Independent Set Orderings for Parallel Matrix Factorization by Gaussian Elimination

Michael R. Leuze
Department of Computer Science
Vanderbilt University
Nashville, Tennessee 37235

Abstract

Commonly used matrix ordering techniques are designed to minimize fill, i.e., they are designed to minimize the number of zero elements which become nonzero during matrix factorization by Gaussian elimination. If Gaussian elimination is to be implemented on a parallel machine, however, minimum fill orderings are not necessarily optimal. Rather, the primary concern is to order a matrix so as to minimize the time required to complete its factorization. An ordering heuristic which appears to perform well with respect to parallel factorization time is one based on finding independent sets of vertices in the matrix adjacency graph.

Keywords. Gaussian elimination, matrix ordering techniques, fill miniminzing.

1 Introduction

Many important applications give rise to systems of linear equations for which the coefficient matrices are large, sparse, and symmetric. Techniques for ordering such matrices as part of a solution process using Gaussian elimination or its symmetric variant, Cholesky factorization, have been studied extensively. The primary goal of all widely-used matrix ordering techniques is to minimize *fill*, i.e., to minimize the number of zero elements which become nonzero during the factorization process. For a Gaussian elimination algorithm implemented for a sequential machine, a minimum fill ordering is optimal with respect to execution time, since the amount of work performed during Gaussian elimination is completely determined by the number of nonzero elements encountered. Unfortunately, the problem of finding a minimum fill ordering has been shown to be NP-complete [39]; consequently, ordering algorithms which approximate minimum fill orderings are heuristic in nature.

The situation changes, however, when Gaussian elimination is implemented to run on a parallel machine. The primary concern is no longer to minimize fill, thus minimizing the total number of scalar operations to be performed, but rather to minimize the time required to complete the matrix factorization. It is, therefore, more difficult to judge orderings for parallel Gaussian elimination than to judge orderings for sequential Gaussian elimination. The efficacy of a sequential ordering is determined entirely by the amount of fill occurring during elimination; a parallel ordering must be evaluated relative to the entire system on which Gaussian elimination executes. The way in which a matrix is partitioned, the way in which matrix components are mapped to processes, the way in which processes are mapped to processors, and the processor computation and communication capabilities all have an impact on the total execution time. It is, therefore, important to judge the relative merits of matrix orderings for parallel Gaussian elimination on the basis of the inherent parallelism potentially exploitable by actual multiprocessor systems. To avoid biases introduced by a particular system, Leuze and Saxton [23] compared parallel orderings using a timing function which assumed an independent process for each row of the matrix. A process could, during a single time step, either send its information to one other process or receive information from one other process and update its row. They demonstrated that minimum fill is not, in general, an appropriate goal for parallel matrix orderings in the following way. An example was given of a matrix for which a number of orderings result in no fill during elimination. The parallel timing function was applied to each such ordering. An ordering which did cause fill was then shown to produce a smaller value with respect to the timing function than any of the orderings which caused no fill. Although this timing function is somewhat unrealistic in that it does not consider the cost of computation, it does demonstrate that other factors are at work in the parallel case and must be considered if good matrix ordering techniques for parallel Gaussian elimination are to be developed.

Intuitively, a parallel ordering technique should seek to minimize the total time required for factorization by maximizing the number of nonzero element eliminations which can be performed simultaneously. This idea has led to the development of ordering techniques based on finding independent sets of vertices in the adjacency graph of a matrix.

Section 2 contains background information needed for an understanding of ordering matrices for parallel factorization. This section reviews commonly used ordering techniques, designed to reduce fill, which have been suggested for use prior to parallel Gaussian elimination. In Section 2, the cost of these ordering techniques is also considered, and the question of mapping factorization processes to a multiprocessor system is examined. A set of timing functions used to compare the behavior of parallel Gaussian elimination on matrices ordered in various ways is presented in Section 3. In Section 4, some experiments comparing commonly used approaches to ordering are described. Results of these experiments indicate that

ordering can significantly reduce the execution time of parallel factorization. Independent set ordering techniques are described in Section 5. Section 5 also contains cost and performance analyses of independent set orderings.

2 Background

In [15], George and Liu discuss four stages in the solution process of large sparse symmetric systems of equations:

1. *Ordering*—Find a "good" symmetric permutation P of the coefficient matrix A which reduces the amount of fill which occurs during the factorization process.

2. *Storage Allocation*—Simulate the numerical factorization of PAP^T to determine the structure of its factor L and create a sparse data structure to hold the nonzero elements of L.

3. *Factorization*—Compute the nonzero elements of L using the data structure of step 2.

4. *Triangular Solution*—Solve the triangular systems $Ly = b$ and $L^T z = y$, and then set $x = P^T z$.

The primary emphasis of this paper is on the development of an ordering technique to minimize the cost of parallel factorization.

2.1 Matrix Ordering Techniques for Reducing Fill

Techniques for ordering matrices for factorization by Gaussian elimination have traditionally attempted to minimize fill. This section describes three ordering algorithms which approach the problem of reducing fill in very different ways: the Cuthill-McKee [4] and reverse Cuthill-McKee [9] orderings, the minimum degree algorithm [34,38], and nested dissection [10,14,15].

The Cuthill-McKee algorithm [4] reduces the fill that can occur during elimination by attempting to reduce the profile of a sparse, symmetric matrix; i.e., it attempts to cluster nonzero elements as close to the main diagonal as possible. Fill cannot occur outside the *envelope*, a region near the main diagonal containing all of the nonzero matrix elements. Since the envelope is generally treated as if it is dense, very simple storage schemes can be used. The Cuthill-McKee algorithm first chooses a vertex of the adjacency graph of the matrix to be ordered and numbers it first. All neighbors of vertex 1 are then numbered in increasing order of degree, where degree refers to the number of edges incident upon a vertex. The unnumbered neighbors of vertex 2 are then numbered in increasing order of degree, and so on, until all vertices of the adjacency graph are numbered. This numbering

of the adjacency graph imposes an ordering on the matrix itself. George [9] discovered that reversing the ordering obtained by the Cuthill-McKee algorithm often results in significantly smaller envelopes, and Liu and Sherman [27] later proved that the reverse Cuthill-McKee ordering of a matrix is, indeed, never worse than the Cuthill-McKee ordering with respect to envelope size.

