = (0, 0, 0)$ to $(n-1, n-1, n-1)$. Cube (i, j, k) represents the multiplication $A(i, k) \cdot B(k, j)$ and its accumulation into $C(i, j)$. The 1-by-1 squares on the top face of the cube, indexed by (i, j), represent $C(i, j)$, and the 1-by-1 squares on the other two faces represent $A(i, k)$ and $B(k, j)$, respectively. The set of all multiplications performed during a segment are some subset (V in Lemma 2.1) of all the cubes. All the $C(i, j)$ needed to store the results are the projections of these cubes onto the "C-face" of the cube (V_z in Lemma 2.1). Similarly, the $A(i, k)$ needed as arguments are the projections onto the "A-face" (V_y in Lemma 2.1), and the $B(k, j)$ are the projections onto the "B-face" (V_x in Lemma 2.1).

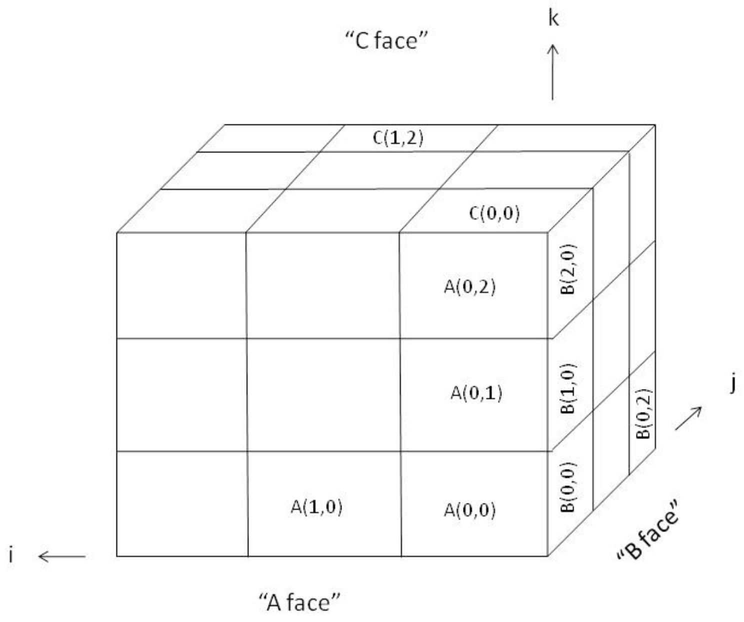


FIG. 2.1. *Geometric Model of Matrix Multiplication*

Now we must bound the maximum number of possibly different $Mem(c(i, j))$ (or corresponding "accumulators"), $Mem(a(i, k))$, and $Mem(b(k, j))$ that can reside in

[2]An intuition for the correctness of this special case of Loomis and Whitney bound is as follows: think of a box of dimensions $a \times b \times c$. Then its (rectangular) projections on the three planes have areas $a \cdot b$, $b \cdot c$ and $a \cdot c$, and we have that its volume $a \cdot b \cdot c$ is equal to the square root of the product of the three areas.

fast memory during a segment. Since we want to accommodate the most general case where input and output arguments can overlap, we need to use a more complicated model than in [ITT04], where no such overlap was possible. To this end, we consider each input or output operand of (2.1) that appears in fast memory during a segment of M slow memory operations. It may be that an operand appears in fast memory for a while, disappears, and reappears, possibly several times (we assume there is at most one copy at a time in the sequential model and at most one for each processor in the parallel model; this assumption is consistent with obtaining a lower bound). For each period of continuous existence of an operand in fast memory, we label its **Root** (how it came to be in fast memory) and its **Destination** (what happens when it disappears):

- **Root R1:** The operand was already in fast memory at the beginning of the segment, and/or read from slow memory. There are at most $2M$ such operands altogether, because the fast memory has size M, and because a segment contains at most M reads from slow memory.
- **Root R2:** The operand is computed (created) during the segment. Without more information, there is no bound on the number of such operands.
- **Destination D1:** An operand is left in fast memory at the end of the segment (so that it is available at the beginning of the next one), and/or written to slow memory. There are at most $2M$ such operands altogether, again because the fast memory has size M, and because a segment contains at most M writes to slow memory.
- **Destination D2:** An operand is *neither* left in fast memory nor written to slow memory, but simply discarded. Again, without more information, there is no bound on the number of such operands.

We may correspondingly label each period of continuous existence of any operand in fast memory during one segment by one of four possible labels Ri/Dj, indicating the Root and Destination of the operand at the beginning and end of the period. Based on the above description, the total number of operands of all types except R2/D2 is bounded by $4M$ (the maximum number of R1 operands plus the number of D1 operands, an upper bound) [3]. The R2/D2 operands, those created during the segment and then discarded without causing any slow memory traffic, cannot be bounded without further information. For our simplest model, adequate for matrix multiplication, LU decomposition, etc., we have no R2/D2 arguments; they reappear when we analyze the QR decomposition in Section 4.2.

Using the set of lattice points (i, j, k) to represent each function evaluation $g_{ijk}(Mem(a(i, k)), Mem(b(k, j)))$, and assuming there are no R2/D2 arguments, then we can use Lemma 2.1 to bound F: We let V be the set of indices (i, j, k) of the g_{ijk} operations, V_z be the set of indices (i, j) of their destinations $c(i, j)$ with $|V_z| \leq 4M$, V_y be the set of indices (i, k) of their arguments $a(i, k)$ with $|V_y| \leq 4M$, and V_x be the set of indices (j, k) of their arguments $b(j, k)$ with $|V_x| \leq 4M$. Then Lemma 2.1 bounds $F = |V| \leq \sqrt{|V_x| \times |V_y| \times |V_z|} \leq \sqrt{(4M)^3}$. Therefore the total number of loads and stores is bounded by $M\lfloor \frac{G}{F} \rfloor = M\lfloor \frac{G}{\sqrt{(4M)^3}} \rfloor \geq \frac{G}{8\sqrt{M}} - M$. This proves the first lower bound:

THEOREM 2.2. *In the notation defined above, and in particular assuming there are no R2/D2 arguments (created and discarded without causing memory traffic) the number of loads and stores needed to evaluate (2.1) is at least* $\frac{G}{8\sqrt{M}} - M$.

[3]More careful but complicated accounting can reduce this upper bound to $3M$.

We may also write this as $\Omega(\#\text{arithmetic\_operations} / \sqrt{M})$ understanding that we only count arithmetic operations required to evaluate the g_{ijk} for $(i,j) \in S_C$ and $k \in S_{ij}$. We note that a more careful, problem-dependent analysis that depends on how much the three arguments can overlap, may sometimes increase the lower bound by a factor of as much as 8, but for simplicity we omit this.

This lower bound is not always attainable, even for dense matrix multiplication: If the matrices are so small that they all fit in fast memory simultaneously, so $3n^2 \leq M$, then the number of loads and stores may be just $3n^2$, which can be much larger than $n^3/\sqrt{M}$. So a more refined lower bound is $\max(G/(8\sqrt{M}) - M, \#\text{inputs} + \#\text{outputs})$. We generally omit this detail from statements of later corollaries.

Theorem 2.2 is a lower bound on bandwidth-cost, the total number of words communicated. But it immediately provides a lower bound on latency-cost as well, the minimum number of messages that need to be sent, where each message may contain many words.

Corollary 2.3. *In the notation defined above, the number of messages needed to evaluate (2.1) is at least* $G/(8M^{3/2}) - 1 = \#\text{evaluations\_of\_}g_{ijk}/(8M^{3/2}) - 1$.

The proof is simply that the largest possible message size is the fast (or local) memory size M, so we divide the lower bound from Theorem 1 by M.

On a parallel computer it is possible for a processor to pack M words into a single message to be sent to a different processor. But on a sequential computer the words to be sent in a single message must generally be located in contiguous memory locations, which depends on the data structures used. This model is appropriate to capture the behavior of real hardware, e.g., cache lines, memory prefetching, disk accesses, etc. This requirement means that to attain the latency-cost lower bound on a sequential computer, rather different matrix data structures may be required than row-major or column-major [BDHS10a, FLPR99, EGJK04, AGW01, AP00].

Finally, we note that real computers typically don't have just one level of memory hierarchy, but many, each with its own underlying bandwidth and latency costs. So it is of interest to minimize *all* communication, between every pair of adjacent levels of the memory hierarchy. As has been noted before [Sav95, BDHS10a], when the memory hierarchy levels are nested (the L2 cache stores a subset of L3 cache, etc.) we can apply lower bounds like ours at every level in the hierarchy.

3. Consequences for BLAS, LU, Cholesky, and LDL^T.

We now show how Theorem 2.2 applies to a variety of conventional algorithms from numerical linear algebra, by which we mean algorithms that would cost $O(n^3)$ arithmetic operations when applied to dense n-by-n matrices, as opposed to Strassen-like algorithms.

It is natural to ask whether algorithms exist that attain these lower bounds. We point out cases where we know such algorithms exist, which are therefore optimal in the sense of minimizing communication. In the case of dense matrices, many optimal algorithms are known, though not yet in all cases. In the case of sparse matrices, little seems to be known.

3.1. Matrix Multiplication and the BLAS.

We begin with matrix multiplication, on which our model in Equation (2.1) is based:

Corollary 3.1. $G/(8\sqrt{M}) - M$ *is the bandwidth-cost lower bound for multiplying explicitly stored matrices* $C = A \cdot B$ *on a sequential machine, where* G *is the number of multiplications performed in evaluating all the* $C_{ij} = \sum_k A_{ik} \cdot B_{kj}$*, and* M *is the fast memory size. In the special case of multiplying a dense* n*-by-*r *matrix times a dense* r*-by-*m *matrix, this lower bound is* $n \cdot r \cdot m/\sqrt{8M} - M$.

This nearly reproduces a result in [ITT04] for the case of two *distinct, dense* matrices, whereas we need no such assumptions; their bound is $\sqrt{8}$ times larger than ours, but as stated before our bound could be improved by specializing it to this case. We note that this result could have been stated for *sparse* A and B in [HK81]: Combine their Theorem 6.1 (their $\Omega(|V|)$ is the number of multiplications) with their Lemma 6.1 (whose proof does not require A and B to be dense).

As noted in the previous section, an independent lower bound on the bandwidth-cost is simply the total number of inputs that need to be read plus the number of outputs that need to be written. But counting the number of inputs is not as simple as counting the number of nonzero entries of A and B: if A and B are sparse, and column i of A is filled with zeros only, then row i of B need not be loaded at all, since C does not depend on it. An algorithm that nevertheless loads row i of B will still satisfy the lower bound. And an algorithm that loads and multiplies by explicitly stored zero entries of A or B will also satisfy the lower bound. Multiplications that involve such zero entries is an optimization sometimes used in practice (e.g., [VDY05]).

When A and B are dense and distinct, there are well-known algorithms mentioned in the Introduction that (nearly) attain the combined lower bound

$$\Omega(\max(n\cdot r\cdot m/\sqrt{M}, \#inputs+\#outputs)) = \Omega(\max(n\cdot r\cdot m/\sqrt{M}, n\cdot r+r\cdot m+n\cdot m)) \ ,$$

see [ITT04] for a more complete discussion. Attaining the corresponding latency-cost lower bound of Corollary 2.3 requires a different data structure than the usual row-major or column-major orders, so that words to be sent in a single message are contiguous in memory, and is variously referred to as *recursive block storage* or storage using *space filling curves*, see [FLPR99, EGJK04, BDHS10a] for discussion. Some of these algorithms also minimize bandwidth-cost and latency-cost for arbitrarily many levels of memory hierarchy. Little seems to be known about the attainability of this lower bound for general sparse matrices.

Now we consider the parallel case, with P processors. Let $nnz(A)$ be the number of nonzero entries of A; then $\mathcal{NNZ} = nnz(A) + nnz(B) + nnz(C)$ is a lower bound on the total memory required to store the inputs and outputs. We need to make some assumption about how this data is spread across processors (each of which has its own memory), since if A, B and C were all stored in one processor, and all arithmetic done there (i.e., no parallelism at all), then no communication would be needed. It is enough to assume either that (1) the memory is balanced among the processors, or that (2) the arithmetic is balanced. In the first case, each processor stores an equal share $\mathcal{NNZ}/P$ of the data (and perhaps at most $o(\mathcal{NNZ}/P)$ more words). Then at least one processor must perform at least G/P multiplications, where G is the total number of multiplications (they can't all be below average); the theorem below will apply to the communication done by this processor. In the second case, each processor does G/P multiplications (give or take $o(G/P)$). Then at least one processor stores at most $\mathcal{NNZ}/P$ words (they can't all be above average); the theorem below will apply to the communication done by this processor. Combining all this with Theorem 2.2 yields[4]

COROLLARY 3.2. *Suppose we have a parallel algorithm on P processors for multiplying matrices $C = A \cdot B$ that is memory-balanced in the sense described above.*

[4]We present the conclusions for the parallel model in asymptotic notation. One could instead assume that each processor had memory of size $M = \mu \cdot \frac{n^2}{P}$ for some constant μ, and obtain the hidden constant of the lower bounds as a function of μ, as done in [ITT04].

Then at least one processor must communicate $\Omega\left(G/\sqrt{P \cdot \mathcal{NNZ}} - \mathcal{NNZ}/P\right)$ *words, where* G *is the number of multiplications* $A_{ij} \cdot B_{kj}$ *performed. In the special case of dense n-by-n matrices, this lower bound is* $\Omega\left(n^2/\sqrt{P}\right)$.

There are again well-known algorithms that attain the bandwidth-cost and latency-cost lower bounds in the dense case, but not in the sparse case.

We next extend Theorem 2.2 beyond matrix multiplication. The simplest extension is to the so-called BLAS3 (Level-3 Basic Linear Algebra Subroutines [BDD+01, BDD+02]), which include related operations like multiplication by (conjugate) transposed matrices, by triangular matrices and by symmetric (or Hermitian) matrices. The last two corollaries apply to these operations without change (in the case of $A^T \cdot A$ we use the fact that Theorem 2.2 makes no assumptions about the matrices being multiplied not overlapping).

More interesting is the BLAS3 operation TRSM, computing $C = A^{-1}B$ where A is triangular. The inner loop of the algorithm (when A is upper triangular) is

$$C_{ij} = (B_{ij} - \sum_{k=i+1}^{n} A_{ik} \cdot C_{kj})/A_{ii} \tag{3.1}$$

which can be executed in any order with respect to j, but only in decreasing order with respect to i. None of this matters for the lower bound, since equation (3.1) still matches Equation (2.1), so the lower bounds apply. To see this, we make the correspondences that C_{ij} is stored at location $c(i,j) = b(i,j)$, A_{ik} is stored at location $a(i,k)$, g_{ijk} multiplies $A_{ik} \cdot C_{kj}$, and f_{ij} performs the indicated sum, subtracts it from B_{ij}, and divides by A_{ii}. The fact that output C_{ij} coincides with the input (so it could be of type R2/D1) does not matter. Sequential algorithms that attain these bounds for dense matrices, for arbitrarily many levels of memory hierarchy, are discussed in [BDHS10a].

We note that our lower bound also applies to the so-called Level 2 BLAS (like matrix-vector multiplication) and Level 1 BLAS (like dot products), but the larger lower bound #inputs + #outputs is attainable.

