1 Introduction

In many scientific and industrial applications there is a need to solve large sparse systems of linear equations. This is the case in fields such as meteorology, oil reservoir modeling and structural analysis.

Treating sparse systems of equations differently from dense systems is important both for the time spent on solving a system and for the amount of storage needed to perform the computations. A large sparse matrix of dimension n may have fewer than 0.1% nonzeros. To store such a matrix as dense would require storing n^2 floating point values of which 99.9% would be zero. Clearly, this is very inefficient and limits severely the size of the problems that one can handle. Also, the amount of work that needs to be performed when solving a sparse system is mainly a function of the number of non-zeros. Thus ignoring that most elements are zero would slow down the computation considerably.

As the size of the problems that need to be solved increases, so does the need for fast computers. A cost efficient way of achieving increased computational speed is by the use of parallel computers. Today there exist parallel computers of several different architectures. However, due to the inherent differences between them one often have to develop new algorithms to solve the same problem on different architectures. The irregularity of sparse matrices makes it especially challenging to design efficient algorithms on parallel computers. Although methods developed for treating sparse matrices on sequential computers are often applicable on parallel computers, there are also many new issues that arise and must be taken into account.

In efficient algorithms for sparse problems on parallel computers one must achieve a good load balance between the processors. This means that the tasks that must be performed are distributed among the processors in such a way that each processor gets roughly the same amount of work. On many types of parallel computers it is difficult to schedule the tasks dynamically while the program is executing. In such a case the load balancing must be performed before the actual computation. A second requirement for efficiency is that the tasks are mapped to the processors in such a way as to reduce the time spent on communication. This may put restrictions on how the tasks are assigned. The time spent on precomputations to speed up the algorithm must be compared to the time spent on solving the actual problem. It is therefore important that the algorithms for load balancing are efficient.

This thesis address issues related to load balancing when performing sparse matrix computations on parallel computers. It consists of the following papers:

I. P. Bjørstad, F. Manne, T. Sørevik, and M. Vajteršic, *Efficient matrix multiplication on SIMD computers*, SIAM J. Matrix Anal. Appl., 13 (1992), pp. 386–401.

II. F. Manne and T. Sørevik, *Optimal partitioning of sequences*, Tech. Report CS-92-62, University of Bergen, Norway, 1992.

III. B. Olstad and F. Manne, *Efficient partitioning of sequences with an application to sparse matrix computations*, 1993.

IV. F. Manne, *An algorithm for computing an elimination tree of minimum height for a tree*, 1992.

V. F. Manne and H. Hafsteinsson, *Efficient sparse Cholesky factorization on a parallel SIMD computer*, 1993.

Paper II has been slightly revised from the technical report to the current paper. The results of IV are based on results from [29] and [30].

The main focus of this thesis is on the design of efficient parallel algorithms for direct methods for solving sparse systems of linear equations. However, we will also present work related to parallel iterative methods. The point of view we take is from the algorithmic side. Most problems that we consider have their origin in the numeric sphere. We do not develop any fundamentally new algorithms for these problems, but investigate instead what needs to be done in order to make standard numeric algorithms execute faster on parallel computers. In most of the work presented here we study how to achieve an even load balance, either by the scheduling of tasks among the processors or by reducing dependencies among the tasks themselves.

The implementations on parallel computers presented in this thesis are performed on a computer classified as a Single Instruction, Multiple Data computer (SIMD). This is a computer where each processor executes the same instruction in lockstep, but on individual data. One advantage of SIMD computers is that once an algorithm has been designed it is relatively easy to to implement it on the computer. This is especially true in terms of achieving synchronization when two processors need to communicate.

As a case study of how to program such computers paper I in this thesis is concerned with how to perform dense matrix multiplication on a SIMD computer. This paper highlights many of the issues that must be considered when using a SIMD computer. One such issue is the speed of memory references compared to the speed of communication.

The next subject we consider is a partitioning problem with applications to load balancing both in parallel and pipelined environments. In II and III we develop two efficient algorithms to solve this problem, as well as a number of variants of it. In III we also demonstrate how a solution to this partitioning problem can be used to speed up sparse matrix-vector multiplication on a SIMD computer. Matrix-vector multiplication is the core of many iterative algorithms for solving sparse linear systems.

Finally, in papers IV and V we look at direct methods for solving sparse linear systems on parallel computers. In particular we consider the use of Cholesky factorization to solve systems of the form $Ax = b$, where A is a sparse symmetric positive definite matrix.