The minimum degree algorithm [34,38] is a greedy heuristic which attempts to reduce the global fill occurring during matrix factorization through a series of local fill reductions. At each step of the algorithm, a row which will result in a small amount of fill is chosen to be numbered next. The minimum degree algorithm is applied to a series of elimination graphs, the first of which is the matrix adjacency graph. A single step of the algorithm consists of choosing the vertex of minimum degree from the current elimination graph, numbering the chosen vertex next, and then producing a new elimination graph by removing the numbered vertex from the current elimination graph and pairwise connecting its remaining neighbors. These induced edges correspond to fill which occurs during the factorization process. This step is repeated until all nodes are numbered. Different minimum degree algorithms result from the application of different tie-breaking rules when the node of minimum degree in the current elimination graph is not unique. A closely related algorithm is the minimum deficiency algorithm [34] which numbers next that node in the current elimination graph which, if chosen, would induce the fewest number of fill edges. In spite of its intuitive appeal, the minimum deficiency algorithm rarely results in much less fill than the minimum degree algorithm. It is, in addition, computationally more complex than the minimum degree algorithm and is, therefore, seldom used in practice.

A third important ordering technique for large, sparse, symmetric matrices is nested dissection [10], a divide-and-conquer algorithm. The nested dissection method has been most frequently applied to matrices arising from finite element problems, such as those matrices which result from rectangular grid discretizations. The basic dissection step involves the choice of a separator, a set of nodes which, when removed from the adjacency graph of a matrix, disconnects the graph into two parts. When ordering the nodes of the adjacency graph, the nodes in one part are numbered first, the nodes in the other part are numbered next, and the nodes of the separator are numbered last. As a result of this numbering, no fill edges will be induced across the separator during the Gaussian elimination process. The two parts of the graph thus represent two independent factorization problems. Recursive application of this basic dissection step results in a technique known as nested dissection. Gilbert [17] has proved that for graphs with n vertices of maximum degree d, there is a nested dissection ordering which produces fill within a factor of $O(d \log n)$ of the minimum fill. Unfortunately, the proof is not constructive and gives no guidance in finding a good ordering.

2.2 Cost of Matrix Ordering Techniques

When implementing an algorithm to solve large sparse systems of equations, the cost of the entire computational process must be considered. It is, for example, unreasonable to apply an ordering heuristic to a matrix if the combined cost of ordering and factorization is greater than the cost of factorization without reordering.

The question of whether it is advantageous to order a matrix prior to factorization has been answered affirmatively by numerous researchers (cf. [4,9,10,27,34,38]), whose collective experience has been that the cost of ordering is very small relative to the cost of factorization. George and Liu state that the "task of finding a 'good' ordering . . . is central to the study of the solution of sparse positive definite systems" [15, p. 5]. For multiprocessor implementations of factorization, the ordering issue is more difficult to address. The need to order will almost certainly still be present in the parallel case, but whether parallel or sequential implementations of ordering algorithms will be more efficient is not clear.

To address this question in part, the complexities of sequential implementations of two algorithms which have been suggested for ordering prior to parallel factorization, the minimum degree algorithm (MIN) and an automatic nested dissection algorithm (NED), were analyzed. A straightforward analysis of the two algorithms is difficult, due to their dynamic nature. The minimum degree algorithm eliminates vertices and adds edges to a graph as it executes, corresponding to fill which occurs during factorization. The automatic nested dissection algorithm is dominated by an iterative technique for finding pseudo-peripheral nodes [15] of connected graph components. The minimum degree algorithm can be shown to be worst-case $O(N^3)$, but the worst-case performance of the pseudo-peripheral node algorithm seems to be unknown [31].

Because of the difficulty in analyzing these two ordering algorithms, an empirical study was undertaken to compare their relative performances. To avoid biases introduced by particular implementations, the minimum number of edge visitations (the dominant operation in each algorithm) necessary to perform each step of each algorithm was counted for various size standard $n \times n$ grid problems. Table 1 compares the edge visitations required by the two algorithms. The number of edge visitations, and hence the execution time, of the minimum degree algorithm appears to be proportional to N^2, as is illustrated in Figure 1. The execution time of the automatic nested dissection algorithm, however, appears to be only slightly worse than $O(N \log N)$, which is consistent with the experience of George and Liu [14]. The complexity of the nested dissection algorithm is illustrated in Figure 2. The problem of implementing an ordering algorithm on a multiprocessor has not, in general, been widely addressed, although George et al. [12,13] have indicated that they will do so. Two other studies have examined the problem to a limited extent. First, Peters [32] has suggested

Table 1 Edge Visitations by Matrix Ordering Algorithms			
n	N	NED	MIN
5	25	204	110
10	100	1391	1550
15	225	3838	6777
20	400	8500	20064
25	625	13716	41593
30	900	22900	87997

