CONTENTS

CONTRIBUTORS

William B. Baringer
Dept. of EECS
UC Berkeley

Barry Boes
327 Arlington Circle
Ridgeland, MS 39157

1

SCALABLE PARALLEL ALGORITHMS FOR UNSTRUCTURED PROBLEMS

Vipin Kumar, Ananth Grama, Anshul Gupta, and George Karypis

Computer Science Department,
University of Minnesota, Minneapolis, Minnesota 55455

ABSTRACT

In this paper we summarize our work on development of parallel algorithms for searching large unstructured trees, and for finding solution of large sparse systems of linear equations. Search of large unstructured trees is at the core of many important algorithms for solving discrete optimization problems. Solution of large sparse systems of equations is required for solving many important scientific computing problems. For both of these domains, we show that highly scalable parallel algorithms can be developed, and these algorithms can obtain high speedup on a large number of processors.

1 INTRODUCTION

The driving force behind high performance computing is the need to solve large scale application problems. To effectively perform such tasks using parallel computers, scalable and portable algorithms must be developed. In this paper, we describe scalable parallel algorithms used in a variety of unstructured domains. Issues of load balancing and reducing processor idling become particularly important in such domains. Furthermore, combined with the unstructured nature of communication patterns, parallel algorithms in these domains tend to be significantly more involved.

2 PARALLEL ALGORITHMS FOR DISCRETE OPTIMIZATION

Discrete optimization problems (DOPs) arise in various applications such as planning, scheduling, computer aided design, robotics, game playing and constraint directed reasoning. Often, a DOP is formulated in terms of finding a (least cost) solution path in a graph from an initial node to a goal node and solved by graph/tree search methods such as branch-and-bound and dynamic programming.

Given that DOP is an NP-hard problem, one may argue that there is no point in applying parallel processing to these problems, as the worst-case run time can never be reduced to a polynomial unless we have an exponential number of processors. However, the average time complexity of heuristic search algorithms for many problems is polynomial. Also, there are some heuristic search algorithms which find suboptimal solutions in polynomial time (e.g., for certain problems, approximate branch-and-bound algorithms are known to run in polynomial time). In these cases, parallel processing can significantly increase the size of solvable problems. Some applications using search algorithms (e.g. robot motion planning, task scheduling) require real time solutions. For these applications, parallel processing is perhaps the only way to obtain acceptable performance. For some problems, optimal solutions are highly desirable, and can be obtained for moderate sized instances in a reasonable amount of time using parallel search techniques (e.g. VLSI floor-plan optimization [1]).

In this section, we provide an overview of our research on parallel algorithms for solving discrete optimization problems.

2.1 Parallel Depth First Search

Depth-first search (DFS), also referred to as Backtracking, is a general technique for solving a variety of discrete optimization problems [17, 3, 1]. Since many of the problems solved by DFS are computationally intensive, there has been a great interest in developing parallel versions of DFS. For most applications, state space trees searched by DFS tend to be highly irregular, and any static allocation of subtrees to processors is bound to result in significant load imbalance among processors.

In [11, 19, 24, 31], we presented and analyzed a number of parallel formulations of DFS which retain the storage efficiency of DFS, and can be implemented on any MIMD multiprocessor. In these parallel formulations, the search space is dynamically distributed among a number of processors. These formulations differ in the methods used for distributing work among processors. To study the effectiveness of these formulations, we have applied these to a number of problems. Parallel formulations of IDA* for solving the 15 puzzle problem yielded near linear speedup on Sequent Balance up to 30 processors and on the Intel Hypercube^{TM} and BBN Butterfly^{TM} up to 128 processors [23, 24, 31]. Parallel formulation of PODEM, which is the best known sequential algorithm for solving the test pattern generation problem, provided linear speedup on a 128 processor Symult^{TM} [3]. Parallel depth-first branch-and-bound for floorplan optimization for VLSI circuits yielded linear speedups on a 1024 processor Ncube^{TM}, a 128 processor Symult^{TM} and a network of 16 SUN workstations [1]. Linear speedups were also obtained for parallel DFS for the tautology verification problem for up to 1024 processors on the Ncube/10^{TM} and the Ncube/2^{TM} [2, 11, 19]. [1] In [14], we have presented new methods for load balancing of unstructured tree computations on large-scale SIMD machines such as CM-2 ^{TM}[2]. The analysis and experiments show that our new load balancing methods provide good speedups for parallel DFS on SIMD architectures. In particular, their scalability is no worse than that of the best load balancing schemes on MIMD architectures.