Cholesky factorization of sparse matrices usually progresses in four separate stages: (1) Ordering, (2) symbolic factorization, (3) numeric factorization and (4) triangular solution. This is true both for sequential and parallel algorithms. In the first stage A must be ordered so that few fill elements are introduced while at the same time making A suitable for parallel algorithms. In IV we consider how to order a sparse matrix A so that its elimination tree is of low height. An efficient algorithm is given that computes an elimination tree of minimum height if the adjacency graph of A is a tree. This is the first non-trivial class of graphs for which the minimum elimination tree height problem has been solved.

In V we develop an algorithm for a SIMD computer that performs the numeric factorization stage of Cholesky factorization. Based on a graph-theoretical model of the tasks that need to be performed, we design an algorithm for assigning the data to the processors to achieve an even load balance. A number of test problems show that this method is superior to other suggested schemes for mapping of data to the processors.

The outline of this presentation is as follows: In Section 2 we give a short overview of different models for parallel computing. We discuss dense matrix multiplication in Section 3 and consider partitioning of sequences in Section 4. In Section 5 we look at parallel sparse Cholesky factorization. Finally, in Section 6 we summarize. Thesis papers I-V follow after this presentation.

2 Parallel computers

A parallel computer consists of a number of processors working together to solve a common task. The problem to be solved is divided into a number of subproblems. All of these are then solved simultaneously, each one on a separate processor. Thus the use of parallel computers offers a possibility for solving the problem faster than on a sequential computer.

Most commercially available parallel computers can be classified according to Flynn's taxonomy [9] as being in one of the two following classes:

1. Single Instruction stream, Multiple Data stream (SIMD).

2. Multiple Instructions stream, Multiple Data stream (MIMD).

On a SIMD computer each processor performs the same instruction but on individual data. The processors can make themselves idle by the means off logical expressions. SIMD computers may have up to several thousand tightly interconnected processors. Examples of such computers are the CM-2, the MasPar family of computers and the DAP computer.

On a MIMD computer each processor has its own individual instruction stream. MIMD computers often have a few but powerful processors. One computer built after this line is the Intel Paragon. An exception is the CM-5 computer with several thousands processors.

Both SIMD and MIMD computers can be further classified depending on whether the processors have local or shared memory. In a shared-memory computer each processor reads and writes to a common area of memory. This requires the use of a tie-breaking scheme if two processors want to access the same area of memory simultaneously. Communication between the processors can be done by writing into designated areas of the memory.

In a local-memory computer each processor has memory which it alone can access. Examples of such computers include all the SIMD computers mentioned above, the Intel Paragon, and the CM-5. Communication in local-memory computers requires that the processors are interconnected by some kind of network. Common fixed networks are hypercubes and grids. It is also possible that the processors have access to a general communication channel. In local-memory computers communication between processors is performed by the use of special communication primitives. Whenever two processors need to communicate they must agree on some kind of synchronization.

This is easily handled on SIMD computers since the processors operate in lockstep. On MIMD computers synchronization can be more difficult to achieve.

The large number of processors in many SIMD computers requires that the processors are organized in a highly structured manner such as a grid. This tends to make SIMD computers less flexible than MIMD computers. An advantage particular to shared memory MIMD computers is that they can support dynamic load balancing. This is often not practical on local memory SIMD computers, where the scheduling of tasks to processors must be done in advance of the computations. For more on the various architectures for parallel computers and their merits see [1, 33].

In papers I, III, and V we will report on experiments performed on the MasPar family of computers. These include the MP-1 computer [5] and the new version, the MP-2. Both the MP-1 and MP-2 computers are local-memory SIMD computers consisting of a large number of processors arranged in a toroidal wrapped grid. Each processor is connected to its 8 nearest neighbors. The processors have two ways of communication: Either by a general point to point communication channel called the router or along the grid lines. Communication along the grid lines is much faster but more restricted than through the router. The processors also support indirect addressing. Other computers with similar features as the MasPar computers include the DAP and the CM-2.

3 Parallel matrix multiplication

The multiplication of dense matrices is an important task in many areas of linear algebra. This is a compute intensive operation where most tasks can be performed independently of each other. Thus it lends itself well to be solved on parallel computers. As a consequence there exists a number of algorithms for matrix multiplication on different types of parallel computers, see [1, 15] for examples.

The standard sequential algorithm for calculating the product $A * B = C$ where A, B and C are $n \times n$ matrices is given by $c_{i,j} = \sum_{0 \leq k < n} a_{i,k} * b_{k,j}$. It follows that the standard matrix multiplication algorithm requires n^3 multiplications and $n^2(n-1)$ additions.