3.2. LU factorization. Independent of sparsity and pivot order, the formulas describing LU factorization are as follows, with the understanding the summations may be over some subset of the indices k in the sparse case, and pivoting has already been incorporated in the interpretation of the indices i, j and k.

$$\begin{aligned} L_{ij} &= (A_{ij} - \sum_{k<j} L_{ik} \cdot U_{kj})/U_{jj} \quad \text{for } i > j \\ U_{ij} &= A_{ij} - \sum_{k<i} L_{ik} \cdot U_{kj} \quad \text{for } i \le j \end{aligned} \tag{3.2}$$

We see that these formulas correspond to our model in Equation (2.1), with $a(i,j) = b(i,j) = c(i,j)$ (since L and U are both inputs and outputs, overwriting A), g_{ijk} identified with multiplying $L_{ik} \cdot U_{kj}$, and f_{ij} summing the operands, subtracting from A_{ij}, and possibly dividing by U_{jj}. The fact that the "outputs" L_{ij} and U_{ij} coincide with the inputs (so they could be of type R2/D1) does not matter, as before.

We discuss the more subtle question of incomplete LU (ILU) in the next section.

A sequential dense LU algorithm that attains this bandwidth-cost lower bound is given by [Tol97], although it does not always attain the latency-cost lower bound

[DGHL08a]. The conventional parallel dense LU algorithm implemented in ScaLAPACK [BCC$^+$97] attains the bandwidth-cost lower bound (modulo an $O(\log P)$ factor), but not the latency-cost lower bound. A parallel algorithm that attains both lower bounds (again modulo a factor $O(\log P)$) is given in [DGX08], where significant speedups are reported. Interestingly, it does not seem possible to attain both lower bounds and retain conventional partial pivoting; a different (but still stable) kind of pivoting is required. We also know of no dense sequential LU algorithm that minimizes bandwidth-cost and latency-cost across multiple levels of a memory hierarchy (unlike Cholesky). There is an elementary reduction proof that dense LU factorization is "as hard as dense matrix multiplication" [DGHL08a], but it does not address sparse or incomplete LU, as does our approach.

3.2.1. How to count operations g_{ijk} carefully. Once an algorithm has completed running, the type Ri/Dj of each argument is well-defined based on the actual sequence of operations performed, but it may be hard to tell by inspecting the source code of the algorithm (or other high level description) which operations to count as g_{ijk} in the total G used in the statement of Theorem 2.2.

A sufficient, but not necessary, condition for counting g_{ijk}, is as follows: If $a(i,k)$ and $b(k,j)$ are originally stored in memory and never modified, then they can only be R1 and not R2; they are always D2. If $c(i,j)$ is only computed once and eventually stored to memory, it can only be D1 and not D2; it could be either R1 or R2, depending on the segment. In this situation, which covers the BLAS, LU and other "complete" factorizations, there are clearly no R2/D2 arguments, and we count all multiplications. (Arguments of type R2/D2 appear later in Section 4, and require different counting techniques.)

In other situations, where it may be difficult to tell which g_{ijk} to count, it may be easier to identify a subset of them that are recognized as satisfying a condition as in the last paragraph, and just count this subset. This may undercount the total number G of g_{ijk}, but still provides a valid lower bound.

We give some examples to illustrate the counting process.

Example 1. Consider incomplete LU (ILU) factorization [Saa96], where some entries of L and U are omitted in order to speedup the computation. In particular, consider *threshold based* ILU, which computes a possible nonzero entry L_{ij} or U_{ij} and compares it to a threshold, storing it only if it is larger than the threshold and discarding it otherwise. Which multiplications $L_{ik} \cdot U_{kj}$ do we count? We may underestimate the total number G of multiplications by simply not counting any multiplications that lead to a value of L_{ij} or U_{ij} that is discarded. Thus we see that analogues of Corollaries 3.1 and 3.2 apply to ILU as well (and later to incomplete Cholesky, etc.).

Example 2. Using only structural information, e.g., about the sparsity patterns of the underlying matrices, it is sometimes possible to deduce that the computed result $f_{ij}(\cdot)$ is exactly zero, and so to possibly avoid a memory reference to location $c(i,j)$ to store the result. This may either be because the values $g_{ijk}(\cdot)$ being accumulated to compute f_{ij} are all identically zero, or, more interestingly, because it is possible to prove there is exact cancellation (independent of the values of the nonzero arguments $Mem(a(i,k))$ and $Mem(b(k,j))$). Here is an example.

Consider a matrix A that is nonzero in its first r rows and columns, and possibly in the trailing $(n-2r)$-by-$(n-2r)$ submatrix; call this submatrix A'. First suppose $A'=0$, so that A has rank at most $2r$, and that pivots are chosen along the diagonal. It is easy to see that the first $2r-1$ steps of Gaussian elimination will generically fill in the entire matrix with nonzeros, but that step $2r$ will cause cancellation to zero (in exact arithmetic) in all entries of A'. If A' starts as a nonzero sparse matrix, then this cancellation will not be to zero but to the sparse LU factorization of A' alone. So one can imagine an algorithm that may or may not recognize this opportunity to avoid work in some or all of the entries of A'. To accommodate all these possibilities, we could, as above, only count those multiplications g_{ijk} (3.2) that contribute to a result L_{ij} or U_{ij} that is stored in memory, possibly underestimating G.

Analogous examples exist for factorizations discussed later, such as LDL^T and QR.

As a short-hand, in Section 4.2 we will sometimes refer to a matrix entry as being *treated as nonzero* if the algorithm assumes that its value could be nonzero in deciding whether to bother performing g_{ijk}. Thus an algorithm for dense matrices treats all entries as nonzero, even if the input matrix is sparse, whereas a sparse factorization algorithm would not.

Example 3. Consider n-by-n boolean matrix multiplication $C = A \cdot B$, where the first column of A and first row of B consist entirely of ones. Then one can deduce that C consists entirely of ones without reading any other columns of A or rows of B. Thus an algorithm could perform as few as n^2 g_{ijk} evaluations (boolean and's) along with $2n + n^2$ loads and stores, or as many as n^3 g_{ijk} evaluations along with $\Omega(n^3/\sqrt{M})$ loads and stores, depending on the algorithm and input matrices. Either way, the theorem applies.

Example 4. It is possible to have no R2/D2 arguments, even if a matrix entry, say $a(i,k)$, requires *no* memory accesses, as long as it is processed in a way like the following: In segment 1, $a(i,j)$ is computed by a formula, and left in fast memory, so it is R2/D1 in segment 1. In segment 2, $a(i,j)$ starts in fast memory at the start of the segment, and is left there at the end, so it is R1/D1 in segment 2. Finally, in segment 3, $a(i,j)$ starts in fast memory at the start of the segment, and discarded before the end, so it is R1/D2 in segment 3. To see that we could potentially have many such arguments, consider the realistic problem of computing the determinant of a matrix A from its LU decomposition, where each entry of A is given by a formula, and we discard the LU decomposition after computing the product $\prod_i U(i,i)$. We give a more systematic way of counting g_{ijk} accurately for examples like this in Section 5.

3.3. Cholesky Factorization.

Now we consider Cholesky factorization. Independent of sparsity and (diagonal) pivot order, the formulas describing Cholesky factorization are as follows, with the understanding the summations may be over some subset of the indices k in the sparse case, and pivoting has already been incorporated in the interpretation of the indices i, j and k.

$$L_{jj} = (A_{jj} - \sum_{k<j} L_{jk}^2)^{1/2} \tag{3.3}$$
$$L_{ij} = (A_{ij} - \sum_{k<j} L_{ik} \cdot L_{jk})/L_{jj} \quad \text{for } i > j$$

It is easy to see that these formulas correspond to our model in Equation (2.1), with g_{ijk} identified with multiplying $L_{ik} \cdot L_{jk}$. As before, the fact that the "outputs" L_{ij} can overwrite the inputs does not matter, and the subtraction from A_{ij}, division by L_{ii}, and square root are all accommodated by Equation (2.1). As before, these formulas are general enough to accommodate incomplete Cholesky (IC) factorization [Saa96].

Dense algorithms that attain these lower bounds are discussed in [BDHS10a], both parallel and sequential, including analyzing one that minimizes bandwidth-cost and latency-cost across all levels of a memory hierarchy [AP00]. We note that there was a proof in [BDHS10a] showing that dense Cholesky was "as hard as dense matrix multiplication" by a method analogous to that for LU.

The bound on Cholesky decomposition applies also to Bunch-Kaufman-type factorizations [BK77]: the symmetric indefinite factorization $A = LDL^T$, where D is block diagonal with 1-by-1 and 2-by-2 blocks, and L is a lower triangular matrix with 1s on the diagonal. If A is positive definite, then all the blocks of D are 1-by-1, and this is essentially the Cholesky decomposition algorithm, and the formulas correspond to our model in Equation (2.1):

$$D_{jj} = A_{jj} - \sum_{k<j} L_{jk}^2 D_{kk} \tag{3.4}$$

$$L_{ij} = \frac{1}{D_{jj}} \left(A_{ij} - \sum_{k<j} L_{ik} \cdot L_{jk} D_{kk} \right) \quad \text{for } i > j \tag{3.5}$$

In the general case, where D has some 2-by-2 diagonal blocks and they are treated as dense (as in standard implementations), the above model captures a subset of the work done (at least half) and the model applies.[5]

3.3.1. Sparse Cholesky Factorization on Matrices whose Graphs are Meshes. Hoffman, Martin, and Rose [HMR73] and George [Geo73] prove that a lower bound on the number of multiplications required to compute the sparse Cholesky factorization of an n^2-by-n^2 matrix representing a 5-point stencil on a 2D grid of n^2 nodes is $\Omega(n^3)$. This lower bound applies to any matrix containing the structure of the 5-point stencil. This yields:

COROLLARY 3.3. *In the case of the sparse Cholesky factorization of the matrix representing a 5-point stencil on a two-dimensional grid of n^2 nodes, the bandwidth-cost lower bound is $\Omega\left(\frac{n^3}{\sqrt{M}}\right)$.*

George [Geo73] shows that this arithmetic lower bound is attainable with a nested dissection algorithm in the case of the 5-point stencil. Gilbert and Tarjan [GT87] show that the upper bound also applies to a larger class of structured matrices, including matrices associated with planar graphs. Recently, David, Demmel, Grigori, and Peyronnet [DDGP10] obtained new algorithms for sparse cases of Cholesky decomposition, that are proven to be communication optimal using our lower bounds.

3.4. Imposing reads and writes. In this example we consider a single linear algebra operation, where inputs are given by formulas and the output is a scalar (e.g., norm of the product of two matrices given by formulas, each used once; computing the determinant of a matrix with entries given by formulas, where one does the LU decomposition and takes the product of the diagonal elements of U, etc.)

Even though this seems to eliminate a large number of reads and writes, we can prove (for this and similar examples) that the communication lower bound is still $\Omega\left(\frac{\#\text{flops}}{\sqrt{M}}\right)$, by using a technique of *imposing reads and writes*: We take an algorithm to which Theorem 2.2 does *not* apply, because it may potentially have R2/D2 operands, and add (impose) memory traffic to eliminate such operands. Then we use Theorem 2.2 to bound below the communication of this modified algorithm, and subtract the amount of imposed communication to get a lower bound for the original algorithm.

[5]One could imagine a nonstandard implementation that took advantage of zero diagonals in 2-by-2 blocks, so a slightly different proof would be needed for this set of inputs of measure zero.

Here is an example. Consider computing $r = \|A \cdot B\|_F^2 = \sum_{ij}(A \cdot B)_{ij}^2$, where $A_{ik} = 1/(i+k)$ and $B_{kj} = k^{1/j}$ are given by formulas. Let $C = A \cdot B$. Whenever the final value of some C_{ij} is computed, squared, and added to r, we impose a write (if it is missing) so that C_{ij} is saved in slow memory, and so has destination D1 instead of possibly D2 (it may still have root R2). Thus no entries of C can be R2/D2. Whenever the value of some A_{ik} or B_{kj} is computed by a formula, we impose a read to get it from a location in slow memory, so it has root R1 instead of R2 (it may still have destination D2). Now, no entries of A or B can be R2/D2. Thus this modified algorithm has lower bound $n^3/(8\sqrt{M}) - M$ by Theorem 2.2.

To get a lower bound for the original algorithm, we need to bound how many reads and writes we imposed. There are clearly at most n^2 imposed writes. If the original algorithm only evaluates each formula for A_{ik} and B_{kj} once, and keeps their computed values in memory if necessary for later use, then the number of imposed reads is $2n^2$, and the communication lower bound for the original algorithm is $n^3/(8\sqrt{M}) - M - 3n^2 = \Omega(n^3/\sqrt{M})$, close to standard dense matrix multiplication.

On the other hand, if the original algorithm evaluates the formulas for A_{ik} and B_{kj} whenever it needs them, so n^3 times, then the communication lower bound for the original algorithm becomes $n^3/(8\sqrt{M}) - M - n^2 - 2n^3$, which degenerates to zero.

4. Orthogonal Factorizations.

In this section we consider algorithms that compute matrix factorizations with at least one orthogonal factor. This includes algorithms that apply sequences of orthogonal transformations to a matrix, which includes the most widely used algorithms for least squares problems (the QR factorization), eigenvalue problems, and the SVD. We need to treat algorithms that apply orthogonal transformations separately because many of the operations to which we would like to apply the model in Equation (2.1) involve R2/D2 arguments, so the model does not directly apply.

We start with the easier case of algorithms that compute the QR factorization without applying orthogonal transformations (e.g., Gram-Schmidt), for which we can use Equation (2.1).

4.1. QR factorization without applying orthogonal transformations.

We first discuss algorithms for computing the QR decomposition whose computations correspond to our model in Equation (2.1). Although Cholesky-QR, classical Gram-Schmidt, and modified Gram-Schmidt do not share the same stability characteristics as when applying orthogonal transformations, they are advantageous in various situations and are used in practice.

Cholesky-QR. Consider an $m \times n$ matrix A. The Cholesky-QR algorithm consists of forming $A^T A$ and computing the Cholesky decomposition of that $n \times n$ matrix. The R factor is the upper triangular Cholesky factor and Q is obtained by solving the equation $A = QR$ using TRSM. The communication lower bounds for TRSM (see Section 3.1) thus apply to the Cholesky-QR algorithm (and reflect at least $\frac{6}{13}$ of the total number of multiplications of the overall dense algorithm). Since the steps of the algorithm (form $A^T A$, Cholesky, TRSM) can all be done with minimal communication, so can the overall algorithm.

Classical Gram-Schmidt. We recall the Gram-Schmidt algorithm for orthonormalizing a set of vectors in an inner product space: Let $Proj_u(v) \equiv \frac{\langle v,u\rangle}{\langle u,u\rangle}u$. Let $\{v_i\}_{i\in[n]}$ be a set of n input vectors in $\mathbb{R}^m$. Then the output of the Gram-Schmidt algorithm is $\{u_i\}_{i\in[n]}$ where

$$u_k \equiv v_k - \sum_{i=1}^{k} Proj_{u_i}(v_k) \tag{4.1}$$

as well as the triangular R factor. Equation (4.1) does not match Equation (2.1). In order to apply Theorem 2.2 here, we consider the inner product $\langle v_i, u_j\rangle$ (which is computed for every $i > j$). The operation g_{ijk} corresponds to the multiplication of the k^{th} element of v_i with the k^{th} element of u_j. Now we have an algorithm that computes $\langle v_i, u_j\rangle$ for all $i > j$, and does some other extra computations. Ignoring all the extra computation, the algorithm agrees with Equation (2.1), with A being the input (R1) vectors $\{v_i\}_i$, B being the output (D1) vectors $\{u_j\}_j$, and C being the dot products which become entries of the output (D1) matrix R.

We can now apply Theorem 2.2 to obtain a lower bound of $\Omega(\frac{mn^2}{\sqrt{M}})$ on the bandwidth-cost (since $\Theta(mn^2)$ flops are performed). This is not matched by existing algorithms [DGHL08a].