Scalability Analysis

From experimental results for a particular architecture and a range of processors alone, it is difficult to ascertain the relative merits of different parallel algorithm-architecture combinations. This is because the performance of different schemes may be altered in different ways by changes in hardware characteristics (such as interconnection network, CPU speed, speed of communication channels etc.), number of processors, and the size of the problem instance being solved [21]. Hence any conclusions drawn from experimental results on a specific parallel computer and problem instance are rendered invalid by changes in any one of the above parameters. Scalability analysis of a parallel algorithm and architecture combination has been shown to be useful in extrapolating these conclusions [20, 24].

[1] Our work on tautology verification, test pattern generation and floorplan optimization received honorable mention for the Gordon Bell Award for outstanding research in practical parallel computing. This is significant considering that this award has historically been given to researchers working on numerical problems.

[2] CM-2 is a registered trademark of the Thinking Machines Corporation.

We have developed a scalability metric, called *isoefficiency* , which relates the problem size to the number of processors necessary for an increase in speedup in proportion to the number of processors used [24]. In general, for a fixed problem size W, increasing the number of processors P causes a decrease in efficiency because parallel processing overhead will increase while the sum of time spent by all processors in meaningful computation will remain the same. Parallel systems that can maintain a fixed efficiency level while increasing both W and P are defined as *scalable* [20]. If W needs to grow as $f(P)$ to maintain an efficiency E, then $f(P)$ is defined to be the **isoefficiency function** for efficiency E and the plot of $f(P)$ with respect to P is defined to be the isoefficiency curve for efficiency E. An important feature of isoefficiency analysis is that it succinctly captures the effects of characteristics of the parallel algorithm as well as the parallel architecture on which it is implemented, in a single expression. The isoefficiency metric has been found to be very useful in characterizing scalability of a number of algorithms [25].

Scalability Analysis of Parallel Depth-First Search

The primary reason for loss of efficiency in our parallel formulations of DFS is the communication overhead incurred by different processors in finding work and in contention over shared resources. By modeling these, we were able to determine the isoefficiency functions (and thus scalability) of a variety of work-distribution schemes[11, 19, 24] for the ring, cube, mesh, network of workstations, and shared memory architectures. Some of the existing parallel formulations were found to have poor scalability. This motivated the design of substantially improved schemes for various architectures. We established lower bounds on the scalability of any possible parallel formulation for various architectures. For each of these architectures, we determined near optimal load balancing schemes. The performance characteristics of various schemes have been proved and experimentally verified in [11, 19, 24, 31]. Schemes with better isoefficiency functions performed better than those with poorer isoefficiency functions for a wide range of problem sizes and processors. Table 1 presents isoefficiency functions of various parallel formulations on various architectures. For a more detailed discussion see [11, 19].

From our scalability analysis of a number of architecture-algorithm combinations, we have been able to gain valuable insights into the relative performance of parallel formulations for a given architecture. For instance, in [1] an implementation of parallel depth first branch and bound for VLSI floorplan optimization is presented, and speedups obtained on a network of 16 workstations. The essential part of this branch-and-bound algorithm is the GRR load balanc-

	ARR	NN	GRR	GRR-M	RP
SM	$O(P^2 \log P)$	$O(P^2 \log P)$	$O(P^2 \log P)$	$O(P \log P)$	$O(P \log^2 P)$
Cube	$O(P^2 \log^2 P)$	$\Theta(P^{\log_2 \frac{1+\frac{1}{\alpha}}{2}})$	$O(P^2 \log P)$	$O(P \log^2 P)$	$O(P \log^3 P)$
Ring	$O(P^3 \log P)$	$\Theta(K^P)$	$O(P^2 \log P)$		$O(P^2 \log^2 P)$
Mesh	$O(P^{2.5} \log P)$	$\Theta(K^{\sqrt{P}})$	$O(P^2 \log P)$	$O(P^{1.5} \log P)$	$O(P^{1.5} \log^2 P)$