There are other ways to perform matrix multiplications. Winograd [36] proposed a method that reduces the number of multiplications at the expense of an increased number of additions compared to the standard algorithm. It requires $n^3/2+n^2$ multiplications and $3n^3/2+2n^2-2n$ additions. This is of interest if additions can be performed faster than multiplications. Strassen [35] proposed a fast recursive algorithm for performing matrix multiplication that requires only $O(n^{2.807})$ operations. This method thus offers an asymptotic speed-up compared to conventional matrix multiplication. However, this comes at a prize since the numeric stability of Strassen's method is somewhat weaker than for the ordinary matrix multiplication algorithm [7, 20].

Many algorithms in linear algebra can be formulated in terms of operations on blocks. This may reduce the amount of data movement both for sequential computers with multi-layered memory and for parallel message-passing computers. Matrix multiplication is among the most used block operations. As an example, dense Cholesky factorization can be formulated as containing a large degree of block matrix multiplications [15].

We choose to study parallel matrix multiplication based on an interest to explore the new range of SIMD computers. In paper I we investigate how matrix multiplication can be performed on the MasPar MP-1 computer. The various aspects considered include issues such as: the relative speed of communication compared to arithmetic and the relative speed between additions and multiplications. In the paper we show that non-standard practical algorithms such as those proposed by Strassen and Winograd can be used to increase the efficiency of parallel matrix multiplication. We also look at how block algorithms can be implemented efficiently, and how suchs algorithms are affected by different schemes for mapping the data onto the processors.

4 Partitioning of sequences

In parallel computing one often has a pool of tasks that need to be distributed among the processors. The objective is to achieve an even load balance between the processors. There might also be a number of constraints on how the tasks are executed. Examples are a partial ordering of the tasks and a start time and deadline for each task. For an overview of different scheduling problems see [16].

Finding an optimal solution to scheduling problems can sometimes be very difficult. Even for two processors with no special constraints on the jobs or the processors, minimizing the total completion time is known to be NP-hard [10]. Thus scheduling problems must have enough restrictions in order to be tractable.

In the papers II and III we consider one such restricted problem. It involves how to partition a sequence of n ordered tasks into p intervals such that the maximum cost of the intervals, measured with a cost function f is minimized. We also show how a number of variants of this problem can be solved.

In III we demonstrate how a solution to this problem can be used to speed up an algorithm suggested by Ogielski and Aiello [31] for sparse matrix-vector multiplication on a SIMD computer. This operation is the core of many iterative algorithms for solving sparse systems such as conjugate gradients [15]. Also the use of partitioned inverses [2] for solving triangular systems of equations, involves repeated applications of sparse matrix, dense vector multiplication.

Another interesting application of this partitioning problem is the following:

Often in communication systems a continuous stream of data packages has to be received and processed in real time. The processing may among other things involve demodulation, error correction, and possibly decryption of each incoming data package before the contents of the package can be accessed [17]. Assume that n computational operations are to be performed in a pipelined fashion on a continuous sequence of incoming data packages. With n processors we may assign one operation to each processor and connect the processors in a chain. This would give high throughput of the system. The time to process the data is now dominated by the processor that has to perform the most time-consuming operation. With this mapping of the operations to the processors, each processor will be idle once it has finished its operation and have to wait for the processor with the most time consuming operation, until it can get a new data package. This is an inefficient utilization of the processors if the time for performing each task varies greatly. Thus to circumvent this we get the following problem: partition n linearly ordered task into p intervals ($p \leq n$) such that the maximal time to finish executing the tasks assigned to each interval is minimized. This way we can achieve good processor efficiency and still hopefully keep the throughput of the system high.

Previous studies of this partitioning problem [3, 6, 18] has lead to an algorithm of time complexity $O(n^2p)$. In II we give an algorithm of complexity $O(p(n-p)\log p)$. The method used is based on finding a series of non-optimal partitionings such that there exists only one way in which each one can be improved. A similar idea is used in IV for partitioning of trees. In III we give a $\Theta(p(n-p))$ dynamic programming algorithm for the same problem. We have chosen to include paper II in this thesis even though the presented algorithm has a higher worst case bound than the one presented in III for two main reasons. The first one is that the algorithms in III *must* perform $p(n-p)$ steps on *any* input, where the algorithm in II might perform fewer than $(n-p)p\log p$ steps depending on the input. Also, there are variants of the general partitioning problem where the algorithm in III is not asymptotically faster than the algorithm in II even in the theoretically worst case.

