A Parallel Formulation of Interior Point Algorithms*

George Karypis, Anshul Gupta, and Vipin Kumar
karypis@cs.umn.edu agupta@cs.umn.edu kumar@cs.umn.edu
Computer Science Department
University of Minnesota, Minneapolis, MN 55455

Technical Report 94-20

Abstract

In recent years, interior point algorithms have been used successfully for solving medium- to large-size linear programming (LP) problems. In this paper we describe a highly parallel formulation of the interior point algorithm. A key component of the interior point algorithm is the solution of a sparse system of linear equations using Cholesky factorization. The performance of parallel Cholesky factorization is determined by (a) the communication overhead incurred by the algorithm, and (b) the load imbalance among the processors. In our parallel interior point algorithm, we use our recently developed parallel multifrontal algorithm that has the smallest communication overhead over all parallel algorithms for Cholesky factorization developed to date. The computation imbalance depends on the shape of the elimination tree associated with the sparse system reordered for factorization. To balance the computation, we implemented and evaluated four different ordering algorithms. Among these algorithms, Kernighan-Lin and spectral nested dissection yield the most balanced elimination trees and greatly increase the amount of parallelism that can be exploited. Our preliminary implementation achieves a speedup as high as 108 on 256-processor nCUBE 2 on moderate-size problems.

Keywords: linear programming, interior point methods, Cholesky factorization, sparse linear systems.

1 Introduction

In this paper we describe a scalable parallel formulation of interior point algorithms. Through our implementation on a 256-processor nCUBE 2 parallel computer, we show that our parallel formulation utilizes hundreds of processors efficiently and delivers much higher performance and speedups than reported earlier. These speedups are a result of our highly efficient parallel algorithm for solving a linear symmetric positive definite system using Cholesky factorization. We also evaluate a number of ordering algorithms for sparse matrix factorization in terms of their suitability for parallel Cholesky factorization.

Linear programming (LP) deals with the problem of minimizing or maximizing a linear function in the presence of linear equality and or inequality constraints. Linear programming is extensively used to model large and complex problems in many fields [11, 2, 49]. However, as the size and

*This work was supported by Army Research Office Contract DA/DAAH04-93-G-0080 at the University of Minnesota Army High Performance Computing Research Center.

complexity of such problems increase, the computational power provided by traditional uniprocessor computer systems is not enough to solve them within reasonable amounts of time. The emergence of parallel computers provides the computational power that can be used to solve large linear programming problems within reasonable time periods.

Traditionally, linear programming problems have been solved using the *simplex method* [6]. Even though this method performs well in practice, its run time is not polynomially bound. In 1984, Karmarkar [26] developed a radically different algorithm for solving linear programming problems, called *projective scaling*. Karmarkar's algorithm is iterative in nature and has polynomial run time. It starts from a point within the feasible set and approaches the optimal solution by moving within this set. Since the development of this first algorithm, several variants have been proposed [45, 5, 46, 44, 42, 41, 64]. Because these algorithms approach the optimal solution while staying within the feasible set, they are called *interior point algorithms*. Research over the past decade has produced evidence that interior point algorithms provide a viable alternative to the simplex method for most medium- to large-size problems, and can be many times faster than the simplex method as the problem size increases [41, 64]. Most recent computational research has concentrated on the *primal-dual logarithmic barrier method* [42], and in particular, on the *predictor-corrector variant* [41, 43, 46] of this method.

The bulk of the computation performed in each iteration of the interior point algorithms is the solution of a symmetric positive definite system. This system is usually solved by obtaining the Cholesky factor of the matrix and then solving the triangular system ([30] chapter 5). Note that the amount of computation performed in each iteration of interior point algorithms is considerably higher than that for the simplex method; however, interior point algorithms require substantially fewer iterations.

Earlier efforts at developing parallel formulations of linear programming algorithms concentrated on dense or nearly dense problems [9, 61, 48, 21, 28]. Eckstein [10] provides a good survey of work on parallel algorithms for dense linear programming problems. For dense matrices, there are efficient parallel formulations for both rank-one updates [9] and Cholesky factorizations [12, 30], making it easy to develop highly scalable formulations for dense LP problems [28, 9]. In contrast, attempts to develop general parallel algorithms for sparse linear programming problems have had limited success. Shu and Wu [60] developed parallel formulations for both the product form of inverses and the LU variant of the revised simplex method on a shared-memory computer. For the former they obtained speedup of 4 on 32 processors, and for the latter, got a speedup of 2 on 8 processors. Housos *et al.* [22] and Saltzman [58] developed parallel formulations of interior point methods for shared-memory architectures, and report good speedups on up to eight processors. Bisseling *et al.* [4] developed a parallel formulation for the dual affine interior point algorithm on a transputer network. They obtained speedup up to 88 on 400 processors for certain classes of LP problems (scheduling problems in oil refineries), and speedup up to 60 on 400 processors for general problems. Speedups are not high particularly for general LP problems, despite the fact that the transputers offer low latency and high bandwidth communication relative to the CPU speed.

To obtain efficient parallel formulations of interior point algorithms, we need to develop an efficient parallel sparse Cholesky factorization algorithm. However, attempts to obtain efficient and highly parallel formulations have had limited success so far. The main reason for that is that the amount of computation relative to the size of the matrix being factored is very small and the communication overhead is relatively high. We have recently developed [19] a parallel sparse Cholesky factorization algorithm based on the multifrontal sequential algorithm [8, 38]. The analysis presented in [19] and our experiments show that good speedup can be achieved by increasing the degree of parallelism and at the same time minimizing the communication overhead.

Parallel Cholesky factorization exploits the sparsity of the matrix to perform more than one elimination at the same time. The columns that can be eliminated in parallel are determined by the elimination tree associated with the ordering. Depending on the ordering, both the amount of fill-in and the number of columns that can be eliminated in parallel may change significantly. However, to minimize communication overhead, our parallel multifrontal algorithm assigns the work associated with subtrees of the elimination tree to group of processors. Due to this assignment, in order to ensure load balance, the amount of computation needed to eliminate the columns associated with each subtree must be roughly the same. We have implemented a variety of existing ordering algorithms and their variants and evaluated their ability to minimize the work load imbalance and the amount of fill-in. Among the orderings we evaluated are minimum degree [16, 20, 17], minimum degree with constraints [36, 35], spectral nested dissection [53], and a nested dissection scheme based on Kernighan-Lin's edge separators [29].

The rest of this paper is organized as follows. Section 2 describes the sequential dual affine algorithm. Section 3 reviews the Cholesky factorization process and describes the multifrontal algorithm. Section 4 describes the various parallel algorithms involved in our parallel dual affine algorithm, including multifrontal, matrix-matrix, and matrix-vector multiplication. Section 5 analyzes the various parallel algorithms. Section 6 describes the test problems used to evaluate the parallel dual affine algorithm. Sections 7 and 8 provide experimental evaluation of the ordering algorithms and the parallel dual affine algorithm respectively. Finally, Section 9 contains concluding remarks.

2 Dual Affine Algorithm

The LP problem to be solved is

$$\begin{array}{ll} \text{Minimize} & c^T x \\ \text{subject to} & Ax = b \\ & x \geq 0. \end{array} \tag{1}$$

Where, A is an $m \times n$ matrix containing m constraints and n variables, c is a vector of size n, b is a vector of size m, and x is the unknown vector of length n.

The corresponding dual problem is

$$\begin{array}{ll} \text{Maximize} & b^T y \\ \text{subject to} & A^T y + z = c \\ & z \geq 0. \end{array} \tag{2}$$

Where A, b, and c are similar to the primal problem, y is the unknown vector of length m and z is the vector of slack variables of length n.

Let Z be an $n \times n$ diagonal matrix, with elements $z_{i,i} = z_i$, let $D = Z^{-1}$, and let x be a vector of primal estimates of length n. Following [44], an iteration of the dual affine interior point algorithm is shown in Program 2.1.

The algorithm starts with an interior point (y, z) of the dual problem Equation 2. In each iteration, new search directions d_y and d_z, a maximum step size α, and a new value for the interior point (y, z) are computed. Also, a solution to the primal problem Equation 1 is computed in step 3. The algorithm terminates when an optimal solution triple (x, y, z) is found.