Table 1 Scalability results of various parallel DFS formulations for different architectures. The various scemes such as ARR, NN, GRR, GRR-M, and RP are described in detail in [11, 19].

ing technique. Our scalability analysis can be used to investigate the viability of using a much larger number of workstations for solving this problem. Note from Table 1 that GRR has an overall isoefficiency of $O(P^2 \log P)$ for this platform. Hence, if we had 1024 workstations on the network, we can obtain the same efficiency on a problem instance which is 10240 ($= \frac{1024^2 \log 1024}{16^2 \log 16}$) times bigger compared to a problem instance being run on 16 processors. This result is of significance, since it indicates that it is indeed possible to obtain good efficiencies with large number of workstations. Scalability analysis also sheds light on the degree of scalability of such a system with respect to other parallel architectures such as hypercube and mesh multicomputers. For instance, the best applicable technique implemented on a hypercube has an isoefficiency function of $O(P \log^2 P)$. With this isoefficiency, we would be able to get identical efficiencies as those obtained on 16 processors by increasing the problem size 400 fold (which is $\frac{1024 \log^2 1024}{16 \log^2 16}$). We can thus see that it is possible to obtain good efficiencies even with smaller problems on the hypercube. We can thus conclude from isoefficiency functions that the hypercube offers a much more scalable platform compared to the network of workstations for this problem.

These parallel formulations were extensively tested on a 1024 node nCUBE2 parallel computer. Table 2 shows average speedup obtained in parallel algorithm for solving instances of the satisfiability problem using different load balancing techniques.

2.2 Parallel Best First Search

The A* algorithm is a well known search algorithm that can use problem-specific heuristic information to prune search space. As discussed in [17], A*

Scheme→ P↓	ARR	GRR-M	RP	NN	GRR
8	7.506	7.170	7.524	7.493	7.384
16	14.936	14.356	15.000	14.945	14.734
32	29.664	28.283	29.814	29.680	29.291
64	57.721	56.310	58.857	58.535	57.729
128	103.738	107.814	114.645	114.571	110.754
256	178.92	197.011	218.255	217.127	184.828
512	259.372	361.130	397.585	397.633	155.051
1024	284.425	644.383	660.582	671.202	

Table 2 Average speedups for various parallel formulations.

is essentially a "best-first" branch-and-bound (B&B) algorithm. A number of researchers have investigated parallel formulations of A*/B&B algorithms [16, 22]. An important component of A*/B&B algorithms is the priority queue which is used to maintain the "frontier" (*i.e.*, unexpanded) nodes of the search graph in a heuristic order. In the sequential A*/B&B algorithm, in each cycle a most promising node from the priority queue is removed and expanded, and the newly generated nodes are added to the priority queue.

In most parallel formulations of A*, different processors concurrently expand different frontier nodes. Conceptually, these formulations can be viewed to differ in the data structures used to implement the priority queue of the A* algorithm. Some formulations are suited only for shared-memory architectures, whereas others are suited for distributed-memory architectures as well. The effectiveness of different parallel formulations is also strongly dependent upon the characteristics of the problem being solved. We have investigated a number of parallel formulations of A*[22]. Some of these formulations are new, and the others are very similar to the ones developed by other researchers. We have tested the performance of these formulations on the 15-puzzle, the traveling salesman problem(TSP), and the vertex cover problem (VCP) on the BBN Butterfly^{TM} multiprocessor. The results for the 15-puzzle and VCP are very similar, but very different from results obtained for the TSP. The reason is that the TSP and VCP generate search spaces that are qualitatively different from each other, even though both problems are NP-hard problems. We have also performed a preliminary analysis of the relationship between the characteristics of the search spaces and their suitability to various parallel formulations[22].

2.3 Speedup Anomalies in Parallel Search

In parallel DFS, the speedup can differ greatly from one execution to another, as the actual parts of the search space examined by different processors are determined dynamically, and can be different for different executions. Hence, for some execution sequences the parallel version may find a solution by visiting fewer nodes than the sequential version thereby giving superlinear speedup, whereas for others it may find a solution only after visiting more nodes resulting in sublinear speedup. It may appear that on the average the speedup would be either linear or sublinear. This phenomenon of speedup greater than P on P processors in isolated executions of parallel DFS has been reported by many researchers. for a variety of problems and is referred to by the term speedup anomaly.