5 Parallel sparse Cholesky-factorization

Consider the linear system of equations $Ax = b$ where A is an $n \times n$ large sparse symmetric positive definite matrix. The are two main ways such a system can be solved: either by a direct method or by iterative methods. Since A is symmetric positive definite, the direct method of choice is Cholesky factorization [13].

There are many settings where Cholesky factorization might be preferred over an iterative method. One such case is when there are multiple right hand sides. Then if the Cholesky factorization LL^T is known the problem reduces to solving two lower triangular systems with multiple right hand sides. This may be more efficient than an iterative method which would require a complete execution of the algorithm for each right hand side. We further note that Cholesky factorization might be advantageous for reasons of numeric stability.

On sparse matrices sequential Cholesky factorization is usually done in four separate stages:

1. *Ordering.* Determine a permutation matrix P so that the Cholesky factor L of PAP^T will suffer little fill.

2. *Symbolic factorization.* Determine the structure of the nonzeros of L and set up a data structure in which to store A and compute the nonzero entries of L.

3. *Numeric factorization.* Insert the nonzeros of A into the data structure and compute the numeric values of L.

4. *Triangular solution.* Solve $Ly = Pb$ and $L^T z = y$, and then set $x = P^T z$.

Most parallel algorithms for performing sparse Cholesky-factorization also operate in the same four stages [19].

Finding a permutation for A in the first stage, so that L suffers little fill is also important for parallel algorithms. The added complication in parallel environments is that PAP^T should be suitable for parallel methods. One way this can be determined is by the use of the *elimination tree* [28, 34]. This is a data structure that mirrors the potential high-level parallelism found in sparse Cholesky-factorization [26]. This is true both for the symbolic and the numeric factorization. The height of the elimination tree is generally considered as measure of goodness, with short trees assumed to be superior to tall ones. However, like the problem of minimizing fill [37], the problem of finding the lowest possible elimination tree is known to be NP-hard [32].

Nested dissection is a method for ordering G that was developed to reduce fill [11, 12] which has been shown to produce low elimination trees [23]. Another approach to finding low elimination trees is to first compute a fill-reducing ordering P. From this a new ordering is then computed such that the resulting elimination tree of minimum height under the restriction that no new fill is introduced [21, 25, 27]. However, little is known about the computation of elimination trees of minimum height for classes of graphs when additional fill is allowed, and how much fill this might cause.

In paper IV we discuss how orderings giving low elimination trees can be found. We present an efficient algorithm that solves the minimum height problem for the class of graphs that are trees. The algorithm is shown to have time complexity $O(n \log n \log d)$, where d is the maximum degree of any node in G. This is the first efficient algorithm for computing an elimination tree of minimum height for a nontrivial class of graphs. In doing so it does not introduce more than $n-1$ fill edges. We also show in the same paper that for any graph there exists a *minimal cutset ordering* giving an elimination tree of minimum height.

The numeric factorization is the most compute intensive operation of Cholesky-factorization. A substantial amount of research has been carried out to perform this operation both on shared-memory and distributed-memory MIMD-computers, see [19] for an overview. However, we are only aware of two papers [14, 22] that have tried to perform the numeric factorization on a SIMD computer.

In paper V we develop and implement an algorithm for performing the numeric factorization on a SIMD computer. This algorithm is similar to the one described in [22]. The algorithm is a *fan-out* algorithm based on computing one outer-product at a time. Thus the algorithm does not explicitly take advantage of the elimination tree. The reason for using a fan-out algorithm that performs the outer-products in a sequential manner is due to the cost of communication on the computer that we use: on the MP-2, the cost of sending one value to each processor in the same column (or row) is little over twice that of sending it to the nearest neighbor. This indicates that once a value has to be sent to another processor it could as well be sent to every processor that might need it. If the outer-products that need to be performed are sufficiently large compared to the processor array, it is possible to keep a large fraction of the processor busy in each outer-product.

The mapping of the data onto the processors is crucial for the performance of the algorithm. We show that obtaining a good load balance can be viewed as a graph coloring problem on a weighted graph. Based on this observation we develop a greedy algorithm that maps the data onto the processors in order to achieve an even load balance. We show that this method is superior to other suggested mapping schemes for a number of test problems from the Harwell-Boeing collection [8] as well as on regular structured problems.

