DIMACS Series in Discrete Mathematics
and Theoretical Computer Science

A Survey of Out-of-Core Algorithms in Numerical Linear Algebra

Sivan Toledo

Abstract. This paper surveys algorithms that efficiently solve linear equations or compute eigenvalues even when the matrices involved are too large to fit in the main memory of the computer and must be stored on disks. The paper focuses on scheduling techniques that result in mostly sequential data accesses and in data reuse, and on techniques for transforming algorithms that cannot be effectively scheduled. The survey covers out-of-core algorithms for solving dense systems of linear equations, for the direct and iterative solution of sparse systems, for computing eigenvalues, for fast Fourier transforms, and for N-body computations. The paper also discusses reasonable assumptions on memory size, approaches for the analysis of out-of-core algorithms, and relationships between out-of-core, cache-aware, and parallel algorithms.

1. Introduction

Algorithms in numerical linear algebra solve systems of linear equations and eigenvalue problems. When the data structures that these algorithms use are too large to fit in the main memory of a computer, the data structures must be stored on disks. Accessing data that is stored on disks is slow. To achieve acceptable performance, an algorithm must access data stored on disks in large contiguous blocks and reuse data that is stored in main memory many times. Algorithms that are designed to achieve high performance when their data structures are stored on disks are called *out-of-core* algorithms.

This paper surveys out-of-core algorithms in numerical linear algebra. The first part of the paper, Section 2, describes algorithms that are essentially clever schedules of conventional algorithms. The second part of the paper, Section 3, describes algorithms whose dependences must be changed before they can can be scheduled for data reuse. The final part of the paper, Section 4, explains several issues that pertain to all the algorithms that are described. In particular, we

1991 *Mathematics Subject Classification.* Primary 65F05, 65F10, 65F15, 65F25, 65F50, 65-02, 68-02.

Key words and phrases. out-of-core algorithms, external-memory algorithms, numerical linear algebra.

This research was performed at the Xerox Palo Alto Research Center and supported in part by DARPA contract number DABT63-95-C-0087 and by NSF contract number ASC-96-26298. The author is on leave from Tel Aviv University.

discuss typical memory sizes, approaches to the analysis of out-of-core algorithms, differences between out-of-core algorithms and cache-aware and parallel algorithms, and abstractions that are useful for designing schedules. The rest of the introduction explains why out-of-core algorithms are useful and why they are not as common as one might expect.

Out-of-core algorithms are useful primarily because they enable users to efficiently solve large problems on relatively cheap computers. Disk storage is significantly cheaper than main-memory storage (DRAM).[1] Since many out-of-core algorithms deliver performance similar to that of in-core algorithms, an out-of-core algorithm on a machine with a small main memory delivers a better price/performance ratio than an in-core algorithm on a machine with enough main memory to solve large problems in core. Furthermore, ubiquitous parallel computing in the form of scalable parallel computers, symmetric multiprocessors, and parallel I/O means that buying enough memory to solve a problem in-core is rarely the best way to speed up an out-of-core computation. If the computation spends most of its time in in-core computations, the best way to speed it up is to add more processors to the computer. If the computation spends most of its time waiting for I/O, the best way to speed it up is often to attach more disks to the computer to increase I/O bandwidth.

Still, one must acknowledge two trends that cause out-of-core algorithms to become less attractive to users. First, the size of computer memories, both DRAM and disks, grows rapidly. Hence, problems that could not be solved in-core in a cost-effective manner a few years ago can be solved in-core today on machines with standard memory sizes. Problems that could not be solved in core on the world's largest supercomputer 20 years ago, the Cray-1, can now be solved in core on a laptop computer. (And not because the Cray-1 had a small memory for its period; it was designed with a memory large enough to make virtual memory unnecessary.) While users' desire to solve ever-larger problems does not seem to wane, including problems too large to fit in memory, growing main memory sizes imply that out-of-core algorithms have become a specialty.

Second, improvements in numerical methods sometimes render out-of-core algorithms obsolete. Some methods are inherently more difficult to schedule out-of-core than others. For example, the conjugate gradient method for iteratively solving sparse linear systems is inherently more difficult to schedule out of core than the iterative methods that were in use prior to the its discovery. When such a method is invented, it renders out-of-core algorithms obsolete. The algorithms described in Sections 2.4 and 3.2 exemplify this phenomenon.

2. Scheduling and Data Layout for Out-of-Core Algorithms

When an algorithm is to to be executed out of core, the ordering of independent operations must be chosen so as to minimize I/O. The schedule must also satisfy the data-flow and control-flow constraints of the algorithm. In addition, the layout of data structures on disks must be chosen so that I/O is performed in large blocks, and so that all or most of the data that is read in one block-I/O operation is used before it is evicted from main memory. When there are multiple processors, the schedule must also expose enough parallelism, and when the data is laid out on

[1]Today (early 1998), DRAM costs about \$5000 per GByte, whereas disks cost about \$100 per Gbyte.

multiple disks, a good schedule reads and writes from multiple disks simultaneously. This section describes techniques for scheduling and data layout that are used in out-of-core numerical linear algebra and the algorithms that use these techniques.

Some algorithms cannot be scheduled for efficient out-of-core execution. These algorithms must be modified, in the sense that their data- or control-flow constraints must be altered so as to admit an efficient schedule. Techniques for modifying algorithms so that their data-flow graphs admit efficient schedules are presented in Section 3.

2.1. An Example: Dense Matrix Multiplication. Several important computations on dense matrices can be scheduled out of core using a simple technique that was discovered very early. We illustrate this technique in some detail using matrix multiplication as an example. Specifically, we show that a naive schedule is inefficient, and that there is a simple schedule that is significantly more efficient than the naive schedule and asymptotically optimal.

The product C of two n-by-n matrices A and B is defined as $c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}$. It is easy to see that each term $a_{ik} b_{kj}$ appears in just one summation, namely, the computation of c_{ij}. In other words, the output values share no common subexpressions, so the algorithm must compute n^3 products. The challenge is to compute them using a schedule that minimizes I/O.

A naive way to implement this algorithm uses a set of three nested loops:

```
naive-matrix-multiply(n,C,A,B)
for i = 1 to n
  for j = 1 to n
    C[i,j] = 0
    for k = 1 to n
      C[i,j] = C[i,j] + A[i,k] * B[k,j]
    end for
  end for
end for
```

Let us analyze the amount of I/O that this schedule performs. We denote the size of main memory by M, and we assume that main memory cannot store more than one half of one matrix. That is, we assume that $M < n^2/2$. (We use this assumption throughout the paper.) All the elements of B are accessed in every iteration of the outer loop. Since at most half of them can remain in memory from one iteration to the next, at least half of them must be read from disk in each iteration. Hence, the total number of words read from disk is at least $n \cdot n^2/2 = n^3/2$. In other words, the number of words transferred[2] is proportional to the work in the algorithm. It is clear that no schedule needs to transfer more than $4n^3$ words, or three reads and one write in every inner iteration. Therefore, our analysis is asymptotically tight: this schedule transfers $\Theta(n^3)$ words.