In [32], we present analytical models and experimental results on the average case behavior of parallel backtracking. We consider two types of backtrack search algorithms: (i) simple backtracking (which does not use any heuristic information); (ii) heuristic backtracking (which uses heuristics to order and prune search). We present analytical models to compare the average number of nodes visited in sequential and parallel search for each case. For simple backtracking, we show that the average speedup obtained is (i) linear when distribution of solutions is uniform and (ii) superlinear when distribution of solutions is non-uniform. For heuristic backtracking, the average speedup obtained is at least linear (i.e., either linear or superlinear), and the speedup obtained on a subset of instances (called difficult instances) is superlinear. We also present experimental results over many synthetic and practical problems on various parallel machines, that validate our theoretical analysis.

3 HIGHLY PARALLEL DIRECT SOLVERS FOR SPARSE LINEAR SYSTEMS

Solving a large sparse system of linear equations is often one of the core computations in large-scale unstructured problems that arise in structural engineering, fluid dynamics, power networks, modeling, linear programming, and various other engineering and scientific applications. Sparse linear systems can be solved using either iterative or direct methods—both have their own advantages and disadvantages and the more suitable method depends on the type of the problem. Although direct methods require more memory and are often costlier than iterative methods, they are important because of their generality and robustness. For solving sparse linear systems arising in certain applications, they are the only feasible methods known.

In [12], we describe an optimally scalable parallel algorithm for factorization of sparse matrices. This algorithm incurs strictly less communication overhead than any known parallel formulation of sparse matrix factorization, and hence, can utilize a higher number of processors effectively. It is well known that dense matrix factorization can be implemented efficiently on distributed-memory parallel computers [29, 18]. However, despite inherent parallelism in sparse sparse direct methods, not much success has been achieved to date in developing their scalable parallel formulations [13, 34]. We have shown that the parallel Cholesky factorization algorithm described in [12] is as scalable as the best parallel formulation of dense matrix factorization on both mesh and hypercube architectures. We also show that our algorithm is equally scalable for sparse matrices arising from two- and three-dimensional finite element problems.

It is difficult to derive analytical expressions for the number of arithmetic operations in factorization and for the size (in terms of number of nonzero entries) of the factor for general sparse matrices. This is because the computation and fill-in during the factorization of a sparse matrix is a function of the the number and position of nonzeros in the original matrix. In the context of the important class of sparse matrices that are adjacency matrices of graphs whose n-node subgraphs have $O(\sqrt{n})$-node separators (this class includes sparse matrices arising out of all two-dimensional finite difference and finite element problems), the contribution of this work can be summarized by Figure 1. A simple fan-out algorithm [8] with column-wise partitioning of an $N \times N$ matrix of this type on p processors results in an $O(Np \log N)$ total communication volume [10] (box A). The communication volume of the column-based schemes represented in box A has been improved using smarter ways of mapping the matrix columns onto processors, such as, the subtree-to-subcube mapping [8] (box B). A number of column-based parallel factorization algorithms [27, 4, 8, 7, 13, 34] have a lower bound of $O(Np)$ on the total communication volume. Since the overall computation is only $O(N^{1.5})$ [9], the ratio of communication to computation of column-based schemes is quite high. As a result, these column-cased schemes scale very poorly as the number of processors is increased [34, 33]. A few schemes with two-dimensional partitioning of the matrix have been proposed [33], and the total communication volume in the best of these schemes [33] is $O(N\sqrt{p}\log p)$ (box C).