The time spent on preprocessing to achieve an even load balance must be compared with the time saved in the numeric factorization algorithm. If the same structural system is to be solved many times then only the numeric factorization and triangular solve have to be performed each time. This is the case in the interior point method for linear programming [4, 24]. In such settings the time spent on computing a good load balance can be averaged over the total time spent on factoring the matrices.

6 Conclusion

The papers in this thesis are mainly in the domain of load balancing on parallel computers. We have investigated aspects related to SIMD-computing and parallel sparse linear algebra.

In paper I we investigated how matrix multiplication could be performed on a SIMD-computer. We showed that non-standard algorithms such as those proposed by Strassen and Winograd can be used to speed up parallel matrix multiplication.

In paper II and III we developed two different algorithms for solving a partitioning problem. The first algorithm was of time complexity $O((n-p)p \log p)$ and the second one of $\Theta((n-p)p)$. It would be interesting to see if these two approaches could be combined into a single algorithm of time complexity $O((n-p)p)$ but such that one did not always need to perform $(n-p)p$ steps.

In paper III we reported on how the partitioning algorithm was used as a heuristic for a two-dimensional partitioning problem to speed up matrix-vector multiplication on a SIMD computer. This method has also been tried in connection with sparse Cholesky-factorization on a SIMD computer. The results, however, were not as good as the those presented in paper V and are therefore not reported here. Initial studies indicates that the two-dimensional partitioning problem might be harder than the one-dimensional case. It would be interesting to know whether the two-dimensional case could be solved efficiently or if the problem is NP-hard.

In paper IV we gave an algorithm for computing an elimination tree of minimum height for a matrix whose adjacency graph is a tree. This algorithm was based on limiting the search area for possible solutions in such a way that there only existed one way in which a given solution could be improved. Whenever it was not possible to improve a solution we shoved that this solution must be optimal. The same basic idea was also used in the paper II for partitioning of sequences. We also gave a proof in IV that for any matrix there exists a minimal cutset ordering giving an elimination tree of minimum height. This is a fact that can be exploited when designing algorithms for computing low elimination trees. It would be of interest to see if clique trees could be combined with the results in this paper into an algorithm to compute an elimination tree of minimum height for chordal graphs.

In paper V we presented an algorithm for performing the numeric factorization in sparse Cholesky-factorization on a SIMD computer. We note that the algorithm does not take advantage of the parallelism given by the elimination tree, although we expect it to execute faster if the elimination tree is low. We have also investigated other algorithms that would take advantage of the elimination tree, but found that on the MP-2 computer these were less efficient than the presented algorithm. This is mainly due to the small difference in time between sending a value to the nearest processor neighbor and the time to broadcast it across a processor column.

We also did a graph-theoretical study of the tasks to be performed in the presented Cholesky-factorization algorithm. From this we developed a load balancing algorithm for assigning the data to the processors in order to reduce the number of parallel steps. Through a number of test problems we showed that this method was superior to previously suggested mapping schemes. Some of the matrices on which we tried the algorithm were very large. As a consequence moving the matrices onto the the processor array was very time consuming. We also found that on the largest matrices the ordering and the symbolic factorization, was slower than expected due to paging problems. It would therefore be interesting to see if these algorithms could also be implemented on a SIMD computer in order to achieve additional speed-up.

References

[1] S. G. Akl, *The design and analysis of parallel algorithms*, Prentice-Hall, 1989.

[2] F. L. Alvarado and R. Schreiber, *Optimal parallel solution of sparse triangular systems*, SIAM J. Sci. Statist. Comput., 14 (1993).

[3] S. Anily and A. Federgruen, *Structured partitioning problems*, Operations Research, 13 (1991), pp. 130–149.

[4] G. Astfalk, I. Lustig, R. Marsten, and D. Shanno, *The interior-point method for linear programming*, IEEE Software, (1992), pp. 61–68.

[5] T. Blank, *The MasPar MP-1 architecture*, in Proceedings of IEEE Compcon Spring 1990, IEEE, February 1990.

[6] S. H. Bokhari, *Partitioning problems in parallel, pipelined, and distributed computing*, IEEE Trans. Comput., 37 (1988), pp. 48–57.

[7] R. P. Brent, *Algorithms for matrix multiplication*, Tech. Report CS 157, Computer Science Department, Stanford University, Stanford, CA, 1960.

[8] I. Duff, R. Grimes, and J. Lewis, *Sparse matrix test problems*, ACM Trans. Math. Software, 15 (1989), pp. 1–14.

[9] M. J. Flynn, *Very high–speed computing systems*, in Proceedings of the IEEE 54, 1966, pp. 1901–1909.