Can we do better? We can. There are schedules that transfer only $\Theta(n^3/\sqrt{M})$ words. The key to these schedules is considering the three matrices as block matrices with block size $b = \sqrt{M/3}$. We denote the b-by-b submatrix of A whose upper left corner is A_{ij} by `A[i:i+b-1,j:j+b-1]`. The following pseudo code implements this

[2]When we refer to "transfers" from here on we mean data transfers between main memory and disk.

improved schedule; it is not hard to prove that the code implements the same data flow graph as `naive-matrix-multiply`, if n is a multiple of b.

```
blocked-matrix-multiply(n,C,A,B)
b = square root of (memory size/3)
for i = 1 to n step b
  for j = 1 to n step b
    fill C[i:i+b-1,j:j+b-1] with zeros
    for k = 1 to n step b
      naive-matrix-multiply(b,
                            C[i:i+b-1,j:j+b-1],
                            A[i:i+b-1,k:k+b-1],
                            B[k:k+b-1,j:j+b-1])
    end for
  end for
end for
```

This schedule is more efficient because each iteration of the inner loop of `blocked-matrix-multiply` accesses only $3b^2 = M$ words but performs b^3 multiplications. The words that are accessed in the inner iteration need to be read from disk only once, because all of them fit within main memory, and written back once.(Actually, a block of C needs to be read and written only every n/b iterations.) The total number of words transferred is bounded by the number of inner iterations times the number of transfers per iteration, or

$$(n/b)^3 \times 2M = (n/\sqrt{M/3})^3 \times 2M = O(n^3/\sqrt{M}) \ .$$

This schedule performs a factor of $\Theta(\sqrt{M})$ less I/O than the naive schedule.

Can we do better yet? We cannot. The following theorem states that when main memory cannot store more than about one sixth of one input matrix, the blocked schedule described above is asymptotically optimal. This result was originally proved by Hong and Kung [**33**]; we give here a novel proof which is both simple and intuitive. Note that the theorem does not bound the number of transfers required to multiply two matrices, only the number of transfers that is required if we multiply the matrices using the conventional algorithm. Other matrix multiplication algorithms, such as Strassen's algorithm, may require fewer transfers.

THEOREM 1. *Any schedule of the conventional matrix multiplication algorithm must transfer $\Omega(n^3/\sqrt{M})$ words between main memory and disk, where the input matrices are n-by-n, M is the size of main memory, and $M < n^2/5.2415$.*

Proof: We decompose a given schedule into phases that transfer exactly M words each, except perhaps the last phase.

We say that c_{ij} is *alive* during phase p if during phase p the schedule computes products $a_{ik}b_{kj}$ for some k, and if some partial sum of these products is either in main memory at the end of the phase or is written to disk during the phase. By the definition of a live c_{ij}, there can be at most $2M$ live c_{ij}'s during a phase. Note that for each triplet (i, j, k), the multiplication $a_{ik}b_{kj}$ must be performed in some phase in which c_{ij} is alive. A multiplication $a_{ik}b_{kj}$ that is performed in a phase in which c_{ij} is not alive does not represent progress toward the scheduling of the data-flow graph, so without loss of generality we assume that there are no such multiplications in the schedule.

There can also be at most $2M$ distinct elements of A and at most $2M$ distinct elements of B in main memory during phase p, because every such element must be either in memory at the beginning of the phase or it must be read from disk during the phase. We denote the set of elements of A that are in main memory during phase p by A_p.

How many multiplications can the schedule perform in phase p?

We denote by S_p^1 the set of rows of A with $\sqrt{M}$ or more elements in A_p, and by S_p^2 the set of rows with fewer elements in A_p. Clearly, the size of S_p^1 is at most $2\sqrt{M}$.

We begin by bounding the number of scalar multiplications during phase p that involve elements of A from rows in S_p^1. Each element of B is used exactly once when a row of A is multiplied by B. Therefore, the number of scalar multiplications in phase p that involve elements of A from rows in S_p^1 is bounded by the size of S_p^1 times the number of elements of B in main memory during the phase, or $2\sqrt{M} \times 2M = 4M^{3/2}$.

We now bound the number of scalar multiplications during phase p that involve elements of A from rows in S_p^2. Each c_{ij} is a product of a row of A and a column of B. The number of multiplications that involve elements of A from rows in S_p^2 is bounded by the number of live c_{ij}'s times the maximal number of elements in a row of S_p^2 that are in A_p, or $2M \times \sqrt{M} = 2M^{3/2}$.

We conclude that the number of multiplications per phase is at most $6M^{3/2}$. Since the total number of multiplications is n^3, the number of phases is at least $\lceil n^3/6M^{3/2} \rceil$, so the total number of words transferred is at least $M(n^3/6M^{3/2} - 1)$ (since the last phase may transfer fewer than M words). For $M < n^2/5.2415$, the number of transfers is $\Omega(n^3/\sqrt{M})$. $\square$

2.2. Dense Matrix Computations. Considering matrices to be block matrices leads to high levels of data reuse in other dense matrix algorithms. In out-of-core implementations the matrices are stored on disks such that b-by-b blocks are contiguous, so I/O operations are done in large blocks.

A Cholesky factorization algorithm, for example, factors a square symmetric positive definite matrix A into a product of triangular factors LL^T. Here the basic operations in the conventional algorithm are multiplication, square roots, and division by square roots. If we interpret the square root operation on a block as computing $A_{II} = L_{II}L_{II}^T$ (the operation is only applied to diagonal elements), and division by a square root as a triangular solve, we again obtain an efficient out-of-core schedule for the conventional algorithm. Interpreting these basic operations in other ways, however, can change the data flow in the algorithm and hence the output that the algorithm computes. For example, if we implement the square root operation by computing a block B_{II} such that $A_{II} = B_{II}^2$, the new algorithm will follow a different data flow path and will compute a different result. In fact, the factor that the new algorithm computes will be block triangular and not triangular.

Algorithms that merely schedule the data flow in the original algorithm are called *partitioned*, and those that follow a different data flow graph and potentially produce different results are referred to as *blocked*. When a partitioned algorithm exists it is generally preferable to a blocked one, because blocked algorithms can be numerically less stable and because they sometimes do more work than the original algorithms.

Both partitioned and blocked algorithms were discovered very early. Rutledge and Rubinstein [**45, 46**] describe the linear algebra library of the Univac. The library was operational since 1952, and Rutledge and Rubinstein attribute its basic design to Herbert F. Mitchell. The library used a partitioned algorithm for matrix multiplication. This algorithm is still widely used today. The library used a blocked algorithm for matrix inversion which is now obsolete. The Univac had a 1,000-word main memory and magnetic tapes served as secondary memory. The algorithms used 10-by-10 blocks, and the data layout and schedule were organized so that both in-core and out-of-core accesses are sequential, since both levels of memory were sequential (the Univac used acoustic delay lines for its main memory).

Other important dense algorithms that can be scheduled for out-of-core execution include the LU factorization with partial pivoting and the QR factorization. In these algorithms, the schedule cannot be generated using a partitioning of the matrix into square blocks, because these algorithms have steps that must access full columns (or full rows). The partitioning scheme that was developed for these algorithms is based on a partitioning of the matrix into vertical slabs of groups of columns.