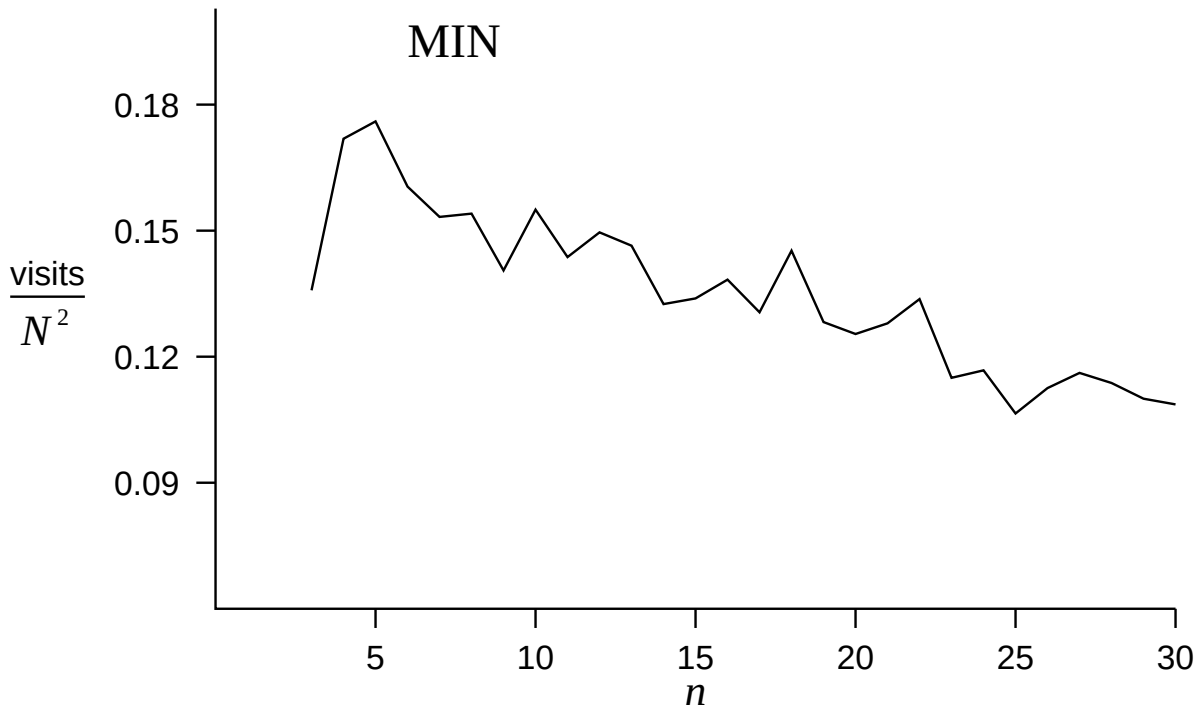

Figure 1: *Complexity of the MIN Algorithm.*

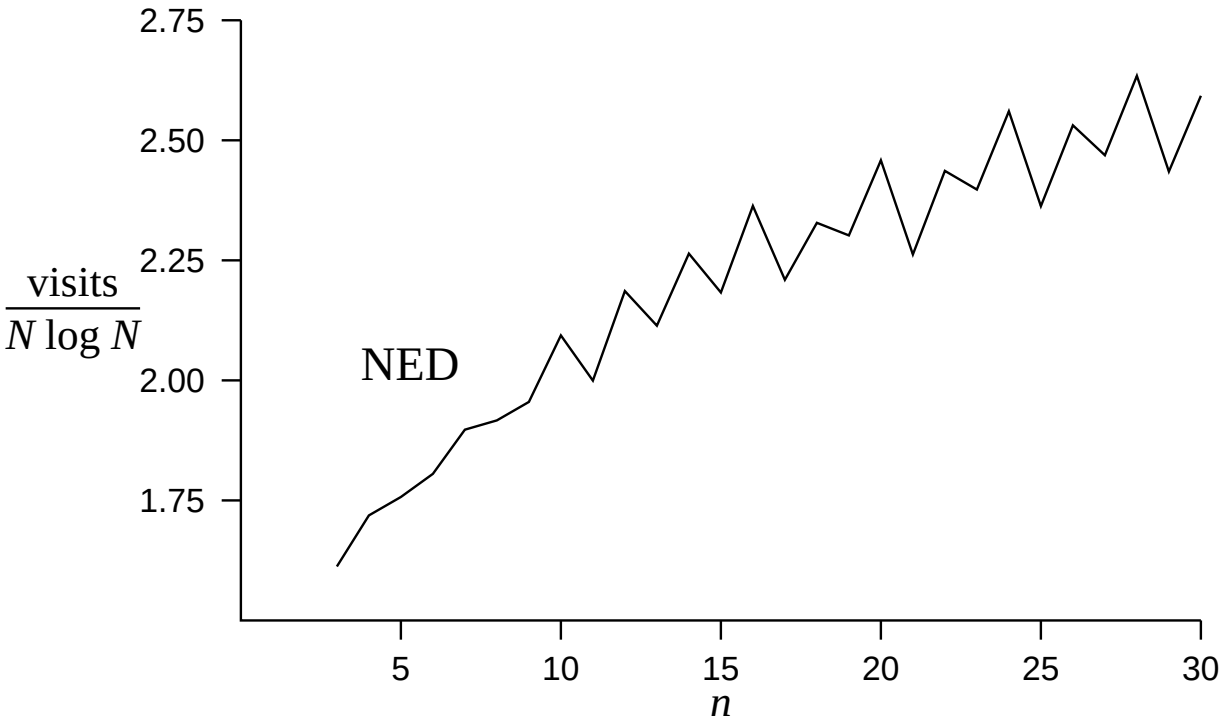

Figure 2: *Complexity of Automatic Nested Dissection.*

parallel algorithms which use a minimum degree heuristic to dynamically select pivots of a large sparse symmetric matrix for processing, thus combining the ordering and factorization phases. Peters' minimum degree program, however, is not a true parallel implementation since it runs on a single processor while all the other processors are involved in factorization. In addition, it appears that Peters' algorithms can be implemented efficiently only on tightly-coupled multiprocessors, since they depend upon access to shared variables. Second, Ng [28] has considered a parallel implementation of nested dissection for a hypercube multiprocessor. The divide-and-conquer nature of nested dissection should be well-suited for execution using successively higher degree subcubes. Ng, however, reported speedups over sequential nested dissection of only two or three, even when 64 processors were used. The general problem of designing and implementing parallel ordering algorithms remains, as George et al. [13] note, a major project and must be considered elsewhere.

2.3 Mapping Factorization Processes to Processors

When applied to a sparse symmetric matrix to be factored in parallel, the goal of an ordering algorithm is to maximize the parallelism which can potentially be exploited by a multiprocessor system. However, unless the factorization processes can be arranged so that interprocess communication is quick and efficient, this potential parallelism may not be re-

alized. Therefore, in a loosely-coupled, local-memory multiprocessor with an incomplete interconnection network, the assignment of processes to processors is an important consideration. (In a tightly-coupled, shared-memory multiprocessor, process placement is of importance only with respect to load-balancing issues. Peters [32] considers this situation.)

Early work on the problem of mapping processes to processors can be divided into three categories based on the techniques used, either graph-theoretic, optimization, or heuristic. Stone [35] developed a graph-theoretic approach based on network flow algorithms (see e.g. [6]) which statically assigns a set of communicating processes in a two-processor system. An enhancement of this technique by Stone [36] and Bokhari [1] allows assignments to be somewhat dynamic; processes may be reassigned at predetermined points in their execution. Extension of this approach to a three-processor system is straightforward [37], but for larger systems, the algorithms quickly become intractable. In addition, this strategy does not necessarily produce balanced assignments.

To address the inadequacies of the network flow approach, Chu et al. [3] discuss formulation of the process assignment problem as an optimization problem to be solved using integer 0–1 programming. Constraints to achieve load balancing, to represent limited storage, and to model real-time processing requirements can be added as appropriate. However, execution times using the integer programming approach can be prohibitive for all but relatively small problems. Consequently, heuristic approaches to the process assignment problem have been developed.