Modified Gram-Schmidt. The argument for the modified Gram-Schmidt is similar to the above. Recall that in this modified algorithm, each v_i is replaced with new vectors, $u_i^{(k)}$, where k is different for each inner product. That is, instead of (4.1) we have the modified algorithm:

$$\begin{aligned}
u_k^{(1)} &= v_k - Proj_{u_1}(v_k)\\
u_k^{(2)} &= u_k^{(1)} - Proj_{u_2}(u_k^{(1)})\\
&\vdots\\
u_k^{(k-2)} &= u_k^{(k-3)} - Proj_{u_{k-2}}(u_k^{(k-3)})\\
u_k &= u_k^{(k-2)} - Proj_{u_{k-1}}(u_k^{(k-2)})
\end{aligned}$$

To apply the model in Equation (2.1), we note that a standard implementation will overwrite $u_k^{(j-1)}$ by $u_k^{(j)}$, so that $a(i,k)$ points to the common location storing the (D1) values $u_k^{(j)}(i)$ for all $1 \le j \le k$. Again, the resulting communication lower bounds $\Omega(\frac{mn^2}{\sqrt{M}})$ are not matched by existing algorithms [DGHL08a].

4.2. Applying Orthogonal Transformations. The case of applying orthogonal transformations is more subtle to analyze for several reasons: (1) There is more than one way to represent the Q factor (e.g., Householder reflections and Givens rotations). (2) The standard ways to reorganize or "block" QR to minimize communication involve using the distributive law, not just summing terms in a different order [BVL87, SVL89, Pug92, Dem97, GVL96]. (3) There may be many intermediate terms that are computed, used, and discarded without causing any slow memory traffic (i.e., are of type R2/D2).

This forces us to use a different argument than [ITT04], but still using Loomis-Whitney, to bound the number of arithmetic operations in a segment. To be concrete, we consider the widely used Householder reflections, in which an n-by-n elementary real orthogonal matrix Q_i is represented as $Q_i = I - \tau_i u_i u_i^T$, where u_i is a column vector called a Householder vector and $\tau_i = 2/\|u_i\|_2^2$. A single Householder reflection Q_i is chosen so that multiplying $Q_i \cdot A$ zeros out selected rows in a particular column of A, and modifies one other row in the same column (for later use, we let r_i be index of this other row).

We furthermore model the way libraries like LAPACK [ABB$^+$92] and ScaLAPACK [BCC$^+$97] may "block" Householder vectors, writing $Q_k \cdots Q_1 = I - U_k T_k U_k^T$, where $U_k = [u_1, u_2, \ldots, u_k]$ is n-by-k and T_k is k-by-k. U_k is nonzero only in the rows being modified, and furthermore column i of U_k is zero in entries r_1,...,r_{i-1} and nonzero in entry r_i.[6] Next, we will apply such block Householder transformations to a (sub)matrix by inserting parentheses as follows: $(I - U \cdot T \cdot U^T) \cdot A = A - U \cdot (T \cdot U^T \cdot A) \equiv A - U \cdot Z$, which is also the way Sca/LAPACK does it. Finally, we overwrite the output onto $A := A - U \cdot Z$, which is how most fast implementations do it, analogously to LU decomposition, to minimize memory requirements. We will also assume that each entry of Z is computed only once.

But we do not need to assume any more commonality with the approach in Sca/LAPACK, in which a vector u_i is chosen to zero out all of column i of A below the diagonal. For example, we can choose each Householder vector to zero out only part of a column at a time, as is the case with the algorithms for dense matrices in [DGHL08a, DGHL08b]. Nor do we even need to assume we are zeroing out any particular set of entries, such as those below the main diagonal as the usual QR algorithm; later this generality will let us apply our result to algorithms for eigenproblems and the SVD.

To get our lower bound, we consider just the multiplications in all the different applications of block Householder transformations $A := A - U \cdot Z$. We argue in Section 4.2.3 that this constitutes a large fraction of all the multiplications in the algorithm (it is a valid lower bound in any event).

There are two challenges to straightforwardly applying our previous approach to the matrix multiplications in all the updates $A := A - U \cdot Z$. The first challenge is that we need to collect all these multiplications into a single set, indexed in an appropriate one-to-one fashion by (i, j, k). The second challenge is that entries of Z may be R2/D2, i.e., they need not be read from or written to memory. Rather, they may be computed on-the-fly from U and A, used and discarded. So we have to account for Z's memory traffic more carefully. Furthermore, each Householder vector (column of U) is created on the fly by modifying certain (sub)columns of A, so it is both an output and an input. Therefore we will have also have to account for U's and A's memory traffic more carefully.

Here is how we address the first challenge: Let index k indicate the number of the Householder vector; in other words $U(:, k)$ are all the entries of k-th Householder vector. Thus, k is not the index of the column of A from which $U(:, k)$ arises (there may be many Householder vectors associated with a given column as in [DGHL08a]) but k does uniquely identify that column. Then the operation $A - U \cdot Z$ may be rewritten as $A(i, j) - \sum_k U(i, k) \cdot Z(k, j)$, where the sum is over the Householder vectors, indexed by k, making up U that both lie in column j and have entries in row i. The use of this index k lets us combine all the operations $A := A - U \cdot Z$ for all different Householder vectors into one collection

$$A(i, j) := A(i, j) - \sum_k U(i, k) \cdot Z(k, j) \tag{4.2}$$

where all operands $U(i, k)$ and $Z(k, j)$ are uniquely labeled by the index pairs (i, k) and (k, j), respectively.

[6]In conventional algorithms for dense matrices (e.g., the implementation available in *LAPACK*[ABB$^+$92]) this means $r_i = i$, and U_k is lower trapezoidal with a nonzero diagonal, but our proof does not assume this.

For the second challenge, we separately handle two cases. The first (easier) case is when the number of R2/D2 Z's is relatively small. We can then use the imposed-writes technique from Section 3.4 and apply Loomis-Whitney to obtain the lower bounds. In the second case, no such bound on the Z's is guaranteed. We then use a "forward-progress" assumption, combined with assuming T is 1×1 to obtain a matching lower bound.

4.2.1. When the number of R2/D2 Z's is not too large. Consider the number of R2/D2 Z's in the entire algorithm, where each R2/D2 Z value is computed once (alternatively, if we allow re-computation, each such value that may be computed several times, and is then counted with corresponding multiplicity). We can impose writes (as in Section 3.4) on each R2/D2 Z element, i.e., writing it to memory when it would have been discarded, making it D1. Thus all A, U and Z arguments are non-R2/D2, allowing as to directly apply Loomis-Whitney by Theorem 2.2. If the number of R2/D2 Z's is bounded above by a constant times the number of inputs plus the number of outputs, we obtain the desired lower bound.

LEMMA 4.1. *Consider dense or sparse QR, done with block Householder transformations of any block size, but at most one Householder transformation per column. Then the number of words moved is at least*

$$\Omega\left(max\left(\frac{\#flops}{\sqrt{M}}, \#inputs + \#outputs\right)\right).$$

Proof. Consider the first block Householder transformation, of block size b_1. From

$$Z(1:b_1,k) = T(1:b_1,1:b_1)\cdot(U(:,1:b_1))^T \cdot A(:,k)$$

and

$$A(:,k) = A(:,k) + U(:,1:b_1)\cdot Z(1:b_1,k)$$

and the fact that $U(i,i)$ is nonzero we see that if $Z(i,k) \neq 0$ then $A(i,k) = A(i,k) + U(i,i)\cdot Z(i,k) + ...$ is generically nonzero.[7] So for the first block Householder transformation, the number of entries in $Z(1:b_1,k)$ is bounded by the number of entries in $A(1:b_1,k)$, which are all TAN. The next block Householder transformation, of block size b_2, is treated similarly, with the number of entries in $Z(b_1+1:b_1+b_2,k)$ bounded by the number of entries in $A(b_1+1:b_1+b_2,k)$.

If we impose writes (as in Section 3.4) on R2/D2 Z entries, then we obtain a lower bound from Theorem 2.2 which must be adjusted to account for the imposed writes. However, since the number of imposed writes is bounded by the number of A entries (which is the number of inputs and outputs), we obtain a lower bound on the number of words moved of

$$\Omega\left(max\left(\frac{\#flops}{\sqrt{M}} - (\#inputs + \#outputs), \#inputs + \#outputs\right)\right),$$

and the result follows. □

[7]We say an element is treated as non-zero *(TAN)* if it is not ignored by the algorithm, even though it may actually contain zero, or an arbitrarily small value. In other words, it was not zeroed out by the algorithm, nor it is assumed to be an input element which is guaranteed to be zero. Otherwise, we say the element is treated as zero *(TAZ)*.

We can conclude a similar bound for reduction to Hessenberg or tridiagonal form: instead of assuming we are doing QR (so that $U(i,i)$ is nonzero, since $A(i,i)$ "accumulates" nonzero entries below it), we could be accumulating into a different, but unique row destination.

Note that the approach of imposing writes does not easily apply to communication-avoiding QR [DGHL08a], since there are potentially $\Theta\left(\frac{\#flops}{block\_size}\right)$ different Z elements.

4.2.2. When the number of R2/D2 Z's is large. We next consider the harder general case, where the number of R2/D2 Z's cannot be bounded by a constant factor times the number of inputs and outputs. We first introduce some notation:

- Let $U(k)$ be the k^{th} column of U (which is the k^{th} Householder vector). We will use $U(k)$ and $U(:,k)$ interchangeably when the context is clear.
- Let col_src_$U(k)$ be the index of the column in which $U(k)$ introduces zeros.
- Let rows_$U(k)$ be the set of indices of rows TAN in $U(k)$. Let row_dest_$U(k)$ be the index of the row in column col_src_$U(k)$ in which nonzero values in that column are accumulated by $U(k)$, and let zero_rows_$U(k)$ be rows_$U(k)$ with row_dest_$U(k)$ omitted.

We will make two central assumptions in this case. First, we assume that the algorithm does not block Householder updates (i.e., all T matrices are 1×1). Second, we assume the algorithm makes "forward progress" which we define below. As explained later, forward progress is a natural property of any efficient implementation that precludes certain kinds of redundant work.

The first assumption means that we are computing $\prod_k (I - \tau_k \cdot U(:,k) \cdot U'(:,k)) \cdot A$, where τ_k is scalar. This seems like a significant restriction, since blocked Householder transformations are widely used in practice. We do not believe this assumption is necessary for the communication lower bound to be valid, but the reason for the assumption is that there exists an artificial example, where by using an $O(n^4)$ algorithm with $O(n^4)$ additional storage (to form and use a T matrix of dimension $O(n^2)$) on a certain matrix, we could arrange to have one segment in which $O(M^2)$ multiplications were performed, thereby creating an obstacle to our proof technique, which depends on bounding the number of multiplications per segment by $O(M^{3/2})$. This (impractical!) variant of QR is not a counterexample to our theorem overall, just our proof technique. We describe this counterexample in detail in Appendix A. Still, we believe this special case gives insight into why blocking techniques will not do better: By using many small Householder transformations (including 2×2, i.e., analogous to Givens rotations) in place of any one larger Householder transformation, and applying these in the right order, very similar memory access patterns as for block Householder transformations can be achieved.

This assumption yields a partial order (PO) in which the Householder updates must be applied to get the right answer. It is only a partial order because if, say, $U(:,k)$ and $U(:,k+1)$ do not "overlap", i.e., have no common rows that are TAN, then $(I - \tau_k \cdot U(:,k) \cdot U'(:,k))$ and $(I - \tau_{k+1} \cdot U(:,k+1) \cdot U'(:,k+1))$ commute, and either one may be applied first (indeed, they may be applied independently in parallel).

Definition 4.2 (Partial Order on Householder vectors (PO)). *Suppose $k_1 < k_2$ and rows_$U(k_1) \cap$ rows_$U(k_2) \neq \{\emptyset\}$, then $U(k_1) < U(k_2)$ in the partial order.*[8]

[8]We note that this relation is transitive. That is, two Householder vectors $U(k_1)$ and $U(k_2)$ are partially ordered if there exists $U(k^*)$ such that $U(k_1) < U(k^*) < U(k_2)$, even if rows_$U(k_1) \cap$ rows_$U(k_2) = \{\emptyset\}$.

Our second assumption is that the algorithm makes forward progress:

Definition 4.3 (Forward Progress (*FP*)). *We say an algorithm which applies orthogonal transformations to zero out entries makes forward progress if the following two conditions hold:*

1. *an element that was deliberately*[9] *zeroed out by one transformation is never again zeroed out or filled by another transformation,*
2. *if*
   (a) $U(k_1), \ldots, U(k_b) < U(\hat{k})$ *in* PO,
   (b) $col\_src\_U(k_1) = \cdots = col\_src\_U(k_b) = c \neq \hat{c} = col\_src\_U(\hat{k})$,
   (c) *and no other* $U(k_i)$ *satisfies* $U(k_i) < U(\hat{k})$ *and* $col\_src\_U(k_i) = c$,
   then

$$rows\_U(\hat{k}) \subset \bigcup_{i=1}^{b} zero\_rows\_U(k_i) \cup \{rows\ of\ column\ c\ that\ are\ TAZ\}\,. \tag{4.3}$$

The first condition holds for most efficient Householder algorithms.[10] It is easy to see that it is necessary to prove any nontrivial communication lower bound, since without it an algorithm could "spin its wheels" by repeatedly filling in and zeroing out entries, doing an arbitrary amount of arithmetic with no memory traffic at all.

The second condition holds for every correct algorithm for QR decomposition. This condition means any later Householder transformation ($U(\hat{k})$) that depends on earlier Householder transformations ($U(k_1), ..., U(k_b)$) creating zeroes in a common column c may operate only "within" the rows zeroed out by the earlier Householder transformations. We motivate this assumption in Appendix B by showing that if an algorithm violates the second condition, it can "get stuck." This means that it cannot achieve triangular form without filling in a deliberately created zero.

We note that *FP* is not violated if an original TAZ entry of the matrix is filled in (so that it is no longer TAZ); this is a common situation when doing sparse QR.

With these assumptions, we begin the argument to bound from below the number of memory operations required to apply the set of Householder transformations. As in the proof of Theorem 2.2, we will focus our attention on an arbitrary segment of computation in which there are $O(M)$ non-R2/D2 entries in fast memory. Our goal will be to bound the number of multiplications in a segment involving R2/D2 entries, since the number of remaining multiplications can be bounded using Loomis-Whitney as before. From here on, let us denote by $Z_2(k, j)$ the element $Z(k, j)$ if it is R2/D2, and by $Z_n(k, j)$ if it is non-R2/D2. We will further focus our attention within the segment on the update of an arbitrary column of the matrix, $A(:, j)$.

Each $Z(k, j)$ in memory is associated with one Householder vector $U(:, k)$ which will update $A(:, j)$. We will denote the associated Householder vector by $U_2(:, k)$ if $Z(k, j) = Z_2(k, j)$ is R2/D2 and $U_n(:, k)$ if $Z(k, j) = Z_n(k, j)$ is non-R2/D2. With this notation, we have the following two lemmas which make it easier to reason about what happens to $A(:, j)$ during a segment.

Lemma 4.4. *If* $Z_2(k, j)$ *is in memory during a segment, then* $U_2(:, k)$ *and the entries* $A(rows\_U(k), j)$ *are also in memory during the segment.*

[9]By deliberately, we mean the algorithm converted a TAN entry into a TAZ entry with an orthogonal transformation. The introduction of a zero due to accidental cancellation (such zero entries are still TAN) is not deliberate.

[10]We note that the first condition of *FP* does not hold for the bulge-chasing process within standard QR iteration or successive band reduction [BLS00b] over multiple bulge chases.