In summary, the simple parallel algorithm with $O(Np\log p)$ communication volume (box A) has been improved along two directions—one by improving

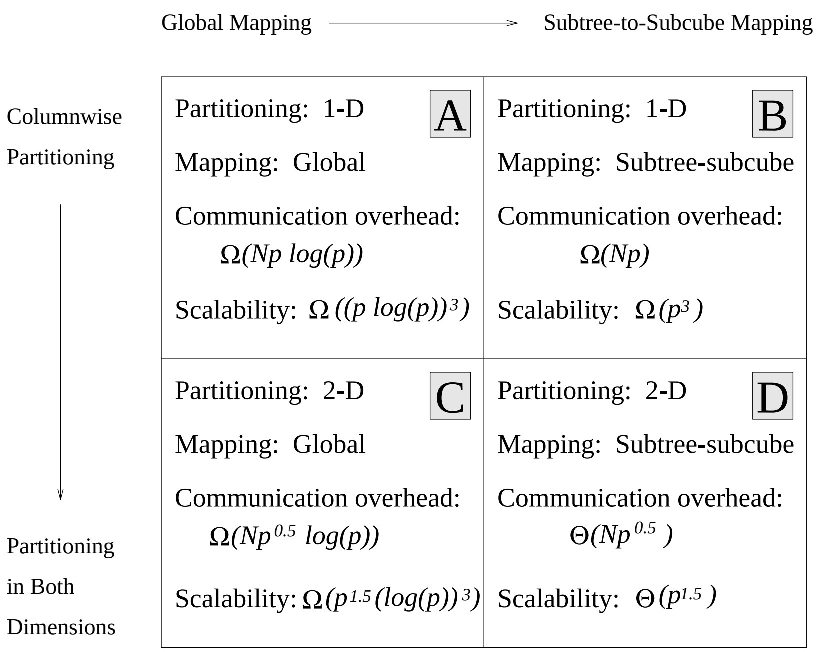

Figure 1 An overview of the performance and scalability of parallel algorithms for factorization of sparse matrices resulting from two-dimensional N-node grid graphs. Box D represents our algorithm, which is a significant improvement over other known classes of algorithms for this problem.

the mapping of matrix columns onto processors (box B) and the other by The algorithm in [12] combines the benefits of improvements along both these lines. The total communication overhead of our algorithm is only $O(N\sqrt{p})$ for factoring an $N \times N$ matrix on p processors if it corresponds to a graph that satisfies the separator criterion. Our algorithm reduces the communication overhead by a factor of at least $O(\log p)$ over the best algorithm implemented to date. It is also significantly simpler in concept as well as in implementation, which helps in keeping the constant factors associated with the overhead term low. Furthermore, this reduction in communication overhead by a factor of $O(\log p)$ results in an improvement in the scalability of the algorithm by a factor of $O((\log p)^3)$; i.e., the rate at which the problem size must increase with the number of processors to maintain a constant efficiency is lower by a factor of $O((\log p)^3)$. This can make the difference between the feasibility and non-feasibility of parallel sparse factorization on highly parallel ($p \geq 256$) computers.

The algorithm presented in [12] is based on the multifrontal principle [5, 26] and the computation is guided by an elimination tree. Independent subtrees of the elimination tree are initially assigned to individual processors. As the computation proceeds up the elimination tree, through pairwise exchanges of data, the matrix is progressively split both horizontally and vertically among the processors resulting in a two-dimensional partitioning. This strategy keeps the communication low in the initial stage of parallel factorization. In the later part, the increase in communication is balanced by a corresponding increase in computation per column, and at the same time the processors do not starve due to narrowing of the elimination tree.

Although, asymptotically scalable, subtree-to-subcube mapping often suffers from significant load-imbalance. In [15], we describe a new elimination tree mapping scheme that minimizes this problem. We assign many subtrees (subforest) of the elimination tree to each processor subcube. These trees are chosen in such a way that the total amount of work assigned to each subcube is as equal as possible. The tree partitioning algorithm uses a set Q that contains the unassigned nodes of the elimination tree. The algorithm inserts the root of the elimination tree into Q, and then it recursively partitions the elimination tree. During each recursive level, the set Q is partitioned into two parts, L and R and each part is checked for acceptability. If so, then L is assigned to half of the processors, and R is assigned to the remaining half. If the partitioning is not acceptable, then one node v of Q is selected and is assigned to all the p processors. Node v is deleted from Q, and its children are inserted into Q. The algorithm then continues by repeating the whole process. Since it assigns subforests of the elimination tree to processor subcubes, we refer to it as subforest-to-subcube mapping scheme. The analysis presented in [15] shows that this new mapping scheme does not increase the overall communication overhead, and its scalability is similar to the earlier algorithm.