Efe [7] surveys a number of heuristic approaches, most of which are based on clustering algorithms which pair processes with large communication costs for placement on the same processor. Efe suggests a clustering approach which attempts not only to minimize interprocess communication but also to achieve a balanced load among the processors. Unfortunately, all of these heuristic approaches make simplifying assumptions which limit the usefulness of their results. Efe, for example, assumes that the multiprocessor system under consideration consists of identical, fully interconnected processors.

In [2], Bokhari takes a different approach to the process assignment problem. Bokhari begins with a problem graph (in which a node corresponds to a process and an edge represents interprocess communication) and a machine graph (in which a node corresponds to a processor and an edge to a physical link between processors) with equal numbers of nodes. He then searches for mappings which maximize the number of communicating pairs of processes which lie on adjacent processors, subject to the constraint that each processor is assigned exactly one process. Initially, an identity mapping is used to assign processes to processors, followed by pairwise exchanges of processes until a local maximum is reached. Occasional random exchanges are made to prevent premature convergence to a suboptimal mapping. Bokhari calls results from the application of his mapping heuristic to structural

problems with from 9 to 49 nodes "acceptable" but speculates that the algorithm may be unsuitable for larger problems.

Leuze and Schach [24] examined a variation of Bokhari's approach which attempts to minimize the total distance from each process to all processes with which it communicates. Bokhari's criterion, the number of communicating processes which lie on adjacent processors, was felt to be inadequate for finding good mappings, since it does not consider the placement of communicating processes which cannot be placed on adjacent processors. In addition, Bokhari's criterion was felt to be inadequate for evaluating mappings since it does not consider communication-link contention which results when messages between communicating processes on non-adjacent processors must be routed through intermediate nodes. Therefore, to evaluate mappings, a model of parallel Gaussian elimination executing on a hypercube multiprocessor was developed. The model assumes that a Gaussian elimination process responsible for one row of the coefficient matrix resides at each node of the hypercube. During each time step, the model passes as many messages as possible, but at most one per link. If two or more messages attempt to use the same link during the same time step, one message is randomly chosen. In a number of tests using a 256-node problem, the mapping heuristic of Leuze and Schach required between 66% and 80% of the time steps required by the heuristic of Bokhari.

Due to the need to examine all possible pairwise interchanges of processes, the heuristics of Bokhari [2] and Leuze and Schach [24] are unacceptably slow for problems of practical size. Therefore, in [25] Leuze and Schach developed a constructive initial assignment heuristic, i.e., a heuristic in which the processes are mapped to the processors one at a time, after which no change is made in the assignment. The heuristic at each step places the unplaced process with the maximum number of neighbors (other processes with which it communicates) on a processor which minimizes the sum of the distances to previously placed neighbors. The heuristic was used to map each of nine parallel Gaussian elimination problems with 256 processes to the processors of an eighth-degree hypercube. The cost of the constructive initial assignment heuristic was approximately two orders of magnitude less than the cost of Bokhari's heuristic [2] in terms of SUN 3/50 CPU time. In addition, in every case, the initial assignment heuristic outperformed Bokhari's heuristic with respect to the model of Gaussian elimination on a hypercube [24], requiring from 70% to 83% as many time steps. Further work with mapping heuristics is needed to determine effective strategies for missized problems with more processes than processors.

A number of researchers at the Oak Ridge National Laboratory [5,8,11,18,19], at the University of Maryland [29,30], and at Yale University [21] have considered the question of how to map dense matrix problems to multiprocessor systems. O'Leary and Stewart [29] proposed three schemes for mapping a matrix to be factored by Cholesky decomposition to a

rectangular grid of processors. The *block* mapping approach partitions the matrix into blocks and then assigns each processor one block so that adjacent blocks lie on adjacent processors. The *reflection* mapping approach assigns individual matrix elements to processors so that adjacent elements lie on adjacent processors. At an edge of the processor array, the mapping is reflected back across the array. This technique is also known as a fold mapping. The *torus* mapping approach is similar to the reflection approach, except that when an edge of the processor array is reached, the mapping wraps around to the opposite edge. Hence, a torus mapping is also known as a wrap mapping.

It has been the common experience of researchers who have examined the mapping problem for dense matrices that wrap mappings are best at balancing processor loads, whether the processor configuration being studied is a linear array or ring [18,21,30], a rectangular grid or torus [30], or a hypercube (where the processors are ordered consecutively by their binary identifiers [5,8,11] or are ordered using a Gray code [19]). Further work is needed to determine whether a wrap mapping is appropriate for a sparse matrix. Wrap mappings may be effectively utilized for the latter rows of a sparse matrix where corresponding portions of the triangular factors are dense, or nearly dense, and they will certainly be useful in balancing processor loads when solving sparse problems. However, as Geist and Romine [8] note, there are other mappings which are as effective as wrap mappings at load balancing, and some of these mappings may result in more efficient communication among the earlier rows of a sparse matrix ordered by an independent set heuristic. The question of how best to map parallel Gaussian elimination processes to a multiprocessor is beyond the scope of this paper and must be considered elsewhere.

3 Parallel Timing Functions

The characteristics of a multiprocessor system and the mapping function being used introduce biases which make it difficult to assess the relative merits of matrix orderings for parallel factorization on the basis of the performance of a particular implementation of a matrix factorization algorithm. Therefore, criteria designed to judge parallel matrix orderings have been machine and implementation independent; these criteria include the performance of an ordering relative to the parallel timing function of Leuze and Saxton [23] and the parallel depth [17,26] of an ordering. Parallel depth may be defined as the longest path in the filled adjacency graph of the matrix through vertices numbered monotonically increasing. In this section, these two criteria, together with a third, are presented using a common framework as a series of three parallel timing functions.

