Tradeoffs Between Parallelism and Fill in Nested Dissection

Claudson F. Bornstein[1]
Bruce M. Maggs[2]
Gary L. Miller[3]

Abstract

In this paper we demonstrate that parallelism and fill can be traded off in orders for Gaussian elimination. While the well-known nested dissection algorithm produces very parallel elimination orders, we show that by reducing the parallelism it is possible to reduce the fill that the orders generate. In particular, we present a new "less parallel nested dissection" algorithm (LPND). We prove that, unlike standard nested dissection, when applied to a chordal graph LPND finds a zero-fill elimination order. Our implementation of LPND generates less fill than state-of-the-art implementations of the nested dissection (METIS), minimum-degree (AMD), and hybrid (BEND) algorithms on a large body of test matrices, at the cost of a small reduction in the paralellism in the orders that it produces. We have also implemented a nested dissection algorithm that is different from METIS and that uses the same separator algorithm used by our implementation of LPND. This algorithm, like LPND, generates less fill than METIS, and on large graphs generates significantly less fill than AMD. The latter comparison is notable, because although it is known that, for certain classes of graphs, minimum-degree produces asymptotically more fill than nested dissection, minimum-degree is believed to produce low-fill orderings in practice. Our experiments contradict this belief.

[1]Universidade Federal do Rio de Janeiro, (claudson@acd.ufrj.br). Claudson was supported in part by NSF Grant CCR-9505472.

[2]School of Computer Science, Carnegie Mellon University, and Akamai Technologies, Inc. Bruce Maggs is supported in part by the Air Force Materiel Command (AFMC) and DARPA under Contracts F19628-93-C-0193 and F19628-96-C-0061, by DARPA Contract N00014-95-1-1246, and by an NSF National Young Investigator Award, No. CCR-94-57766, with matching funds provided by NEC Research Institute and Sun Microsystems.

[3]Department of Computer Science, Carnegie Mellon University, (glmiller@cs.cmu.edu). Gary Miller is supported in part by NSF Grants CCR-9505472 and CCR-9706572.

1 Introduction

Among the most common problems to be solved in both operations research and scientific computing is that of finding a vector x that satisfies a linear system of equations $Ax = b$, where A is a known matrix and b is a known vector. Systems of this sort arise in scientific computing when modeling physical systems, such as the flow of air over an airfoil, and in operations research when solving optimization problems, such as assigning production tasks to machines in a workshop. Often these systems are quite large, making the task of solving them quickly on even the most powerful computers challenging. Two common approaches to solving linear systems in practice are direct methods such as Gaussian elimination, and iterative methods such as conjugate gradient. This work focuses on Gaussian elimination.

The basic step in Gaussian elimination is to subtract a multiple of one row of the matrix A from another row of A. In particular, in order to eliminate entry A_{ji}, a multiple of the ith row of A is subtracted from the jth row of A. Hence, the entries in row i in all columns k that have non-zero entries are subtracted from the corresponding entries in the jth row. When the entry A_{ik} is non-zero, but the entry A_{jk} is zero, we say that *fill* occurs, because the new value of A_{jk} is no longer zero. In general, fill is considered to be undesirable because it increases the amount of memory required to store the linear system and because there is a strong correlation between the amount of fill and the total number of floating point operations (the *work*) performed in completing the elimination. Minimizing memory usage is crucial because in practice the size of the largest system that can feasibly be solved is typically limited by the size of a computer's main memory. Unfortunately, the process of elimination may turn a sparse matrix A into a dense matrix U.

The solution to the system of equations $Ax = b$ does not depend on the order in which the rows of A (and hence b) appear. Hence the rows of A and b can be permuted, and then Gaussian elimination can be applied to the permuted system rather than to $Ax = b$. The permutation specifies the order in which the original rows of the matrix A are to be processed, and different orders may create wildly different amounts of fill and work. Minimizing either the fill or the work by performing symmetric permutations of symmetric matrices, that is, by re-ordering the rows and columns of the matrix, however, has been shown to be NP-hard [28].

Gaussian elimination admits several forms of parallelism. The operation of eliminating all entries A_{ji}, for $j > i$, is called pivoting on A_{ii}. Each entry A_{ji} can be eliminated in parallel, and throughout this paper we assume that within a pivoting step, this parallelism is exploited. Furthermore, if two rows are independent, i.e., if $A_{ik} = A_{ki} = 0$, then the operation of pivoting on A_{ii} and A_{kk}

simultaneously is well-defined and can be performed in parallel. It is this form of parallelism that we trade off for fill and work in this paper.

1.1 Our results

In order to evaluate the contribution of this paper, it is helpful to make a distinction between two separate parallel algorithmic issues: the first is the parallelism in the algorithm that finds an elimination order, while the second is the parallelism inherent in the order itself. This paper focuses on the second issue. In particular, we study tradeoffs between parallelism and fill in elimination orders.

Note, however, that even an entirely sequential algorithm that produces a parallel elimination order may be useful in a parallel computing context, for performing the elimination is usually far more computationally intensive than finding the ordering. Furthermore, it is not unusual to repeatedly solve the system of equations $Ax = b$ for a fixed matrix A but with different vectors b. In this case the time to compute a parallel order for A may be amortized over the repeated solutions of the equation.

One of the primary goals in designing a parallel algorithm is to find an algorithm that is work-efficient, i.e., an algorithm whose processor-time product (the work) is close to the time required by the fastest sequential algorithm for solving the same problem. In order to achieve this goal, it is not uncommon to sacrifice some parallelism to reduce the work. A similar phenomena arises in the context of elimination orders. In order to achieve orders with near-minimum fill, it is sometimes necessary to sacrifice some parallelism.

In Section 2.2 we modify the nested dissection algorithm to obtain a similar algorithm (LPND) that produces orders that are typically less parallel than the orders nested dissection produces. We show that, unlike nested dissection, when applied to a chordal graph, the new algorithm produces zero fill. While the obtainment of zero-fill orders is a desirable feature of any ordering algorithm it also has one intrinsic disadvantage, namely the fact that zero-fill orders might be far from optimal in terms of the number of parallel elimination steps they require. The ordering algorithm itself is similar to the traditional nested dissection algorithm and lends itself to parallelism.

Although the algorithm is very similar to nested dissection, we cannot yet provide performance guarantees in terms of fill or work for classes of graphs other than chordal graphs.

In Section 3, we describe experiments with an implementation of LPND and other ordering heuristics on popular benchmark matrices. We observe that both LPND and nested dissection perform well on average, in terms of fill and work. Furthermore, all methods that we experimented with are usually significantly better in terms of fill and work than the minimum-degree algorithm on these benchmarks.

For these benchmarks we observe that nested dissection is actually very competitive in terms of fill with the best known orders. With only a small reduction in the parallelism, LPND obtains elimination orders with even less fill.

1.2 Previous work

All of the work discussed in this paper is restricted to *symmetric positive definite matrices*. If a matrix is symmetric positive definite, then during the course of Gaussian elimination, no matter how the variables are ordered, no pivot element A_{ii} will have value zero. Symmetric positive definite matrices are not uncommon in practice. Furthermore, there are a variety of approaches for dealing with a zero-valued pivot, including exchanging the row with another row, or replacing the pivot value with a small non-zero value (and later adjusting the final solution). We also ignore other numerical stability issues, thus allowing the matrix A to be viewed as a graph. An $n \times n$ symmetric matrix A can be viewed as an undirected graph consisting of n vertices, $\{1, \ldots, n\}$, with an edge between vertices i and j if the matrix entry $A_{ij} = A_{ji}$ is non-zero. The operation of pivoting on A_{ii} then corresponds to the graph operation of removing vertex i from the graph and adding edges between all of the neighbors of i so that they form a clique [23, 25]. To see why such edges must be added, note that in pivoting on A_{ii}, a non-zero entry may be created at A_{jk} any j and k such that $A_{ji} \geq 0$ and $A_{ik} \geq 0$, i.e., for any pair of neighbors, j and k, of i. The newly added edges are called *fill edges* (or just *fill*). Also, note that the pivoting operation preserves the symmetry of the submatrix representing the vertices that have not yet been eliminated. Throughout this paper we view matrices as graphs and use this view to describe the various algorithms.