Proof. Since $Z_2(k,j)$ is discarded before the end of the segment and may not be re-computed later, the entire $A(:,j) = A(:,j) - U(:,k) \cdot Z_2(k,j)$ computation has to end within the segment. Thus, all entries involved must be resident in memory. $\square$

However, even if a $Z_n(k,j)$ is in memory during a segment, the $U_n(:,k) \cdot Z_n(k,j)$ computation will possibly not be completed during the segment, and therefore the $U_n(:,k)$ vector and corresponding entries of $A(:,j)$ may not be completely represented in memory.

LEMMA 4.5. *If $Z_2(k_1,j)$ and $Z_2(k_2,j)$ are in memory during a segment, and $U(k_1) < U(k) < U(k_2)$ in the* PO*, then $Z(k,j)$ must also be in memory during the segment.*

Proof. This follows from our first assumption that all T matrices are 1×1 and the partial order is imposed. Since $U(k_1) < U(k)$, $Z(k,j)$ cannot be fully computed before the segment. Since $U(k) < U(k_2)$, $U(:,k) \cdot Z(k,j)$ has to be performed in the segment too, at least "enough"[11] to carry the dependency, so $Z(k,j)$ cannot be fully computed after the segment. Thus, $Z(k,j)$ is computed during the segment and therefore must exist in memory. $\square$

Emulating the arithmetic operations in a segment. Roughly speaking, our goal now is to bound the number of $U_2(r,k) \cdot Z_2(k,j)$ multiplications by the number of multiplications in a different matrix multiplication $\widehat{U} \cdot \widehat{Z}$ where we can bound the number of $\widehat{U}$ entries by the number of U entries in memory, and bound the number of $\widehat{Z}$ entries by the number of A entries plus the number of Z_n entries in memory, which lets us use Loomis-Whitney.

Given a particular segment and column j, we construct $\widehat{U}$ by first partitioning the $U_2(:,k)$ by their col_src_$U(k)$ and then collapsing each partition into one column of $\widehat{U}$. Likewise, collapse $Z(:,j)$ by partitioning its rows corresponding to the partitioned columns of U and taking the union of TAN entries in each set of rows to be the TAN entries of the corresponding row of $\widehat{Z}(:,j)$. More formally,

DEFINITION 4.6 ($\widehat{U}$ and $\widehat{Z}$). *For a given segment of computation and column j of A, we set $\widehat{U}(r,c)$ to be TAN if there exists a $U_2(:,k)$ in fast memory such that $c =$ col_src_$U(k)$ and $r \in$ rows_$U(k)$. We set $\widehat{Z}(c,j)$ to be TAN if there exists a $Z_2(k,j)$ in fast memory such that $c =$ col_src_$U(k)$.*

We will "emulate" the computation $A(:,j) = A(:,j) - \sum U_2(:,k) \cdot Z_2(k,j)$ with the related computation $A(:,j) = A(:,j) - \sum \widehat{U}(:,c) \cdot \widehat{Z}(c,j)$ in the following sense: we will show that the number of multiplications done by $U_2(:,k) \cdot Z_2(k,j)$ is within a factor of 2 of the number of multiplications done by $\widehat{U}(:,c) \cdot \widehat{Z}(c,j)$, which we will be able to bound using Loomis-Whitney.

The following example illustrates this construction on a small matrix, where K_2 contains three indices (i.e., there are three Householder vectors that were computed

[11]Note that, if $U(:,k)$ is $U_n(:,k)$, not all *rows_U*(k) rows of $A(:,j)$ must be updated, but enough for $Z_2(k_2,j)$ to be computed and $U_2(:,k_2) \cdot Z_2(k_2,j)$ to be applied correctly. Also, a partial sum $(U(\text{stuff},k))^T \cdot A(\text{stuff},j)$ may have been computed before the beginning of the segment and used in the segment to compute $Z_n(k,j)$, but the final $Z_n(k,j)$ value cannot be computed until the segment.

to zero entries in the second column of A); just TAN patterns are shown.

$$U(:,K_2) = \begin{bmatrix} & & \\ & & \bullet \\ & & \\ & \bullet & \bullet \\ \bullet & \bullet & \\ & & \\ \bullet & & \\ \bullet & & \\ \bullet & & \end{bmatrix} \quad \Rightarrow \quad \widehat{U}(:,2) = \begin{bmatrix} \\ \bullet \\ \\ \bullet \\ \bullet \\ \\ \bullet \\ \bullet \\ \bullet \end{bmatrix}$$

Note that we do not care what the TAN values of $\widehat{U}$ and $\widehat{Z}$ are; this computation has no hope of getting a correct result because the rank of $\widehat{U} \cdot \widehat{Z}$ is generally less than the rank of the subset of $U \cdot Z$ it replaces. We emulate in this way only to count the memory traffic. We establish the following results with this construction.

LEMMA 4.7. *$\widehat{U}(:,c)$ has at least half as many TAN entries, and at most as many TAN entries, as the columns of U from which it is formed.*

Proof. The sets zero_rows_$U(k)$ for k in a partition (i.e., with the same col_src_$U(k)$) must be disjoint by the forward progress assumption, and there are at least as many of these rows as in all the corresponding row_dest_$U(k)$, which could potentially all coincide. By Lemma 4.4, we know that complete $U_2(:,k)$ are present (otherwise they could, for example, all be Givens transformations with the same destination row, and if zero rows were not present, they would all collapse into one row). And so since every entry of zero_rows_$U(k)$ contributes to a TAN entry of $\widehat{U}(:,c)$, and zero_rows_$U(k)$ constitutes at least half of the TAN entries of $U(k)$, $\widehat{U}(:,c)$ has at least half as many TAN entries as the corresponding columns of U.

If all the $U_2(:,k)$ being collapsed have TAN entries in disjoint sets of rows, then $\widehat{U}(:,c)$ will have as many entries TAN as all the $U(:,k)$. $\square$

Because each TAN entry of $U(:,k)$ contributes one scalar multiplication to $A(:,j) = A(:,j) - \sum U_2(:,k) \cdot Z_2(k,j)$ and each TAN entry of $\widehat{U}(:,c)$ contributes one scalar multiplication to $A(:,j) = A(:,j) - \sum \widehat{U}(:,c) \cdot \widehat{Z}(c,j)$, we have the following corollary.

COROLLARY 4.8. *$\widehat{U}(:,c) \cdot \widehat{Z}(c,j)$ does at least half as many multiplications as all the corresponding $U_2(:,k) \cdot Z_2(k,j)$.*

In order to bound the number of $\widehat{U} \cdot \widehat{Z}$ multiplications in the segment, we must also bound the number of $\widehat{Z}$ entries available.

LEMMA 4.9. *The number of TAN entries of $\widehat{Z}(:,j)$ is bounded by the number of $A(:,j)$ entries plus the number of $Z_n(:,j)$ entries resident in memory.*

Proof. Our goal is to construct an injective mapping $\mathcal{I}$ from the set of of $\widehat{Z}(:,j)$ entries to the union of the sets of $A(:,j)$ and $Z_n(:,j)$ entries. Consider the set of $Z(k,j)$ entries (both R2/D2 and non-R2/D2) in memory as vertices in a graph G. Each vertex has a unique label k (recall that j is fixed), and we also give each vertex two more non-unique labels: 2 or n to denote whether the vertex is $Z_2(k,j)$ or $Z_n(k,j)$ and col_src_$U(k)$ to denote the column source of the corresponding Householder vector. A directed edge (k_1,k_2) exists in the graph if $U(:,k_1) < U(:,k_2)$ in the *PO*. Note that all the vertices labeled both 2 and c are $Z_2(k,j)$ that lead to $\widehat{Z}(c,j)$ being TAN in Definition 4.6.

For all values of $c = \text{col\_src\_}U(k)$ appearing as labels in G, in order of which node labeled c is earliest in PO (not necessarily unique), find a (not necessarily unique) node k with label $\text{col\_src\_}U(k) = c$, that has no successors in G with the same label c. If this node is also labeled n, then we let $\mathcal{I}$ map $\widehat{Z}(c,j)$ to $Z_n(k,j)$. If node k is labeled 2, then we let $\mathcal{I}$ map $\widehat{Z}(c,j)$ to $A(\text{row\_dest\_}U(k),j)$. By Lemma 4.4, this entry of A must be in fast memory.

We now argue that this mapping $\mathcal{I}$ is injective. The mapping into the set of $Z_n(k,j)$ entries is injective because each $\widehat{Z}(c,j)$ can be mapped only to an entry with column source c. Suppose the mapping into the $A(:,j)$ entries is not injective, and let $\widehat{Z}(c,j)$ and $\widehat{Z}(\hat{c},j)$ be the entries which are both mapped to some $A(r,j)$. Then there are entries $Z_2(k,j)$ and $Z_2(\hat{k},j)$ such that $c = \text{col\_src\_}U(k)$, $\hat{c} = \text{col\_src\_}U(\hat{k})$, $r = \text{row\_dest\_}U(k) = \text{row\_dest\_}U(\hat{k})$, and neither k nor $\hat{k}$ have successors in G with the same column source label.

Since $\text{rows\_}U(k)$ and $\text{rows\_}U(\hat{k})$ intersect, they must be ordered with respect to the PO, so suppose $U(k) < U(\hat{k})$. Consider the second condition of *FP*. In this case, premises (2a) and (2b) hold, but the conclusion (4.3) does not. Thus, premise (2c) must not hold, so there exists another Householder vector $U(k^*)$ such that $c = \text{col\_src\_}U(k^*)$ and $r \in \text{zero\_rows\_}U(k^*)$.

Again, because their nonzero row sets intersect, each of these Householder vectors must be partially ordered. By the first condition of *FP*, since $\text{row\_dest\_}U(k) \in \text{zero\_rows\_}U(k^*)$, we have $U(k) < U(k^*)$. Also, since $U(k^*)$ satisfies (2a), we have $U(k^*) < U(\hat{k})$. Thus, $U(k) < U(k^*) < U(\hat{k})$, and by Lemma 4.5, $Z(k^*,j)$ must also be in fast memory and therefore in G. Since $Z(k^*,j)$ is a successor of $Z(k,j)$ in G, we have a contradiction. □

THEOREM 4.10. *An algorithm which applies orthogonal transformations to annihilate matrix entries, does not compute T matrices of dimension 2 or greater for blocked updates, maintains forward progress as in Definition 4.3, and performs G flops of the form $U \cdot Z$, has a bandwidth cost of at least*

$$\Omega\left(\frac{G}{\sqrt{M}}\right) - M \quad \textit{words.}$$

In the special case of a dense m-by-n matrix with $m \geq n$, this lower bound is $\Omega(mn^2/\sqrt{M})$.

Proof. We first argue that the number of A, U, and Z_n entries available during a segment are all $O(M)$.

Every $A(i,j)$ operand is destined either to be output (i.e., D1) or converted into a Householder vector. Every $A(i,j)$ operand is either read from memory (i.e., R1) or created on the fly due to sparse fill-in. So the only possible R2/D2 operands from A are entries which are filled in and then immediately become Householder vectors, and hence become R2 operands of U. We bound the number of these as follows.

All U operands are eventually output, as they compose Q. So there are no D2 operands of U (recall that we may only compute each result $U(i,k)$ once, so it cannot be discarded). So all R2 operands $U(i,k)$ are also D1, and so there are at most $2M$ of them (since at most M can remain in fast memory, and at most M can be written to slow memory, by the end of the segment). This also bounds the number of R2/D2 operands $A(i,j)$, and so bounds the total number of $A(i,j)$ operands by $6M$ (the sum of $2M$ = maximum number of D1 operands plus $2M$ = maximum number of R1 operands plus $2M$ = maximum number of R2/D2 operands).

The number of Z_n entries available in a segment is bounded by $2M$ because by definition, all entries are non-R2/D2.

From Lemma 4.7, the number of $\widehat{U}$ entries available is $O(M)$ because it is bounded by the number of U_2 entries which is in turn bounded by the number of U entries. From Lemma 4.9, the number of $\widehat{Z}$ entries available is $O(M)$ because it is bounded by the sum of the number of entries of A and of Z_n.

Thus, since the number of entries of each operand available in a segment are $O(M)$, by Lemma 2.1 (Loomis-Whitney), the number of $\widehat{U} \cdot \widehat{Z}$ scalar multiplications is bounded by $O\left(M^{3/2}\right)$. By Corollary 4.8, the number of $U \cdot Z$ scalar multiplications within a segment is also bounded by $O\left(M^{3/2}\right)$.

Since there are $O(M)$ $Z_n(k,j)$ operands in a segment, the Loomis-Whitney argument bounds the number of multiplies involving such operands by $O(M^{3/2})$, so with the above argument that bounds the number of multiplies involving R2/D2 $Z(k,j)$ operands, the total number of multiplies involving both R2/D2 and non-R2/D2 Z entries is $O\left(M^{3/2}\right)$.

The rest of the proof is similar to before: A lower bound on the number of segments is then $\lfloor \#\text{multiplies}/O\left(M^{3/2}\right)\rfloor \geq \#\text{multiplies}/O\left(M^{3/2}\right) - 1$, so a lower bound on the number of slow memory accesses is $M \cdot \lfloor \#\text{multiplies}/O\left(M^{3/2}\right)\rfloor \geq \Omega\left(\#\text{multiplies}/M^{1/2}\right) - M$. For dense m-by-n matrices with $m \geq n$, the conventional algorithm does $\Theta(mn^2)$ multiplies. □

4.2.3. Discussion of QR Model. It is natural to wonder whether the G operations in Theorem 4.10 capture the majority of the arithmetic operations performed by the algorithm, which would allow us to deduce that the lower bound is as large as possible. The G operations are just the multiplications in all the different applications of block Householder transformations $A := A - U \cdot Z$, where $Z = T \cdot U^T \cdot A$. We argue that under a natural "genericity assumption" this constitutes a large fraction of all the multiplications in the algorithm (although this is not necessary for our lower bound to be valid). Suppose $(U^T \cdot A)(k,j)$ is nonzero; the amount of work to compute this is at most proportional to the total number of entries stored (and so treated as nonzeros) in column k of U. Since T is triangular and nonsingular, this means $Z(k,j)$ will be generically nonzero as well, and will be multiplied by column k of U and added to column j of A, which costs at least as much as computing $(U^T \cdot A)(k,j)$. The cost of the rest of the computation, forming and multiplying by T and computing the actual Householder vectors, are lower order terms in practice; the dimension of T is generally chosen small enough by the algorithm to try to assure this. Thus, for example, there are both a total of $\Theta(mn^2)$ multiplies done by dense QR factorization on an m-by-n matrix (with $m \geq n$), as well as $\Theta(mn^2)$ multiplies counted in our lower bound.

4.3. Eigenvalue and Singular Value Problems.

Standard algorithms for computing eigenvalues and eigenvectors, or singular values and singular vectors (the SVD), start by applying orthogonal transformations to both sides of A to reduce it to a "condensed form" (Hessenberg, tridiagonal or bidiagonal) with the same eigenvalues or singular values, and simply related eigenvectors or singular vectors [Dem97]. This section presents communication lower bounds for these reductions, and then discusses whether analogous lower bounds apply to algorithms that work on the condensed forms.

Later, in section 6, we discuss eigenvalues algorithms that attain these lower bounds for dense matrices. For the symmetric eigenproblem and SVD, there are such algorithms that begin by reduction to a condensed form. But for the nonsymmetric eigenproblem, the only known algorithm attaining the expected lower bound does not initially reduce to condensed form, and is not based on QR iteration [DDH07, BDD11].