All three timing functions assume that the coefficient matrix is partitioned by rows and that corresponding to each row is a process with the sole responsibility of maintaining and

updating that row. Timing function 1 is very similar to the parallel timing function of Leuze and Saxton [23], which allows, during a single time step, a process either to send its information to one other process or to receive information from one other process and update its row. The timing functions are illustrated in Figure 4. using the sparse matrix of Figure 3. In Figure 3, nonzeros are denoted by $*$ while fill elements are denoted by

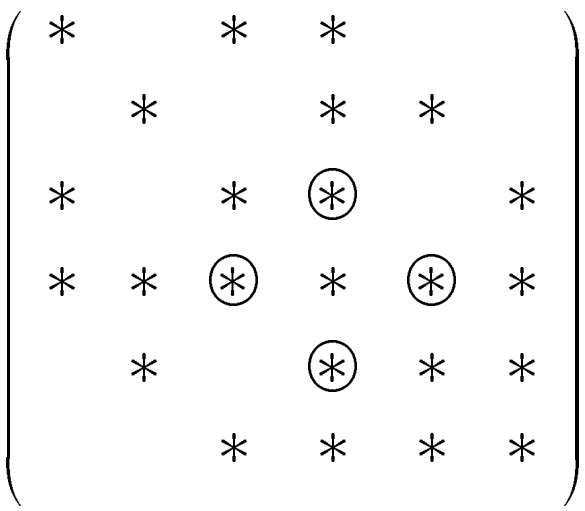

Figure 3: *A Sparse, Symmetric Matrix.*

$\circledast$. In Figure 4, the solid lines represent edges in the matrix adjacency graph and the dashed lines represent fill edges induced during the elimination process. Numbers inside the circles indicate the ordering of the vertices, whereas the numbers associated with the edges represent times at which messages are sent or received. Timing function 1 is illustrated in Figure 4a. Note that messages are received according to the order in which they are sent (for example, process 4 receives a message from process 2 before receiving a message from process 1) and that delays between the sending and the receiving of a message may occur (for example, process 3 sends its information to process 4 during time step 2, but the information is not received until time step 3). The total time required to complete factorization of the 6×6 matrix of Figure 3 is six time units if timing function 1 is used.

Timing function 2 assumes that during a single time step, a process can either send its row information to any number of other processes or receive information from one other process and update its row. From Figure 4b, note that the time required to complete factorization of the matrix is five time units when timing function 2 is used.

The value resulting from the application of timing function 3 to an ordered matrix is equivalent to the parallel depth of the ordering. Timing function 3 assumes that during a

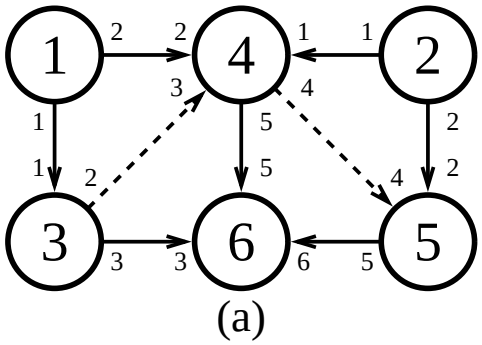

Figure 4: *Parallel Timing Functions.*

single time step, a process can either send a message to any number of other processes or receive a message from any number of other processes and update its row. The use of timing function 3 is illustrated in Figure 4c.

These timing functions are designed to measure the degree of inter-row data dependencies of an ordering. Timing values reflect the inherent parallelism of an ordering, which may or may not be exploitable by a specific machine due to limitations on the number of processors, incomplete interconnection networks, or restrictions on communication capabilities, such as the number of simultaneous sends or receives a processor can initiate. Better comparisons of matrix orderings can be made for specific machines if these limitations are modeled in the timing function.

4 The Effectiveness of Ordering for Parallel Factorization

The timing functions of Section 3 can be used to help answer the question of whether the anticipated costs of parallel factorization are reduced significantly by ordering. These timing functions, which attempt to measure the parallelism which can be exploited during the factorization process, were applied to three 256×256 test matrices that might result from the discretization of a region with a rectangular mesh. Matrix 1 corresponds to a 16×16 rectangular grid, matrix 2 to an 8×32 rectangular grid, and matrix 3 to a 4×64 rectangular grid. In the first test, the three test matrices were unordered (i.e., the ordering was a "natural" one with nodes of the rectangular meshes numbered consecutively from left-to-right, top-to-bottom). The results of this test are summarized in Table 2. An

Table 2 Timing Results for Unordered Matrices

	Average degree of filled graph	Timing function		
		1	2	3
Matrix 1 (16×16)	30.117	480	255	255
Matrix 2 (8×32)	15.555	472	255	255
Matrix 3 (4×64)	7.898	444	255	255

examination of these results indicates that this ordering is among the worst possible for parallel factorization. The parallel depth of each filled adjacency graph (given by timing function 3) is at a maximum. (Timing function 2 gives this same result since its value is bounded above by one less than the number of nodes in the adjacency graph and below by the value of timing function 3.) The timing functions were next applied to ten random orderings of each of the three test matrices. These results are summarized in Table 3.

Table 3 Timing Results for Randomly Ordered Matrices
(means and standard deviations)

	Average degree of filled graph	Timing function		
		1	2	3
Matrix 1 (16×16)	48.419 ± 3.823	233.0 ± 12.229	130.0 ± 4.899	120.3 ± 7.088
Matrix 2 (8×32)	43.620 ± 4.773	216.4 ± 17.328	124.2 ± 7.099	113.3 ± 9.627
Matrix 3 (4×64)	26.085 ± 3.287	158.8 ± 15.047	92.2 ± 7.193	82.9 ± 7.992

Finally, the performances of three ordering heuristics, the reverse Cuthill-McKee method (RCM), the minimum degree algorithm (MIN), and nested dissection (NED), relative to the timing functions were tested. Experimental results appear in Table 4.

Table 4 Timing Results from Matrix Ordering Experiment

	Average degree of filled graph	Timing function		
		1	2	3
16×16 rectangular grid				
RCM	22.188	480	255	255
MIN	14.531	85	60	46
NED	19.102	99	66	45
8×32 rectangular grid				
RCM	14.844	472	255	255
MIN	11.617	103	61	57
NED	16.383	86	59	37
4×64 rectangular grid				
RCM	7.859	444	255	255
MIN	7.359	228	143	142
NED	11.656	58	41	25

Information included in Table 4 concerning the average degree of the filled adjacency graph, and hence the amount of fill occurring in the matrix during Gaussian elimination, clearly demonstrates that amount of fill is not strongly correlated with performance relative to the three timing functions.