The computationally demanding step of each iteration of the dual affine algorithm is computing d_y (step 1). This is done by solving the following linear system of equations

$$M d_y = b, \tag{3}$$

1.	$d_y = (AD^2A^T)^{-1}b$
2.	$d_z = -A^T d_y$
3.	$x = -Z^{-2} d_z$
4.	$\alpha = \gamma \min\{-z_j/(d_z)_j : (d_z)_j < 0, j = 0, \ldots, n-1\}$
5.	$y = y + \alpha d_y$
6.	$z = c - A^T y$

Program 2.1: The computation performed in each iteration of the dual affine interior point algorithm.

where $M = AD^2A^T$. Note that M is symmetric positive definite, and thus d_y can be obtained by first factoring M using Cholesky factorization [16, 7] and then solving two triangular systems. Cholesky factorization is the most expensive part of the entire computation. In our experiments, around 90 percent of the time is spent in solving Equation 3.

Besides solving Equation 3, in each iteration of the dual affine algorithm, a matrix-matrix multiplication is performed to obtain M, and in both steps 2 and 6, a matrix-vector multiplication is performed. The rest of computation in each iteration is fairly small, an element-wise vector multiplication in step 3, a vector addition in step 5, and finding the minimum of n elements in step 4.

3 Cholesky Factorization

In most linear programming problems, the matrix A is sparse, resulting in a sparse M. During the factorization of M, when a column is subtracted from another column, some of the zero elements in the latter may become nonzero. These nonzero elements are said to be generated as a result of fill-in. Different orderings of the rows and columns of M result in different amounts of fill-in. A poor ordering of M can significantly increase the amount of fill-in, which in turn translates to a significant increase in the factorization time. Hence, it is desirable to find a good ordering (*i.e.*, a permutation matrix P so that the Cholesky factor L of PMP^T has small fill-in). The problem of finding the best ordering for M that minimizes the amount of fill-in is NP-complete [66], therefore a number of heuristic algorithms for ordering have been developed. In particular, minimum degree ordering [16, 20, 17] is found to have low fill-in.

Even though the values of matrix D change in each iteration, matrix A remains the same, and thus, the sparsity structure of M also remains the same. Thus, the algorithm for finding a good ordering needs to be applied only once, and the same ordering can be used in each iteration of the dual affine algorithm.

For a given ordering of sparse matrix M, there exists an *elimination tree* [37], which shows the precedence relations among columns with respect to Cholesky factorization. The elimination tree of a matrix M contains a node for each column of M. A "parent" relation between the nodes is defined as follows:

$$Parent[j] = \begin{cases} \min\{i | l_{i,j} \neq 0, i > j\} \\ \\ 0, \text{ for } j = m \end{cases}$$

Here $l_{i,j}$ is the element in the row i and column j of the Cholesky factor L. Note that *Parent*[j] is the row index of the first subdiagonal nonzero in column j. Column j can be eliminated only after the columns that are descendents of j in the elimination tree have been eliminated. Figure 1 shows a factor matrix L and its elimination tree. In parallel implementations of Cholesky factorization, elimination trees help to determine the columns that can be eliminated independently. For example, in the elimination tree shown in Figure 1(c), columns 1, 2, and 3 can be eliminated concurrently, as they are not dependent upon each other. Different orderings of M result in different elimination trees as illustrated in Figure 1(c) and 1(d).

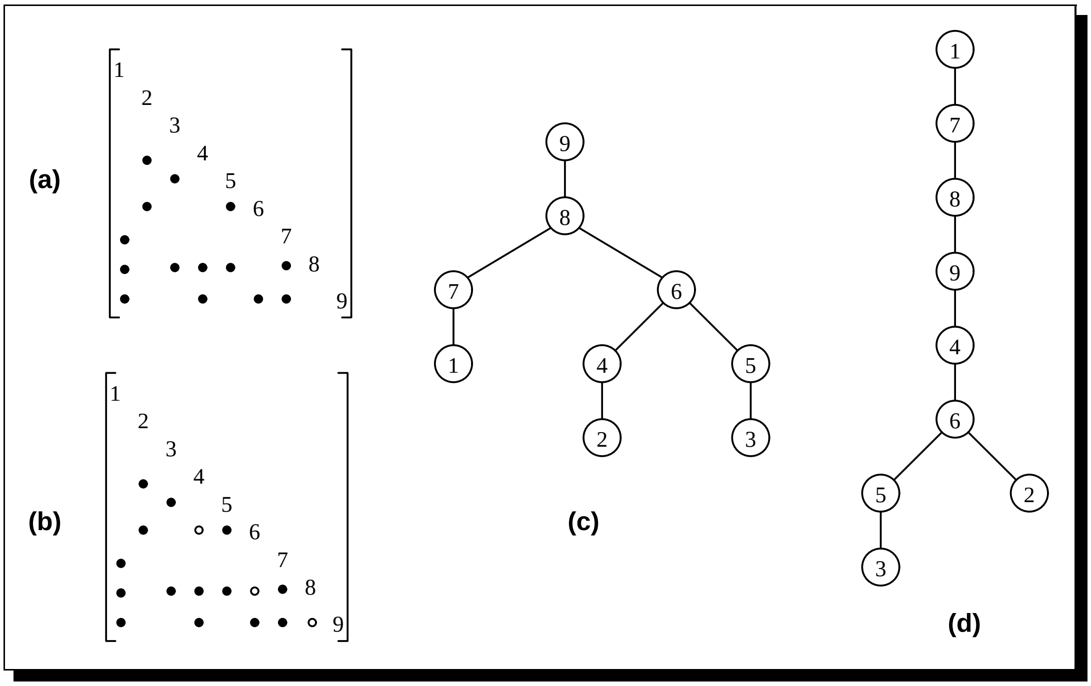

Figure 1: The elimination tree associated with a factor matrix L. (a) shows the matrix M ('$\bullet$' denote nonzero entries), (b) shows the factor matrix L ('$\circ$' denote fill-in), (c) shows the elimination tree corresponding to L, and (d) shows another elimination tree that corresponds to the matrix that has been ordered as follows: $\{3, 5, 2, 6, 4, 9, 8, 7, 1\}$.

Having determined an ordering for M, the next step is to numerically factor M. Depending upon how the nonzero elements of the Cholesky factor are stored and accessed, there are several algorithms to perform Cholesky factorization [20, 25]. They are *row Cholesky*, *column Cholesky*, and *submatrix Cholesky*. Each method has its advantages and disadvantages depending on the memory access pattern, vectorization, and other considerations. The *multifrontal* method [8, 38] is a form of submatrix Cholesky, in which single elimination steps are performed on a sequence of small, dense *frontal matrices*, which are summed to L. The main advantage of multifrontal methods is that the frontal matrices are dense, and therefore during elimination they can efficiently utilize vector processors. Moreover, the localization of memory references in these methods is advantageous in exploiting cache or machines with virtual memory and paging. Our parallel Cholesky factorization algorithm is based on the multifrontal method. In the next section we briefly describe the multifrontal method. For further details the reader should refer to the excellent tutorial by Liu

[38].

3.1 Multifrontal Method

Let M be an $m \times m$ symmetric positive definite matrix and L be its Cholesky factor. Let T be its elimination tree and define $T[i]$ to represent the set of descendants of the node i in the elimination tree T. Consider the jth column of L. Let $i_0, i_1, \ldots, i_r$ be the row subscripts of the nonzeros in $L_{\bullet,j}$ with $i_0 = j$ (*i.e.*, column j has r off-diagonal nonzeros).