Over the years a number of algorithms for producing elimination orders with low fill and work have been proposed. The most popular of these are *minimum degree* [19, 21, 27] and *nested dissection* [9, 10, 18]. At each step, the minimum-degree heuristic selects a vertex with the smallest *degree* (number of neighbors), removes this vertex from the graph by pivoting on it (making its neighborhood a clique), and then looks for the next vertex with the smallest degree in the new graph. It proceeds in this fashion until all vertices have been selected. The intuition behind this heuristic is that if a node has small degree, then the clique formed on its neighbors will be small. The orders produced typically create low fill and work. There exist, however, minimum-degree orders with suboptimal amounts of fill. For example, minimum degree can produce orders with $\Omega(n^{\log_3 4})$ fill for the toroidal $n \times n$ mesh [6], while the optimal order for the same graph requires only $\Theta(n \cdot \log n)$ fill.

Nested dissection, on the other hand, examines the graph as a whole before ordering it. Unlike minimum degree, nested dissection orders the vertices of the graph backwards, that is, it begins by deciding which vertices should be eliminated last. Nested dissection works by selecting a *balanced separator*, i.e., a set of vertices that, when removed from the graph, partitions the graph into two or more connected components, each of which has at most a constant fraction of the total number of vertices in the graph. The vertices in the separator are placed last in the elimination order. Then nested dissection recursively orders each of the connected components until the whole graph has been ordered. Because the separator is eliminated last, and there are no edges connecting vertices in different connected components, no fill can be created between different components as they are eliminated. Hence, the various components can be eliminated either sequentially or in parallel without affecting the quality of the order in terms of both fill and work. A small separator is desirable because when the connected components are eliminated they typically turn the separator into a clique. Choosing balanced separators reduces the number of parallel stages, as discussed below. For planar graphs and graphs with bounded genus [11, 18], and for graphs with bounded degree [1], nested dissection has been shown to produce orders that have fill within a poly-logarithmic factor of the optimum.

Even though minimum-degree algorithms can produce elimination orders that introduce more fill in the worst-case than nested dissection, minimum degree is usually preferred to nested dissection in practice, and is said to produce orders with less fill. However, the “in practice” wisdom is changing. The currently accepted champion algorithms are hybrids of nested dissection and minimum degree [3, 14] and benefit from the strengths of both methods but do not, however, provide any provable performance guarantees.

Most implementations of nested dissection actually use a minimum-degree heuristic to order small subgraphs. In particular, the recursive nested dissection ordering is stopped when the subgraph to be ordered is smaller than some fixed size. The small subgraphs are then ordered using the minimum-degree heuristic. This optimization significantly reduces the running times of the nested dissection codes, without generating more work or fill. BEND [14], one of the hybrid algorithms we examined, works in a similar fashion. It begins by identifying and removing separators to decompose the graph into small subgraphs, just as nested dissection does. It considers any subgraph with less than $n/32$ vertices (where n is the total number of vertices in the graph being ordered) to be "small enough". It orders the vertices within these subgraphs using the minimum-degree heuristic. However, unlike regular nested dissection, BEND then merges the vertices in all of the separators that it has found and re-orders them by applying the minimum-degree heuristic to the graph that remains after the elimination of the small subgraphs.

1.3 Parallel elimination

The performance of Gaussian elimination on parallel computers can be improved by generalizing the basic elimination step to allow for more than one vertex to be eliminated at a time. In order for such an operation to make sense, the vertices that are eliminated in one step must form an *independent set*, i.e., a set of vertices no two of which are adjacent. As an example, in the nested dissection algorithm a set of vertices consisting of one vertex from each of the connected components constitutes an independent set, and hence can be eliminated in parallel. The *height* of a graph is the minimum number of parallel elimination steps needed to eliminate all of the vertices of the graph.

Finding the height of a graph is NP-hard [24], but minimum-height orders may be found for specific classes of graphs. Aspvall and Heggernes [5] present an algorithm that finds elimination orders with minimum height for interval graphs, but the orders produced have not been analyzed in terms of fill. A family of chordal graphs for which any minimum-height order must produce fill that is more than a constant factor larger than the total number of nonzeros (graph edges plus fill) produced by a minimum-fill order is presented in [4]. Manne [20] shows how to produce minimum-height orders for trees with fill linear in the number of edges in the tree.

The height of an order is a measure of its parallelism. The more parallel an order, the smaller its height. Some graphs are inherently more sequential than others. For example, the vertices of a *clique* (a graph in which every pair of vertices is connected by an edge) must be eliminated sequentially, because a clique contains no independent sets of size greater than one. More generally, if a graph contains a clique, or if a clique is created during the course of elimination, then that clique must be eliminated sequentially. More parallelism can be exploited by using a method other than Gaussian elimination for eliminating entire cliques in parallel. For example, both Csanky's algorithm and Newton iteration can be used to quickly invert a dense matrix (corresponding to a clique) in parallel [17, Section 2.4]. This suggests modifying Gaussian elimination so that when a large dense subgraph is encountered, a different algorithm is used to eliminate it. In any single *stage* of such a hybrid algorithm, an independent set of cliques may be eliminated in parallel. We do not trade this form of parallelism for fill.

2 Parallelism and fill minimization

We have designed an algorithm, which we call "less parallel nested dissection" (LPND), that is a variant of nested dissection. The term "less parallel" refers not to the ordering algorithm itself, but rather to the elimination orders it produces. LPND produces orders with low fill but usually with less parallelism than the orders produced by standard nested dissection algorithms. While still using separators to guide the ordering process, the algorithm does not always order a separator last, but instead relies on the separator only to prevent fill between the different connected components it induces in the graph. In particular, once all but one of the components are eliminated, then the requirement that the separator vertices be ordered last no longer exists. Instead, LPND recurses on the subgraph formed by the last of the components along with the, as of yet unordered, separator vertices. (The actual algorithm is more involved, and sometimes treats the last component exactly as nested dissection does, ordering it before the separator.)

Although the orders produced by LPND may be less parallel than those produced by nested dissection, the ordering algorithm itself is similar to nested dissection, and lends itself to parallel implementation. We have not, however, exploited this avenue yet.

We have been able to analyze our new algorithm for the class of *chordal graphs*. A graph is said to be *chordal* if every cycle of length greater than or equal to four in the graph has a *chord*, i.e., an edge between two cycle vertices that are non-adjacent in the cycle. Chordal graphs are of special interest since they are the graphs that have zero-fill elimination orders. Unlike nested dissection, our algorithm produces zero-fill orders when applied to chordal graphs. In the proof that LPND produces no fill for chordal graphs we substitute the use of balanced separators with *minimal separators*, that is, sets of vertices that are not only *separators*, that is, vertices whose removal disconnects the graph, but are also required not to contain any smaller separators. These separators need not be balanced. Note that minimal separators are easily found and given any separator one can be easily reduce it to a minimal separator. The theoretical analysis for LPND for classes of graphs other than chordal graphs is open. On the other hand, the experimental results use balanced separators from standard packages such as Chaco and Metis.