We extend our argument from the last section as follows. We can have some arbitrary interleaving of (block) Householder transformations applied on the left:

$$A = (I - U_L \cdot T_L \cdot U_L^T) \cdot A = A - U_L \cdot (T_L \cdot U_L^T \cdot A) \equiv A - U_L \cdot Z_L$$

and the right:

$$A = A \cdot (I - U_R \cdot T_R \cdot U_R^T) = A - (A \cdot U_R \cdot T_R) \cdot U_R^T \equiv A - Z_R \cdot U_R^T \quad .$$

Combining these, we can write

$$A(i,j) = A(i,j) - \sum_{k_L} U_L(i,k_L) \cdot Z_L(k_L,j) - \sum_{k_R} Z_R(i,k_R) \cdot U_R(j,k_R) \tag{4.4}$$

Of course there are lots of possible dependencies ignored here, much as we wrote down a similar formula for QR. At this point we can apply either of the two approaches in the last section: we can either assume (1) the number of R2/D2 Z_L's and Z_R's is bounded by the number of inputs and outputs $O(I+O)$ (see Section 4.2.1), or (2) all T matrices are 1x1 and we make "forward progress" (see Section 4.2.2). In case (1) it is straightforward to see that the same lower bound on the number of words moved applies as in Lemma 4.1: $\Omega(\max(\#\text{flops}/M^{1/2}, I+O))$

Case (2) requires a little more discussion to clarify the definitions of Partial Order (Definition 4.2) and forward progress (Definition 4.3): There will be two partial orders, one for U_L and one for U_R. In parts 1 and 2 of Definition 4.3, we insist that no transformation (from left or right) fills in or re-zeros out an entry deliberately zeroed out by another transformation (left or right). This implies that there is an ordering between left and right transformations, but we do not need to use this order for our counting argument. We also insist that part 3 of Definition 4.3 hold independently for the left and for the right transformations.

With these minor changes, we see that the lower bound argument of Section 4.2.2 applies independently to $U_L \cdot Z_L$ and $Z_R \cdot U_R^T$. In particular, insisting that left (right) transformations cannot fill in or re-zeros out entries deliberately zeroed out by right (left) transformations means that number of arithmetic operations performed by the the left and right transformations can be bounded independently and added. This leads to the same lower bound on the number of words moved as before (in a Big-Oh sense).

This lower bound applies to the conventional algorithms in LAPACK [ABB+92] and ScaLAPACK [BCC+97] for reduction to Hessenberg, tridiagonal and bidiagonal forms. See Section 6 for a discussion of which lower bounds are attained.

The lower bound also applies to reduction of a pair (A,B) to upper Hessenberg and upper triangular form: This is done by a QR decomposition of B (to which the lower bound for QR factorization applies), multiplying $Q^T A$ (to which we can again apply the QR lower bound argument (as long as the Householder vectors comprising Q satisfy the conditions of forward progress with respect to entries of B), and then reducing A to upper Hessenberg form (to which the argument in this section applies) while keeping B in upper triangular form. Since this involves filling in entries of B and zeroing them out again, our argument does not directly apply, but this is a fraction of the total work, and so would not change the lower bound in a Big-Oh sense.

Our lower bound also applies to the first phase of the successive-band-reduction algorithm of Bischof, Lang, and Sun, [BLS00a, BLS00b], namely reduction to narrow band form, because this satisfies our requirement of forward progress. However, the second phase of successive band reduction does not satify our requirement of forward progress, because it involves *bulge chasing*, i.e., repeatedly creating nonzero entries outside the band and zeroing them out again. Thus only one "pass" of bulge-chasing satifies forward-progress, not multiple passes. But since the first phase does asymptotically more arithmetic than the second phase, our lower bound based just on the first phase cannot be much improved (see Section 6 for more discussion of these and other algorithms).

Now we consider the rest of the eigenvalue or singular value problem. Once a symmetric matrix has been reduced to tridiagonal form T, it of course requires much less memory to store, just $O(n)$. Assuming M is at least a few times larger than n, there are a variety of classical algorithms to compute some or all of T's eigenvalues also using just $O(n)$ fast memory. So in the common case that n is at least a few times smaller than the fast memory size M, this can be done with as many slow memory references as there are inputs and outputs, which is a lower bound. A similar discussion applies to the SVD of a bidiagonal matrix B. Once the eigenvectors of T or singular vectors of B have been computed, they must be multiplied by the orthogonal matrices used in the reduction to get the final eigenvectors or singular vectors of A. Our previous analysis of applying Householder transformations applies here, as long as the Householder vectors satisfy forward progress with respect to the matrix from which they were computed. For example, in the two-phase successive band reduction algorithm, the lower bound does not apply to updating the eigenvector matrix with Householder vectors computed in the second phase (involving bulge-chasing), but it does apply to updating the eigenvectors with Householder transformations from the first phase (which satisfy forward progress).

Finally we consider the more challenging computation of the eigenvalues and eigenvectors of a Hessenberg matrix H. Our analysis applies to one pass (of bulge chasing) of standard QR iteration on a dense upper Hessenberg matrix to find its eigenvalues, but this does $O(n^2)$ flops on $O(n^2)$ data, and so does not improve the trivial lower bound of the input size. As discussed above, multiple bulge chasing passes do not satisfy our forward progress definition. We conjecture that improvements of Braman, Byers and Mathias [BBM02a, BBM02b] to combine m passes into one increase the flop count to $O(mn^2)$, while maintaining forward progress, letting us get a lower bound of $\Omega(mn^2/M^{1/2})$. This starts to get interesting as soon as $m > M^{1/2}$. In practice, for numerical reasons, m is usually chosen to be 256 or lower, which limits the applicability of this result.

5. Lower bounds for more general computations.

We next demonstrate how our lower bounds can be applied to more general computations where any or all of the following apply:

1. We might do a sequence of basic operations (matrix multiplication, LU, etc.).
2. The outputs of one operation are the inputs to a later one but do not necessarily need to be saved in slow memory,
3. The inputs may be computed by formulas (like $A(i,j) = 1/(i+j)$) requiring no memory traffic.
4. The ultimate output written to slow memory may just be a scalar, like the norm or determinant of a matrix.
5. An algorithm might compute but discard some results rather than save them to memory (e.g., ILU might discard entries of L or U whose magnitudes falls below a threshold).

In particular we would like a lower bound where we are allowed to arbitrarily interleave all the instructions from all basic operations in the computation together, and so get a lower bound for a global optimization of the entire program. For example, if two different matrix multiplications share a common input matrix, is it worth trying to interleave instructions from these two different matrix multiplications?

A natural question is whether it is good enough to just use optimal implementations of the basic operations, like matrix multiplication, to attain the global lower bound. This would clearly be the simplest way to implement the program. We know from experience that this is not always the case. For example, LU itself can be decomposed in many ways in terms of operations like matrix multiplication. Yet only recently have optimal LU algorithms been constructed. Previous LU algorithms did not attain optimal bandwidth-cost and latency-cost, even when each of their composing operations had optimal bandwidth-cost and latency-cost.

We give some examples, such as computing matrix powers, where it is indeed good enough to use repeated calls to an optimal matrix multiplication, as opposed to needing a new algorithm, and another example where the straightforward composition does not suffice, and a more careful interleaving of the computation is needed in order to attain the lower bound.

5.1. The Sequential Case.

5.1.1. Classical and Modified Gram-Schmidt. The classical and modified Gram-Schmidt orthogonalization algorithms discussed in Section 4.1 are often used just to generate an orthonormal basis of the subspace spanned by the input vectors. In this case, the triangular matrix R may not be written to slow memory. In order to apply Theorem 2.2, we impose writes (as described in Section 3.4) of the entries of R. For n vectors of length m, these $O(n^2)$ imposed writes are a lower order term compared to the communication lower bound $\Omega(mn^2/\sqrt{M})$.

5.1.2. A sequence of basic linear algebra operations. In the following example, we compose a sequence of basic linear algebra operations where intermediate outputs are used as inputs later, and never written to memory (e.g., computing consecutive powers of a matrix, or repeated squaring). Again, even though this seems to eliminate a large number of reads and writes, we show that in some cases the lower bound is still $\Omega\left(\frac{\#\text{flops}}{\sqrt{M}}\right)$, by imposing reads and writes and merging all the operations into a single set satisfying Equation (2.1). This means that in such cases we can simply call a sequence of individually optimized linear algebra routines and do asymptotically as well as we would do with any arbitrary interleaving.

Corollary 5.1 (Consecutive powers of a matrix). *Let A be an n-by-n matrix, and let Alg be a sequential algorithm that computes $A^2 = A \cdot A$, $A^3 = A^2 \cdot A$, ... , $A^t = A^{t-1} \cdot A$, but only needs to save A^t in slow memory. Let G be the total number of multiplications performed (e.g., $G = (t-1)n^3$ if A is dense), where we assume that each entry of each A^i is computed at most once. Then no matter how the operations of Alg are interleaved, its bandwidth-cost lower bound is $\Omega(\frac{G}{\sqrt{8M}} - M - (t-2)n^2)$ (if the A^i are sparse, we can subtract less than $(t-2)n^2$ and get a better lower bound).*

Proof. We give two proofs, each of which may be applied to other examples. For the first proof, we show how all the operations $A^2 = A \cdot A$, ... , $A^t = A^{t-1} \cdot A$, may be combined into one set to which Equation (2.1), and so Theorem 2.2, applies. For Equation (2.1) to apply, we must show that all the inputs, outputs and multiplications can be indexed by one index set (i, j, k) in the one-to-one manner described in section 2; this is most easily seen by writing all the operations as

$$\begin{pmatrix} A^2 \\ A^3 \\ \vdots \\ A^t \end{pmatrix} = \begin{pmatrix} A \\ A^2 \\ \vdots \\ A^{t-1} \end{pmatrix} \cdot A$$

Recall that Equation (2.1) permits inputs and output to overlap, and "$a(i,k)$" and "$b(k,j)$" inputs to overlap, but the "$a(i,k)$" inputs alone must be indexed one-to-one, and similarly the "$b(k,j)$" inputs alone must be indexed one-to-one; this is the case above.

Next, we impose writes of all the intermediate results $A^2, ..., A^{t-1}$, yielding a new algorithm Alg'. This means that there are no R2/D2 arguments, so Theorem 2.2 applies to Alg'. Thus the bandwidth-cost lower bound of Alg' is $\frac{G}{\sqrt{8M}} - M$, and the bandwidth-cost lower bound of Alg is lower by the number of imposed writes, at most $(t-2)n^2$ (less if the matrices are sparse).

Now we present a second proof, which uses the Loomis-Whitney-based analysis of a segment more directly. We let $\#A_i$ be the number of entries of A^i in fast memory during a segment of Alg'. From the definition of a segment, we can bound $\sum_{i=1}^t \#A_i \le 4M$. Applying Loomis-Whitney to each multiplication $A^{i+1} = A^i \cdot A$ that one might do (some of) during a segment, we can bound the number of multiplications during a segment by $F = \sum_{i=1}^{t-1} \sqrt{\#A_{i+1} \cdot \#A_i \cdot \#A_1}$. We can now bound F subject to the constraint $\sum_{i=1}^t \#A_i \le 4M$, yielding

$$\begin{aligned} F &= \sum_{i=1}^{t-1} \sqrt{\#A_{i+1} \cdot \#A_i \cdot \#A_1} \\ &= \sqrt{\#A_1} \cdot \sum_{i=1}^{t-1} \sqrt{\#A_{i+1} \cdot \#A_i} \\ &\le \sqrt{\#A_1} \cdot \sqrt{\sum_{i=1}^{t-1} \#A_{i+1}} \cdot \sqrt{\sum_{i=1}^{t-1} \#A_i} \quad \text{... by the Cauchy} - \text{Schwarz inequality} \\ &\le \sqrt{4M} \cdot \sqrt{4M} \cdot \sqrt{4M} = 8\sqrt{M^3} \end{aligned}$$

This yields the ultimate bandwidth-cost lower bound of $G/(8\sqrt{M}) - M$. □

Both proof techniques also apply to repeated squaring: $A_{i+1} = A_i^2$ for $i = 1, ..., t-1$, the first proof via the identity

$$\begin{pmatrix} A^2 & & & \\ & A^4 & & \\ & & \ddots & \\ & & & A^{2^t} \end{pmatrix} = \begin{pmatrix} A & & & \\ & A^2 & & \\ & & \ddots & \\ & & & A^{2^{t-1}} \end{pmatrix} \cdot \begin{pmatrix} A & & & \\ & A^2 & & \\ & & \ddots & \\ & & & A^{2^{t-1}} \end{pmatrix}$$

and the second proof by bounding the number of multiplications during a segment by maximizing $F = \sum_{i=1}^{t-1} \sqrt{\#A_i \cdot \#A_i \cdot \#A_{i+1}}$ subject to $\sum_{i=1}^t \#A_i \le 4M$ (here $\#A_i$ denotes the number of entries of $A^{2^{i-1}}$ available during a segment).

5.1.3. Interleaved vs. Phased Sequences of Operations. In some cases, one can combine and interleave basic linear algebra operations, (e.g., a sequence of matrix multiplications) so that the resulting algorithm no longer agrees with Equation (2.1), although the algorithms for performing each of the basic linear algebra operations separately do agree with Equation (2.1). This may lead to an algorithm whose minimum communication is *not* proportional to #flops, but asymptotically better.

Before giving an example, we first observe that a "phased" algorithm, consisting of a sequence of calls to individually optimized basic linear algebra operations (like matrix multiplication), where each such basic linear algebra operation (phase) must complete before the next can begin, can offer no such asymptotic improvements. Indeed, if we perform Alg_1, ... ,Alg_t in phases, where Alg_i has bandwidth-cost lower bound B_i, then the sequence has bandwidth-cost lower bound $B = \sum_{i=1}^{t} B_i - 2(t-1)M$. If each B_i is proportional to the operation count of Alg_i, then B is proportional to the total operation count. (The modest improvement $2(t-1)M$ arises since we can possibly avoid a little communication by Alg_{i+1} using the results left in fast memory by Alg_i.)

Let us now look at an example, where the interleaved algorithm can do asymptotically less communication than the phased algorithm: Consider computing the dense matrix multiplications $C^{(k)} = A \cdot B^{(k)}$ for $k = 1, 2, ..., t$ where $B_{i,j}^{(k)} = \sqrt[k]{B_{i,j}}$.

The idea is that having both $A_{i,k}$ and $B_{k,j}$ in fast memory lets us do up to t evaluations of g_{ijk}. Moreover, the union of all these tn^3 operations does not match Equation (2.1), since the inputs $B_{k,j}$ cannot be indexed in a one-to-one fashion. However, we can still give a non-trivial lower bound as follows, analyzing the algorithm segment by segment. Let us begin with the lower bound, then show an algorithm attaining this lower bound.

No operands in a segment are R2/D2. By the same argument as in Section 2, a maximum of $4M$ arguments of A, B and any $C^{(i)}$'s are available during a segment. We want to bound the number of g_{ijk}'s that we can do during such a segment. Let $\#A, \#B$ and $\#C^{(i)}$ denote the number of each type of argument available during the segment. Then by Loomis-Whitney (applied t times) the maximum number of g_{ijk}'s is bounded by $F = \sum_{i=1}^{t} \sqrt{\#A \cdot \#B \cdot \#C^{(i)}}$. We want to maximize F subject to the constraint $\#A + \#B + \sum_{i=1}^{t} \#C^{(i)} \le 4M$. Applying Cauchy-Schwarz as before yields

$$\begin{aligned} F &= \sqrt{\#A} \cdot \sqrt{\#B} \cdot \sum_{i=1}^{t} \sqrt{\#C^{(i)}} \le \sqrt{\#A} \cdot \sqrt{\#B} \cdot \sqrt{\sum_{i=1}^{t} \#C^{(i)}} \cdot \sqrt{t} \\ &\le \sqrt{4M} \cdot \sqrt{4M} \cdot \sqrt{4M} \cdot \sqrt{t} = 8\sqrt{tM^3} \end{aligned}$$

The number of segments is thus at least $\left\lfloor \frac{tn^3}{8M^{3/2}t^{1/2}} \right\rfloor$ and the number of memory operations at least $\frac{t^{1/2}n^3}{8M^{1/2}} - M$. This is smaller than the "phased" lower bound for t matrix multiplications in sequence, $\frac{tn^3}{8\sqrt{M}} - tM$, by an asymptotic factor of $\Theta(\sqrt{t})$.