The *subtree update matrix* at column j, W_j for M is defined as

$$W_j = -\sum_{k \in T[j]-\{j\}} \begin{pmatrix} l_{j,k} \\ l_{i_1,k} \\ \vdots \\ l_{i_r,k} \end{pmatrix} (l_{j,k}, l_{i_1,k}, \ldots, l_{i_r,k}). \tag{4}$$

Note that W_j contains outer product contributions from those previously eliminated columns that are descendants of j in the elimination tree. The jth *frontal matrix* F_j is defined to be

$$F_j = \begin{pmatrix} m_{j,j} & m_{j,i_1} & \cdots & m_{j,i_r} \\ m_{i_1,j} & & & \\ \vdots & & 0 & \\ m_{i_r,j} & & & \end{pmatrix} + W_j. \tag{5}$$

Thus, the first row/column of F_j is formed from $M_{\bullet,j}$ and the subtree update matrix at column j. Having formed the frontal matrix F_j, the algorithm proceeds to perform one step of elimination on F_j that gives the nonzero entries of the factor of $L_{\bullet,j}$. In particular, this elimination can be written in matrix notation as

$$F_j = \begin{pmatrix} l_{j,j} & 0 \\ l_{i_1,j} & \\ \vdots & I \\ l_{i_r,j} & \end{pmatrix} \begin{pmatrix} 1 & 0 \\ & \\ 0 & U_j \end{pmatrix} \begin{pmatrix} l_{j,j} & l_{i_1,j} & \cdots & l_{i_r,j} \\ & & & \\ 0 & & I & \end{pmatrix}. \tag{6}$$

where $l_{\bullet,j}$ are the nonzero elements of the Cholesky factor of column j. The matrix U_j is called *update matrix* for column j and is formed as part of the elimination step.

In practice, W_j is never computed using Equation 4, but is constructed from the update matrices as follows. Let $c_1, \ldots, c_s$ be the children of j in the elimination tree, then

$$F_j = \begin{pmatrix} m_{j,j} & m_{j,i_1} & \cdots & m_{j,i_r} \\ m_{i_1,j} & & & \\ \vdots & & 0 & \\ m_{i_r,j} & & & \end{pmatrix} \mathbin{\updownarrow\!\!\!\!+} U_{c_1} \mathbin{\updownarrow\!\!\!\!+} \cdots \mathbin{\updownarrow\!\!\!\!+} U_{c_s} \tag{7}$$

where $\mathbin{\updownarrow\!\!\!\!+}$ is called the *extend-add operator*, and is a generalized matrix addition. The extend-add operation is illustrated in the following example. Consider the following two update matrices of M:

$$R = \begin{pmatrix} a & b \\ c & d \end{pmatrix}, \quad S = \begin{pmatrix} p & q & r \\ s & t & x \\ y & z & w \end{pmatrix}$$

where $\{2, 5\}$ is the index set of R (*i.e.*, the first row/column of R corresponds to the second row/column of M, and the second row/column of R corresponds to the fifth row/column of M), and $\{1, 3, 5\}$ is the index set of S. Then

$$R \mathbin{\updownarrow\!\!\!\!+} S = \begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & a & 0 & b \\ 0 & 0 & 0 & 0 \\ 0 & c & 0 & d \end{pmatrix} + \begin{pmatrix} p & 0 & q & r \\ 0 & 0 & 0 & 0 \\ s & 0 & t & x \\ y & 0 & z & w \end{pmatrix} = \begin{pmatrix} p & 0 & q & r \\ 0 & a & 0 & b \\ s & 0 & t & x \\ y & c & z & d+w \end{pmatrix}.$$

Note that the submatrix $U_{c_1} \mathbin{\updownarrow\!\!\!\!+} \cdots \mathbin{\updownarrow\!\!\!\!+} U_{c_s}$ may have fewer rows/columns than W_j, but if it is properly extended by the index set of F_j, it becomes the subtree update matrix W_j.

The process of forming F_j from the nonzero structure elements of column j of M and the updates matrices is called *frontal matrix assembly operation*. Thus, in the multifrontal method, the elimination of each column of M involves the assembly of a frontal matrix and one step of elimination.

4 Parallel Dual Affine Algorithm

In developing a scalable parallel formulation for the dual affine algorithm, a number of issues related to processor's topology, data distribution, and implementation of efficient algorithms for the various steps need to be addressed.

4.1 Cholesky Factorization

As discussed in Section 3, the most computationally demanding step of the dual affine algorithm is the solution of Equation 3 using Cholesky factorization. Therefore, developing an efficient parallel formulation for the Cholesky factorization is essential to the success of the parallel dual affine algorithm.

The performance of any parallel sparse Cholesky factorization algorithm is highly dependent upon the ordering algorithm. Due to sparsity, many columns of the matrix can be factored concurrently; thus, a good ordering must maximize the concurrency in the factorization processes, in addition to minimizing the fill-in [32, 31, 24, 51, 20, 30]. The computation associated with distinct subtrees of a node in the elimination tree can be performed in parallel; thus, the shape of the elimination tree determines the degree of concurrency available in the factorization. The number of tasks that can be performed in parallel increases as the elimination tree becomes more balanced.

The minimum degree (MD) ordering has been a good choice in serial implementations of Cholesky factorization. This was because the fill-in incurred by minimum degree algorithm is usually less than that incurred by the other orderings. However, minimum degree produces orderings that are highly unbalanced, which limits the speedup that can be obtained from parallel Cholesky factorization algorithms such as the parallel multifrontal algorithm we developed.

Several ordering algorithms based on graph partitioning have been developed including minimum degree with constraints [36], spectral nested dissection [53], and Kernighan-Lee [29]. These algorithms operate in the undirected graph $G = (V, E)$ constructed from M as described in Appendix A. The basic step in all these algorithms is the partitioning of G by removing a set of nodes S. The set of removed nodes are numbered last in the ordering (*i.e.*, after the other nodes have been numbered). Furthermore, the nodes in each remaining component are then ordered by recursively applying the same graph partitioning algorithm. In contrast to the minimum degree algorithm, the graph partitioning algorithms take a global view of the matrix; thus, they tend to yield more balanced elimination trees. However, the size of the set S (called the *separator*), affects the amount of fill-in. Low fill-in is achieved if S is small. In most graphs, a separator of order $\sqrt{m}$ is considered to be good. Also, the height of the elimination tree is determined by the relative size of the subgraphs in which the original graph is partitioned. If the subgraphs have roughly the same size, then the height of the elimination tree is minimized [24, 34]. We implemented these three ordering algorithms and evaluated their suitability for our parallel formulation of the multifrontal algorithm for the sparse matrices arising in LP problems (Section 7).

For the numerical Cholesky factorization step we use a highly parallel and scalable formulation of the multifrontal algorithm that we developed recently [19]. A brief description of parallel algorithm is as follows:

Consider a p-processors hypercube-connected computer. Before the beginning of the algorithm, the elimination tree is converted to a binary tree using an algorithm described in [27]. This is a preprocessing step applied before the execution of the parallel dual affine algorithm.

In order to factor the sparse matrix in parallel, portions of the elimination tree are assigned to processors using the standard subtree-to-subcube assignment strategy. With subtree-to-subcube assignment, all p processors in the system cooperate to factor the frontal matrix associated with the root node of the elimination tree. The two subtrees of the root are assigned to subcubes of $p/2$ processors each. Each subtree is further partitioned recursively using the same strategy. Thus, the p subtrees at a depth of $\log p$ levels are each assigned to individual processors. Each processor can process this part of the tree completely independently without any communication overhead. At the end of the local computation, each processor stores the columns of L assigned to it, and the update matrix corresponding to the root of its subtree. Next, pairs of processors (P_{2j} and P_{2j+1} for $0 \leq j < p/2$) perform a parallel extend-add on their update matrices, say Q and R, respectively. At the end of this operation, P_{2j} and P_{2j+1} roughly equally share $Q \mathbin{\#} R$. More precisely, all even columns of $Q \mathbin{\#} R$ go to P_{2j} and all odd columns of $Q \mathbin{\#} R$ go to P_{2j+1}. At the next level, subcubes of two processors each perform a parallel extend-add. Each subcube initially has one update matrix. The matrix resulting from the extend-add on these two update matrices is now split among four processors. To effect this split, all even rows are moved to the subcube with the lower processor labels and all odd rows are moved to the subcube with the higher processor labels. During this process, each processor needs to communicate only once with its counterpart in the other subcube. After this (second) parallel extend-add each of the processors has a block of the update matrix roughly one-fourth the size of the matrix. Note that both the rows and the columns of the update matrix are distributed among the processors in a cyclic fashion which helps in maintaining load balance. Similarly, in subsequent parallel extend-add operations, the update matrices are alternatively split along the columns and rows.