Barron and Swinnerton-Dyerm [**4**] described in 1960 a partitioned algorithm for LU factorization with partial pivoting. They implemented a factorization algorithm for the EDSAC 2 using magnetic tapes to store the matrix. Their algorithm, which is now called the block-column right-looking algorithm, is still used today to improve data reuse in caches. The most widely used out-of-core LU factorization algorithm today, the block-column left-looking algorithm, was proposed in 1981 by Du Cruz, Nugent, Reid and Taylor [**17**]. Their algorithm was developed for the NAG library as a factorization algorithm for virtual-memory machines. The I/O behavior of the right- and left-looking algorithms is similar in terms of reads but the right-looking algorithm performs more writes. A somewhat different algorithm was proposed by the author recently [**53**]. This algorithm partitions the matrix recursively, rather than into fixed block columns, and performs asymptotically fewer reads and writes than either the right- or left-looking algorithms. It is always faster than the right-looking algorithm and it is faster than the left-looking algorithm when factoring large matrices on machines with a small main memory.

A steady stream of recent implementations of dense out-of-core factorization codes [**7, 16, 22, 26, 30, 34, 49, 50, 52, 54, 61**] serves as a testimony to the need for out-of-core dense solvers; most of these papers describe parallel out-of-core algorithms and implementations.

The first systematic comparison of conventional and partitioned schedules was published by McKellar and Coffman in 1969 [**39**], in the context of virtual memory. They assumed that the matrices are stored by blocks for the partitioned algorithms and showed that the partitioned algorithms generate asymptotically fewer page faults (I/O transfers) than conventional algorithms. Fischer and Probert [**25**] performed a similar analysis for Strassen's matrix multiplication algorithm.

Hong and Kung proved in 1981 [**33**] that the partitioned schedule for matrix multiplication is asymptotically optimal in a model of I/O that allows redundant computation and does not assume that I/O is performed in blocks. They also proved lower bounds for other problems that are discussed in this survey, such as FFTs. Their bounds were later extended to more complex models that assume that I/O is performed in blocks on multiple independent disks by Aggarwal and Vitter [**1**] and by Vitter and Shriver [**58**].

The misperception that partitioned schedules require a data layout by blocks caused partitioned schedules to go out of fashion in the 1970s. Virtual memory was becoming a standard way to solve problems larger than memory. The convenience of laying out matrices in two-dimensional arrays that are laid out by a compiler in column- or row-major order favors storing matrices by column or row, not by block. The trend to abandon partitioned schedules was apparently also fueled by the fact that on machines of that period, simply using all the words that are fetched by a page fault once or twice was enough to achieve high processor utilization. The high level of data reuse offered by partitioned algorithms was not deemed necessary. An influential 1972 paper by Moler [**40**] is typical of that period. In it, he describes a schedule for LU factorization that access entire columns of the matrix, so that all the words fetched in a page fault are used by the algorithm. The schedule is not efficient in the sense that when a typical page is fetched, its content is used once, and the page is evicted. Moler dismisses the analysis of McKellar and Coffman on the basis that it requires layout by blocks, which is inconvenient in Fortran. He ignores the 1960 paper by Barron and Swinnerton-Dyerm [**4**] that showed a partitioned schedule that accesses entire columns and does not require a block layout. Papers describing similar schedules for other problems were published in the following years [**18, 19, 59**].

Schedules that guarantee sequential access or access to large contiguous blocks were always important. In the early days, out-of-core storage used tapes, a sequential medium, so sequential access was of paramount importance. When disks became the main medium for out-of-core storage, completely sequential access lost its importance, but access to large contiguous blocks remains important. But sequential or mostly sequential access can be achieved even with partitioned algorithms, and even when the matrix is laid out in columns.

2.3. Sparse Matrix Factorizations. Sparse factorization algorithms can be classified into categories based on two criteria: what assumptions they make on sparsity patterns and whether they are row/column methods or frontal methods. Some algorithms assume that the nonzeros are clustered in relatively large dense blocks (block methods), some assume that the nonzeros are clustered around the main diagonal (band and envelope methods), while others do not assume any particular sparsity pattern (general sparse methods). We explain the difference between row/column and frontal methods later.

Band solvers were popular until well into the 1970s, and were later refined into envelope solvers. These methods have the property that at any given point in the factorization, the rows and columns of the matrix can be decomposed into three classes: done, active, and inactive. The rows/columns that are done have been factored and they need not be read again. The active part consists of rows/columns that either have been factored but are needed for updates to later rows/columns (in a left-looking algorithm), or of rows/columns that have been updated but not factored (in a right-looking algorithm). The inactive part of the matrix has not been yet read at all. Some band methods assume that the active part of the matrix can be stored in core. These methods require $\Theta(m^2)$ words of in core storage, where m is the bandwidth of the matrix, and compute the factors in one pass over the matrix. Riesel [**43**] describes a 1955 implementation of a sparse elimination algorithm on the BESK computer at the SAAB Aircraft Company in Sweden. It seems that he used such a banded algorithm, although he does not describe the algorithm in detail. His solver was used to solve linear systems with 214 variables, which did not fit within the BESK's 8192-word magnetic drum. Cantin [**8**] describes another such algorithm from the mid 1960s, which he attributed to E. Wilson.

When the bandwidth of the matrix is large, the active part of the matrix cannot be stored in core. The matrix is then partitioned into dense blocks and a partitioned algorithm is used. These algorithms are essentially identical to dense partitioned algorithms, except that they do not operate on zero blocks. Cantin [**8**], Mondkar and Powell [**41**] and Wilson, Bathe and Doherty [**60**] all describe such algorithms in the early 1970s. The same idea can be used when the nonzeros are clustered in large dense blocks even when the blocks themselves are not clustered near the main diagonal. Crotty [**14**] described such a solver in 1982 for problems arising from the boundary-elements method in linear elasticity.

General sparse algorithms that are row/column based are usually left-looking. These algorithms can, at least in principle, be implemented out of core using a block-column partitioned approach, similar to that used for dense matrices. The performance of such a method can be significantly worse than the performance of similar methods for dense and band matrices, however. In dense and band matrices, the columns that are needed to update a given column or block of columns are contiguous. In a general sparse algorithm, they are not. Therefore, a simple out-of-core left-looking algorithm may need to read many small noncontiguous blocks of data. To the best of my knowledge, implementations of such algorithms have not been described in the literature, perhaps for this reason.

The general sparse algorithms that have been described are frontal or multifrontal, rather than row/column based algorithms. Unlike row/column algorithms that only store original rows and columns of A, partially updated rows/columns of A, and rows/columns of the factors, multifrontal algorithms (including simple frontal ones) store original rows/columns, rows/columns of the factors, and a stack of sparse matrices called frontal matrices. The frontal matrices are sparse, but all their rows and columns have the same structure, so they can be stored and manipulated as dense matrices. If the stack of frontal matrices fits in core, the algorithm can be implemented out-of-core as a single-pass algorithm. Interestingly, the maximum size of the stack depends on the ordering of independent operations in the algorithm. Liu [**36**] describes techniques to schedule the algorithm so as to minimize the maximum size of the stack so that the need for an out-of-core algorithm is eliminated in many cases. When the stack is stored out of core, the same algorithm can reduce the amount of I/O required for out-of-core stack operations. In another paper [**37**], Liu describes another technique that can reduce the amount of memory required for in-core factorization. This technique switches between column Cholesky and multifrontal Cholesky to reduce running time and/or memory requirements.