We implemented our new parallel sparse multifrontal algorithm on a 1024-processor Cray T3D parallel computer. Each processor on the T3D is a 150Mhz Dec Alpha chip, with peak performance of 150MFlops for 64-bit operations (double precision). However, the peak performance of most level three BLAS routines is around 50 Mflops. The processors are interconnected via a three dimensional torus network that has a peak unidirectional bandwidth of 150Bytes per second, and a very small latency. Even though the memory on T3D is physically distributed, it can be addressed globally. That is, processors can directly access (read and/or write) other processor's memory. T3D provides a library interface to this capability called SHMEM. We used SHMEM to develop a lightweight message passing system. Using this system we were able to achieve unidirectional data transfer rates up to 70Mbytes per second. For the computation performed during the dense Cholesky factorization, we used single-processor implementation of BLAS primitives. These routines are part of the standard scientific library on T3D, and they have been fine tuned for the Alpha chip. The algorithm was tested on matrices from a variety of sources. Four matrices (BCSSTK30, BCSSTK31, BCSSTK32, and BCSSTK33) come from the Boeing-Harwell matrix set. MAROS-R7 is from a linear programming problem taken from NETLIB. COPTER2 comes from a model of a helicopter rotor. CUBE35 is a $35 \times 35 \times 35$ regular three-dimensional grid. NUG15 is from a linear programming problem derived from a quadratic assignment problem obtained from AT&T. In all of our experiments, we used spectral nested dissection [30] to order the matrices.

The performance obtained by our multifrontal algorithm in some of these matrices is shown in 3. The operation count shows only the number of operations required to factor the nodes of the elimination tree.

				Number of Processors			
Problem	n	$\|L\|$	OPC	64	256	512	1024
2pt MAROS-R7	3136	1345241	720M	1.41	3.02	4.07	4.48
FLAP	51537	4192304	940M	1.27	2.87	3.83	4.25
BCSSTK33	8738	2295377	1000M	1.30	2.90	4.36	6.02
BCSSTK30	28924	5796797	2400M	1.48	3.59	5.56	7.54
BCSSTK31	35588	6415883	3100M	1.45	3.97	6.26	7.93
BCSSTK32	44609	8582414	4200M	1.51	4.16	6.91	8.90
COPTER2	55476	12681357	9200M	1.94	5.76	9.55	14.78
CUBE35	42875	11427033	10300M	2.26	6.46	10.33	15.70
NUG15	6330	10771554	29670M		7.54	12.53	19.92

Table 3 The performance of sparse direct factorization on Cray T3D. For each problem the table contains the number of equations n of the matrix A, the original number of nonzeros in A, the nonzeros in the Cholesky factor L, the number of operations required to factor the nodes, and the performance in gigaflops for different number of processors.

On large enough problems, one of our algorithms [15] delivers up to 20 GFLOPS for medium-size structures and linear programming problems on a Cray T3D. To the best of our knowledge, this is the first parallel implementation of sparse Cholesky factorization that has delivered speedups of this magnitude and has been able to benefit from several hundred processors. Our work so far has focussed on Cholesky factorization of symmetric positive definite matrices; however, the same ideas can be adapted for performing Gaussian elimination on diagonally dominant matrices that are almost symmetric in structure [6] and for solving sparse linear least squares problems [28].

REFERENCES

[1] S. Arvindam, Vipin Kumar, and V. Nageshwara Rao. Floorplan optimization on multiprocessors. In *Proceedings of the 1989 International Conference on Computer Design*, 1989. Also published as Technical Report ACT-OODS-241-89, Microelectornics and Computer Corporation, Austin, TX.

[2] S. Arvindam, Vipin Kumar, and V. Nageshwara Rao. Efficient parallel algorithms for search problems: Applications in VLSI CAD. In *Proceedings of the Frontiers 90 Conference on Massively Parallel Computation*, 1990.

[3] S. Arvindam, Vipin Kumar, V. Nageshwara Rao, and Vineet Singh. Automatic test pattern generation on multiprocessors. *Parallel Computing*, 17(12):1323–1342, December 1991.

[4] Cleve Ashcraft, S. C. Eisenstat, J. W.-H. Liu, and A. H. Sherman. A comparison of three column based distributed sparse factorization schemes. Technical Report YALEU/DCS/RR-810, Yale University, New Haven, CT, 1990. Also appears in *Proceedings of the Fifth SIAM Conference on Parallel Processing for Scientific Computing*, 1991.