Our experiments show that LPND algorithm generates less work and fill, on average, than state-of-the-art implementations of other ordering heuristics, including minimum-degree, nested dissection, and a hybrid of minimum-degree and nested dissection.

2.1 Why isn't nested dissection optimal?

In order to understand the intuition behind the new algorithm, consider a nested dissection order for a grid with width n and height k, $k \ll n$, such as the one in Figure 1. (In this grid each cycle of length four has been triangulated.) In breaking the grid down into approximately n/k subgraphs (of size approximately k^2), nested dissection would likely select roughly n/k vertical separators with k vertices each. The $k \times k$ subgraphs would be recursively ordered, causing fill between consecutive separators. Each separator would itself become a clique. At this point a considerable amount of fill, roughly $O(n \cdot k \cdot \log k)$, has already been introduced, but the remaining graph (the separators) has a zero-fill elimination order. Any elimination order with low height, however, requires an additional $\Omega(n \cdot k)$ fill. This explains in part why nested dissection produces

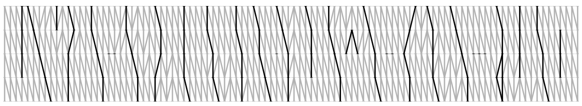

Figure 1: Partial elimination of a graph, according to a nested dissection order. Dark edges represent edges between vertices that have not been eliminated yet, while gray edges correspond to edges in which at least one of the endpoints has been eliminated.

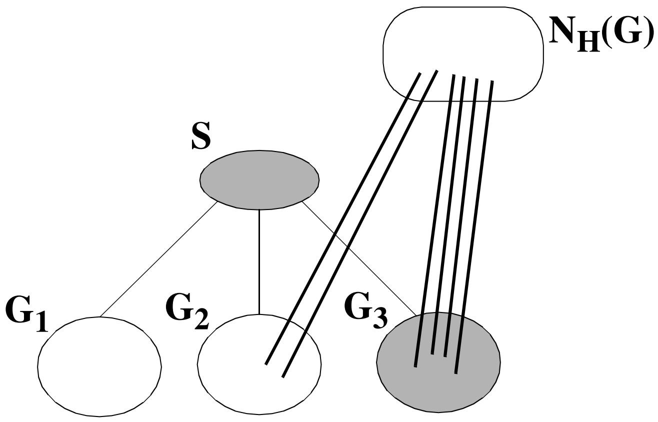

Figure 2: The LPND algorithm would order the vertices in G_1 and in G_2 before the vertices in G_3. The vertices in G_3 would be left to be ordered last, along with the vertices in S.

orders with higher amounts of fill than other ordering heuristics on some graphs.

2.2 A less parallel nested dissection algorithm

Our less parallel nested dissection algorithm was motivated by prior work in which we designed and implemented an algorithm to parallelize elimination orders generated by other heuristics [7]. The algorithm guarantees not to create fill that is more than linear in the size of the input graph (which might be a chordal completion graph produced by another ordering heuristic). As expected, we saw that the parallelism of minimum-degree orders could be improved, but sometimes at the cost of more fill and work. To our surprise, however, we found that our parallelizing algorithm actually made the nested dissection orders *less* parallel but also created less fill and work. This suggested to us to design a less parallel variant of nested dissection that produces less fill.

We propose the following algorithm, which takes a subgraph G of a graph H and the set $T = H \setminus G$ of vertices to be eliminated after the vertices of G and produces an elimination order for the vertices of G.

The algorithms works by finding a separator and the corresponding components the separator breaks the graph into. It then orders the components, specifying which of the components should be eliminated last. Finally, it decides whether to do a regular nested dissection step, meaning to order all the components in parallel, before the separator vertices, or to do a "less parallel" step, meaning to order the vertices in all but the last component in parallel, and then to recurse on the last component of the graph along with the, as of yet unordered, separator vertices.

Initially, the set T is empty, and G is the entire input graph to be ordered. In the recursive steps, the algorithm receives a set T of vertices that belong to separators from previous steps and are to be ordered after the vertices that remain in G, the subgraph to be ordered in the current recursive step.

The algorithm starts by finding a minimal separator S of G and the connected components G_k of $G \setminus S$. If a *non-trivial* minimal separator, that is, one that partitions the graph in more than one component, cannot be found, then it must be the case that G is a clique. In this case its vertices are ordered according to increasing degrees, that is, according to the number of neighbors each vertex has in T. This is not only a good heuristic for ordering the vertices in a clique, but obtains a zero-fill order for the vertices of the clique if the set of neighbors of G in T also forms a clique, as will be shown in Lemma 1.

At each step, if a non-trivial minimal separator can be found, the algorithm computes the number of neighbors that each component G_k has in T. If at least two components have the maximum number of neighbors in T over all components G_k, then the algorithm recurses on each component G_k as a subgraph of $G_k \cup S \cup T$ (with a set $T' = T \cup S$), exactly as nested dissection would. Vertices in all components are ordered so as to be eliminated before the vertices in S, exactly as nested dissection would number them. Unlike nested dissection, we also prescribe an order for the vertices within the separator. The vertices in S are ordered after the vertices within the various components according to increasing degrees in the graph induced by $S \cup T$, like a minimum-degree heuristic would: a vertex of S with the smallest degree is ordered first among the vertices in S and is eliminated, updating the degrees of its neighbors and potentially affecting the choice of the next vertex of S to be ordered.

If, on the other hand, a subgraph G_i has a number of neighbors in T that is larger than the number of neighbors that any other component has in T, then the algorithm treats G_i and S in a different manner. All components of $G \setminus S$ except for G_i are recursively ordered, just as before. However, instead of recursing on G_i, the algorithm recurses on the subgraph $G_i \cup S$ of $G_i \cup S \cup T$, after inserting edges corresponding to the fill created by the elimination of the other components of $G \setminus S$. The vertices in $G_i \cup S$ are numbered after the vertices in the other components of $G \setminus S$.

Figure 2 illustrates the situation in which the rightmost component in the figure has the largest number of neighbors in T, and is left to be ordered along with the vertices in the separator S, after the other two components are ordered.

We start by proving a lemma that shows that, under certain conditions, the step of the LPND algorithm that orders the vertices of a clique does not introduce any fill. Let $N_H(G)$ be the set of vertices of H that are not in a subgraph G of H and that have neighbors in G.

Lemma 1 *Let C be a clique in a chordal graph H, and let $N_H(C)$ also be a clique. Then every minimum-degree elimination order for the vertices of C creates zero fill.*

Proof. Since both C and $N_H(C)$ are cliques, we have to show that if v and w are vertices in C such that $d(w) \geq d(v)$ then w is adjacent to all neighbors of v, which implies that the set of neighbors of any vertex with minimum degree in C forms a clique in H, and thus can be eliminated from H with zero fill.

Assume by contradiction that this is not the case. Since C is a clique, it must be the case that there exists a vertex $n_v \notin C$ that is adjacent to v but not to w. Also, since $d(w) \geq d(v)$, there must exist a vertex $n_w \notin C$ that is adjacent to w but not to v. But then n_v and n_w are both in $N_H(C)$, and thus are adjacent. This forms a chordless cycle of length four and contradicts the assumption that H is chordal. ■

The next lemmas allow us to show that the LPND algorithm produces perfect elimination orders for chordal graphs.