Between two successive extend-add operations, the parallel multifrontal algorithm may perform several steps of dense Cholesky elimination. The communication taking place in this phase is the standard communication in grid-based dense Cholesky factorization.

4.2 Matrix-Matrix and Matrix-Vector Multiplication

In the dual affine algorithm, M is computed as the product of AD^2A^T during the first step of the algorithm shown in Program 2.1. The parallel Cholesky factorization algorithm described in Section 4.1 requires that the matrix M be distributed among the processors according to the subtree-to-subcube assignment of the elimination tree. In this scheme, groups of columns corresponding to the lower part of the tree are assigned to the same processor, and the columns corresponding to the upper part of the elimination tree are shared among the subcubes of processors. This mapping is very different than the mapping used by many standard matrix-matrix and matrix-vector multiplication algorithms. It is well known [30, 3] that parallel matrix-matrix and matrix-vector multiplication algorithms are more efficient when a two-dimensional distribution of the matrix is used. Therefore, we have implemented both the matrix-matrix and the matrix-vector multiplication using a two-dimensional distribution of M (and consequently A). This distribution is described in Section 4.2.1. Section 4.4 describes the data redistribution phase that is used to redistribute M in the form required by the Cholesky factorization.

4.2.1 Data Distribution

We consider the processors of the parallel computer as forming a logical two-dimensional array of size $\sqrt{p} \times \sqrt{p}$ embedded in a hypercube-connected computer.

Matrix M is partitioned among the processors in a two-dimensional fashion. Each processor stores a two-dimensional subblock of M. Since M is sparse, the selection of subblocks to be assigned to each processor must be done so that the work is balanced among the processors. Specifically, M should be distributed in such a way that the number of nonzero elements is roughly the same at each processor. This is crucial because the amount of computation performed during the matrix-matrix multiplication in step 1 depends on the nonzero elements of M assigned to each processor.

A widely used distribution strategy that tends to evenly distribute sparse matrices is the *cyclic mapping* (CM) [30]. In this mapping, element $m_{i,j}$ is assigned to processor ($i \bmod \sqrt{p}, j \bmod \sqrt{p}$). We implemented this mapping. But as p increased, the percentage difference ((max−min)/min) in the amount of nonzero elements of M assigned to each processor increased significantly. Even on $p = 64$, the difference was greater than 90 percent for most problems. The reason for the load imbalance is that M has some dense subblocks along its main diagonal. The size of these subblocks varies, but they remain small. The cyclic mapping assigns elements from these subblocks to the same processors (the ones on the diagonal of the $\sqrt{p} \times \sqrt{p}$ processor grid) and thus creates load imbalances. As a result, in cyclic mapping, the processors along the main diagonal get more work than the processors in the second diagonal. The processors in the second diagonal get more work than those in the third, and so on.

In order to balance the distribution of nonzero elements of M, we developed the following mapping. Element $m_{i,j}$ is mapped to processor $(f[i], g[j])$ where the $1 \times m$ mapping arrays f and g are constructed as follows. Initially, $f[i] = i \bmod \sqrt{p}$ (as in the cyclic mapping). Then we randomly permute the elements of f. A similar procedure is used for g and a different random permutation is obtained. This new mapping is called *random cyclic mapping* (RCM). Since different permutation arrays are used for rows and columns of M, this mapping does not map the diagonal of M to the processors on the diagonal of the processor grid. Like cyclic mapping, this mapping assigns elements to each processor from an equal number of rows and columns of M. This is important because it increases neither the communication overhead nor the memory requirements during matrix-matrix and matrix-vector multiplication. Table 1 shows the percentage difference on the number of nonzero elements of M assigned to each processor for both the cyclic mapping and the random cyclic mapping. From this table we can see that random cyclic mapping distributes the nonzero elements of M much better than cyclic mapping. Note that on small matrices (*e.g.*, the first two matrices in Table 1) and on large number of processors, both mappings lead to significant load imbalances. But in each case, RCM does considerably better than CM.

The remaining data are distributed as follows. Vectors y, z, d_z, d_y, c, b, and x are stored in the processors along the first column of the logical two dimensional processor grid. Elements

Name	m	Mapping	Number of Processors 4	16	64	256	1024
80bau3b	2262	CM	55%	113%	295%	1088%	8600%
		RCM	9%	19%	53%	188%	1200%
bnl12	2324	CM	41%	78%	212%	689%	2967%
		RCM	7%	29%	47%	174%	800%
greenbea	2392	CM	15%	40%	93%	234%	631%
		RCM	3%	22%	43%	117%	492%
cre-d	8927	CM	12%	27%	60%	142%	287%
		RCM	3%	14%	46%	96%	195%

Table 1: Percentage difference in the number of nonzero elements of M assigned to each processor for the cyclic mapping (CM) and the random cyclic mapping (RCM). These differences where computing as the ratio (max−min)/min.

y_i, b_i, and $(d_y)_i$ are stored in processor $(g[i], 0)$, while elements z_i, c_i, x_i, and $(d_z)_i$ are stored in processor $(i \bmod \sqrt{p}, 0)$. The reason for this data mapping will become apparent when we describe the implementation of the matrix-vector multiplication algorithm in Section 4.2.3.

4.2.2 Matrix-Matrix Multiplication

In each iteration of the dual affine algorithm, the matrix $M = AD^2A^T$ needs to be computed. Recall, that D is a diagonal matrix; thus,

$$m_{i,j} = \sum_{k=0}^{n-1} a_{i,k} d_k^2 a_{k,j}^T.$$

In our formulation, matrix M is partitioned among the processors using the random-cyclic mapping described in Section 4.2.1. In order for element $m_{i,j}$ to be computed, the processor responsible for it needs to receive elements of rows i of A and column j of A^T (which is the same as the row j of A) and also the entire vector z (recall $D = Z^{-1}$). Note that A does not change from one iteration to another. So, the time to perform the matrix multiplication will be reduced if each processor stores the rows of A it needs. In our implementation, each processor stores the rows of A that it needs to compute all the $m_{i,j}$ elements of M assigned to it. In random cyclic mapping , each processor contains $m_{i,j}$ elements for $m/\sqrt{p}$ rows and $m/\sqrt{p}$ columns. Hence, each processor has to store at most $2m/\sqrt{p}$ rows of A. Even though this distribution of A requires more memory, the savings in communication time is significant.

In each iteration z changes, and needs to be distributed from the processors of the first column (where it resides) to all the other processors. This can be done in two steps as follows. Each processor in the first column broadcasts its local part of z to all the processors along its row. Now, all the processors along each column contain different parts of z and they perform an all-to-all broadcast of their local part of z. At the end of this step, each processors has the entire z vector.

4.2.3 Matrix-Vector Multiplication

In steps 2 and 6, (Program 2.1), the dual affine algorithm performs a matrix-vector multiplication of the form $u = A^T v$, where v is an m element vector (either d_y or y). We use a two-dimensional partitioning of A^T to perform this operation. Each processor has a block of A^T and it performs a subset of the dot product computations for a group of elements of u [30].

As discussed in Section 4.2.2, each processor stores the rows of A that are required during the multiplication. Specifically, for each element $m_{i,j}$ assigned to it, each processor stores row i and row j of A. The rows of A due to j are essentially the columns of A^T; thus, each processor already stores parts of A^T. In particular, processor $P_{i,j}$ stores all columns k of A^T such that $g[k] = j$. So the columns of A^T are already partitioned among the processors. However, the processors along each column of the processor grid have the same columns of A^T. Now, the processors along each column of the processor grid select elements from their columns of A^T in a cyclic fashion. Note, that the distribution of A^T does not require any additional space. The matrix-vector product is performed as follows.