[5] Iain S. Duff and J. K. Reid. The multifrontal solution of indefinite sparse symmetric linear equations. *ACM Transactions on Mathematical Software*, 9:302–325, 1983.

[6] Iain S. Duff and J. K. Reid. The multifrontal solution of unsymmetric sets of linear equations. *SIAM Journal on Scientific and Statistical Computing*, 5(3):633–641, 1984.

[7] G. A. Geist and E. G.-Y. Ng. Task scheduling for parallel sparse Cholesky factorization. *International Journal of Parallel Programming*, 18(4):291–314, 1989.

[8] A. George, M. T. Heath, J. W.-H. Liu, and E. G.-Y. Ng. Sparse Cholesky factorization on a local memory multiprocessor. *SIAM Journal on Scientific and Statistical Computing*, 9:327–340, 1988.

[9] A. George and J. W.-H. Liu. *Computer Solution of Large Sparse Positive Definite Systems*. Prentice-Hall, Englewood Cliffs, NJ, 1981.

[10] A. George, J. W.-H. Liu, and E. G.-Y. Ng. Communication results for parallel sparse Cholesky factorization on a hypercube. *Parallel Computing*, 10(3):287–298, May 1989.

[11] Ananth Grama, Vipin Kumar, and V. Nageshwara Rao. Experimental evaluation of load balancing techniques for the hypercube. In *Proceedings of the Parallel Computing '91 Conference*, pages 497–514, 1991.

[12] Anshul Gupta and Vipin Kumar. A scalable parallel algorithm for sparse matrix factorization. Technical Report 94-19, Department of Computer Science, University of Minnesota, Minneapolis, MN, 1994. A short version appears in *Supercomputing '94 Proceedings*. TR available in *users/kumar* at anonymous FTP site *ftp.cs.umn.edu*.

[13] M. T. Heath, E. G.-Y. Ng, and Barry W. Peyton. Parallel algorithms for sparse linear systems. *SIAM Review*, 33:420–460, 1991. Also appears in K. A. Gallivan et al. *Parallel Algorithms for Matrix Computations*. SIAM, Philadelphia, PA, 1990.

[14] George Karypis and Vipin Kumar. Unstructured Tree Search on SIMD Parallel Computers. Technical Report 92–21, Computer Science Department, University of Minnesota, 1992. Appears in IEEE Transactions on Parallel and Distributed Systems, Volume 5, Number 10, pp. 1057-1072, October 1994. A short version appears in *Supercomputing '92 Proceedings*, pages 453–462, 1992. Available via anonymous ftp from ftp.cs.umn.edu at users/kumar/lb-SIMD.ps.

[15] George Karypis and Vipin Kumar. A high performance sparse Cholesky factorization algorithm for scalable parallel computers. Technical Report TR 94-41, Department of Computer Science, University of Minnesota, Minneapolis, MN, 1994. Submitted to the *Eighth Symposium on the Frontiers of Massively Parallel Computation*, 1995.

[16] V. Kumar and L. N. Kanal. Parallel branch-and-bound formulations for and/or tree search. *IEEE Transactions Pattern Analysis and Machine Intelligence*, PAMI–6:768–778, 1984.

[17] Vipin Kumar. Depth-first search. In Stuart C. Shapiro, editor, *Encyclopaedia of Artificial Intelligence: Vol 2*, pages 1004–1005. John Wiley and Sons, New York, NY, 1987. Revised version appears in the second edition of the encyclopedia to be published in 1992.

[18] Vipin Kumar, Ananth Grama, Anshul Gupta, and George Karypis. *Introduction to Parallel Computing: Design and Analysis of Algorithms*. Benjamin/Cummings, Redwood City, CA, 1994.

[19] Vipin Kumar, Ananth Grama, and V. Nageshwara Rao. Scalable load balancing techniques for parallel computers. *Journal of Parallel and Distributed Computing*, 22(1):60–79, July 1994. Also available as Technical Report 91-55 (November 1991), Department of Computer Science, University of Minnesota, Minneapolis, MN. Available via anonymous ftp from ftp.cs.umn.edu at users/kumar/lb-MIMD.ps.