Lemma 2 *Let G be an induced subgraph of a chordal graph H, such that $N_H(G)$ is a clique. Let S be a minimal separator of G. If G_1 and G_2 are distinct connected components of $G \setminus S$ such that $|N_H(G_1)| \leq |N_H(G_2)|$ then $N_H(G_1)$ is a clique.*

Proof. Assume for the sake of contradiction that $N_H(G_1)$ is not a clique. The vertices of $N_H(G_1)$ are contained in either S or $N_H(G)$. Every minimal separator of a chordal graph is a clique, and since G is an induced subgraph of H it is also chordal. Therefore S is a clique. Since both S and $N_H(G)$ are cliques, but $N_H(G_1)$

is not, there must exist vertices s_1 in S and t_1 in $N_H(G)$ that are in $N_H(G_1)$ but are not adjacent to each other. Since S is a minimal separator, all other components G_k of $G \setminus S$ have at least one vertex adjacent to s_1, for otherwise s_1 could be removed from S.

If t_1 is adjacent to vertices in both G_1 and G_2 then we can find a chordless cycle in H by considering a shortest path between s_1 and t_1 going only through s_1, t_1 and vertices in G_1, and another shortest path between s_1 and t_1 going only through s_1, t_1 and vertices in G_2. Since S is a separator, there are no edges between vertices in G_1 and vertices in G_2. Therefore, the paths taken together form a cycle with more than 3 vertices and without a chord, contradicting the fact that H chordal.

Now consider the case in which t_1 is not adjacent to any vertex in G_2. Since $|N_H(G_1)| \leq |N_H(G_2)|$ there exists a vertex of G_2 that is adjacent to a vertex t_2 in $N_H(G)$ that is not in $N_H(G_1)$. In this case, take a shortest path between s_1 and t_1 through G_1. Note that there can be no chords on a shortest path. The vertex t_2 is adjacent to t_1 since $N_G(H)$ is a clique. Either t_2 is also adjacent to s_1, in which case the path in G_1 along with the path $s_1 - t_2 - t_1$ is chordless (a contradiction), or s_1 and t_2 are not adjacent. In this case we consider the cycle formed by the shortest path from s_1 to t_1 through G_1, along with the edge (t_1, t_2) and a shortest path between t_2 and s_1 through G_2. This cycle is again chordless, a contradiction. Hence, $N_H(G_1)$ must be a clique. ■

Lemma 3 *Let G be an induced subgraph of a chordal graph H, such that $N_H(G)$ is a clique. Then given G and a set of vertices containing $N_H(G)$, the LPND algorithm orders the vertices of G so that they are eliminated from H with zero fill.*

Proof. If G is a clique then it has no non-trivial minimal separators, and the algorithm orders the graph using minimum degree, so that we can apply Lemma 1 proving the result. Assume G is not a clique. Then it has a minimal separator S that will be used by the algorithm to partition G into connected components $G_1, \ldots, G_n$, $n > 1$. Since G is an induced subgraph of H it is also chordal. Every minimal separator of a chordal graph is a clique, and thus S is a clique.

Let G_k be a subgraph that has the maximum number of neighbors in $N_H(G)$. First, assume that all other components have fewer neighbors in $N_H(G)$ than G_k. By Lemma 2 the set of neighbors $N_H(G_i)$ of each of these components forms a clique in H, so that, by induction, these components can be ordered with zero fill. After ordering all these components the algorithm recursively orders $G_k \cup S$ as a subgraph of $G_k \cup S \cup (H \setminus G)$. Since the set of neighbors of $G_k \cup S$ in $H \setminus G$ is contained in $N_H(G)$, it is also a clique by hypothesis. By induction, $G_k \cup S$ can be ordered and eliminated from H with zero fill.

Now consider the case in which two or more components have the maximum number of neighbors in $N_H(G)$. In this case, Lemma 2 can be applied to each component G_i of $G \setminus S$ (since for each G_i there is a G_j such that $|N_H(G_i)| \leq |N_H(G_j)|$), proving that the set of neighbors $N_H(G_i)$ forms a clique. Again, by induction, these components can be ordered with zero fill. After all components of $G \setminus S$ are eliminated the vertices in S are the only ones that still remain. But S is a clique, and so is $N_H(S)$, since $N_H(G)$ is clique. Thus we can apply Lemma 1. ■

Corollary 1 *The LPND algorithm produces a perfect elimination order when applied to a chordal graph.*

Proof. Let H be a chordal graph. If H is a clique, then any order will produce zero fill. Otherwise, the algorithm finds a minimal separator S. S is a clique since H is chordal. The algorithm orders the vertices of S last. As long as all other vertices have been eliminated, the vertices in S can be eliminated in any order with zero fill. The algorithm will recurse on each of the components of $H \setminus S$, and by Lemma 3 it will order each of these components with zero fill. ■

Unlike nested dissection, the algorithm we described will not necessarily generate orders with low height. In particular, given a graph consisting only of a simple path of length n, this algorithm will not introduce any fill. Therefore, any order it produces must have height at least $n/2$.

3 Experimental results

We have implemented the algorithm described in Section 2.2 and performed experiments that compare the orders it produces with the orders produced by state-of-the-art implementations of a number of other heuristics. We proceed to describe a few aspects of the implementation of the LPND algorithm.

Separators:

The algorithm only requires minimal separators, without any balance constraints. However, in the actual implementation the algorithm tries to obtain separators with a bounded amount of imbalance allowed, thus mimicking nested dissection.

Obtaining small balanced separators is an NP-hard problem on its own, but there are provably good approximation algorithms for finding separators as well as a number of algorithms that, in practice, produce very good separators. We use vertex separators produced by Chaco[1] [12, 13], a graph partitioning algorithm by Hendrickson and Leland at Sandia Labs.

Chaco is a multilevel algorithm that works by coarsening the graph into smaller and smaller graphs that approximate the initial graph. The smallest such graph is partitioned. This partition is mapped to a partition of the next smallest graph, from which the coarse graph was obtained. The process proceeds, as partitions are propagated back to finer and finer graphs. Every so many of these mapping steps, a variant of the Kernigan-Lin algorithm is applied as a local refinement method that improves the partitioning. Chaco uses this combination of heuristics to directly produce high quality edge and vertex separators. Parameters to Chaco affect our algorithm and the orders it produces. Among the parameters to Chaco is the permissible fraction of imbalance between the partitions to be produced. Chaco uses randomization, most noticeably in the coarsening steps. This makes our ordering algorithm also sensitive to the initial random seeds used. Even though Chaco produces good separators, we noticed that we obtain even better solutions, at the cost of speed, by running Chaco more than once with different random seeds on the same input graph and then choosing the smallest separator produced.

Minimum degree:

Most descriptions of nested dissection do not prescribe any ordering for vertices within a given separator because these vertices usually end up forming a clique after the components into which the separator breaks the graph have been eliminated. It is natural however, to try to minimize the fill from vertices in a separator to vertices higher up in the separator tree. An important part of the LPND algorithm is the use of a constrained minimum-degree rule to order the vertices within

[1] http://www.cs.sandia.gov/CRF/chac.html

separators and to order the vertices in subgraphs that could not be further divided. The constrained minimum-degree algorithm orders the vertices in the subgraphs by eliminating first the vertex of minimum degree among the vertices of the subgraph, while however considering the degrees measured in the whole graph.

We also use constrained minimum degree as the ordering heuristic for subgraphs that are smaller than some threshold. This widely adopted practice allows us to reduce the amount of time needed to produce our orders, while not affecting the quality of the orders produced by much.

Sorting recursive subproblems:

The step that differentiates the LPND algorithm from nested dissection involves choosing whether to treat all subgraphs defined by the separator identically, or to save one subgraph for last and merge it with the separator. As we have shown with Corollary 1, the choice of which subgraph should be left for last is clear when the graph being ordered is already a chordal graph. In practice we made the conditions under which the algorithm deviates from nested dissection more strict. Instead of requiring a component to have a number of neighbors outside the subgraph being ordered that is larger than the number of neighbors of any other component, in the implementation, we actually require that this component have at least some constant fraction more neighbors than any other component. This is a conservative measure that tries to avoid making bad decisions based on the imperfect information that is available during the computation by defaulting to the nested dissection behavior.