We next show that this bound is indeed attainable, using a different blocked matrix multiplication algorithm whose block sizes b_1 and b_2 depend on M and t (see Algorithm 1). The bandwidth-cost count for this algorithm is as follows. In the innermost loop we read/write t blocks of $C^{(1)}, ..., C^{(t)}$, of $M/3t$ words each. So we have $2M/3$ reads/writes for the innermost loop. Before this loop we read two blocks (of A and B) of $M/3$ words each. This adds up to $O(M)$ read/writes. This is performed $\frac{n^3}{b_1^2 b_2}$ times. So the total bandwidth-cost count is $O\left(M \cdot \left(\frac{n^3}{b_1^2 b_2}\right)\right) = O\left(\frac{\sqrt{t}n^3}{\sqrt{M}}\right)$.

Algorithm 1 Matrix-Matrices multiplication

```
1: $b_1 = \sqrt{M/3t}$, $b_2 = \sqrt{Mt/3}$, {so $b_1 b_2 = M/3$ }
2: Break $A$ into blocks of size $b_1 \times b_2$.
3: Break $B$ into blocks of size $b_2 \times b_1$.
4: Break each $C^{(i)}$ into blocks of size $b_1 \times b_1$.
5: Do block matrix multiplication, where the innermost loop reads in a block of $A$,
   a block of $B$, and one block each of $C^{(1)}, ...., C^{(t)}$, and updates each $C^{(i)}$ :
6: for $i = 1$ to $n/b_1$ do
7:   for $j = 1$ to $n/b_1$ do
8:     for $k = 1$ to $n/b_2$ do
9:       Read block $A_{i,k}$ and block $B_{k,j}$
10:      for $m = 1$ to $t$ do
11:        Read block $C^{(m)}_{i,j}$
12:        $C^{(m)}_{i,j} += A_{i,k} \cdot (B^{(m)}_{k,j})$        ...{$(B^{(m)}_{k,j})$ is recomputed each time }
13:        Write $C^{(m)}_{i,j}$
14:      end for
15:    end for
16:  end for
17: end for
```

5.2. The Parallel Case. The techniques in the above Section 5.1 for composing sequential linear algebra operations can be extended to the parallel case in two different ways. When we impose reads and writes to get an algorithm to which our previous lower bounds apply, we need to decide which processor's memory will participate in those reads and writes. The first option is to create a "twin processor" for each processor, whose memory will hold this data. This doubles the number of processors to which the previous lower bound applies, and also requires us to bound the total memory per processor not by $\mathcal{NNZ}/P$ (again assuming memory is balanced among processors) but by the maximum of $\mathcal{NNZ}/P$ and the largest number of reads and writes imposed on any processor. The second option is to have all the imposed reads and writes be in the local processor's memory. This keeps the number of processors constant, but increases $\mathcal{NNZ}/P$ by adding the largest number of imposed reads and writes on each processor. The details are algorithm-dependent. For example, similar to the sequential case, we obtain a tight lower bound for repeated matrix multiplication and for repeated matrix squaring.

5.3. Applications to Graph Algorithms. Matrix multiplication algorithms are used to solve many graph related problems. Thus our lower bounds may hold, as long as the matrix multiplication algorithm that is used agrees with Equation (2.1). The bounds, however, do not apply when using Strassen-like algorithm (e.g., [YZ05]).

In some cases, one can directly match the flops performed by an algorithm to Equation (2.1), and obtain a communication lower bound (e.g., computing All-Pairs-Shortest-Path using repeated squaring gives an arithmetic count of $\Theta\left(n^3 \log n\right)$ and bandwidth-cost of $\Theta\left(\frac{n^3 \log n}{\sqrt{M}}\right)$).

We next consider, for example, matrix-multiplication-like recursive algorithms for finding the shortest path between any pair of vertices in a graph (the All-Pairs Shortest-Path problem). For tight upper and lower bounds for the bandwidth-cost of Floyd-Warshall and other related algorithms, see [MPP02]. The algorithm works as follows [CLRS01]. Let $l_{ij}^{(m)}$ be the minimum weight of any path from vertex i to vertex j that contains at most m edges, where the weight of the edge (i,j) is $w_{ij} = l_{ij}^{(1)}$. Then $l_{ij}^{(m)} = \min_{1\le k\le n}\left(l_{ik}^{(m-1)} + w_{kj}\right)$, and the recursive naive algorithm for the All-Pairs Shortest-Path problems performs exactly these $\Theta(n^4)$ computations. If all values $l_{ij}^{(m)}$ are written to slow memory, then, by Theorem 2.2, the bandwidth-cost lower bound is $\Omega\left(\frac{n^4}{\sqrt{M}}\right)$. Although this may not be the case —some of the intermediate values may never reach the slow memory— there are fewer than n^3 intermediate $l_{ij}^{(m)}$ values. Thus, by imposing reads and writes, the bandwidth-cost lower bound is $\Omega\left(\frac{n^4}{\sqrt{M}}\right)$ (note that here, similar to the repeated matrix multiplication arguments of Corollary 5.1, after imposing writes, no two g_{ijk} operations use the same two inputs, so Equation 2.1 applies). Similarly, the $\Theta(n^3\log n)$ recursive algorithm for APSP has $O(n^2\log n)$ intermediate values, therefore, by Theorem 2.2 and imposing reads and writes, the bandwidth-cost lower bound is $\Omega\left(\frac{n^3\log n}{\sqrt{M}}\right)$.

Note that these lower bounds are attainable. As noted before (see e.g., [CLRS01]) any matrix powering algorithm can be converted into a APSP algorithm, by using '+' instead of '$*$' and 'min' instead of summation. Starting with any of the communication-avoiding optimal matrix-multiplication algorithms (e.g., [FLPR99]) guarantees a bandwidth-cost upper bound of $O\left(\frac{n^4}{\sqrt{M}}\right)$ and $O\left(\frac{n^3\log n}{\sqrt{M}}\right)$ respectively. Using recursive-block data structure further guarantees optimal latency-cost for both algorithms.

The above repeated-matrix-squaring-like algorithm may, in some cases, perform better than the communication-avoiding implementation of Floyd-Warshall algorithm [MPP02]. Consider the problem of finding the neighbors within distance t of every vertex.

One can use the above repeated-matrix-squaring-like algorithm for $\log t$ phases, obtaining a running time of $\Theta(n^3\log t)$ and communication cost $\Theta\left(\frac{n^3\log t}{\sqrt{M}}\right)$ for dense graphs. For sparse input graphs this may be further reduced. For example, when G is a union of cycles and paths, the running time and communication bandwidth-cost are $O(n^2 2^t)$ and $O\left(\frac{n^2 2^t}{\sqrt{M}}\right)$ (as the degree of a vertex of the ith phase is at most 2^{2^i}).

If, however, we use the Floyd-Warshall algorithm for this purpose, we have to run it all the way through, regardless of the input graph, resulting in running time of $\Theta(n^3)$ and communication cost of $\Theta\left(\frac{n^3}{\sqrt{M}}\right)$ (assuming the above communication-avoiding implementation). Thus, for $t = o(\log n)$ the repeated-matrix-squaring-like algorithm performs better for constant-degree inputs, both from flops count and from communication bandwidth-cost perspectives.

6. Attaining the lower bounds, and open problems.

A major problem is to find algorithms that attain the lower bounds described in this paper, for bandwidth and latency costs, for the various linear algebra problems, for dense and sparse matrices, and for sequential and parallel machines. And since real computers generally have many levels of memory hierarchy, and possibly levels of parallelism as well (cores on a chip, chips in a node, nodes in a rack, racks in a room...) we would ideally like to minimize communication between all of them simultaneously (i.e., between L1 and L2 cache, between L2 cache and main memory, between memories of different processors, and so on). It is easy to see that our lower bounds can be applied hierarchically to this situation, for example, by treating L1 and L2 cache as “fast memory” and L3 cache and DRAM as “slow memory”, to bound below memory traffic between L3 and L2 cache.

Tables 6.1 and 6.2 summarize the current state-of-the-art (to the best of our knowledge) for the communication cost of algorithms for dense matrices. To summarize, in the dense sequential case (Table 6.1), for most important problems, the lower bounds are attained for 2 levels of memory hierarchy (excluding Gram–Schmidt and modified Gram–Schmidt algorithms), but fewer are attained so far for multiple levels, at least without constant factor increases in the amount of arithmetic. In the dense parallel case (Table 6.2), for most important problems, the lower bounds are also attained (again, excluding Gram–Schmidt and modified Gram–Schmidt algorithms), assuming minimal memory $O(n^2/P)$ per processor, and modulo polylog$P$ terms. Again, some of these algorithms do a constant factor times as much arithmetic as their conventional counterparts.

However, only a few of these communication-optimal algorithms appear in standard libraries like LAPACK [ABB$^+$92] and ScaLAPACK [BCC$^+$97]; the complexity of ScaLAPACK implementations in Table 6.2 is taken from [BCC$^+$97, Table 5.8]. (Other libraries may well attain similar bounds [GGHvdG01, vdG].) Several of the papers cited below report large speedups compared to these standard libraries.

When there is enough memory per processor for $c > 1$ copies of the data ($M = cn^2/p$ instead of $M = n^2/p$), the lower bound on the number of words decreases by a factor of $c^{1/2}$ and the lower bound on the number of messages decreases by a factor $c^{3/2}$. So far only a few algorithms are known that achieve these smaller lower bounds, for dense matrix multiplication and (just for the number of words) LU decomposition [SD11, MT99, DNS81, ABG$^+$95]. (We note that c cannot be arbitrarily large; the proof breaks down when the lower bound on the number of messages reaches 1, i.e., c reaches $p^{1/3}$.)

We note that in practice, a collection of words must be stored in contiguous locations in order to be transferred as a single message at maximum bandwidth; this is a consequence of common hardware design limitations. On a parallel computer, the processor can in principle repack locally stored noncontiguous data into a separate contiguous region before sending it to another processor. But on a sequential computer, the data structure must have the property that desired data (a submatrix, say) is already stored contiguously. But if a matrix is stored row-wise or column-wise, then most submatrices (those not consisting of complete rows or columns) will not have this property. This means that in order to achieve the lower bound on the number of messages, sequential algorithms must not store matrices row-wise or column-wise, but block-wise. And in order to to minimize the number of messages when there is more than one level of memory hierarchy, these blocks must themselves be stored block-wise, leading to data structures known by various names in the literature, such as *recursive block layout* or storage using *space-filling curves* or *Morton-ordered quadtree matrices* [EGJK04]. The algorithms referred to in Table 6.1 as minimizing the number of messages assume such data-structures are used.

One may imagine that sequential algorithms that minimize communication for any number of levels of memory hierarchy might be very complex, possibly depending not just on the number of levels, but their sizes. It is worth distinguishing a class of algorithms, called *cache-oblivious* [FLPR99], that can sometimes minimize communication between all levels (at least asymptotically) independent of the number of levels and their sizes. These algorithms are recursive, for example multiplying two n-by-

Algorithm	Two Levels of Memory		Multiple Levels of Memory	
	Minimizes # words moved	and #messages	Minimizes # words moved	and # messages
BLAS3	usual blocked or recursive algorithms [Gus97, FLPR99]		usual (nested) blocked or recursive algorithms [Gus97, FLPR99]	
Cholesky	LAPACK (with $b = M^{1/2}$) [Gus97] [AP00] [BDHS10a]	[Gus97] [AP00] [BDHS10a]	[Gus97] [AP00] [BDHS10a]	[Gus97] [AP00] [BDHS10a]
LU with pivoting	LAPACK (rarely) [Tol97] [DGX08] [DGX10]	[DGX08] [DGX10]	[Tol97]	?
QR	LAPACK (rarely) [FW03] [EG98] [DGHL08a]	[FW03] [DGHL08a]	[FW03] [EG98]	[FW03]
Eig, SVD	[BDD11]		[BDD11]	

TABLE 6.1

Sequential $\Theta(n^3)$ algorithms attaining communication lower bounds. We separately list algorithms that attain the lower bounds for 2 levels of memory hierarchy, and multiple levels. In each of these cases, we separately list algorithms that only minimize the number of words moved, and algorithms that also minimize the number of messages.

Algorithm	Reference	Factor exceeding lower bound for #words_moved	Factor exceeding lower bound for #messages
Matrix-multiply	[Can69]	1	1
Cholesky	ScaLAPACK	$\log P$	$\log P$
LU with pivoting	[DGX08, DGX10]	$\log P$	$\log P$
	ScaLAPACK	$\log P$	$(n/P^{1/2}) \log P$
QR	[DGHL08a]	$\log P$	$\log^3 P$
	ScaLAPACK	$\log P$	$(n/P^{1/2}) \log P$
SymEig, SVD	[BDD11]	$\log P$	$\log^3 P$
	ScaLAPACK	$\log P$	$n/P^{1/2}$
NonymEig	[BDD11]	$\log P$	$\log^3 P$
	ScaLAPACK	$P^{1/2} \log P$	$n \log P$

TABLE 6.2

Parallel $\Theta\left(\frac{n^3}{P}\right)$ flops algorithms with $M = \Theta\left(\frac{n^2}{P}\right)$ memory per processor: In this case the common lower bounds for all algorithms listed are #words_moved = $\Omega(n^2/P^{1/2})$ and #messages = $\Omega(P^{1/2})$ (both refer the number of words and messages sent by at least one processor to some other processors). The table shows the factors by which the listed algorithms exceed the respective lower bound, i.e., the ratio upper_bound / lower_bound (so 1 is optimal). ScaLAPACK refers to [BCC+ 97]. All entries are to be interpreted in a Big-O sense.

n matrices by recursively multiplying $\frac{n}{2}$-by-$\frac{n}{2}$ submatrices and adding these partial products. Provided a recursive block layout described above is used, these algorithms may also minimize the number of messages independent of the number of levels of memory hierarchy. All the algorithms cited in Table 6.1 that work for arbitrary levels of memory hierarchy are cache-oblivious. (In practice one does not recur down to 1-by-1 submatrices because of the high overhead. Also, some cache-oblivious algorithms require a constant factor more arithmetic operations than non-oblivious alternatives [FW03]. So "pure" cache-obliviousness is not a panacea.)

We now discuss these tables in more detail. There is a very large body of work on many of these algorithms, and we do not pretend to have a complete list of citations. Instead we refer just to papers where these algorithms first appeared (to the best of our knowledge), with or without analysis of their communication costs (often without), or to survey papers.

Best understood are dense matrix-multiplication, other BLAS routines, and Cholesky, which have algorithms that attain (perhaps modulo polylog$P$ factors) both bandwidth and latency lower bounds on parallel machines, and on sequential machines with multiple levels of memory hierarchy. The optimal sequential Cholesky algorithm cited in Table 6.1 was presented in [Gus97, AP00], but first analyzed later in [BDHS10a]. The algorithm in [AP00, BDHS10a] is cache-oblivious, but whether or not the recursive algorithm in [Gus97] minimizes communication for many levels of memory hierarchy depends on the implementation of the underlying BLAS library that it uses. The complexity of ScaLAPACK's parallel Cholesky cited in Table 6.2 assumes that the largest possible block size is chosen ($NB \approx n/\sqrt{P}$ in line "PxPOSV" in [BCC$^+$97, Table 5.8]).