[20] Vipin Kumar and Anshul Gupta. Analyzing the scalability of parallel algorithms and architectures: A survey. In *Proceedings of the 1991 International Conference on Supercomputing*, 1991. Also appears in September 1994 issue of JPDC. Available via anonymous ftp from ftp.cs.umn.edu at users/kumar/survey-scalability.ps.

[21] Vipin Kumar and Anshul Gupta. Analyzing scalability of parallel algorithms and architectures. *Journal of Parallel and Distributed Computing (special issue on scalability)*, 22(3):379–391, September 1994. A short version of the paper appears in the Proceedings of the 1991 International Conference on Supercomputing. Available via anonymous ftp from ftp.cs.umn.edu at users/kumar/survey-scalability.ps.

[22] Vipin Kumar, K. Ramesh, and V. Nageshwara Rao. Parallel best-first search of state-space graphs: A summary of results. In *Proceedings of the 1988 National Conference on Artificial Intelligence*, pages 122–126, August 1988.

[23] Vipin Kumar and V. N. Rao. Scalable parallel formulations of depth-first search. In Vipin Kumar, P. S. Gopalakrishnan, and L. N. Kanal, editors, *Parallel Algorithms for Machine Intelligence and Vision*. Springer-Verlag, New York, NY, 1990.

[24] Vipin Kumar and V. Nageshwara Rao. Parallel depth-first search, part II: Analysis. *International Journal of Parallel Programming*, 16 (6):501–519, December 1987.

[25] Vipin Kumar and Vineet Singh. Scalability of Parallel Algorithms for the All-Pairs Shortest Path Problem: A Summary of Results. In *Proceedings of the International Conference on Parallel Processing*, 1990. An extended version appears in *Journal of Parallel and Distributed Processing*, 13:124–138, 1991. Available via anonymous ftp from ftp.cs.umn.edu at users/kumar/shortest-path.ps.

[26] J. W.-H. Liu. The multifrontal method for sparse matrix solution: Theory and practice. Technical Report CS-90-04, York University, Ontario, Canada, 1990. Also appears in *SIAM Review*, 34:82–109, 1992.

[27] Robert F. Lucas, Tom Blank, and Jerome J. Tiemann. A parallel solution method for large sparse systems of equations. *IEEE Transactions on Computer Aided Design*, CAD-6(6):981–991, November 1987.

[28] Pontus Matstoms. *The multifrontal solution of sparse linear least squares problems*. PhD thesis, Department of Mathematics, Linkoping University, S-581 83 Linkoping, Sweden, March 1992.

[29] Dianne P. O'Leary and G. W. Stewart. Assignment and scheduling in parallel matrix factorization. *Linear Algebra and its Applications*, 77:275–299, 1986.

[30] Alex Pothen, H. D. Simon, and K.-P. Liou. Partioning sparce matrices with eigenvectors of graphs. *SIAM Journal of Mathematical Analysis and Applications*, 11(3):430–452, 1990.

[31] V. Nageshwara Rao and V. Kumar. Parallel depth-first search, part I: Implementation. *International Journal of Parallel Programming*, 16 (6):479–499, December 1987.

[32] V. Nageshwara Rao and Vipin Kumar. On the efficicency of parallel backtracking. *IEEE Transactions on Parallel and Distributed Systems*, 4(4):427–437, April 1993. Also available as Technical Report TR 90-55, Department of Computer Science, University of Minnesota, Minneapolis, MN. Available via anonymous ftp from ftp.cs.umn.edu at users/kumar/suplin.ps.

[33] Edward Rothberg. Performance of panel and block approaches to sparse Cholesky factorization on the iPSC/860 and Paragon multicomputers. In *Proceedings of the 1994 Scalable High Performance Computing Conference*, May 1994.

[34] Robert Schreiber. Scalability of sparse direct solvers. Technical Report RIACS TR 92.13, NASA Ames Research Center, Moffet Field, CA, May 1992. Also appears in A. George, John R. Gilbert, and J. W.-H. Liu, editors, *Sparse Matrix Computations: Graph Theory Issues and Algorithms* (An IMA Workshop Volume). Springer-Verlag, New York, NY, 1993.