From Section 4.2.1, we know that vector v (*i.e.*, either d_y or y), is initially distributed among the processors of the first column, so that element v_i goes to processor $P_{g[i],0}$. Processor $P_{i,0}$ sends its part of vector v of size $m/\sqrt{p}$ to processor $P_{i,i}$. Next, processor $P_{i,i}$ sends the just received part of vector v to all the processors along its column by performing an one-to-all broadcast along the columns. Consider processor $P_{i,j}$. This processor stores all columns k of A^T such that $g[k] = j$. Also, $P_{i,j}$ will receive parts of vector v from $P_{j,j}$. Now because of the way v is distributed along the first column of processors, the part of v received by $P_{j,j}$ from $P_{j,0}$ consists of all elements v_l such that $g[l] = j$. Therefore, each processor has the appropriate parts of vector v to perform $n/\sqrt{p}$ partial dot-products. At the end of this step, the partial dot-products are added to processors along the first column by a single-node accumulation. At this point, processors along the first column have $n/\sqrt{p}$ elements of the result vector u distributed in a cyclic fashion.

4.3 The Rest of the Computation

The computation performed in steps 3 through 5 in Program 2.1 is done by the processors at the first column of the processor grid. Z is a diagonal matrix, and its diagonal consists of vector z. Since, vectors z and d_z are already aligned (Section 4.2.1), step 3 involves no communication. In step 4, the minimum of $-z_j/(d_z)_j$ needs to be computed for $(d_z)_j < 0$. This is done in two steps. First each processor finds the minimum among its local $-z_j/(d_z)_j$ values, and then the global minimum value α is found by a global-min reduction operation involving the processors of the first column. This involves a single-node accumulation and a single-node broadcast to distribute α. Finally, in step 5 a vector addition is performed. Since both vectors y and d_y are aligned, this step does not involve any communication.

4.4 Redistribution of M

Before, the parallel multifrontal algorithm for Cholesky factorization of M can be invoked, matrix M needs to be redistributed according to the column assignment imposed by the elimination tree. This can be done efficiently, provided that all processors know where each column of M goes. Since each processor knows which columns of M it stores during the factorization, information about the remaining columns can be obtained by performing an all-to-all broadcast in the beginning of the algorithm. Since this information does not change in each iteration, the communication overhead of this step is incurred only once, and is relatively small.

Now the redistribution of M can be done in $\log p$ steps as follows. In step i for $i = 0, \ldots, \log p - 1$, each processor partitions whatever elements of M it stores into two buckets. One bucket contains elements for columns that need to go to processors whose ID differs in the ith bit and the other bucket to processors whose ID has the same ith bit. Then each processor keeps the second bucket, and sends the first bucket to the processor whose ID differs in the ith bit. At the end of each step, each processor has two buckets, one local and one newly received. It merges the buckets, and goes to the next step. At the end of the $\log p$ steps, each processor has the columns of M assigned to it.

5 Performance Analysis

Precise analysis of the performance of parallel algorithms for the sparse dual affine algorithm is difficult because both A and M are unstructured sparse matrices. However, under some relatively mild assumptions, important information can be derived about the performance of our formulation.

For the rest of this analysis, we use the following definitions.

- p is the number of processors available to solve the problem.
- m is the number of constraints in the LP problem.
- n is the number of variables in the LP problem.
- c_A is the average number of nonzero elements in each column of A.
- r_M is the average number of nonzero elements in each row of M.
- t_c is the time for a unit of computation.
- t_s is the message startup time. This is the time required to setup a message transfer operation. The startup time is independent of the size of the transmitted message.
- t_w is the per word transfer time. This is the time required to send 4 bytes between two adjacent processors. If m words is sent then the transfer time is mt_w.

Furthermore, we will assume that the parallel computer has a hypercube interconnection network and cut-through routing.

Cholesky Factorization

Analyzing, the communication overhead of the algorithm for non-planar graphs is particularly hard. The performance of our parallel Cholesky factorization algorithm has been analyzed in [19]. It is shown there that, for matrices whose corresponding graphs are planar, the time spent by each processor for communication is $\Theta(m/\sqrt{p})$. This overhead is smaller than the overheads of other schemes for parallel Cholesky factorization [39, 40, 1, 54, 55, 62, 15, 14, 23, 20, 59, 65, 47, 18, 57]. The experimental results in [19] suggest that our scheme is superior to other existing schemes even for non-planar graphs. But the communication overhead for non-planar graphs should be somewhat higher than $\Theta(m/\sqrt{p})$. Hence $\Theta(m/\sqrt{p})$ can be taken as a lower bound.

Recall from Section 4.4 that our dual affine algorithm requires a redistribution of the rows of M prior to factorization. This redistribution is done in $\log p$ steps, and in each step half the number of elements of M are sent to a neighbor processor. Therefore, this redistribution takes $\Theta((mr_M/p)\log p)$ time in each iteration. From our experiments, we observed that the redistribution cost is quite small, usually much less than 5 percent of the time spent in factorization by each processor.

Matrix-Matrix Multiplication

In order to compute the parallel run time of our matrix-matrix multiplication algorithm described in Section 4.2.2, we need to compute the time each processor spends in performing multiplications.

The total number of multiplications required to compute AA^T can be computed from the outer-product formulation of AA^T [7], where

$$AA^T = \sum_{k=0}^{n-1} A_{\bullet,k} A^T_{k,\bullet}.$$

In the outer-product formulation, the kth column of A is multiplied with the kth row of A^T. Since each column has c_A nonzero elements, the kth row of A^T is multiplied by a nonzero element c_A times. Since each row of A^T has c_A nonzeros, each outer product requires c_A^2 multiplications. Therefore, a total of $n(c_A)^2$ multiplications are involved in computing AA^T, and it takes $\Theta(nc_A^2)$ time to compute M on a sequential computer. If we assume that the nonzero elements of M are evenly distributed among the p processors, then each processor spends $\Theta(nc_A^2/p)$ time in computing M.

The only communication step involved in the matrix-matrix multiplication algorithm is the broadcast of the vector z. As described in Section 4.2.2 this is done in two steps. The first step takes $\Theta((n/\sqrt{p})\log p)$ time, while the second step takes $\Theta(n)$ time [30]. Therefore, the total run time of a matrix-matrix multiplication is

$$\Theta\left(\frac{nc_A^2}{p}\right) + \Theta\left(\frac{n}{\sqrt{p}}\log p\right) + \Theta(n). \tag{8}$$

Matrix-Vector Multiplication

In the matrix-vector algorithm described in Section 4.2.3, each processor is responsible for computing partial dot-products for $n/\sqrt{p}$ elements. Since each processor has nc_A/p nonzero elements of A on the average, the computation performed by each processor is $\Theta(nc_A/p)$.

Assuming that m is sufficiently large, the point-to-point communication step required by the algorithm takes at most $\Theta(m/\sqrt{p})$ time. The single-node broadcast takes $\Theta((m/\sqrt{p})\log p)$ time. Finally, the single-node accumulation takes $\Theta((n/\sqrt{p})\log p)$ time. Therefore, the total execution time of a matrix-vector multiplication is

$$\Theta\left(\frac{nc_A}{p}\right) + \Theta\left(\frac{m}{\sqrt{p}}\log p\right) + \Theta\left(\frac{n}{\sqrt{p}}\log p\right). \tag{9}$$

6 Test Problems

In all the experiments reported in this paper, we used real linear programming models that are available through NETLIB [13]. Table 2 shows the characteristics of our experimental models. The models shown is this table were pre-processed to remove empty constraints [41].

7 Experimental Evaluation of Orderings

We experimentally evaluated the following ordering algorithms: minimum degree (MD), the minimum degree ordering with constraints [36, 35], which we call minimum degree based nested dis-

Name	Constraints m	Variables n	Nonzeros in A	Nonzeros in AA^T
woodw	1098	8418	37487	20421
cycle	1093	3371	21234	27714
ken-11	14695	21349	70354	33880
d2q06c	2171	5831	33081	26991
pilot	1441	4860	44375	61538
gosh	3790	13451	101876	10117
pilot87	2030	6465	76616	122550
pds-06	9882	28655	82269	39061
cre-d	8927	69980	312626	181670
maros-r7	3136	9408	152690	330472

Table 2: Statistics for the NETLIB problems used to evaluate the performance of the various ordering algorithms and the parallel dual affine algorithm.