We also implemented a version of the Lipton, Rose, and Tarjan (LRT) nested dissection algorithm [18] that uses the same separator and minimum-degree algorithms that we used in implementing our own algorithm. In the tables that follow, we report the results obtained for our LPND algorithm, our implementation of the LRT algorithm, as well as the AMD[2] algorithm [2], the BEND algorithm (with code obtained from Rothberg [14, 15]), and the nested dissection algorithm in METIS-3.0[3] [16]. As for the results reported for the approximate minimum deficiency algorithm (AMMF), these are quoted from [26], and are not available for all the test matrices. Any unavailable results are represented with a "-" in the tables that follow.

The "bcsstk" matrices used in our experiments come from structural engineering problems, and were obtained from the Harwell-Boeing collection, and from Timothy Davis's "University of Florida Sparse Matrix Collection" [8] (the matrices were provided to Davis by Roger Grimes, at Boeing). The "cfd", gearbox and struct3 matrices were provided by Rothberg, while the nasasrb, pwt, and shuttle-eddy matrices came from NASA. All of these matrices can be obtained from the University of Florida Sparse Matrix collection. The "sf" matrices come from the simulation of an earthquake in the San Fernando Valley [22]. The "g" matrices are $n \times k$ grids, while the "hex" matrices are meshes of hexagons. The CAR, hsct16k, 50K, 172K and 178K matrices came from Olaf Storaasli at NASA Langley.

The number of vertices and edges in each graph/matrix can be found in Table 1. Under the label "c-vertices" this table lists the number of vertices in the compressed graph that is obtained by identifying sets of adjacent vertices that have the same neighborhood into a *supernode*. We applied the BEND algorithm to order each of

Matrix	vertices	edges	c-vertices	$\|L\|/10^3$	work/10^6
shuttle-eddy	10429	46585	10363	330	18
sf10	7294	44922	7294	531	71
g64x1024	65536	129984	65536	1351	83
pwt	36519	144794	36515	1494	140
g256x256	65536	130560	65536	1675	190
bcsstk35	30237	709963	6611	2776	394
bcsstk37	25503	557737	7093	2693	477
bcsstk36	23052	560044	4351	2555	483
hex64	24576	147456	24576	2080	562
hsct16k	16152	376432	7911	2479	578
bcsstk33	8738	291583	4344	1880	590
struct3	53570	560062	41644	4452	706
bcsstk30	28924	1007284	9289	3892	936
bcsstk32	44609	985046	14821	5067	959
bcsstk31	35588	572914	17403	4183	1159
sf5	30169	190377	30169	3873	1275
50K	49790	1253700	11224	7313	2235
nasasrb	54870	1311227	24954	9765	2820
3dtube	45330	1584144	15909	17906	12740
cfd1	70656	878854	70656	22386	13360
CAR	263574	6292129	61983	34352	13970
gearbox	153746	4463329	56175	38089	22020
172K	172400	7229603	36062	35601	24700
cfd2	123440	1482229	123440	39026	28710
hex256	393216	2359296	393216	50091	41620
178K	178044	6277614	145112	132100	183110
sf2	378747	2509064	378747	188240	339500

Table 1: Matrices and statistics for a BEND order

these matrices, and use the numbers obtained to normalize the results that are presented in the next tables. The results for the BEND order are also included in Table 1. The fill and work entries indicate the number of nonzeros in the submatrix L below and including the diagonal and the amount of work to decompose each matrix according to the BEND order. This and the remaining tables that we present show the matrices sorted according to increasing amounts of work. Finally, the height entries correspond to the reordering height for the BEND order.

Tables 2 and 3 present the number of nonzeros and work for decomposing each matrix according to the various orders produced. Both LRT and LPND used **two** calls to Chaco per separator. These results are presented as ratios with respect to the BEND order. Numbers smaller than 1 indicate an order that compares favorably with the BEND order. In an attempt to summarize the results in each table, we included the average, geometric mean and standard deviation of these normalized results.

These results indicate that both LRT and LPND orders are on average slightly better than the BEND order. However, the differences observed are of only a few percent, making it difficult to draw any strong conclusions. The running times for both LRT and LPND are comparable and are significantly higher than those for the BEND algorithm. Table 4 compares the amount of time to produce the BEND and the LPND orders for the various matrices. It is unclear how much the BEND orders could improve if the algorithm were given more time to compute the orders. However, our results show that reductions in both fill and work can be obtained by investing more time finding elimination orders.

The fill and work results for the AMD orders stand out. Although occasionally better in individual cases, the AMD algorithm produces orders that are significantly worse than the orders produced by the remaining algorithms. This is consistent with the results obtained by Hendrickson and Rothberg in [14], where they found that their implementation of the nested dissection algorithm was significantly better than the minimum-degree algorithms, and

[2]code from ftp://ftp.cise.ufl.edu/pub/umfpack/AMD/

[3]code from ftp://ftp.cs.umn.edu/dept/users/kumar/metis/

Ratio	AMD	AMMF	METIS	LRT	LPND
shuttle-eddy	0.9910	-	1.1144	1.0139	0.9415
sf10	1.2726	-	1.0737	1.0205	0.9905
g64x1024	1.0553	-	1.0771	0.9686	0.8757
pwt	1.0656	1.1205	0.9295	0.8740	0.8754
g256x256	1.1772	-	1.0512	0.9056	0.8986
bcsstk35	0.9843	0.9645	1.1284	0.9944	0.9733
bcsstk37	1.0395	1.0154	1.1682	0.9564	0.9567
bcsstk36	1.0695	1.0375	1.1887	1.0145	1.0028
hex64	1.1644	-	0.9699	0.9105	0.9103
hsct16k	1.0808	-	1.1361	1.0276	1.0080
bcsstk33	1.3680	1.2757	1.2271	1.0929	1.0775
struct3	1.1441	1.1185	1.0274	0.9468	0.9667
bcsstk30	0.9903	0.8869	1.1407	1.0196	0.9102
bcsstk32	0.9840	0.9541	1.1310	1.0135	0.9824
bcsstk31	1.3286	1.0525	1.0549	0.9449	0.9575
sf5	1.3541	-	1.0320	0.9595	0.9521
50K	1.2086	-	1.1721	1.0003	0.9982
nasasrb	1.2242	-	1.0837	1.0086	0.9560
3dtube	1.4719	1.5511	1.0314	0.9548	0.9548
cfd1	1.6856	1.3106	1.0205	0.8597	0.8047
CAR	1.1028	-	1.0631	0.9769	0.9846
gearbox	1.2748	1.2614	1.0015	0.9227	0.9166
172K	0.8196	-	1.0203	0.9249	0.9010
cfd2	1.9220	1.6969	0.9977	0.9218	0.9411
hex256	1.0948	-	0.9070	0.8420	0.8420
178K	1.7301	-	0.9272	0.8650	0.8645
sf2	2.0333	-	0.8558	0.8311	0.8306
AVERAGE	1.2458	1.1727	1.0567	0.9545	0.9361
GEOMEAN	1.2163	1.1520	1.0528	0.9523	0.9341
STDEV	0.2959	0.2400	0.0911	0.0650	0.0618