Another technique, similar in motivation to Liu's techniques, is described by Eisenstat, Schultz and Sherman [**20**]. They propose a technique that discards elements of the factor and recomputes them as necessary, instead of writing them to disk and reading them later. The method performs more work than the conventional factorizations, it solve a linear system but does not produce the factors, and it may fail for very large matrices. But like Liu's techniques, it does allow one to solve in-core linear systems that would otherwise have to be solved out of core.

Out-of-core implementations of multifrontal and related factorization methods are described by George and Rashwan [**28**], by George, Heath, and Plemmons [**29**], by Bjørstad [**5**], and by Rothberg and Schreiber [**44**]. Some of these implementations can handle frontal matrices larger than main memory by partitioning them, and recent ones (e.g. [**44**]) use Liu's technique for reducing the size of the stack and hence the amount of I/O performed. Rothberg and Schreiber use another technique to minimize I/O. Their algorithm can switch from a multifrontal method to a left-looking method based on a criterion for estimating I/O costs. The general idea is similar to that of Liu [**37**], except that Liu switches from a column factorization to a multifrontal one to save space and Rothberg and Schreiber switch in the other direction to minimize I/O.

2.4. Sparse Iterative Methods. Iterative algorithms maintain a state vector x and repeatedly update it, $x^{(t)} = Fx^{(t-1)}$, where F represents the update rule. When the update in each iteration is a multiplication by a sparse matrix A, such as in Jacobi relaxation, a simple technique called *covering* can be used to schedule the computations out-of-core. The main idea is to load into main memory a subset x_I of the state vector x, perform more than one iteration on at least some elements of x_I, then to write back the updated elements of x to disk. For example, when the underlying graph of A is a $\sqrt{n}$-by-$\sqrt{n}$ mesh, this method can perform $\Theta(\sqrt{M})$ iterations while transferring only $\Theta(n)$ words. In contrast, the naive schedule that applies each iteration to the entire state vector before starting the next iteration transfers $\Theta(n)$ words per iteration. The same technique can be used when the state-vector update represents a sparse triangular or block triangular linear system.

A program called PDQ-5 used this technique to implement an out-of-core solver for two-dimensional problems arising in nuclear reactor design [**42**]. The program was written in the early 1960's by W. R. Cadwell, L. A. Hageman, and C. J. Pfeifer. It ran on a Philco-2000 using magnetic tapes to store the state vector. The program used a line successive overrelaxation (line-SOR) method, in which the update rule represents the solution of a block triangular linear system. Once the first two lines in the domain (first two blocks) are relaxed, the first line is relaxed again.

Hong and Kung [**33**] showed upper and lower bounds on the use of this technique on regular meshes.

Unfortunately, the majority of modern iterative methods cannot be scheduled for data reuse. Methods that cannot be scheduled for significant in-core data reuse include implicit time-stepping schemes, symmetric successive overrelaxation, conjugate gradient, alternating directions, and multigrid algorithms (see [**55**] for lower bounds for several of these). Indeed, in a 1964 survey paper on computers in nuclear reactor design [**15**], Cuthill remarked that alternating directions iterative methods converge faster than line-SOR methods, but are more difficult to implement efficiently out-of-core.

Consequently, since the mid 1960s most iterative solvers have been implemented in core. Schedules that access data mostly sequentially have been developed for virtual memory systems, however. Wang, for example, shows such a schedule for alternating directions [**59**]. Another idea that has emerged recently is to modify the data flow of iterative algorithms that cannot be scheduled for data reuse so that they can be scheduled. This idea is explored in Section 3.2.

2.5. Fast Fourier Transforms (FFTs). The FFT can be scheduled for out-of-core execution fairly efficiently. Theoretically, both lower and upper bounds for the number of words transferred is $\Theta(n \log n / \log M)$ [**33, 1**]. In practice, the

$\log n / \log M$ factor is a small constant, rarely larger than 2, so asymptotic bounds are not particularly useful.

Soon after the introduction of the FFT algorithm by Cooley and Tukey, Gentleman and Sande [**27**] presented an FFT algorithm which is suitable for out-of-core applications. The algorithm works by arranging the input n vector in a two-dimensional array, performing an FFT on every row, scaling the array, transposing the array and performing an FFT on every row again (every column of the original array). Assuming that primary memory can hold at least one row and one column of the array, the algorithm requires only two passes over the data and an out-of-core matrix transposition. The algorithm did not gain much acceptance, but it was rediscovered and implemented on several machines. See [**3**] for a survey and a description of an implementation on the Cray X-MP.

Although the total amount of I/O that this algorithm performs is not large relative to the size of the data set, it often suffers from low processor utilization. The total amount of work per datum in the FFT, $\Theta(\log n)$, does not allow for extensive data reuse. With only a moderate level of data reuse, I/O often dominates the running time.

Other, less efficient, out-of-core FFT algorithms were proposed by Singleton [**51**] and Brenner [**6**]. Eklundh [**21**] noticed that an efficient out-of-core matrix transposition algorithm is a key to out-of-core two-dimensional FFT algorithms, and suggested one. Others have also proposed algorithms for matrix transposition, for example [**2, 48, 56**].

More recently, a series of papers by Cormen, Wegmann, and Nicols describe parallel out-of-core FFT algorithms that use multiple disks [**11, 12, 13**].

The monograph by Van Loan [**57**] describes numerous FFT algorithms, including out-of-core algorithms, together with extensive references. Bailey's paper [**3**] also presents a good survey of out-of-core FFT algorithms.

2.6. N-Body Computations. N-body or fast multipole algorithms are tree based algorithms for evaluating integrals. In a recent paper, Salmon and Warren [**47**] describe a parallel out-of-core N-body code. The code uses several scheduling techniques to achieve high out-of-core performance. The most important technique is a special ordering of points in space, and hence data accesses to them, which enhances data reuse. The ordering is based on a self-similar space-filling curve. This ordering was developed to enhance locality in caches and in distributed memory parallel computers, but it proved suitable for an out-of-core implementation. The other important algorithmic technique that they use is essentially loop fusion. They fuse several passes over the data that were originally separate into as few traversals as possible, in order to minimize I/O. They comment that this scheduling technique enhances performance but reduces the modularity of the code, because tasks that are conceptually unrelated are now coded together. They report good performance using a cluster of Pentium Pro computers.

Most implementations of out-of-core algorithms in numerical linear algebra either perform I/O data transfers explicitly or rely on the virtual memory system to perform data transfers. In contrast, Salmon and Warren use a hybrid scheme that can best be described as a user-level paging mechanism.