Name	MD	MND	KLND	SND
woodw	**47683**	53712	53216	58644
cycle	79083	109138	**77365**	91060
ken-11	**133650**	152074	222168	212109
d2q06c	155351	172086	**120886**	123229
pilot	**192871**	228245	242596	236911
gosh	**267775**	310040	341322	338217
pilot87	**455200**	541250	558701	572659
pds-06	**573263**	688704	988000	1493175
cre-d	856898	**856330**	905755	1131841
maros-r7	**1252577**	1649706	1370906	1357307

Table 3: The number of nonzeros in the Cholesky factor for four ordering algorithms.

section (MND), the spectral nested dissection [53] (SND), and a nested dissection scheme based on Kernighan-Lin's edge separators [29] (KLND).

Table 3 shows the number of nonzero elements in the Cholesky factor L for the test problems of Table 2. The amount of fill-in produced by MD is in most cases smaller than that produced by the other orderings. However, for many problems, the amount of fill-in produced by MND, KLND, and SND is only moderately larger than that of MD. Table 4 shows the number of operations performed during the Cholesky factorization of M for each of the four orderings. For most of the problems, the orderings that have the smaller fill-in also require the smaller number of operations. One notable exception is d2q06c for which SND requires the smallest number of operations even though KLND has the smallest fill-in for this problem. The reason is that the number of operations is the sum of the squares of the nonzero elements in each column of the Cholesky factor L. KLND has some columns that are denser than SND; thus, the square of those adds up to a higher operation count.

In the context of our parallel multifrontal algorithm, the another important way of evaluating orderings is to consider the elimination trees they generate. The elimination trees generated by MD are usually very deep and highly unbalanced. On the other hand, the elimination trees generated by KLND and SND have small height and they are quite balanced. However, the amount of work associated with each node of the elimination tree can vary significantly. Thus, any subtree-to-subcube work assignment algorithm can potentially distribute the work quite unevenly, even if each subtree has the same number of nodes. Thus, for any given ordering, there will be some inherent load imbalance associated with it, and this will put an upper bound on the achievable efficiency, *i.e.*, even if there is no communication overhead, efficiency will not be one. Therefore, a desirable ordering is the one in which the amount of work associated with the subtrees of the elimination tree at the same level is roughly the same.

To evaluate the load balancing properties of the ordering algorithms, we wrote a sequential program that simulates our parallel multifrontal algorithm, and reports the maximum achievable efficiency. This was done because we wanted to study the characteristics of the orderings in the absence of communication overheads incurred by the actual parallel algorithm. Our simulator takes as input the same binary elimination tree as the parallel algorithm and logically distributes the work in the same fashion. The time required to perform the Cholesky factorization T_p is equal to the time required to eliminate the root node of the elimination tree plus the maximum of the times required to eliminate all the nodes in each of its two subtrees. The time to eliminate a subtree is computed in the same fashion. However, when the whole subtree is locally assigned to a single processor, then the time to eliminate the subtree is equal to the sum of the time required to eliminate all the nodes in this subtree. Having computed the time to perform the factorization, the maximum achievable efficiency is computed as the sum of the time required to eliminate all the columns divided by pT_p. Because each nCUBE 2 processor has a total of 16MB of memory, we performed the simulation on a Sun Sparc-2 workstation.

Table 5 shows these efficiencies for the test problems of Table 2. For 16 processors, for most of the problems, MD produced orderings whose efficiency is comparable to the efficiency achieved

Name	MD	MND	KLND	SND
woodw	**3,461,499**	4,301,889	3,657,900	5,170,396
cycle	5,713,494	9,015,082	**4,398,097**	6,468,351
ken-11	**4,074,171**	4,651,280	10,620,980	8,962,257
d2q06c	29,696,436	34,553,561	14,911,941	**12,866,482**
pilot	**42,380,243**	57,822,291	67,974,968	58,684,694
gosh	**51,583,850**	63,173,750	76,440,590	64,244,883
pilot87	**195,776,414**	244,039,517	214,215,718	263,492,967
pds-06	**208,223,868**	302,185,761	495,597,665	1,192,302,692
cre-d	297,383,390	**288,993,512**	312,874,063	498,409,786
maros-r7	**557,860,867**	1,068,417,963	691,770,450	675,704,811

Table 4: The number of operations performed during the Cholesky factorization for four ordering algorithms.

Name	Ordering	Number of Processors 16	64	256	Name	Ordering	Number of Processors 16	64	256
woodw	MD	26.62	20.53	24.41	gosh	MD	31.93	23.23	12.26
	MND	77.57	57.61	55.53		MND	81.67	51.37	43.11
	KLND	76.64	57.35	49.26		KLND	85.05	56.57	50.99
	SND	84.14	73.92	67.21		SND	59.20	27.70	29.02
cycle	MD	58.86	49.58	44.07	pilot87	MD	91.17	81.49	71.02
	MND	55.55	50.79	41.70		MND	82.87	73.72	65.59
	KLND	64.89	58.97	48.45		KLND	80.89	79.69	75.80
	SND	76.59	74.77	65.33		SND	71.51	60.27	58.05
ken-11	MD	57.92	58.49	45.45	pds-06	MD	65.04	47.93	23.92
	MND	45.63	43.13	39.00		MND	49.78	36.32	49.07
	KLND	69.21	66.33	55.53		KLND	76.05	66.09	48.42
	SND	58.26	73.61	55.95		SND	40.97	70.96	60.08
d2q06c	MD	75.28	56.67	49.60	cre-d	MD	58.92	33.37	27.01
	MND	90.90	74.16	80.21		MND	73.62	44.73	23.68
	KLND	79.50	76.93	47.92		KLND	82.86	65.39	48.51
	SND	68.60	62.73	53.76		SND	72.91	55.90	43.51
pilot	MD	66.55	55.74	50.42	maros-r7	MD	83.77	74.59	64.94
	MND	69.54	62.96	63.27		MND	76.13	75.20	75.24
	KLND	94.11	89.64	89.24		KLND	83.74	77.05	77.42
	SND	88.06	71.92	60.88		SND	77.23	78.50	76.10

Table 5: Maximum achievable efficiency of the parallel multifrontal algorithm due to load imbalance. The efficiencies were computed by simulation on a Sun Sparc-2 workstation.

by the other ordering algorithms. However, as the number of processors increases, the maximum achievable efficiency of MD decreases dramatically. For example, for gosh an efficiency of only 12.26 can be obtained when 256 processors are used. However, the efficiency achieved by the other three orderings is usually higher and does not decrease considerably as p increases.

It should be clear from the results shown in Tables 3 and 5, that MND, KLND, and SND improve the maximum achievable efficiency at the expense of a slight to moderate increase in the amount of fill-in. In many cases, the efficiency improvement outweighs the penalty due to increased fill-in and the additional computation associated with it. As the experimental results presented in Section 8 indicate, in most cases the increased load balance reduces the overall execution time as p increases; thus, increasing the speedup obtained by using a parallel computer.

8 Experimental Results of Parallel Dual Affine Algorithm

We implemented the dual affine algorithm on the nCUBE 2 parallel computer, a hypercube-connected message-passing system. Each processor of nCUBE 2, is rated at 2MFlop (double precision). The message startup time t_s (*i.e.*, the time for sending zero bytes) is roughly 180μs . The channel bandwidth is 2Mbytes/sec; hence, t_w (the time it takes to send a word of four bytes to a neighbor processor) is 2.0μs . Note that the time to send a message of size k words between nearest neighbors is $t_s + kt_w$.

Table 6 shows the run time of our parallel dual affine algorithm on 1, 16, 64, and 256 processors for the four orderings discussed in Appendix A. The run times shown are for a single iteration of the algorithm. Since the computation performed at each iteration is the same, the reported run times are accurate indicators of the performance of the overall parallel interior point algorithm.