Table 2: Fill relative to the BEND order

Ratio	AMD	AMMF	METIS	LRT	LPND
shuttle-eddy	0.9642	-	1.2855	1.1027	0.8422
sf10	1.9313	-	1.1294	1.0865	0.9736
g64x1024	1.0160	-	1.1778	0.9716	0.6271
pwt	1.2305	1.3761	0.8017	0.7194	0.7215
g256x256	1.3736	-	1.1728	0.8519	0.8087
bcsstk35	0.9721	0.9110	1.2905	1.0233	0.9642
bcsstk37	1.1150	1.0106	1.4479	0.8705	0.8869
bcsstk36	1.2833	1.1255	1.4609	1.0786	1.0075
hex64	1.2301	-	0.8330	0.7139	0.7137
hsct16k	1.2415	-	1.3558	1.0901	1.0337
bcsstk33	2.1035	1.6197	1.5193	1.2053	1.1730
struct3	1.5455	1.4126	1.0895	0.9266	0.9841
bcsstk30	1.0057	0.7022	1.2592	1.1182	0.7630
bcsstk32	0.9885	0.9305	1.3432	1.1273	1.0396
bcsstk31	2.5004	1.3932	1.0267	0.8645	0.9077
sf5	2.1812	-	1.0776	0.8996	0.8824
50K	2.2367	-	1.4872	1.0224	1.0336
nasasrb	1.6918	-	1.2582	1.1411	0.9199
3dtube	2.3571	2.4563	0.9521	0.8673	0.8673
cfd1	3.3323	1.7663	1.3421	0.8293	0.6635
CAR	1.6600	-	1.0014	0.9556	1.0193
gearbox	2.1353	2.0058	1.0595	0.9382	0.9260
172K	0.6138	-	0.9425	0.7854	0.7186
cfd2	4.7510	3.4315	1.1961	0.9871	1.0355
hex256	1.1499	-	0.8400	0.6540	0.6540
178K	3.8589	-	1.0334	0.9012	0.8941
sf2	5.5022	-	0.8999	0.8315	0.8031
AVERAGE	1.9249	1.5493	1.1586	0.9468	0.8838
GEOMEAN	1.6615	1.4120	1.1399	0.9359	0.8726
STDEV	1.1934	0.7459	0.2100	0.1441	0.1409

Table 3: Work relative to the BEND order

that the BEND algorithm was a few percent better than their own implementation of the nested dissection algorithm.

Tables 5 and 6 show the reordering height and number of reordering stages for each of the orders. The results for the AMMF order are not available. The results for the two largest cases are omitted, for they would take too long to compute. As expected, we can see in the tables that the orders produced by the LPND algorithm are less parallel than the corresponding LRT orders while the AMD orders are even less parallel than the LPND orders in terms of height, but in most cases are more parallel than the LPND orders in terms of the number of stages. This difference indicates that AMD orders are probably likely to produce larger cliques than the LPND orders, as corroborated by Table 7, which lists the maximum front size, i.e., the maximum size of the neighborhood of for each order (again, except for the AMMF orders).

3.1 Nested dissection versus minimum degree

Although we believe that nested dissection is a better approach for producing elimination orders than minimum degree, we cannot deny that, on certain graphs, minimum degree does generate better elimination orders in terms of both fill and work.

Figures 3 and 4 show the number of nonzeros and the amount of work for decomposing grids according to various ordering algorithms. In these experiments we used rectangular two-dimensional grid graphs of various aspect ratios, each of which had a total of 2^{16} vertices. Four ordering algorithms were applied to each graph, namely nested dissection (ND), the hybrid algorithm that we mentioned previously (BEND) [14, 15], our "less parallel" variation of nested dissection (LPND) and a version of minimum degree (AMD) [2]. We also applied a special-purpose algorithm that works only on two-dimensional grids, and uses diagonal separators as a basis for the ordering. The results for all orders are normalized to the results for this special-purpose order.

On graphs with large aspect ratios, the minimum-degree algorithm does best in terms of fill, but poorly in terms of parallelism. On these same graphs, the nested dissection and the hybrid algorithm produce orders that are substantially more parallel, but require more fill and work. For each of the orders produced by these algorithms we computed the minimum-height order that is equivalent to the order produced, in the sense that the new order produces the same fill edges. The heights of these minimum-height orders are shown in Figure 5, again, relative to the special-purpose order. We note that the LPND algorithm produces very sequential orders that are only a few percent more parallel than those produced by AMD. At least part of the additional fill experienced by the nested dissection and hybrid orders is a function of the additional parallelism in these orders.

We also performed an additional experiment. We took a point ($\log(n/k) = 8$) in the curve in Figure 4 for which AMD produces the order with the smallest amount of work. Then, with a fixed aspect ratio of $1 : 256$, we ordered grids with total number of nodes varying from 2^8 to 2^{20} vertices. The results obtained for the various algorithms are shown in Figures 6 and 7, and indicate an interesting trend: even though AMD produces better orders for small grids, the larger the grid, the worse the order it produces is, relative to the orders produced by both nested dissection and LPND. We also observe that the orders produced by BEND have pretty much the same slope for large enough graphs. This is very likely due to the fact that BEND partitions the graph into only a small number of pieces and uses a minimum-degree heuristic to order the vertices within each piece. For large graphs, some tuning in the number and size of these pieces can probably be done so that BEND will produce better orders.

Matrix	BEND(secs)	Slow-down: LPND/BEND
shuttle-eddy	0.981	3.258
sf10	0.962	3.534
g64x1024	4.730	4.599
pwt	3.049	3.938
g256x256	4.407	4.975
bcsstk35	7.365	0.351
bcsstk37	5.539	0.610
bcsstk36	5.400	0.313
hex64	2.805	3.646
hsct16k	3.322	2.079
bcsstk33	2.133	1.915
struct3	7.131	3.565
bcsstk30	7.680	0.893
bcsstk32	10.142	0.837
bcsstk31	7.063	1.684
sf5	3.491	5.916
50K	11.594	0.465
nasasrb	11.817	1.805
3dtube	12.485	0.988
cfd1	13.385	8.632
CAR	42.983	0.850
gearbox	33.132	1.378
172K	47.357	0.688
cfd2	18.237	13.141
hex256	41.574	6.313
178K	54.393	9.461
sf2	59.876	17.186

Table 4: Amount of time to compute BEND order, in seconds, and ratio between the time to compute the LPND order and the BEND order

Matrices	BEND	Ratio to BEND			
	height	AMD	METIS	LRT	LPND
shuttle-eddy	659	1.2914	0.5083	0.4431	1.1897
sf10	703	1.1280	0.7923	0.8990	0.9900
g64x1024	979	2.8509	0.4474	0.5455	2.0409
pwt	778	1.2134	0.7339	0.6530	0.7558
g256x256	995	1.6251	0.7477	0.8151	0.9980
bcsstk35	1084	1.1642	0.8948	0.8792	1.0563
bcsstk37	1162	1.1472	1.0602	0.8614	1.0258
bcsstk36	1255	1.2271	0.9291	0.9211	0.9793
hex64	1226	0.9013	0.8303	0.8051	0.8051
hsct16k	1307	1.4667	0.8898	0.9082	1.2754
bcsstk33	1534	1.1682	0.8149	0.7640	0.7823
struct3	1440	1.0708	0.7590	0.6479	0.9868
bcsstk30	1800	1.5356	0.7500	0.7361	1.4728
bcsstk32	1774	1.3850	0.7813	0.7418	1.2272
bcsstk31	1641	1.3924	0.7867	0.7569	1.0049
sf5	1830	1.2792	0.8322	0.8066	1.0443
50K	2716	1.0670	0.7323	0.6116	0.6554
nasasrb	3360	1.4372	0.4946	0.4726	1.2711
3dtube	3477	1.4498	0.7636	0.8332	0.8332
cfd1	3438	2.3040	0.9389	0.7833	1.5852
CAR	3824	1.2644	0.7858	0.8413	1.1462
gearbox	3899	1.4334	0.9336	0.9646	1.1749
172K	4790	0.9305	0.7931	0.7975	0.8929
cfd2	5538	1.7143	0.7346	0.6844	1.1685
hex256	5423	0.8388	0.8044	0.7640	0.7533
178K	-	-	-	-	-
sf2	-	-	-	-	-
AVERAGE		1.3714	0.7816	0.7575	1.0846
STDEV		0.4301	0.1380	0.1348	0.2999

Table 5: Reordering height relative to the BEND order

4 Final remarks

With all the recent work on finding low-fill elimination orders it has become fairly clear that, while a number of approaches provide low-fill orders, the best ones rely, at least to some extent, on finding small, balanced or only slightly unbalanced separators.