[10] M. R. Garey and D. S. Johnson, *Complexity results for multiprocessor scheduling under resource constraints*, SIAM J. Comput., 4 (1975), pp. 397–411.

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

[12] A. George and J. W. H. Liu, *An automatic nested dissection algorithm for irregular finite element problems*, SIAM J. Numer. Anal., 15 (1978), pp. 1053–1069.

[13] ——, *Computer Solutions of Large Sparse Positive Definite Systems*, Prentice-Hall, 1981.

[14] J. R. Gilbert and R. Schreiber, *Highly parallel sparse Cholesky factorization*, SIAM J. Sci. Statist. Comput., 13 (1992), pp. 1151–1172.

[15] G. H. Golub and C. F. V. Loan, *Matrix Computations*, North Oxford Academic, 2 ed., 1989.

[16] R. L. Graham, E. L. Lawler, J. K. Lenstra, and A. H. G. R. Kan, *Optimization and approximation in deterministic sequencing and scheduling: A survey*, Annals of Discrete Mathematics, 5 (1979), pp. 287–326.

[17] F. Halsall, *Data Communications, Computer Networks and OSI*, Addison Wesley, 1988.

[18] P. Hansen and K.-W. Lih, *Improved algorithms for partitioning problems in parallel, pipelined, and distributed computing*, IEEE Trans. Comput., 41 (1992), pp. 769–771.

[19] M. T. Heath, E. Ng, and B. Peyton, *Parallel algorithms for sparse linear systems*, SIAM Rev., 33 (1991), pp. 420–460.

[20] N. J. Higham, *Exploiting fast matrix multiplication within the level 3 BLAS*, ACM Trans. Math. Software, 16 (1990), pp. 352–368.

[21] J. A. G. Jess and H. G. M. Kees, *A data structure for parallel L/U decomposition*, IEEE Trans. Comput., C-31 (1982), pp. 231–239.

[22] S. G. Kratzer and A. J. Cleary, *Sparse matrix factorization of SIMD parallel computers*, Tech. Report SRC-TR-92-063, Supercomputing Research Center, 1992.

[23] C. E. Leiserson and J. G. Lewis, *Orderings for parallel sparse symmetric factorization*, in Parallel Processing for Scientific Computing, G. Rodrigue, ed., SIAM, 1989, pp. 27–32.

[24] R. Levkovitz and G. Mitra, *Solution of large scale linear programs: A review of hardware, software, and algorithmic issues*, Tech. Report TR/06/92, Dept. of Mathematics and Statistics, Brunel University, 1992.

[25] J. G. Lewis, B. W. Peyton, and A. Pothen, *A fast algorithm for reordering sparse matrices for parallel factorization*, SIAM J. Sci. Statist. Comput., 10 (1989), pp. 1146–1173.

[26] J. W. H. Liu, *Computational models and task scheduling for parallel sparse Cholesky factorization*, Parallel Comput., 3 (1986), pp. 327–342.

[27] ——, *Reordering sparse matrices for parallel elimination*, Parallel Comput., 11 (1989), pp. 73–91.

[28] ——, *The role of elimination trees in sparse factorization*, SIAM J. Matrix Anal. Appl., 11 (1990), pp. 134–172.

[29] F. Manne, *Reducing the height of an elimination tree through local reorderings*, Tech. Report CS-91-51, University of Bergen, Norway, 1991.

[30] ——, *An algorithm for computing an elimination tree of minimum height for a tree*, Tech. Report CS-91-59, University of Bergen, Norway, 1992.

[31] A. T. Ogielski and W. Aiello, *Sparse matrix computations on parallel processor arrays.* To appear in SIAM J. Sci. Statist. Comput., 1993.

[32] A. Pothen, *The complexity of optimal elimination trees*, Tech. Report CS-88-13, Pennsylvania State University, 1988.

[33] M. J. Quinn, *Designing efficient algorithms for parallel computers*, McGraw-Hill, 1987.

[34] R. Schreiber, *A new implementation of sparse Gaussian elimination*, ACM Trans. Math. Software, 8 (1982), pp. 256–276.

[35] V. Strassen, *Gaussian elimination is not optimal*, Numer. Math, 13 (1969), pp. 354–356.

[36] S. Winograd, *A new algorithm for inner product*, IEEE Trans. Comput., C-18 (1968), pp. 693–694.

[37] M. Yannakakis, *Computing the minimum fill-in is NP-complete*, SIAM J. Alg. Discrete Meth., 2 (1981), pp. 77–79.