		Number of Processors						Number of Processors			
Name	Ordering	1	16	64	256	Name	Ordering	1	16	64	256
woodw	MD	20.91	3.34	1.58	0.83	gosh	MD	124.52	34.68	13.08	7.80
	MND	22.46	2.33	1.16	0.71		MND	139.89	12.86	6.51	3.09
	KLND	21.33	2.10	0.98	0.61		KLND	153.16	13.76	6.26	2.70
	SND	24.00	2.61	1.32	0.82		SND	137.99	24.33	15.68	5.28
cycle	MD	13.71	2.30	1.25	0.82	pilot87	MD	360.81	28.52	11.68	5.76
	MND	23.57	3.36	1.69	1.02		MND	442.71	33.50	13.08	5.87
	KLND	12.21	1.29	0.68	0.43		KLND	380.28	29.61	10.90	5.24
	SND	15.51	2.55	1.25	0.80		SND	444.76	34.87	13.48	6.24
ken-11	MD	22.10	2.20	0.97	0.47	pds-06	MD	324.03	37.13	18.66	12.59
	MND	23.52	2.05	0.80	0.45		MND	350.23	48.92	25.24	7.05
	KLND	31.35	2.71	0.96	0.59		KLND	722.31	58.92	22.61	10.35
	SND	27.32	2.50	0.92	0.48		SND	2017.32	190.29	35.24	14.66
d2q06c	MD	50.82	5.99	3.35	1.64	cre-d	MD	546.21	57.95	27.20	11.36
	MND	57.37	5.75	2.70	1.38		MND	587.88	50.92	21.22	9.30
	KLND	28.84	3.32	1.59	1.04		KLND	560.82	45.88	18.08	7.60
	SND	25.94	2.89	1.62	0.97		SND	741.83	68.82	21.41	10.03
pilot	MD	91.78	9.90	4.18	2.24	maros-r7	MD	890.26	102.78	52.87	20.04
	MND	113.37	12.83	5.23	2.48		MND	1437.82	99.56	28.80	11.76
	KLND	129.59	10.15	4.50	2.02		KLND	989.32	77.67	22.85	8.98
	SND	127.59	10.13	4.25	2.17		SND	980.73	67.30	20.00	8.23

Table 6: Execution time in seconds for one iteration of the parallel dual affine interior point algorithm. The run times were obtained on an nCUBE2 parallel computer.

Table 7 shows the actual speedup achieved by the parallel system. These speedups were computed by dividing the best sequential run time among the four orderings, with the run time of the particular ordering. That is, the speedup for orderings i is computed as

$$S_i = \frac{\text{Minimum of the serial run time of all orderings}}{\text{Parallel run time for ordering } i} \tag{10}$$

For instance, the speedup of 9.0 for woodw on 16 processors using MND was obtained by dividing 20.91 (the best sequential run time for this problem) with 2.33 (the run time for MND when 16 processors are used). An alternative method is to compute speedup for an ordering as the ratio of the serial run time and the parallel run time of the same ordering. The speedups computed via this method will be higher than those computed using Equation 10. However, only Equation 10 gives an accurate measure of the performance improvement due to parallelization.

The test problems shown in Table 6 can be classified into two groups according to their size. The problems in the first group are fairly small. For these problems, the number of nonzeros in the Cholesky factor ranges from 47000 to 135000. Furthermore, the serial execution time for all these problems is less than one minute. For problems of these size, even sequential interior point algorithms may not be the best way to solve them, and the simplex method may be faster [41]. Nevertheless, speedup in the range of 16–27 is obtained for all of them on 64 processors.

Name	Ordering	Number of Processors 16	64	256
woodw	MD	6.3	13.2	25.2
20.91	MND	9.0	18.0	29.5
	KLND	10.0	21.3	**34.3**
	SND	8.0	15.8	25.5
cycle	MD	5.3	9.8	14.9
12.21	MND	3.6	7.2	12.0
	KLND	9.5	18.0	**28.4**
	SND	4.8	9.8	15.3
ken-11	MD	11.2	27.6	47.0
22.10	MND	10.8	27.6	**49.1**
	KLND	8.2	23.0	37.5
	SND	8.9	24.0	46.1
d2q06c	MD	4.3	7.7	15.8
25.24	MND	4.5	9.6	18.8
	KLND	7.8	16.3	24.9
	SND	9.0	16.0	**26.7**
pilot	MD	9.3	22.0	41.0
91.78	MND	7.2	17.5	37.0
	KLND	9.0	20.4	**45.3**
	SND	9.1	21.6	42.3

Name	Ordering	Number of Processors 16	64	256
gosh	MD	3.6	9.5	16.0
124.52	MND	9.7	19.1	40.3
	KLND	9.1	19.9	**46.1**
	SND	5.1	7.9	23.6
pilot87	MD	12.7	30.9	62.6
360.81	MND	10.8	27.6	61.5
	KLND	12.2	33.1	**68.9**
	SND	10.3	26.8	57.8
pds-06	MD	8.7	17.4	25.7
324.03	MND	6.6	12.8	**46.0**
	KLND	5.5	14.3	31.3
	SND	1.7	9.2	22.1
cre-d	MD	9.4	20.1	48.1
546.21	MND	10.7	25.7	58.7
	KLND	11.9	30.2	**71.8**
	SND	7.9	25.5	54.4
maros-r7	MD	8.7	17.2	44.4
890.26	MND	8.9	30.9	75.7
	KLND	11.5	39.0	99.1
	SND	13.2	44.5	**108.2**

Table 7: Actual speedup obtained by our parallel interior point algorithm for the four orderings on the different LP problems. The number under the name of each model is the sequential run time using the best ordering algorithm.

The problems in the second group have matrices in which the number of nonzeros in the Cholesky factor ranges from 190000 to 1250000. For these problems, speedups are in the range of 45-108 on 256 processors. In particular for maros-r7, a speedup of 108 was obtained on 256 processors. To our knowledge, no previous work exists in parallel interior point algorithms, where speedup that high was obtained for general LP problems. However, the size of even these problems is relatively small compared to the problems for which sequential interior point algorithms excel. Significantly larger problems (more than 5-100 million nonzeros in the Cholesky factor) are solved using interior point methods [41]. For these bigger problems, we expect to obtain significantly better speedup.

From our experiments we have seen that there are two sources of overhead in our parallel Cholesky factorization. The first overhead is due to load imbalance that can limit the maximum achievable efficiency. As discussed in Section 7, MND, KLND, and SND produce orderings that are significantly more balanced than MD, but normally have a higher amount of fill-in. The overall performance of an ordering is determined by the interplay between load imbalance and fill-in, as illustrated by the following example. Table 8 combines information from Tables 4, 5, and 6 for the model gosh. From this table we see that MD requires the smallest amount of computation to factor M, which is reflected in its smallest serial execution time. For this problem, KLND has the highest operation count, and thus the highest sequential run time. Now consider the case for $p = 16$. KLND has the maximum achievable efficiency due to load imbalance (85%). However, the

		Maximum Achievable Efficiency Due to Load Imbalance Number of Processors			Run Time Number of Processors			
Ordering	Operation Count	16	64	256	1	16	64	256
MD	51,583,850	32%	23%	12%	124.52	34.68	13.08	7.80
MND	63,173,750	82%	51%	43%	139.89	12.86	6.51	3.09
KLND	76,440,590	85%	57%	51%	153.16	13.76	6.26	2.70
SND	64,244,883	59%	28%	29%	137.99	24.33	15.68	5.28

Table 8: Consolidated information about gosh.

run time of MND is the smallest (12.86 seconds). Despite higher fill-in, MND performs better than MD because its upper bound on efficiency due to load imbalance is 82% compared to only 32% for MD. KLND performs worse (higher run time) than MND, because KLND performs 20% more computation whereas it is only slightly more balanced than MND. The moderately higher load balance for KLND is not sufficient to offset the extra computation. For $p = 64$, KLND performs better than MND. In this case, the extra efficiency gained by using 64 processors was significant to offset the increase in the operation count. This becomes more evident for $p = 256$, in which case, KLND performs significantly better than MND. From the above discussion we see that: (a) better load balance can be achieved at the expense of increase in the computation required to factor M, and (b) as p increases, this increase in the sequential run time is compensated by the increased concurrency, and the factorization takes less time.