While minimum-degree is a good heuristic, in the sense that minimum-degree orders are easily computable, requiring very little running time, and produce relatively low fill, it is undeniable that they are not anymore, taken alone, the heuristic of choice. With today's large matrices it is worthwhile to invest running time in computing lower-fill elimination orders. The nested dissection and hybrid algorithms we experimented with produces orders that are in the same ballpark in terms of both fill and work.

It is unclear how the LPND algorithm would compare with the BEND algorithm if BEND were given more time to produce its separators, or to improve on its ordering. Although the LPND orders are better in terms of fill and work, on average, than the BEND orders, the difference in performance is not large enough, at least for the matrices in our benchmark, to allow a definite answer as to which ordering method is best.

For large enough graphs, such as the largest grids we experimented with, BEND will need to be "re-tuned" in order to produce orders with low-enough fill. The larger the graph, the lower the fill and the work we expect to see for orders produced by either nested dissection or LPND, relative to both minimum-degree and BEND orders.

References

[1] A. Agrawal, P. Klein, and R. Ravi. Cutting down on fill using nested dissection: provably good elimination orderings. In A. George, J. Gilbert, and J. W. H. Liu, editors, *Graph Theory and Sparse Matrix Computation*, volume 56 in the IMA Volumes in Mathematics and its Applications, pages 31–55. Springer–Verlag, 1993.

[2] P. Amestoy, T. Davis, and I. Duff. An approximate minimum degree ordering algorithm. *SIAM Journal of Matrix Analysis and Applications*, 17(4):886–905, 1996.

[3] C. Ashcraft and J. Liu. Robust ordering of sparse matrices using multisection. Technical Report ISSTECH-96-002, Boeing information and support services, 1996.

[4] B. Aspvall. Minimizing elimination tree height can increase fill more than linearly. *Information Processing Letters*, 56:115–120, 1995.

[5] B. Aspvall and P. Heggernes. Finding minimum height elimination trees for interval graphs in polynomial time. *BIT*, 34:484–509, 1994.

[6] P. Berman and G. Schnitger. On the performance of the minimum degree ordering for Gaussian elimination. *SIAM Journal of Matrix Analysis and Applications*, 11(1):83–88, January 1990.

[7] C. F. Bornstein, B. M. Maggs, G. Miller, and R. Ravi. Parallel gaussian elimination with linear work and fill. In *Proceedings of the Thirty-Eighth Annual Symposium on Foundations of Computer Science*, pages 274–283, October 1997.

Matrices	BEND stages	Ratio to BEND AMD	METIS	LRT	LPND
shuttle-eddy	26	1.5000	0.8077	0.6538	1.4615
sf10	31	0.7742	1.0323	0.8387	1.1290
g64x1024	47	2.3191	0.6596	0.5319	2.0213
pwt	43	0.5814	0.6512	0.6977	0.7674
g256x256	46	1.1304	0.7174	0.5435	0.7391
bcsstk35	29	1.1034	1.0345	0.9655	1.1379
bcsstk37	27	1.1481	1.0741	1.0000	1.1481
bcsstk36	24	1.0000	1.0833	0.8333	0.9583
hex64	17	0.7059	1.7059	1.0588	1.0588
hsct16k	25	1.1200	0.9200	0.7600	1.1200
bcsstk33	22	1.0455	0.8636	0.8636	0.9545
struct3	30	0.9667	0.9000	0.7667	1.0333
bcsstk30	24	1.5833	0.9583	0.8750	1.6667
bcsstk32	27	1.4815	1.1111	0.9630	1.7778
bcsstk31	30	0.9667	1.0000	1.0000	1.2333
sf5	38	0.7632	1.0789	1.0526	1.3158
50K	36	0.7778	1.5000	0.8611	0.9722
nasasrb	28	1.1786	1.0714	0.6786	1.5714
3dtube	18	0.9444	1.2778	1.0000	1.0000
cfd1	35	1.3714	1.0857	0.7143	1.4857
CAR	40	1.4000	0.8500	0.8250	1.0500
gearbox	30	1.1000	1.0667	1.0667	1.3000
172K	26	1.0385	1.2308	0.8462	1.1154
cfd2	40	0.9500	0.8500	0.7000	1.4750
hex256	22	0.7273	1.6818	1.0909	1.0909
178K	-	-	-	-	-
sf2	-	-	-	-	-
AVERAGE		1.1071	1.0485	0.8475	1.2233
STDEV		0.3642	0.2717	0.1617	0.3108

Table 6: Number of reordered stages relative to the BEND order

Ratio	AMD	METIS	LRT	LPND
shuttle-eddy	0.8462	1.1748	1.1049	0.7762
sf10	1.5135	1.0980	1.1858	0.9155
g64x1024	0.7766	1.0266	1.0160	0.6755
pwt	1.1351	1.1577	0.8649	0.8559
g256x256	1.0974	1.1103	1.0179	1.0179
bcsstk35	0.8814	1.0442	1.0977	0.9953
bcsstk37	1.0174	1.2046	0.8301	0.9093
bcsstk36	1.2705	1.0989	1.2425	0.9757
hex64	0.9349	0.7850	0.7064	0.7064
hsct16k	1.2077	1.3910	1.2322	1.2688
bcsstk33	1.3870	1.1152	1.0295	1.0295
struct3	1.4507	1.5045	1.0067	1.0942
bcsstk30	1.0294	1.2003	1.1796	0.8497
bcsstk32	0.9521	1.2801	1.1915	1.2305
bcsstk31	1.9695	0.9791	0.9310	0.9727
sf5	1.5416	1.0567	0.8492	0.8480
50K	1.8794	1.4710	0.9317	0.9807
nasasrb	1.5339	1.3339	1.3403	1.1016
3dtube	1.3759	0.8693	0.8693	0.8693
cfd1	2.0487	1.4912	1.0974	0.8772
CAR	1.3997	0.9236	1.0102	1.0993
gearbox	1.4588	1.1647	1.1712	1.1712
172K	0.8997	0.8886	0.8886	0.8886
cfd2	2.3176	1.3871	1.2229	1.1431
hex256	0.9235	0.8566	0.6939	0.6939
178K	-	-	-	-
sf2	-	-	-	-
AVERAGE	1.3139	1.1445	1.0285	0.9578
STDEV	0.4106	0.2047	0.1721	0.1613

Table 7: Front sizes relative to the BEND order

[8] T. Davis. University of Florida sparse matrix collection. *http://www.cise.ufl.edu/ ~davis/sparse/*, 1994.

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

[10] J. A. George and J. W. Liu. An automatic nested dissection algorithm for irregular finite element problems. *SIAM Journal of Numerical Analysis*, 15:1053–1069, 1978.