3. Data-Flow Transformation Techniques

When the data flow of an algorithm does not admit an efficient out-of-core schedule, the designer must develop an algorithm with a more suitable data-flow graph. The new algorithm may be a "transformation" of the original algorithm, or it may be completely new. By transformation I mean that the two algorithms are equivalent in some sense, although there is no clear distinction between transformed algorithms and completely new algorithms.

Numerous parallel algorithms can be thought of as transformations of conventional sequential algorithms. Parallel summation is probably the simplest example. An algorithm that adds n numbers to a running sum has a deep data-flow graph that does not admit a good parallel schedule. An algorithm that sums two $n/2$ groups of numbers recursively and then adds up the two partial sums has a data flow graph with depth $\log n$ that admits an efficient parallel schedule. If addition is associative, the two algorithms are algebraically equivalent. Floating-point addition is not associative, but the two algorithms are essentially equivalent for most practical purposes. This is not always the case. Some sequential algorithms are significantly more stable than their (transformed) parallel versions. For example, the data-flow graph of Modified Gram-Schmidt orthogonalization contains less parallelism than the data-flow graph of Classical Gram-Schmidt, but it is significantly more accurate. Transformed algorithms sometimes perform significantly more work (arithmetic) than their sequential counterparts. For example, parallel algorithms for solving tridiagonal systems of linear equations usually perform at least twice the amount of arithmetic that sequential algorithms perform. Another example is Gauss-Seidel relaxation, which often converges more slowly when the unknowns are reordered to maximize parallelism (e.g., red-black ordering of a grid instead of a natural ordering).

Algorithms can sometimes be transformed so that their data-flow admits an efficient out-of-core schedule, but they are not as common as algorithms that are transformed to introduce parallelism. One obvious reason for this is that parallel algorithms have been investigated much more intensively than out-of-core algorithms. Another possible reason is that work inefficiency, and to some extent instability, are often tolerated in parallel algorithms. The common argument goes something like this: "this parallel algorithm performs 4 times more work than the sequential algorithm, but if you have 100 processors, it would still 25 be times faster than a sequential algorithm that uses only one processor." Although this argument fails if you have only 4 or 8 processors, it is valid if you have a 100, and some users do have applications that they must parallelize on large numbers of processors. The same reasoning cannot be applied to out-of-core algorithms, because the in-core/out-of-core performance ratio is fixed, not unbounded like the number of processors. For example, if running an algorithm which cannot reuse in-core data out of core is 10 times slower than running it in core, and if a "transformed" algorithm with improved data reuse performs 4 times more arithmetic, it will solve problems at most 2.5 times faster than the original algorithm. For many users the difference is not significant enough to switch to an out-of-core algorithm, especially if the out-of-core algorithm is numerically less stable.

Still, there are interesting examples of algorithms that have been transformed to improve data reuse, which this section describes.

3.1. Dense Eigensolvers. The most widely-used dense eigensolvers are based on the QR algorithm and its variants. In these algorithms, the matrix is first reduced to either a tridiagonal matrix (in the symmetric case) or to an upper Hessenberg matrix (in the nonsymmetric case) using symmetric orthogonal transformations. The algorithm then computes the eigenvalues of the reduced matrix iteratively.

It appears that there is no way to schedule the symmetric reduction with a significant level of data reuse, although I am not aware of any formal lower bound. But the algorithm can be replaced by a block algorithm that offers a high level of data reuse. The transformed algorithm performs more work than the original. It is not considered to be less stable, since both apply only orthogonal transformations to the matrix. In the symmetric case, the rest of the algorithm can be performed in-core, since it operates on a tridiagonal matrix (with $\Theta(n)$ nonzeros, as opposed to the original dense matrix with has $\Theta(n^2)$ nonzeros). In the nonsymmetric case, computing the eigenvalues of the Hessenberg matrix involves an iterative algorithm in which every iteration accesses $\Theta(n^2)$ words and performs only $\Theta(n^2)$ work. I am not aware of an efficient way to implement this algorithm out-of-core.

In general terms, The blocked algorithm for the reduction of a symmetric matrix to a tridiagonal one works as follows. It first reduces the matrix to a block tridiagonal matrix, rather than to a true tridiagonal matrix. The reduced matrix can then be reduced to a new block tridiagonal matrix with a smaller block size, until we get a true tridiagonal matrix. The blocked algorithm performs more work than the original scalar algorithm.

Grimes and Simon [**31**] describe such an algorithm for dense symmetric generalized eigenvalue problems, motivated by quantum mechanical bandstructure computations. Their algorithm works by reducing the matrix into a band matrix using block Householder transformation. They assume that the band matrix fits within primary memory and reduce it to a tridiagonal matrix using an in-core algorithm. This variant is more efficient than the general scheme we described above because their algorithm reduces the original matrix not to a general block tridiagonal matrix, but to a band matrix. They report on the performance of the algorithm on a Cray X-MP with a solid state storage device (SSD) as secondary storage.

Dubrulle [**18**] describes an implementation of the symmetric reduction to tridiagonal algorithm with a sequential access pattern, but without introducing any data reuse. This paper is similar in motivation to Moler's paper [**40**].

3.2. Iterative Linear Solvers. Early algorithms were all implemented out-of-core, but advances in iterative linear solvers made out-of-core implementations increasingly difficult. Early algorithms were relatively easy to schedule out of core. More advanced techniques that were introduced later, such as conjugate gradient and multigrid, however, are difficult to schedule out-of-core. The same techniques can sometimes be applied to conjugate-gradient-like algorithms that compute eigenvalues of sparse matrices.

Many modern iterative methods cannot be scheduled for significant data reuse because each element of their state vectors in iteration t depends on all the elements of the state vector in iteration $t-1$. In conjugate gradient and similar algorithms, these dependences occur through a global summation; in multigrid and other multilevel algorithms, the dependences occur through a traversal of a hierarchical data structure.

Three different techniques have been proposed to overcome these problems. The first technique retains the basic structure of the algorithm, but tries to reduce the number of iterations by making each iteration more expensive. When each iteration requires one pass over the out-of-core data structure, this approach reduces I/O. Mandel [**38**], for example, describe an out-of-core Krylov-subspace algorithm (i.e., conjugate-gradient-like) with a domain-decomposition preconditioner for solving finite-elements problems. The preconditioner accelerates convergence by making every iteration computationally more expensive.

The second approach is to algebraically transform algorithms so that they can be scheduled for data reuse. The effect of these transformations is to delay information propagation through the state vector. All of the algorithms in this category perform more work than the original algorithms, and they are generally less numerically stable. On the other hand, they are equivalent to the original algorithms in exact arithmetic (so convergence is not a concern, only accuracy) and transformed algorithms can usually be derived almost mechanically from existing ones.

Leiserson, Rao, and the author [**35, 55**] introduced the notion of blocking covers, which they used to design out-of-core multigrid algorithms. These algorithms have not been implemented. The author also developed transformation techniques for conjugate gradient and related algorithms [**55**]. These algorithms have been successfully implemented out-of-core. These conjugate-gradient algorithms are closely related to multistep conjugate-gradient algorithms [**9**] whose goal is to eliminate synchronization points (i.e., global summations) in parallel algorithms.