More recently, optimal dense LU and QR algorithms have been proposed that attain both bandwidth and latency lower bounds in parallel or sequentially (with just 2 levels of memory hierarchy). LAPACK is labeled "rarely" because only for some matrix dimensions n and fast memory sizes M is it possible to choose a block size b to attain the lower bound. Interestingly, conventional partial pivoting must apparently be replaced by a different (but still stable) pivoting scheme in order to minimize latency costs in LU [DGX08, DGX10]; we can retain partial pivoting if we only want to minimize bandwidth [Tol97]. Similarly, we must apparently change the standard representation of the Q matrix in QR in order to minimize both latency and bandwidth costs [DGHL08a]; we can retain the usual representation if we only want to minimize bandwidth costs in the sequential case [EG98]. Both [EG98] and [FW03] are cache oblivious, but only [FW03] also minimizes latency costs; however it triples the arithmetic operation count to do so. See the above references for large speedups reported over algorithms that do not try to minimize communication. The ideas behind communication-optimal dense QR first appear in [GPS88], and include [BLKD07, GG05, EG98]; see [DGHL08a] for a more complete list of references.

ScaLAPACK's parallel symmetric eigensolver and SVD routine also minimize bandwidth cost (modulo a $\log P$ factor), but not the latency cost, sending $O(n/P^{1/2})$ times as many messages. ScaLAPACK's nonsymmetric eigensolver communicates much more, indeed just the Hessenberg QR iteration has n-times higher latency cost. LAPACK's symmetric and nonsymmetric eigensolvers and SVD minimize neither bandwidth nor latency costs, moving $O(n^3)$ words. Recently proposed randomized algorithms in [BDD11, DDH07] for the symmetric and nonsymmetric eigenproblems, generalized nonsymmetric eigenproblems and SVD do attain the desired communication cost (modulo polylog$P$ factors) but at the cost of doing a possibly large constant factor more arithmetic. (This is in contrast to the new dense LU and QR algorithms, which do at most $O(n^2)$ more arithmetic operations than the $O(n^3)$ operations done by their conventional counterparts.) In [BDD11] it is also pointed out that appropriate variants of the "successive band reduction" approach in [BLS00a, BLS00b] can also minimize communication, at least in the sequential case for the symmetric eigenproblem and SVD, for a much smaller increase in the arithmetic operation count (nearly no increase, if eigenvalues/singular values alone are desired).

The eigenvalue algorithms mentioned above use randomization to implement a URV decomposition that reveals the rank with high probability; here U and V are orthogonal and R is upper triangular, with the large singular values "in the upper left corner" of R, and the small singular values "in the lower right corner." In fact we can perform an implicit randomized rank-revealing URV factorization on an arbitrary product $\prod_i A_i^{\pm 1}$ without the need to multiply or invert any of the factors A_i, and so retain numerical stability.

Devising algorithms that attain the communication lower bounds while performing QR with column pivoting, LU with complete pivoting, or LDL^T factorization with any pivoting remain work in progress. It also remains an open problem to design parallel algorithms (besides matrix multiplication and LU decomposition) that can take advantage of extra memory (a multiple of the minimal n^2/p per processor) to further reduce communication. Finally, finding optimal algorithms for heterogenous computers (eg CPUs and GPUs), where each processor has a different fast memory size, bandwidth, latency and floating point speed, remains open.

It is possible to extend our lower bound results to many Strassen-like algorithms [BDHS10b] for matrix multiplication, which are attained by the natural recursive sequential implementations, and are attainable in parallel as well. But the lower bound proof is significantly different than the one used in this paper. By using recursive algorithms in [DDH07], it is possible to compute LU, QR and other factorizations while doing asymptotially as little arithmetic and commucation (at least sequentially) as Strassen-like matrix multiplication. But it remain an open problem to extend the lower bounds to any implementation of "Strassen-like LU", "Strassen-like QR", etc.

For the Cholesky factorization of sparse matrices, whose sparsity structure satisfy certain graph-theoretic conditions (having "good separators"), the lower bounds can also be attained [DDGP10]. For sparse matrix algorithms more generally, the problems are open.

We note that for sufficiently rectangular dense matrices (e.g., matrix-vector multiplication) or for sufficiently sparse matrices (e.g., multiplying diagonal matrices), our lower bound may be lower than the trivial lower bound (#inputs + #outputs) and so not be attainable. In this case the natural question is whether the maximum of the two lower bounds is attainable (as it is for dense matrix multiplication).

Acknowledgments.
The authors thank Cleve Ashcraft, Robert Luce, Nicholas Knight, Edgar Solomonik, and Sivan Toledo for a number of helpful comments, questions and discussions, as well as the anonymous reviewers for their valuable comments.

REFERENCES

[ABB$^+$92] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov, and D. Sorensen. *LAPACK's user's guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1992. Also available from http://www.netlib.org/lapack/.

[ABG$^+$95] R. C. Agarwal, S. M. Balle, F. G. Gustavson, M. Joshi, and P. Palkar. A three-dimensional approach to parallel matrix multiplication. *IBM J. Res. Dev.*, 39:575–582, September 1995.

[AGW01] B. S. Andersen, F. Gustavson, and J. Wasniewski. A recursive formulation of Cholesky factorization of a matrix in packed storage format. *ACM Transactions on Mathematical Software*, 27(2):214–244, jun 2001.

[AP00] N. Ahmed and K. Pingali. Automatic generation of block-recursive codes. In *Euro-Par '00: Proceedings from the 6th International Euro-Par Conference on Parallel Processing*, pages 368–378, London, UK, 2000. Springer-Verlag.

[Ash91] C. Ashcraft. A taxonomy of distributed dense LU factorization methods. Boeing Computer Services Technical Report ECA-TR-161, March 1991.

[Ash93] C. Ashcraft. The fan-both family of column-based distributed Cholesky factorization algorithms. In John R. Gilbert Alan George and Joseph W. H. Liu, editors, *Graph Theory and Sparse Matrix Computation*, volume 56 of *IMA Volumes in Mathematics and its Applications, Springer-Verlag*, pages 159–190, 1993.

[AV88] A. Aggarwal and J. S. Vitter. The input/output complexity of sorting and related problems. *Commun. ACM*, 31(9):1116–1127, 1988.

[BBF$^+$07] M. A. Bender, G. S. Brodal, R. Fagerberg, R. Jacob, and E. Vicari. Optimal sparse matrix dense vector multiplication in the I/O-model. In *SPAA '07: Proceedings of the nineteenth annual ACM symposium on parallel algorithms and architectures*, pages 61–70, New York, NY, USA, 2007. ACM.

[BBM02a] K. Braman, R. Byers, and R. Mathias. The Multi-Shift QR Algorithm, Part I: Maintaining Well Focused Shifts and Level 3 Performance. *SIAM J. Matrix Anal. App.*, 23(4):929–947, 2002.

[BBM02b] K. Braman, R. Byers, and R. Mathias. The Multi-Shift QR Algorithm, Part II: Aggressive Early Deflation. *SIAM J. Matrix Anal. App.*, 23(4):948–973, 2002.

[BCC$^+$97] L. S. Blackford, J. Choi, A. Cleary, E. DAzevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley. *ScaLAPACK Users' Guide*. SIAM, Philadelphia, PA, USA, May 1997. Also available from http://www.netlib.org/scalapack/.

[BDD$^+$01] L. S. Blackford, J. Demmel, J. Dongarra, I. Duff, S. Hammarling, G. Henry, M. Heroux, L. Kaufman, A. Lumsdaine, A. Petitet, R. Pozo, K. Remington, R. C. Whaley, Z. Maany, F. Krough, G. Corliss, C. Hu, B. Keafott, W. Walster, and J. Wolff v. Gudenberg. Basic Linear Algebra Subprograms Techical (BLAST) Forum Standard. *Intern. J. High Performance Comput.*, 15(3-4), 2001.

[BDD$^+$02] L. S. Blackford, J. Demmel, J. Dongarra, I. Duff, S. Hammarling, G. Henry, M. Heroux, L. Kaufman, A. Lumsdaine, A. Petitet, R. Pozo, K. Remington, and R. C. Whaley. An updated set of Basic Linear Algebra Subroutines (BLAS). *ACM Trans. Math. Soft.*, 28(2), June 2002.

[BDD11] G. Ballard, J. Demmel, and I. Dumitriu. Communication-optimal parallel and sequential eigenvalue and singular value algorithms. EECS Technical Report EECS-2011-14, UC Berkeley, February 2011.

[BDHS10a] G. Ballard, J. Demmel, O. Holtz, and O. Schwartz. Communication-optimal parallel and sequential Cholesky decomposition. *SIAM J. Sci. Comput.*, 32:3495–3523, 2010.

[BDHS10b] G. Ballard, J. Demmel, O. Holtz, and O. Schwartz. Graph Expansion and Communication Costs of Algorithms, 2010. Submitted.

[BK77] J. Bunch and L. Kaufman. Some stable methods for calculating inertia and solving symmetric linear systems. *Mathematics of Computation*, 31(137):163–179, January 1977.

[BLKD07] A. Buttari, J. Langou, J. Kurzak, and J. J. Dongarra. A class of parallel tiled linear algebra algorithms for multicore architectures. Technical Report 191, LAPACK Working Note, September 2007.

[BLS00a] C. H. Bischof, B. Lang, and X. Sun. Algorithm 807: The SBR Toolbox—software for successive band reduction. *ACM Transactions on Mathematical Software*, 26(4):602–616, December 2000.

[BLS00b] C. H. Bischof, B. Lang, and X. Sun. A framework for symmetric band reduction. *ACM Transactions on Mathematical Software*, 26(4):581–601, December 2000.

[BVL87] C. Bischof and C. Van Loan. The WY representation for products of Householder matrices. *SIAM J. Sci. Stat. Comp.*, 8(1), 1987.

[BZ88] Y. D. Burago and V. A. Zalgaller. *Geometric Inequalities*, volume 285 of *Grundlehren der Mathematische Wissenschaften.* Springer, Berlin, 1988.

[Can69] L. Cannon. *A cellular computer to implement the Kalman filter algorithm.* PhD thesis, Montana State University, Bozeman, MN, 1969.

[CLRS01] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms.* The MIT Press, 2nd edition, 2001.

[CR06] R. A. Chowdhury and V. Ramachandran. Cache-oblivious dynamic programming. In *SODA '06: Proceedings of the seventeenth annual ACM-SIAM symposium on discrete algorithms*, pages 591–600, New York, NY, USA, 2006. ACM.

[DDGP10] P.-Y. David, J. Demmel, L. Grigori, and S. Peyronnet. Brief announcement: Lower bounds on communication for sparse Cholesky factorization of a model problem. In *22nd ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2010.

[DDH07] J. Demmel, I. Dumitriu, and O. Holtz. Fast linear algebra is stable. *Numerische Mathematik*, 108(1):59–91, 2007.

[Dem96] J. Demmel. CS 267 Course Notes: Applications of Parallel Processing. Computer Science Division, University of California, 1996. http://www.cs.berkeley.edu/∼demmel/cs267.

[Dem97] J. Demmel. *Applied Numerical Linear Algebra.* SIAM, 1997.

[DGHL08a] J. Demmel, L. Grigori, M. Hoemmen, and J. Langou. Communication-optimal parallel and sequential QR and LU factorizations. UC Berkeley Technical Report EECS-2008-89, Aug 1, 2008; Submitted to SIAM. J. Sci. Comp., 2008.

[DGHL08b] J. Demmel, L. Grigori, M. Hoemmen, and J. Langou. Implementing communication-optimal parallel and sequential QR and LU factorizations. Submitted to SIAM. J. Sci. Comp., 2008.

[DGX08] J. Demmel, L. Grigori, and H. Xiang. Communication-avoiding Gaussian elimination. Supercomputing 08, 2008.

[DGX10] J. Demmel, L. Grigori, and H. Xiang. CALU: A communication optimal LU factorization algorithm. EECS Technical Report EECS-2010-29, UC Berkeley, March 2010. Submitted to SIAM J. Matrix Anal. Appl.

[DNS81] Eliezer Dekel, David Nassimi, and Sartaj Sahni. Parallel matrix and graph algorithms. *SIAM Journal on Computing*, 10(4):657–675, 1981.

[EG98] E. Elmroth and F. Gustavson. New serial and parallel recursive QR factorization algorithms for SMP systems. In B. Kågström et al., editor, *Applied Parallel Computing. Large Scale Scientific and Industrial Problems.*, volume 1541 of *Lecture Notes in Computer Science*, pages 120–128. Springer, 1998.

[EGJK04] E. Elmroth, F. Gustavson, I. Jonsson, and B. Kågström. Recursive blocked algorithms and hybrid data structures for dense matrix library software. *SIAM Review*, 46(1):3–45, March 2004.

[FLPR99] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran. Cache-oblivious algorithms. In *FOCS '99: Proceedings of the 40th Annual Symposium on Foundations of Computer Science*, page 285, Washington, DC, USA, 1999. IEEE Computer Society.

[FW03] J. D. Frens and D. S. Wise. QR factorization with Morton-ordered quadtree matrices for memory re-use and parallelism. *SIGPLAN Not.*, 38(10):144–154, 2003.

[Geo73] A. George. Nested dissection of a regular finite element mesh. *SIAM J. Numer. Anal.*, 10:345–363, 1973.

[GG05] B. C. Gunter and R. A. Van De Geijn. Parallel out-of-core computation and updating of the QR factorization. *ACM Trans. Math. Softw.*, 31(1):60–78, 2005.

[GGHvdG01] J. A. Gunnels, F. G. Gustavson, G. M. Henry, and R. A. van de Geijn. FLAME: Formal Linear Algebra Methods Environment. *ACM Transactions on Mathematical Software*, 27(4):422–455, December 2001.

[GPS88] G. H. Golub, R. J. Plemmons, and A. Sameh. Parallel block schemes for large-scale least-squares computations. In *High-speed computing: scientific applications and algorithm design*, pages 171–179, Champaign, IL, USA, 1988. University of Illinois Press.

[GT87] J. R. Gilbert and R. E. Tarjan. The analysis of a nested dissection algorithm. *Numerische Mathematik*, pages 377–404, 1987.

[Gus97] F. G. Gustavson. Recursion leads to automatic variable blocking for dense linear-algebra algorithms. *IBM J. Res. Dev.*, 41(6):737–756, 1997.

[GVL96] G. Golub and C. Van Loan. *Matrix Computations.* Johns Hopkins University Press, Baltimore, MD, 3rd edition, 1996.

[HK81] J. W. Hong and H. T. Kung. I/O complexity: The red-blue pebble game. In *STOC '81: Proceedings of the thirteenth annual ACM symposium on theory of computing*, pages 326–333, New York, NY, USA, 1981. ACM.

[HMR73] A. J. Hoffman, M. S. Martin, and D. J. Rose. Complexity bounds for regular finite difference and finite element grids. *SIAM J. Numer. Anal.*, 10:364–369, 1973.

[IT02] D. Irony and S. Toledo. Trading replication for communication in parallel distributed-memory dense solvers. *Parallel Processing Letters*, 12(1):79–94, 2002.

[ITT04] D. Irony, S. Toledo, and A. Tiskin. Communication lower bounds for distributed-memory matrix multiplication. *J. Parallel Distrib. Comput.*, 64(9):1017–1026, 2004.