From a comparison of Tables 2, 3, and 4, it is obvious that proper ordering of sparse matrices can lead to significant reductions in the cost of parallel factorization. The timings for the matrices when "unordered" are identical to the timings when ordered using the reverse Cuthill-McKee algorithm; this is not unexpected since this "natural" ordering tends to cluster all nonzero elements relatively near the main diagonal, thus inhibiting parallelism. On average, the random orderings were significantly better for parallel factorization than

were the reverse Cuthill-McKee and "natural" orderings but were significantly worse than the nested dissection ordering and significantly worse than the minimum degree ordering for matrices 1 and 2. Over the test cases considered, factorization preceded by ordering with the nested dissection algorithm was completed in 30% to 51% of the time required to factor a randomly ordered matrix. It appears, therefore, that ordering of sparse matrices can significantly increase the inherent parallelism of the factorization process.

The performance of the ordering algorithms was, for the most part, as expected. Profile reduction algorithms in general, and the reverse Cuthill-McKee algorithm in particular, are not well-suited for parallel Gaussian elimination. Since all nonzero elements are clustered near the main diagonal, there is a strong tendency for a given row to depend on information from a number of its immediate predecessors. The parallel depth of a matrix ordered by the reverse Cuthill-McKee method is generally large. Consequently, parallel Gaussian elimination proceeds from the upper left portion of the coefficient matrix to the lower right portion with very little opportunity to exploit parallelism.

The minimum degree algorithm produced a relatively good ordering for matrix 1 but a surprisingly poor ordering for matrix 3. This result is more surprising in light of the fact that incorporated into the minimum degree algorithm was a tie-breaking rule which should have helped to decrease the parallel depth of the ordered matrix. When the vertex of minimum degree in the elimination graph was not unique, ties were broken by choosing that vertex with the smallest number of previously ordered neighbors. Further analysis and experimentation will be required to explain the behavior of, and perhaps to suggest an alternative tie-breaking rule for, the minimum degree algorithm.

Of the three fill-reducing ordering techniques tested, nested dissection performed consistently best. This result was expected, since the divide-and-conquer nature of nested dissection produces a matrix in which initially a number of smaller factorization problems can be solved independently.

5 Independent Set Orderings

5.1 Description

Matrix orderings which are well-suited for parallel Gaussian elimination should have a low degree of inter-row data dependencies. Parallel depth should be as small as possible. An intuitively appealing technique for producing matrix orderings with these characteristics is to maximize the number of independent operations that can be performed simultaneously. It is, therefore, the goal of independent set orderings to arrange a coefficient matrix in such a way as to permit the simultaneous execution of as many operations as possible during

parallel Gaussian elimination.

An independent set ordering algorithm is applied to a series of elimination graphs, the first of which is the adjacency graph of the matrix of interest. Each stage of the algorithm consists of first finding a maximal independent set of vertices in the current elimination graph. A set of vertices $S \subseteq V$ in a graph $G = (V, E)$ is said to be *independent* if no pair of vertices is adjacent in G. S is said to be a *maximal* independent set if it is not contained in a larger independent set; see e.g. [33]. A *maximum* independent set is an independent set of greatest cardinality. Since Karp [22] has shown the *maximum* independent set problem to be NP-hard, heuristics for finding large *maximal* independent sets must be used. Once an independent set is found in the current elimination graph, its vertices are numbered next in an arbitrary fashion, and each newly numbered vertex is then removed from the elimination graph by pairwise connecting each of its neighbors, thus forming a new elimination graph. These induced edges correspond to fill which occurs during the elimination process. Vertices can be removed in a completely arbitrary order, since they form an independent set. This process continues until all vertices of the adjacency graph are numbered.

Variations of the independent set ordering algorithm result from the use of different techniques for finding a maximal independent set of vertices in an elimination graph; two techniques for finding maximal independent sets are considered in this work. The first approach is a greedy algorithm. Initially all vertices of the elimination graph are unmarked. An unmarked node is selected to be added to the independent set after which it and all of its neighbors are marked. This selection process continues until all vertices of the elimination graph are marked. The vertices selected in this manner form a maximal independent set. A second approach is to find a vertex cover for the elimination graph, i.e., a set of vertices such that every edge in the graph is incident upon at least one vertex in the set; the complement of a minimal vertex cover is a maximal independent set. To find the largest possible maximal independent set, the vertex cover should be as small as possible. (Empirical evidence suggests that it is easier to find a small vertex cover than it is to find a large independent set.) The following vertex cover complement algorithm can be used. A temporary graph is initially equivalent to the current elimination graph. The vertex of maximum degree in the temporary graph is found, and the corresponding vertex in the elimination graph is marked. This vertex and all of its incident edges are then removed from the temporary graph. The process of finding and removing the vertex of maximum degree continues until the temporary graph contains no edges. The unmarked nodes in the elimination graph then form a maximal independent set.

5.2 Cost of Independent Set Orderings

An analysis of the complexity of the independent set algorithm using a vertex cover complement approach (IVC) similar to the analyses of the minimum degree and automatic nested dissection algorithms in Section 2.2 was performed. The independent set algorithm, like the minimum degree algorithm, eliminates vertices and adds edges to a graph as it executes, corresponding to fill which occurs during factorization. As before, the minimum number of edge visitations necessary to perform each step of the independent set algorithm was counted for various size standard $n \times n$ grid problems. Table 5 reports the edge visitations required by the algorithm. As was the case for the minimum degree algorithm, the number of edge

Table 5 Edge Visitations by the Independent Set Algorithm

n	N	IVC
5	25	246
10	100	3546
15	225	13743
20	400	43238
25	625	105042
30	900	186383

visitations, and hence the execution time, of the independent set algorithm appears to be proportional to N^2, as is illustrated in Figure 5. Although the independent set algorithm is the most costly of the three ordering algorithms considered, its order of complexity appears to be the same as that of the minimum degree algorithm. This is not surprising, since both algorithms involve the selection of vertices which are then removed from an elimination graph by deleting all incident edges after pairwise connecting all adjacent vertices. Because of the similarity of the two algorithms, it is hoped that many of the improvement techniques which have been applied to the minimum degree algorithm [16], resulting in an overall reduction of ordering time by more than a factor of 50, will be applicable to the independent set ordering algorithm.

5.3 Performance of Independent Set Orderings

The timing functions of Section 3 were applied to the three test matrices of Section 4 ordered by two different independent set methods, one using a greedy algorithm to find maximal independent sets (IGR), the other using a vertex cover complement algorithm (IVC). Experimental results appear in Table 6.