[11] J. R. Gilbert and R. E. Tarjan. The analysis of a nested dissection algorithm. *Numerische Mathematik*, 50:377–404, 1987.

[12] B. Hendrickson and R. Leland. The Chaco user's guide: Version 2.0. Technical Report SAND94–2692, Sandia National Laboratories, 1994.

[13] B. Hendrickson and R. Leland. A multilevel algorithm for partitioning graphs. In *Proceedings of the 1995 International Conference on Supercomputing*, November 1995.

[14] B. Hendrickson and E. Rothberg. Improving the runtime and quality of nested dissection ordering. Technical Report SAND96-0868, Sandia National Labs, March 1996. To appear in SIAM J. Sci. Stat. Comput.

[15] B. Hendrickson and E. Rothberg. Effective sparse matrix ordering: Just around the BEND. In *Eight SIAM Conference on Parallel Processing for Scientific Computing*, March 1997.

[16] G. Karypis and V. Kumar. Metis. Unstructured graph partitioning and sparse matrix ordering system. *http://www.www.cs.umn.edu/~karypis/metis/metis.html*, 1995.

[17] F. T. Leighton. *Introduction to Parallel Algorithms and Architectures: Arrays • Trees • Hypercubes*. Morgan Kaufmann, San Mateo, CA, 1992.

[18] R. J. Lipton, D. J. Rose, and R. E. Tarjan. Generalized nested dissection. *SIAM Journal of Numerical Analysis*, 16:346–358, 1979.

[19] J. W. Liu. Modification of the minimum degree algorithm by multiple elimination. *ACM Transactions on Mathematical Software*, 12:141–153, 1985.

[20] F. Manne. An algorithm for computing an elimination tree of minimum height for a tree. Technical Report CS-91-59, University of Bergen, Norway, 1991.

[21] H. M. Markowitz. The elimination form of the inverse and its application to linear programming. *Management Sci.*, 3:255–269, 1957.

[22] D. O'Hallaron and J. Shewchuk. Properties of a family of parallel finite element simulations. Technical Report CMU-CS-96-141, School of Computer Science, Carnegie Mellon University, 1996.

[23] S. Parter. The use of linear graphs in Gaussian elimination. *SIAM review*, 3:364–369, 1961.

[24] A. Pothen. The complexity of optimal elimination trees. Technical Report CS-88-16, Department of Computer Science, The Pennsylvania State University, 1988.

[25] D. Rose. Triangulated graphs and the elimination process. *J. Math. Anal. Appl.*, 32:597–609, 1970.

[26] E. Rothberg and S. Eisenstat. Node selection strategies for bottom-up sparse matrix ordering. *SIAM Journal of Matrix Analysis and Applications*, 19(3):862–695, 1998.

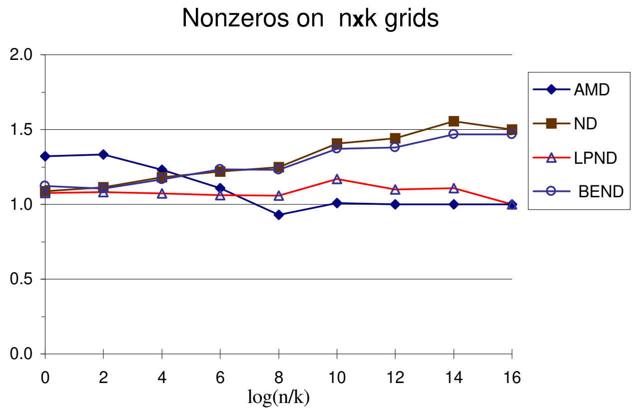

Figure 3: Number of nonzeros introduced by different elimination orders for $n \times k$ rectangular grids, with a total of 2^{16} vertices, relative to a special-purpose order.

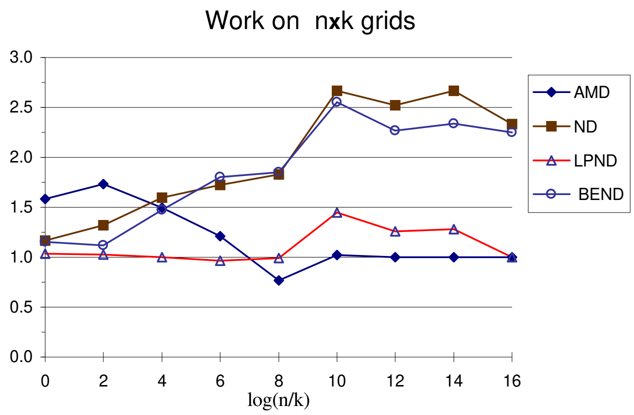

Figure 4: Work for different orders relative to a special-purpose order.

[27] W. F. Tinney and J. W. Walker. Direct solutions of sparse network equations by optimally ordered triangular factorization. *J. Proc. IEEE*, 55:1801–1809, 1967.

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

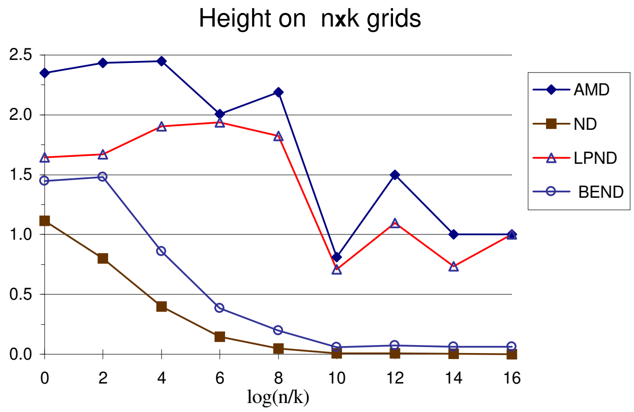

Figure 5: Height relative to a special-purpose order.

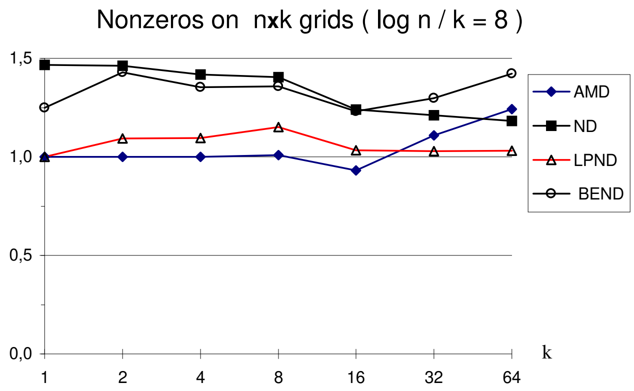

Figure 6: Number of nonzeros introduced by different elimination orders for $n \times k$ rectangular grids with a fixed aspect ratio (1 : 256), and sizes varying from 2^8 to 2^{20} nodes.

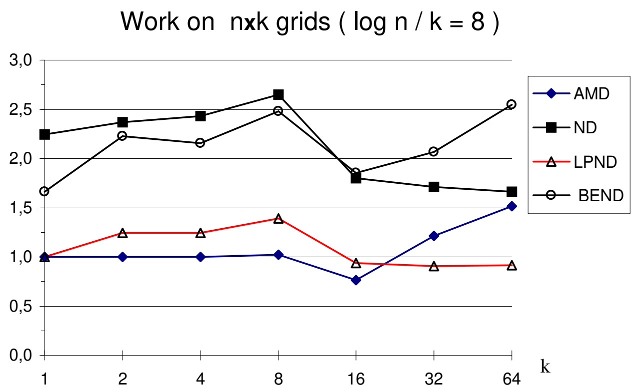

Figure 7: Work for different orders relative to a special-purpose order, for different size grids with a fixed aspect ratio of 1 : 256.