The third technique for removing dependencies from iterative algorithms relies on the structure of the conjugate gradient algorithm. In the conjugate gradient and related algorithms, global summations are used to incrementally construct a polynomial that minimizes some error norm, and hence, ensures optimal convergence. It is also possible to choose the polynomial without computing global sums. The polynomial that we choose can be either fixed for all linear systems (as in so-called stationary iterations and in certain polynomial preconditioners), or it can depend on certain properties of the coefficient matrix (as in Chebychev iteration, which depends on the extreme eigenvalues of the matrix).

Fischer and Freund proposed such out-of-core methods based on polynomial preconditioning [**24**] and on an inner-product free Krylov-subspace method [**23**]. Both methods perform a small number of conjugate-gradient iterations to approximate the spectrum of the matrix. This approximation is used to construct a family of polynomials that is used in a polynomial preconditioner in one method, and in an inner-product free Krylov-subspace algorithm in another method. Both methods compute far fewer inner products than popular Krylov-subspace algorithms and should therefore be easier to implement out-of-core. Generally speaking, these methods are not as numerically robust as the (hard to schedule) algorithms that they are supposed to replace.

4. Concluding Remarks

This section discusses a few general issues that pertain to all of the algorithms described in this paper.

4.1. How Much Main Memory Should Computers Have? The costs of main memory and disks dictate a certain reasonable range of ratios of disk to main memory sizes. Disk storage is about 50 times cheaper today than DRAM storage.[3] Therefore, computers are likely to have main memory sizes that are, say, 10 to 1000 times smaller than their disks. Computers with less memory than this are poorly designed, since their main memories, and hence their performance, can be significantly increased at a modest increase in their total cost.

In terms of out-of-core algorithms, and especially in terms of asymptotic data-reuse bounds, these disk/memory size rations imply that main memories are very large. For example, using the conservative assumptions that computers have main memories that are at least 1/1000 the size of their disks, and that the data set size n is at least $1,000,000$, we conclude that computers have enough main memory to hold $\Theta(\sqrt{n})$ data items in core. Smaller data sizes can usually be handled in core today, and smaller memories imply poor design, as explained above.

This analysis implies that in some cases we cannot evaluate algorithms by comparing asymptotic data-reuse bounds. The asymptotic I/O bound for the FFT, for example, $\Theta(n \log n / \log M)$, essentially reduces to $\Theta(n)$ since $\log n / \log M \leq 2$ on any reasonable computer. We cannot use the asymptotic bound to compare the algorithm to another algorithm that performs asymptotically only $\Theta(n)$ I/O's.

4.2. Analyzing I/O and Data Reuse. Four approaches for analyzing I/O counts and levels of data reuse have been used in the literature.

The most obvious approach is to simply count read and write operations exactly. This approach is the most tedious, and it rarely predicts the impact of I/O on the total execution time accurately. The time it takes to service I/O requests depends on many factors, including the size of the requests, whether data is prefetched or fetched on demand, whether the requests access data sequentially, etc. Therefore, exactly analyzing the number of I/O requests or the total amount of data that is read and written usually does not produce accurate running-time estimates.

A more relaxed approach is to asymptotically bound the number of I/O's in the algorithm. This is easier, of course, than counting I/O's exactly, especially when one wishes to prove lower bounds. The usual warnings that apply to any asymptotic bound are applicable here, with the added caveat that one must assume that main memories are large, as explained above. This reduces some asymptotic bounds, such as the ones for FFT and for sorting, to $\Theta(n)$.

Another approach, which is applicable only when the algorithm performs repeated sequential accesses to the data, is to count the number of times the algorithm accesses the data sets, commonly referred to as the number of passes. When applicable, this approach can predict running times very accurately. The approach can be extended somewhat by considering additional access patterns, such as out-of-core matrix transposition, as "primitives". We can then state, for example, that a certain FFT algorithm performs two passes over the data and one matrix transposition. In my opinion, such statements are more useful in practice than asymptotic bounds.

The last approach, which is widely used in the literature on dense-matrix algorithms, analyzes the so-called *level-3 fraction* in the algorithm. The level-3 fraction is the fraction of arithmetic operations that can be represented by matrix-matrix operations, mainly multiplication, factorization, and triangular solves. The rationale behind this approach is that matrix-matrix operations can be scheduled for a

[3]The DRAM/disk price ratio fluctuates (it was 100 two years ago, for example), but these fluctuations are too small to invalidate the argument that we present.

high level of data reuse, and that thus, an algorithm with a high level-3 fraction is likely to run well on a computer with caches or as an out-of-core algorithm.

The trouble with this approach is that it predicts what fraction of the work can be expected to enjoy *some* level of data reuse, but it does not predict how much data reuse can be achieved. One can thus find in the literature algorithms with a high level-3 fraction, but in which the matrix-matrix operations have a constant level of data reuse, say 2 or 4. These algorithms will clearly be inefficient when executed out-of-core. On the other hand, the level-3 fraction is a good indicator of cache performance, since even a modest level of data reuse is sufficient to reduce cache miss rates significantly on many computers (this is changing as cache miss penalties are growing).

4.3. Out-of-Core and Other Architecture-Aware Algorithms. Main memory and disks are just two levels of what is often an elaborate memory hierarchy. The memory systems of today's workstations and personal computers consist of a register file, two levels of cache, main memory, and disks. High-end machines often have three levels of cache, and their main memories and file systems are distributed. Algorithms that are designed to exploit one level of the memory system often exploit other levels well. For example, an out-of-core algorithm that is designed to exploit the speed of main memory when the data is stored on disks can sometimes be used to exploit caches and even the register file. Similarly, an algorithm that is designed to run well on a distributed-memory parallel computer can sometimes be efficient as an out-of-core algorithm. In particular, many partitioned dense-matrix algorithms can effectively exploit all of these architectural features. But this is not always true. The main memory/disk relationship is different enough from the cache/main memory relationship that some algorithms that are efficient out-of-core do not exploit caches efficiently, or vice versa. Algorithms that run well on a distributed-memory parallel computer do not always translate into efficient out-of-core algorithms, and vice versa.

At least until recently, the cache/main-memory bandwidth ratio has been much lower than the main-memory/disk bandwidth ratio. Therefore, algorithms with a modest level of data reuse exploit caches well, but are inefficient when implemented out-of-core. Algorithms which trade more work for a high level of data reuse are often useful out of core, but are inefficient in core, because the gain in cache effectiveness is more than offset by the loss in work efficiency. The cache/main-memory bandwidth ratio is increasing, so it is possible that in the future there will not be any difference between cache-aware and out-of-core algorithms. I believe, however, that it is more likely that new main memory technologies will improve main memory bandwidth.

Some algorithms have schedules that are efficient on distributed-memory parallel computers but have no efficient out-of-core schedules. The most important group of algorithms in this class are iterative linear solvers. Many of these algorithms can exploit a large number of processors and can be scheduled so that the amount of interprocessor communication is asymptotically smaller than the amount of work. For example, a conjugate gradient solver for a 2-dimensional problem of size n can theoretically exploit $n/\log n$ processors in a work-efficient manner, with a computation/communication ratio of $\sqrt{n/p}$. The high computation/communication ratio ensures that the algorithm runs efficiently even when interprocessor communication is significantly slower than the computation speed, which is common.