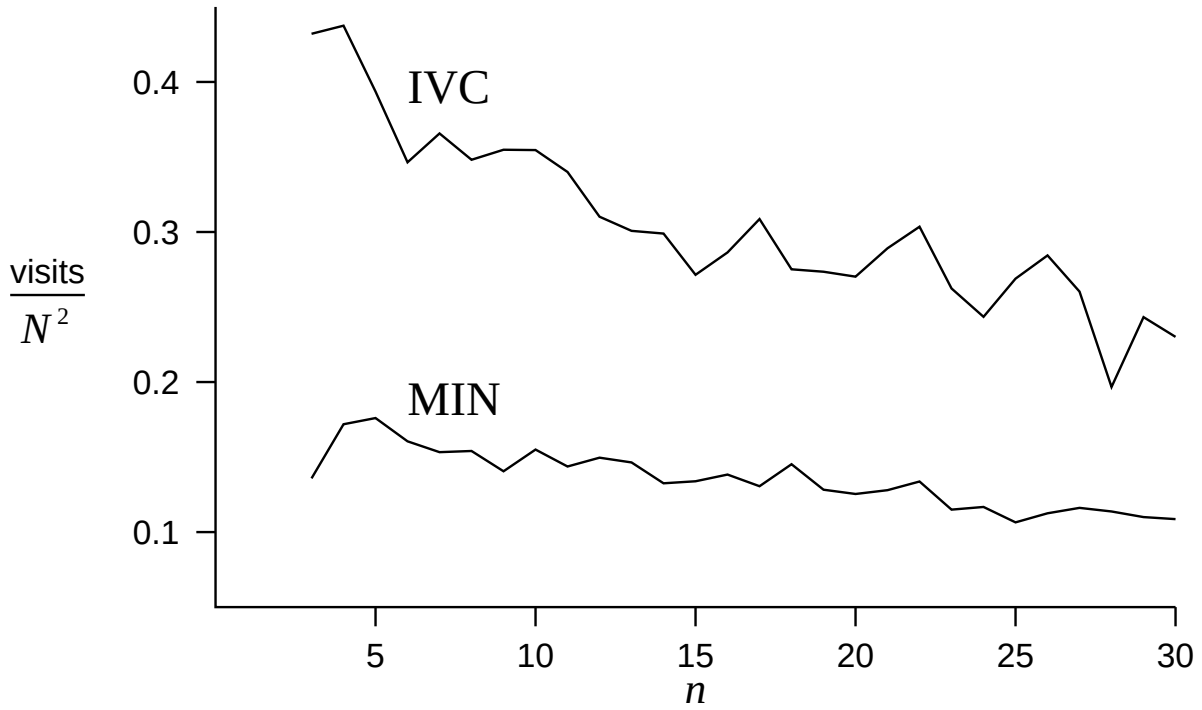

Figure 5: *Complexity of the MIN and IVC Algorithms.*

Table 6 Timing Results from Matrix Ordering Experiment

	Average degree of filled graph	Timing function 1	2	3
16×16 rectangular grid				
IGR	21.594	126	81	57
IVC	14.648	84	59	36
8×32 rectangular grid				
IGR	17.141	102	66	46
IVC	13.383	74	55	32
4×64 rectangular grid				
IGR	10.648	57	40	20
IVC	10.008	50	37	21

As expected, the independent set algorithms performed well relative to the ordering algorithms designed to minimize fill. In almost all cases, the only exception being the application of timing function 3 to orderings of matrix 3, the independent set ordering using the vertex cover complement algorithm gave better results than the independent set ordering using the greedy algorithm. This finding would tend to indicate that an appropriate ordering strategy is to find the largest possible maximal independent set of vertices in each elimination graph, since the vertex cover complement algorithm generally finds larger independent sets than the greedy algorithm. It is not clear, however, that this is an optimal strategy since the ultimate goal of a parallel ordering technique is not maximization of the width of the factorization precedence graph (number of operations that can be performed in parallel) but rather minimization of the depth of the precedence graph (parallel depth). Choosing a smaller maximal independent set, or even an independent set that is not maximal, may simplify the task of finding independent sets in subsequent elimination graphs and thus reduce the total number of elimination graphs (parallel depth) required by the ordering process.

6 Conclusions

The primary goal of the majority of widely-used matrix ordering heuristics is to minimize the fill that occurs during Gaussian elimination. Minimum fill orderings, however, are not necessarily optimal for Gaussian elimination implemented for a parallel machine. The problem of finding good orderings for parallel factorization has not been addressed sufficiently. Independent set orderings are designed to arrange a matrix in such a way as to maximize the number of elimination operations that can be performed simultaneously, thus, it is hoped, decreasing the total time required for parallel factorization of the matrix.

Preliminary tests of independent set orderings are encouraging. Performance of the independent set ordering using the vertex cover complement algorithm was superior to that of nested dissection, the leading contender for producing parallel orderings from among the fill-reducing heuristics, for all three parallel timing functions applied to all three test matrices.

In addition to the problem of mapping, further work is needed in a number of areas related to independent set orderings. The performance of independent set ordering techniques relative to the performance of other ordering heuristics, particularly nested dissection, should be tested using more general matrices of practical importance. The question of optimal size for maximal independent sets should also be addressed, since the ultimate goal of independent set orderings is not simply to find the maximum number of operations that can be performed in parallel but rather to minimize the total number of independent sets that must be found

during the ordering process. Finally, the best way to order vertices within an independent set should be explored. The techniques discussed in Section 5 assume an arbitrary ordering. While vertex numbering within an independent set has no effect on timing function 3, it can influence the total number of time steps required for elimination if either timing function 1 or timing function 2 is used.

Acknowledgements

The author gratefully acknowledges the contributions of Jeremy P. Spinrad and Stephen R. Schach to this work.

References

[1] Shahid H. Bokhari. Dynamic processor scheduling with dynamic reassignment. *IEEE Trans. on Software Engineering*, SE-5(4):341–349, July 1979.

[2] Shahid H. Bokhari. On the mapping problem. *IEEE Trans. on Computers*, C-30(3):207–214, March 1981.

[3] Wesley W. Chu, Leslie J. Holloway, Min-Tsung Lau, and Kemal Efe. Task allocation in distributed data processing. *Computer*, 13(11):57–69, November 1980.