[LW49] L. H. Loomis and H. Whitney. An inequality related to the isoperimetric inequality. *Bulletin of the AMS*, 55:961–962, 1949.

[MPP02] J. P. Michael, M. Penner, and V. K. Prasanna. Optimizing graph algorithms for improved cache performance. In *In Proc. Intl Parallel and Distributed Processing Symp. (IPDPS 2002), Fort Lauderdale, FL*, pages 769–782, 2002.

[MT99] W. F. McColl and A. Tiskin. Memory-efficient matrix multiplication in the BSP model. *Algorithmica*, 24:287–297, 1999.

[Pug92] C. Puglisi. Modification of the Householder method based on compact WY representation. *SIAM J. Sci. Stat. Comput.*, 13(3):723–726, 1992.

[Saa86] Y. Saad. Communication complexity of the Gaussian elimination algorithm on multiprocessors. *Linear Algebra Appl.*, 77:315–340, 1986.

[Saa96] Y. Saad. *Iterative Methods for Sparse Linear Systems.* PWS Publishing Co., Boston, 1996.

[Sav95] J. E. Savage. Extending the Hong-Kung model to memory hierarchies. In *COCOON*, pages 270–281, 1995.

[SD11] E. Solomonik and J. Demmel. Communication-optimal parallel 2.5D matrix multiplication and LU factorization algorithms. EECS Technical Report EECS-2011-10, UC Berkeley, February 2011.

[SVL89] R. Schreiber and C. Van Loan. A storage efficient WY representation for products of Householder transformations. *SIAM J. Sci. Stat. Comput.*, 10:53–57, 1989.

[Tol97] S. Toledo. Locality of reference in LU decomposition with partial pivoting. *SIAM J. Matrix Anal. Appl.*, 18(4):1065–1081, 1997.

[vdG] R. van de Geijn. PLAPACK: Parallel Linear Algebra Package. www.cs.utexas.edu/users/plapack.

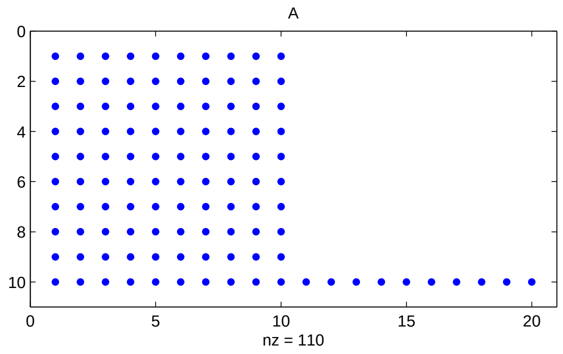
[VDY05] R. Vuduc, J. Demmel, and K. Yelick. OSKI: A library of automatically tuned sparse matrix kernels. In *Proc. of SciDAC 2005, J. of Physics: Conference Series.* Institute of Physics Publishing, June 2005.

[YZ05] R. Yuster and U. Zwick. Fast sparse matrix multiplication. *ACM Trans. Algorithms*, 1(1):2–13, 2005.

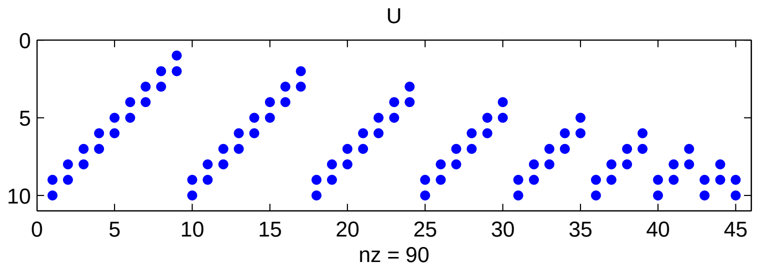
Appendix A. Motivation for 1×1 T matrices.

In the proof of Theorem 4.10, we make an assumption that all T matrices involved in blocked Householder updates of the form $A = (I - UTU^T)A$ are 1×1. This assumption seems like a significant restriction, since blocked Householder transformations are widely used in practice. We do not believe this assumption is necessary for the communication lower bound to be valid, but the reason for the assumption is that we discovered an artificial example, where by using an $O(n^4)$ algorithm with $O(n^4)$ additional storage (to form and use a T matrix of dimension $O(n^2)$) on a certain matrix, we could arrange to have one segment in which $O(M^2)$ multiplications were performed, thereby creating an obstacle to our proof technique, which depends on bounding the number of multiplications per segment by $O(M^{3/2})$. This (impractical!) variant of QR is not a counterexample to our theorem overall, just our proof technique. We present the example here.

Consider an $n \times 2n$ matrix $A = [A_1, A_2]$ where A_1 is square and dense and A_2 is square and nonzero only in its last row. Then A has the following sparsity structure:



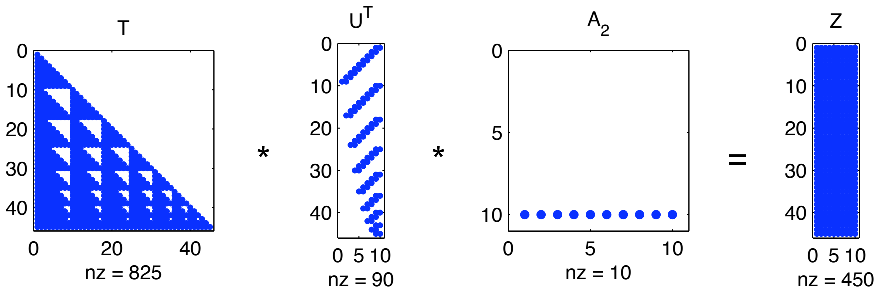
Suppose we perform QR on A using Givens rotations, where we work column by column, rotating the bottom row into the one above it, that row into the one above it, and so on to the diagonal entry. There will be $\frac{n(n-1)}{2}$ Givens rotations, and if we index them as described in Section 4, then the U matrix will have $\frac{n(n-1)}{2}$ columns with the following sparsity structure:



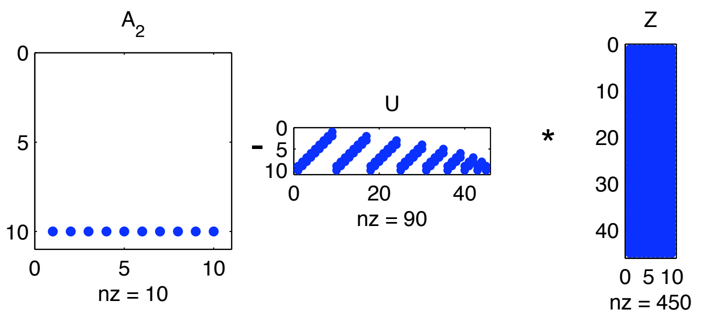
Suppose the computation is organized so that the triangularization of A_1 is completed before any updates are done to A_2. By letting T be as large as possible and following equation (4.2) the update of A_2 can be written as

$$A_2 = A_2 - \sum_k U(:,k) \cdot Z(k,:)$$

where $Z = TU^TA_2$. Here, T is $\frac{n(n-1)}{2} \times \frac{n(n-1)}{2}$ and at least 25% nonzero, and the sparsity structure of Z and its factors are given below.



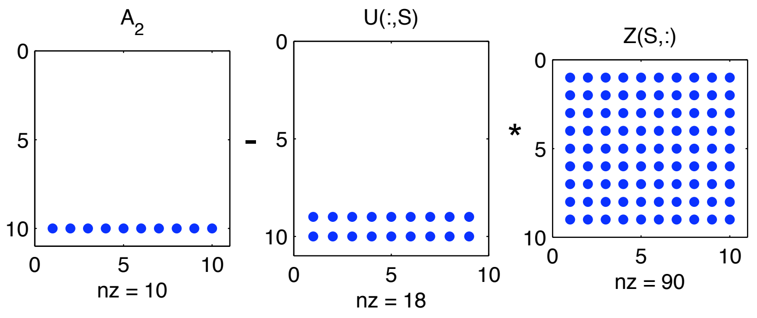
Thus, the computation $A_2 = A_2 - UZ$ has the following structure:



The full computation will completely fill in A_2, but suppose in one segment we only update the last two rows of A_2 by choosing the subset S of the columns of U and corresponding rows of Z which effect only the last two rows. Note that $|S| = n - 1$. In this case, the subcomputation is of the form

$$A_2(n-1:n, j) = A_2(n-1:n, j) - \sum_{k \in S} U(n-1:n, k) \cdot Z(k, j)$$

which has the following sparsity structure:



Further, all the entries of Z can be R2/D2. Each $Z(k, j)$ is the dot product of a row of (TU^T) with a column of A_2. Due to the sparsity structure of A_2, this is just one scalar multiplication. If we pre-compute (TU^T) and store it in slow memory, then in one segment we could do the following:

```
1: read $A_2(n,:)$, $(TU^T)(S,n)$, and $U(n-1:n,S)$ from slow memory
2: **for** $j = 1$ to $n$ **do**
3:   **for** $k \in S$ **do**
4:     $Z(k,j) = (TU^T)(k,n) \cdot A_2(n,j)$
5:     $A_2(n-1:n,j) = A_2(n-1:n,j) - U(n-1:n,k) \cdot Z(k,j)$
6:     discard $Z(k,j)$
7:   **end for**
8: **end for**
```

In this way, if $n = M/5$, then the last two rows of A_2, the entries of the last column of (TU^T) in rows $k \in S$, and the last two rows of U in columns $k \in S$ can all fit into fast memory simultaneously, and the $O(M^2)$ corresponding Z entries can be computed, used to update A_2, and discarded as shown above.

Thus, modeling the computation with the form $A = A - U \cdot Z$ and allowing unbounded T matrices, this example shows that it is possible to execute $O(M^2)$ flops within a single segment (allowing only $O(M)$ memory operations). This motivates our restriction that the T matrices are all 1×1. However, we do not believe this is a counterexample to the lower bound result because it involves the computation and storage of $O(n^4)$ entries of T.

Appendix B. Motivating Forward Progress.

We argue in this appendix that the second condition of *FP* (Definition 4.3) is a reasonable assumption. Under some mild assumptions, if an algorithm violates this condition, it will "get stuck" (i.e., be unable to complete the QR algorithm without violating the first condition of *FP*).

Let

$$X = \bigcup_{i=1}^{b} \text{zero\_rows\_}U(k_i) \cup \{\text{rows of column } c \text{ that are TAZ}\}$$

as in (4.3), and define

$$Y = \bigcup_{i=1}^{b} \text{rows\_}U(k_i) \cup \{ \text{ rows of column c that are TAZ}\}.$$

Note that X is a strict subset of Y, and that Y must include a row with a nonzero entry of column c of A, by the second condition of *FP*, after applying Householder transformations $k_1, ..., k_b$.

Suppose that the second condition is violated. There are two ways this could happen:

1. rows_$U(\widehat{k})$ are disjoint from Y.
2. rows_$U(\widehat{k})$ and Y intersect.

First suppose that (1) holds. Then since $U(:, \widehat{k})$ effects a disjoint set of rows as $U(:, k_1), ..., U(:, k_b)$, the only reason $\widehat{k}$ could be later in the PO than $k_1, ..., k_b$ (as opposed to being unordered as with respect to these Householder transformations) is that $\widehat{k}$ depends on some other k' such that $k_1, ...k_b < k' < \widehat{k}$, in which case we replace $\hat{k}$ by k'. We may continue this process until we find a $\widehat{k}$ satisfying (2). So now assume $\widehat{k}$ satisfies (2). Since the second condition does not hold, $rows\_U(\widehat{k})$ intersects $Y - X$, i.e., both contain a row of column c of A that was not zeroed out by Householder transformations $k_1, ..., k_b$ (or TAZ).

So rows_$U(\widehat{k})$ must lie in rows in column c not zeroed out by $k_1, .., k_b$, including at least one that "accumulated" nonzeros from other entries in column c. This implies that three rows of columns c and $\widehat{c}$ must look as follows after transformation $\widehat{k}$: $A_{tmp} = \begin{bmatrix} x & x \\ x & 0 \\ 0 & x \end{bmatrix}$ where x denotes a TAN entry and either

1. The first row is in $Y - X$,
   $A_{tmp}(1, 1)$ is the entry of column c into which nonzeros were accumulated,
   $A_{tmp}(2, 1)$ is not touched by Householder transformations $k_1, ..., k_b$,
   $A_{tmp}(3, 1)$ is zeroed out by Householder transformations $k_1, ..., k_b$,
   $A_{tmp}(1, 2)$ is the entry of column $\widehat{c}$ into which $A_{tmp}(2, 2)$ was accumulated

2. The second row is in $Y - X$,
   $A_{tmp}(2, 1)$ is the entry of column c into which nonzeros were accumulated,
   $A_{tmp}(1, 1)$ is not touched by Householder transformations $k_1, ..., k_b$,
   $A_{tmp}(3, 1)$ is zeroed out by Householder transformations $k_1, ..., k_b$,
   $A_{tmp}(1, 2)$ is the entry of column $\widehat{c}$ into which $A_{tmp}(2, 2)$ was accumulated,
   In both cases entries $A_{tmp}(3, 1)$ and $A_{tmp}(2, 2)$ have been deliberately zeroed out.

Note that in order to conclude that each x is nonzero, we are assuming that having a row_dest_$U(k)$ be TAZ is forbidden. An algorithm may permute rows for free (i.e., we do not enumerate those in the set of Householder vectors). Instead of accumulating into a TAZ entry, the algorithm must accumulate into a TAN entry and then permute (this does less arithmetic and causes less fill, so is preferable anyway).

Now we are "stuck", i.e., it is impossible to reduce this submatrix to upper triangular form without violating the first two conditions of *FP*. To see why, consider the entire rows corresponding to the pattern given by A_{tmp}. Note that given any subset of rows of an upper triangular nonsingular matrix, the leftmost nonzero must be the only nonzero in that column. Find the leftmost nonzero within the rows associated with A_{tmp}. If the leftmost nonzero is in column c, then it is impossible to zero out either $A_{tmp}(1,1)$ or $A_{tmp}(1,2)$ without filling in $A_{tmp}(3,1)$ or $A_{tmp}(2,2)$. If the leftmost nonzero is not in column c, then no matter in which row it lies, the Householder transformations $k_1,\dots,k_b$ and $\hat{k}$ will have filled in at least one other nonzero in the column. Again, any further transformation to reduce that column to one nonzero will involve either row 2 or row 3 and cause fill-in on a deliberately-zeroed-out entry, violating the first condition of *FP*.

For this to hold, we assume either that later Householder transformations are either restricted to this subset of rows, or that other rows used are nonzero in both columns c and $\hat{c}$. It is possible, using another row with special sparsity, to create the pattern above and then obtain triangular form without filling in any zeroes which were previously deliberately created. Consider the following example, where we have a 4×2 matrix with the following sparsity structure:

$$\begin{bmatrix} x & x \\ x & x \\ x & x \\ x & z \end{bmatrix}$$

where x means nonzero (TAN) and z means zero (TAZ). Suppose we create the pattern of A_{tmp} above with two Givens rotations to obtain

$$\begin{bmatrix} x & x \\ 0 & x \\ x & 0 \\ x & z \end{bmatrix}$$

where 0 denotes a TAZ entry that was deliberately created. Then we can rotate (3,1) into (4,1), which causes no fill since (3,2) and (4,2) are both TAZ:

$$\begin{bmatrix} x & x \\ 0 & x \\ 0 & 0 \\ x & z \end{bmatrix}$$

Then we rotate (4,1) into (1,1). This fills in (4,2), but since it was originally zero, we haven't violated forward progress. Then we remove the fill by rotating (4,2) into (2,2) and we obtain triangular form.

This completes the motivation for insisting on the second condition of *FP*.