As explained in Section 3.2, these algorithms cannot be scheduled for any significant level of data reuse. Even though in both computational environments a processor can compute much faster than it can communicate (with other processors or with the disk), there is a crucial difference between the two environments. On the parallel computer, data that is not local is still being updated, but in the out-of-core environment, data that is not in core is not being updated.

There are also algorithms that can be scheduled for a high level of data reuse but cannot be scheduled in parallel, but they are not common in practice.

4.4. Abstractions for Scheduling. Discovering and implementing good schedules is difficult. Abstractions help us discover, understand, and implement complex schedules that allow algorithms to run efficiently out-of-core and in parallel. Many of the schedules and algorithms that this paper describes can be expressed in terms of simple abstractions.

Many efficient schedules for dense matrix algorithms can be expressed in terms of block-matrix algorithms. That is, the schedule is expressed as an algorithm applied to matrices consisting of submatrices rather than matrices of scalars.

Efficient out-of-core FFT algorithms are expressed in terms of multiple FFTs on smaller data sets and a structured permutation, namely, matrix transposition.

Schedules for general sparse matrix factorizations are usually expressed (and generated) using *elimination trees*, which are essentially compact approximate representations of the data-flow graphs of the algorithms. They are compact mostly because they represent dependences between rows and/or columns, not between individual nonzeros of the factors. Some elimination trees represent actual dependences, some represent a superset of the dependences.

While these are the most common abstractions one finds in the literature to describe schedules, there are a few others that deserve mention.

Recursion is a powerful abstraction that has not been used extensively by designers of numerical software, perhaps because of the influence of Fortran (which did not support procedural recursion until recently). The author designed a recursive schedule for dense LU factorization with partial pivoting that is more efficient than schedules based on block algorithms [**53**]. Gustavson applied the same technique to several other dense matrix algorithms [**32**].

Cormen developed an abstraction for describing structured permutations in terms of matrix operations on indices [**10**].

Van Loan uses an abstraction based on Kronecker products to describe FFT algorithms and their schedules [**57**].

References

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

[2] Mordechai Ben Ari. On transposing large $2^n \times 2^n$ matrices. *IEEE Transactions on Computers*, C-28(1):72–75, 1979. Due to a typo, the pages in that issue are marked as Vol. C-27.

[3] David H. Bailey. FFTs in external or hierarchical memory. *The Journal of Supercomputing*, 4(1):23–35, 1990.

[4] D. W. Barron and H. P. F. Swinnerton-Dyerm. Solution of simultaneous linear equations using a magnetic tape store. *Computer Journal*, 3:28–33, 1960.

[5] Petter E. Bjørstad. A large scale, sparse, secondary storage, direct linear equation solver for structural analysis and its implementation on vector and parallel architectures. *Parallel Computing*, 5(1):3–12, 1987.

[6] N. M. Brenner. Fast Fourier transform of externally stored data. *IEEE Transactions on Audio and Electroacoustics*, AU-17:128–132, 1969.

[7] Jean-Philippe Brunet, Palle Pedersen, and S. Lennart Johnsson. Load-balanced LU and QR factor and solve routines for scalable processors with scalable I/O. In *Proceedings of the 17th IMACS World Congress*, Atlanta, Georgia, July 1994. Also available as Harvard University Computer Science Technical Report TR-20-94.

[8] Gilles Cantin. An equation solver of very large capacity. *International Journal for Numerical Methods in Engineering*, 3:379–388, 1971.

[9] A. T. Chronopoulos and C. W. Gear. s-step iterative methods for symmetric linear systems. *Journal of Computational and Applied Mathematics*, 25:153–168, 1989.

[10] Thomas H. Cormen. *Virtual Memory for Data-Parallel Computing*. PhD thesis, Massachusetts Institute of Technology, 1992.

[11] Thomas H. Cormen. Determining an out-of-core FFT decomposition strategy for parallel disks by dynamic programming. In Michael T. Heath, Abhiram Ranade, and Robert S. Schreiber, editors, *Algorithms for Parallel Processing, volume 105 of IMA Volumes in Mathematics and its Applications*, pages 307–320. Springer-Verlag, 1998.

[12] Thomas H. Cormen and David M. Nicol. Out-of-core FFTs with parallel disks. *ACM SIGMETRICS Performance Evaluation Review*, 25(3):3–12, December 1997.

[13] Thomas H. Cormen, Jake Wegmann, , and David M. Nicol. Multiprocessor out-of-core FFTs with distributed memory and parallel disks. In *Proceedings of IOPADS '97*, pages 68–78, San Jose, California, November 1997.

[14] J. M. Crotty. A block equation solver for large unsymmetric matrices arising in the boundary integral equation method. *International Journal for Numerical Methods in Engineering*, 18:997–1017, 1982.

[15] Elizabeth Cuthill. Digital computers in nuclear reactor design. In Franz L. Alt and Morris Rubinoff, editors, *Advances in Computers*, volume 5, pages 289–348. Academic Press, 1964.

[16] J. J. Dongarra, S. Hammarling, and D. W. Walker. Key concepts for parallel out-of-core LU factorization. Technical Report CS-96-324, University of Tennessee, April 1996. LAPACK Working Note 110.

[17] J. J. Du Cruz, S. M. Nugent, J. K. Reid, and D. B. Taylor. Solving large full sets of linear equations in a paged virtual store. *ACM Transactions on Mathematical Software*, 7(4):527–536, 1981.

[18] A. A. Dubrulle. Solution of the complete symmetric eigenproblem in a virtual memory environment. *IBM Journal of Research and Development*, pages 612–616, November 1972.

[19] A. A. Dubrulle. The design of matrix algorithms for Fortran and virtual storage. Technical Report G320-3396, IBM Palo Alto Scientific Center, November 1979.

[20] S. C. Eisenstat, M. H. Schultz, and A. H. Sherman. Software for sparse Gaussian elimination with limited core memory. In Ian. S. Duff and C. W. Stewart, editors, *Sparse Matrix Proceedings*, pages 135–153. SIAM, Philadelphia, 1978.

[21] J. O. Eklundh. A fast computer method for matrix transposing. *IEEE Transactions on Computers*, C-21(7):801–803, 1972.

[22] Charbel Farhat. Large out-of-core calculation runs on the IBM SP2. *NAS News*, 2(11), August 1995.

[23] Bernd Fischer and Roland W. Freund. An inner product-free conjugate gradient-like algorithm for hermitian positive definite systems. In *Proceedings of the Cornelius Lanczos 1993 International Centenary Conference*, pages 288–290. SIAM, December 1993.

[24] Bernd Fischer and Roland W. Freund. On adaptive weighted polynomial preconditioning for hermitian positive definite matrices. *SIAM Journal on Scientific Computing*, 15(2):408–426, 1994.

[25] Patrick C.. Fischer and Robert L. Probert. A not one matrix multiplication in a paging environment. In *ACM '76: Proceedings of the Annual Conference*, pages 17–21, 1976.