[4] E. Cuthill and J. McKee. Reducing the bandwidth of sparse symmetric matrices. In *Proceedings of the 24th National Conference of the Association for Computing Machinery*, pages 157–172. ACM Publications, 1969.

[5] George J. Davis. Column LU factorization with pivoting on a hypercube multiprocessor. Technical Report ORNL–6219, Oak Ridge National Laboratory, Oak Ridge, TN, 1985.

[6] Jack Edmonds and Richard M. Karp. Theoretical improvements in algorithmic efficiency for network flow problems. *J. ACM*, 19(2):248–264, April 1972.

[7] Kemal Efe. Heuristic models of task assignment scheduling in distributed systems. *Computer*, 15(6):50–56, June 1982.

[8] G. A. Geist and C. H. Romine. LU factorization algorithms on distributed memory multiprocessor architectures. Technical Report ORNL/TM–10383, Oak Ridge National Laboratory, Oak Ridge, TN, 1987.

[9] Alan George. Computer implementation of the finite element method. Technical Report STAN–CS–208, Stanford University, 1971.

[10] Alan George. Nested dissection of a regular finite element mesh. *SIAM Journal of Numerical Analysis*, 10:345–363, 1973.

[11] Alan George and Eleanor Chu. Gaussian elimination with partial pivoting and load balancing on a multiprocessor. Technical Report ORNL/TM–10323, Oak Ridge National Laboratory, Oak Ridge, TN, 1987.

[12] Alan George, Michael T. Heath, Joseph Liu, and Esmond Ng. Sparse Cholesky factorization on a local-memory multiprocessor. Technical Report ORNL/TM–9962, Oak Ridge National Laboratory, Oak Ridge, TN, 1986.

[13] Alan George, Michael T. Heath, Joseph Liu, and Esmond Ng. Solution of sparse positive definite systems on a shared-memory multiprocessor. Technical Report ORNL/TM–10260, Oak Ridge National Laboratory, Oak Ridge, TN, 1987.

[14] Alan George and Joseph W.-H. Liu. An automatic nested dissection algorithm for irregular finite element problems. *SIAM Journal of Numerical Analysis*, 15:1053–1069, 1978.

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

[16] Alan George and Joseph W.-H. Liu. The evolution of the minimum degree algorithm. Technical Report ORNL/TM–10452, Oak Ridge National Laboratory, Oak Ridge, TN, 1987.

[17] John R. Gilbert. Some nested dissection order is nearly optimal. Technical Report 86–767, Department of Computer Science, Cornell University, 1986.

[18] Michael T. Heath. Parallel Cholesky factorization in message passing multiprocessor environments. Technical Report ORNL–6150, Oak Ridge National Laboratory, Oak Ridge, TN, 1985.

[19] Michael T. Heath and C. H. Romine. Parallel solution of triangular systems on distributed memory multiprocessors. Technical Report ORNL/TM–10384, Oak Ridge National Laboratory, Oak Ridge, TN, 1987.

[20] John W. Huang and Omar Wing. Optimal parallel triangularization of a sparse matrix. *IEEE Trans. on Circuits and Systems*, CAS-26:726–732, September 1979.

[21] Ilse C. F. Ipsen, Youcef Saad, and Martin H. Schultz. Complexity of dense-linear-system solution on a multiprocessor ring. *Lin. Algebra Appl.*, 77:205–239, 1986.

[22] R. M. Karp. Reducibility among combinatorial problems. In R. E. Miller and J. W. Thatcher, editors, *Complexity of Computer Computations*, pages 85–104. Plenum Press, New York, 1972.

[23] Michael R. Leuze and Lawrence V. Saxton. On minimum parallel computing times for Gaussian elimination. *Congressus Numerantium*, 40:169–179, 1983.

[24] Michael R. Leuze and Stephen R. Schach. On the placement of parallel processes. Technical Report 86–13, Department of Computer Science, Vanderbilt University, Nashville, TN, 1986.

[25] Michael R. Leuze and Stephen R. Schach. A constructive initial assignment algorithm for mapping parallel Gaussian elimination processes to a hypercube multiprocessor. Technical Report 87–3, Department of Computer Science, Vanderbilt University, Nashville, TN, 1987.

[26] Joseph W. H. Liu. Computational models and task scheduling for parallel sparse Cholesky factorization. *Parallel Computing*, 3:327–342, 1986.

[27] Joseph W. H. Liu and Andrew H. Sherman. Comparative analysis of the Cuthill-McKee and the reverse Cuthill-McKee ordering algorithms for sparse matrices. *SIAM Journal of Numerical Analysis*, 13:198–213, 1975.

[28] Esmond Ng. personal communication, 1987.

[29] Dianne P. O'Leary and G. W. Stewart. Data-flow algorithms for parallel matrix computations. *Comm. ACM*, 28(8):840–853, August 1985.

[30] Dianne P. O'Leary and G. W. Stewart. Assignment and scheduling in parallel matrix factorization. *Lin. Algebra Appl.*, 77:275–299, 1986.

[31] Jan K. Pachl. Finding pseudoperipheral nodes in graphs. *J. Computer and System Sciences*, 29:48–53, 1984.

[32] Frans J. Peters. Parallel pivoting algorithms for sparse symmetric matrices. *Parallel Computing*, 1:99–110, 1984.

[33] Edward M. Reingold, Jurg Nievergelt, and Narsingh Deo. *Combinational Algorithms*. Prentice-Hall, Englewood Cliffs, NJ, 1977.

[34] D. J. Rose. A graph-theoretic study of the numerical solution of sparse positive definite systems of linear equations. In R. C. Read, editor, *Graph Theory and Computing*. Academic Press, New York, 1972.

[35] Harold S. Stone. Multiprocessor scheduling with the aid of network flow algorithms. *IEEE Trans. on Software Engineering*, SE-3(1):85–93, January 1977.

[36] Harold S. Stone. Critical load factors in two-processor distributed systems. *IEEE Trans. on Software Engineering*, SE-4(3):254–258, May 1978.

[37] Harold S. Stone and Shahid H. Bokhari. Control of distributed processes. *Computer*, 11(7):97–106, July 1978.

[38] W. F. Tinney. Comments on using sparsity techniques for power system problems. In *Sparse Matrix Proceedings, IBM Research Report RAI 3–12–69*, 1969.

[39] Mihalis Yannakakis. Computing the minimum fill-in is NP-complete. *SIAM Journal on Algebraic and Discrete Methods*, 2:77–79, 1981.