[26] Nikolaus Geers and Roland Klee. Out-of-core solver for large dense nonsymmetric linear systems. *Manuscripta Geodetica*, 18(6):331–342, 1993.

[27] W. M. Gentleman and G. Sande. Fast Fourier transforms for fun and profit. In *Proceedings of the AFIPS*, volume 29, pages 563–578, 1966.

[28] Alan George and Hamza Rashwan. Auxiliary storage methods for solving finite element systems. *SIAM Journal on Scientific and Statistical Computing*, 6(4):882–910, 1985.

[29] J. A. George, M. T. Heath, and R. J. Plemmons. Solution of large-scale sparse least squares problems using auxiliary storage. *SIAM Journal on Scientific and Statistical Computing*, 2(4):416–429, 1981.

[30] Roger G. Grimes. Solving systems of large dense linear equations. *The Journal of Supercomputing*, 1(3):291–299, 1988.

[31] Roger G. Grimes and Horst D. Simon. Solution of large, dense symmetric generalized eigenvalue problems using secondary storage. *ACM Transactions on Mathematical Software*, 14(3):241–256, 1988.

[32] F. G. Gustavson. Recursion leads to automatic variable blocking for dense linear-algebra algorithms. *IBM Journal of Research and Development*, pages 737–755, November 1997.

[33] J.-W. Hong and H. T. Kung. I/O complexity: the red-blue pebble game. In *Proceedings of the 13th Annual ACM Symposium on Theory of Computing*, pages 326–333, 1981.

[34] Kenneth Klimkowski and Robert van de Geijn. Anatomy of an out-of-core dense linear solver. In *Proceedings of the 1995 International Conference on Parallel Processing*, pages III:29–33, 1995.

[35] Charles E. Leiserson, Satish Rao, and Sivan Toledo. Efficient out-of-core algorithms for linear relaxation using blocking covers. *Journal of Computer and System Sciences*, 54(2):332–344, 1997.

[36] Joseph W. H. Liu. On the storage requirement in the out-of-core multifrontal method for sparse factorization. *ACM Transactions on Mathematical Software*, 12(3):249–264, 1986.

[37] Joseph W. H. Liu. The multifrontal method and paging in sparse Cholesky factorization. *ACM Transactions on Mathematical Software*, 15(4):310–325, 1989.

[38] Jan Mandel. An iterative solver for p-version finite elements in three dimensions. *Computer Methods in Applied Mechanics and Engineering*, 116:175–183, 1994.

[39] A. C. McKeller and E. G. Coffman, Jr. Organizing matrices and matrix operations for paged memory systems. *Communications of the ACM*, 12(3):153–165, 1969.

[40] Cleve B. Moler. Matrix computations with Fortran and paging. *Communications of the ACM*, 15(4):268–270, 1972.

[41] Digambar P. Mondkar and Graham H. Powell. Large capacity equation solver for structural analysis. *Computers and Structures*, 4:699–728, 1974.

[42] C. J. Pfeifer. Data flow and storage allocation for the PDQ-5 program on the Philco-2000. *Communications of the ACM*, 6(7):365–366, 1963.

[43] Hans Riesel. A note on large linear systems. *Mathematical Tables and other Aids to Computation*, 10:226–227, 1956.

[44] Edward Rothberg and Robert Schreiber. Efficient, limited memory sparse Cholesky factorization. Unpublished manuscript, May 1997.

[45] J. Rutledge and H. Rubinstein. High order matrix computation on the UNIVAC. Presented at the meeting of the Association for Computing Machinery, May 1952.

[46] Joseph Rutledge and Harvey Rubinstein. Matrix algebra programs for the UNIVAC. Presented at the Wayne Conference on Automatic Computing Machinery and Applications, March 1951.

[47] John Salmon and Michael S. Warren. Parallel out-of-core methods for N-body simulation. In *Proceedings of the 8th SIAM Conference on Parallel Processing for Scientific Computing, CD-ROM*, Minneapolis, March 1997.

[48] Ulrich Schumann. Comments on "A fast computer method for matrix transposing" and application to the solution of Poisson's equation. *IEEE Transactions on Computers*, C-22(5):542–544, 1973.

[49] David S. Scott. Out of core dense solvers on Intel parallel supercomputers. In *Proceedings of the Fourth Symposium on the Frontiers of Massively Parallel Computation*, pages 484–487, 1992.

[50] David S. Scott. Parallel I/O and solving out of core systems of linear equations. In *Proceedings of the 1993 DAGS/PC Symposium*, pages 123–130, Hanover, NH, June 1993. Dartmouth Institute for Advanced Graduate Studies.

[51] R. C. Singleton. A method for computing the fast Fourier transform with auxiliary memory and limited high-speed storage. *IEEE Transactions on Audio and Electroacoustics*, AU-15:91–98, 1967.

[52] M. M. Stabrowski. A block equation solver for large unsymmetric linear equation systems with dense coefficient matrices. *International Journal for Numerical Methods in Engineering*, 24:289–300, 1982.

[53] Sivan Toledo. Locality of reference in LU decomposition with partial pivoting. *SIAM Journal on Matrix Analysis and Applications*, 18(4):1065–1081, 1997.

[54] Sivan Toledo and Fred G. Gustavson. The design and implementation of SOLAR, a portable library for scalable out-of-core linear algebra computations. In *Proceedings of the 4th Annual Workshop on I/O in Parallel and Distributed Systems*, pages 28–40, Philadelphia, May 1996.

[55] Sivan A. Toledo. *Quantitative Performance Modeling of Scientific Computations and Creating Locality in Numerical Algorithms*. PhD thesis, Massachusetts Institute of Technology, 1995.

[56] R. E. Twogood and M. P. Ekstrom. An extension of Eklundh's matrix transposition algorithm and its application in digital image processing. *IEEE Transactions on Computers*, C-25(9):950–952, 1976.

[57] Charles Van Loan. *Computational Frameworks for the Fast Fourier Transform*. SIAM, Philadelphia, 1992.

[58] Jeffry Scott Vitter and Elizabeth A. M. Shriver. Algorithms for parallel memory (in two parts) I: Two-level memories, and II: Hierarchical multilevel memories. *Algorithmica*, 12(2–3):110–169, 1994.

[59] H. H. Wang. An ADI procedure suitable for virtual storage systems. Technical Report G320-3322, IBM Palo Alto Scientific Center, January 1974.

[60] Edward L. Wilson, Klaus-Jürgen Bathe, and William P. Doherty. Direct solution of large systems of linear equations. *Computers and Structures*, 4:363–372, 1974.

[61] David Womble, David Greenberg, Stephen Wheat, and Rolf Riesen. Beyond core: Making parallel computer I/O practical. In *Proceedings of the 1993 DAGS/PC Symposium*, pages 56–63, Hanover, NH, June 1993. Dartmouth Institute for Advanced Graduate Studies. Also available online from `http://www.cs.sandia.gov/~dewombl`.

Xerox Palo Alto Research Center, 3333 Coyote Hill Road, Palo Alto, CA 94304.
E-mail address: `toledo@parc.xerox.com, sivan@math.tau.ac.il`