The second source of overhead is due to communication. In particular, our parallel multifrontal algorithm incurs two types of communication overhead. The first is due to the extend-add operation and the second is due to the elimination steps performed on the frontal matrix. From our experiments we found that due to high message startup time on nCUBE 2, the communication overhead due to the elimination steps is significantly higher and reduces overall efficiency. The performance of dense Cholesky for each elimination step of a frontal matrix of size $k \times k$ depends on the relative values of t_s, t_w, and t_c. In each step of parallel elimination, a processor performs $(k^2/p)t_c$ computation, and sends data of size $k/\sqrt{p}$ at the communication cost of $t_s + (k/\sqrt{p})t_w$. Because, the size of these dense frontal matrices is small (k is less than 200 in the problems used in our experiments), on nCUBE 2, t_s is significant higher than the data transmission time $(k/\sqrt{p})t_w$ and the time spent in computation $(k^2/p)t_c$. If t_s is small (as in transputers, Cray T3D), then the communication overhead will be reduced significantly.

From the results in Table 7 we note that KLND performs significantly better than the other orderings for most problems. Even for problems when KLND is not the best ordering, it is not much worse than the best scheme for these problems. However, determination of the best ordering may be domain dependent, and requires a more extensive study.

9 Conclusion

In this paper we described a parallel formulation for the dual affine interior point algorithm. Since the type of operations performed by the primal-dual algorithm are similar to the dual affine algorithm, we expect the parallel primal-dual algorithm to exhibit performance similar to the dual affine algorithm. The preliminary experimental results presented here show that our algorithm achieves substantial speedups on moderate-size problems. Speedup would have been higher if the message startup time (t_s) was smaller. On architectures with smaller t_s, such as CM-5 (with active messages) and Cray T3D we expect the speedup to be significantly better. To evaluate the quality of our serial implementation we compared the performance obtained by our interior point code to that of LOQO [63], running on a Sun Sparc-2 workstation. LOQO is a publicly available solver for linear programming problems based on interior point methods. Our 256-processor nCUBE 2 implementation is 8 times faster for woodw, 21 times faster for pilot87, and 44 times faster for maros-r7.

Even though, the parallel formulation has been developed on a hypercube-connected computer, the algorithm can be easily ported to architectures like CM-5 (whose fat-tree network provides the bisection width equivalent to that of a hypercube) and Cray-T3D (which has substantial bisection width). We are currently working on developing ports to these architectures. We expect to obtain much higher overall performance on these architectures as they have higher channel bisection width and low message startup times (5μs on CM-5 with active messages, and 2μs on Cray-T3D) and much faster CPUs (100MFlop peak on CM-5 and 150MFlop peak on T3D).

From the results presented in this paper, it is also clear that ordering plays a significant role in determining the overall performance of the dual affine algorithm. The choice of best ordering may be problem dependent and the best ordering for problems arising from finite element/finite difference computation may be different than those for LP problems. Among the orderings we evaluated in this paper, it seems that KLND does consistently better than either MD, MND, and SND for the LP problems we tested. However, we believe that more research is needed to determine the best ordering. In the current implementation, we computed the ordering as a preprocessing step, on a serial computer. Since we are able to get good speedup on the Cholesky factorization (which is done in each iteration), the time spent in ordering (which is done only once) can start dominating the time to solve the entire LP problem. Hence, there is a need to develop parallel formulations for the ordering algorithms.

Acknowledgments

We would like to thank Dr. Jonathan Eckstein for his guidance in interior point methods. We are also grateful to Dr. Sartaj Sahni for providing access to a 64-processors nCUBE 2 at the University of Florida, and Sandia National Labs for providing access to their 1024-processor nCUBE-2.

References

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

[2] M. S. Bazaraa, J. J. Jarvis, and H. D. Sherali. *Linear Programming and Network Flows.* Wiley, 1990.

[3] D. P. Bertsekas and J. N. Tsitsiklis. *Parallel and Distributed Computation: Numerical Methods.* Prentice-Hall, Englewood Cliffs, NJ, 1989.

[4] R. H. Bisseling, T. M. Doup, and L. Loyens. *A parallel Interior Point algorithm for linear programming on a network of transputers. Annals of Operations Research*, 43:51–86, 1993.

[5] I. C. Choi, C. L. Monma, and D. F. Shanno. *Further Development of a Primal-Dual Interior Point Method. ORSA Journal on Computing*, 2(4):304–311, Fall 1990.

[6] G. B. Dantzig. *Linear Programming and Extensions.* Princeton University Press, Princeton, NJ, 1963.

[7] I. S. Duff, M. Erisman, and J. K. Reid. *Direct Methods for Sparse Matrices.* Oxford University Press, Oxford, UK, 1990.

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

[9] J. Eckstein, L. C. Polymenakos, R. Qi, V. I. Ragulin, and S. A. Zenios. *Data-Parallel Implementations of Dense Linear Programming Algorithms.* Technical Report TMC-230, Thinking Machines Corporations, Cambridge, MA, 1992.

[10] J. Eckstein. *Large-Scale Parallel Computing, Optimization, and Operations Research: A survey. ORSA CSTS Newsletter*, 14(2), Fall 1993.

[11] S.-C. Fang and S. Puthenpura. *Linear Optimization and Extensions.* Prentice Hall, Englewood Cliffs, NJ, 1993.

[12] K. A. Gallivan, R. J. Plemons, and A. H. Sameh. *Parallel Algorithms for Dense Linear Algebra Computations.* In *Parallel Algorithms for Matrix Computations*, pages 1–82. SIAM, 1990.

[13] D. M. Gay. *Electronic Mail Distribution of Linear Programming Test Problems. ¡mathematical Programming Society COAL Newsletter*, December 1985.

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

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

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

[17] A. George and J. W.-H. Liu. *The evolution of the minimum degree ordering algorithm. SIAM Review*, 31(1):1–19, March 1989.

[18] J. R. Gilbert and R. Schreiber. *Highly Parallel Sparse Cholesky Factorization. SIAM Journal on Scientific and Statistical Computing*, 13:1151–1172, 1992.

[19] A. Gupta and V. Kumar. *A Scalable Parallel Algorithm for Sparse Matrix Factorization.* Technical Report 94-19, Computer Science Department, University of Minnesota, April 1994.

[20] M. T. Heath, E. Ng, and B. W. Payton. *Parallel Algorithms for Sparse Linear Systems. SIAM Review*, 33(3):420–460, 1990.

[21] H. F. Ho, G. H. Chen, S. H. Lin, and J. P. Sheu. *Solving Linear Programming on Fixed-Size Hypercubes.* In *International Conference on Parallel Processesing*, pages 112–116, 1988.

[22] E. C. Housos, C. C. Huang, and J.-M. Liu. *Parallel Algorithms for the AT&T KORBX System. AT&T Technical Journal*, 68(3):37–47, 1989.

[23] L. Hulbert and E. Zmijewski. *Limiting communication in parallel sparse Cholesky factorization. SIAM Journal on Scientific and Statistical Computing*, 12(5):1184–1197, September 1991.

[24] J. Jess and H. Kees. *A data structure for parallel L/U decomposition. IEEE Transactions on Computers*, C-31:231–239, 1982.

[25] H.-W. Jung, R. E. Marsten, and M. J. Saltzman. *Numerical Factorization Methods for Interior Point Algorithms. ORSA Journal on Computing*, 6(1):94–105, Winter 1994.

[26] N. Karmarkar. *A New Polynomial-Time Algorithm for Linear Programming. Combinatorica*, 4(8):373–395, 1984.

[27] G. Karypis, A. Gupta, and V. Kumar. *Ordering and load balancing for parallel factorization of sparse matrices*. Technical Report (in preparation), Department of Computer Science, University of Minnesota, Minneapolis, MN, 1994.

[28] G. Karypis and V. Kumar. *Performance and Scalability of Parallel Formulations of the Simplex Method for Dense Linear Programming Problems*. Technical report, Computer Science Department, University of Minnesota, Minneapolis, MN, April 1994.

[29] B. W. Kernighan and S. Lin. *An efficient heuristic procedure for partitioning graphs. The Bell System Technical Journal*, 1970.

[30] V. Kumar, A. Grama, A. Gupta, and G. Karypis. *Introduction to Parallel Computing: Design and Analysis of Algorithms*. Benjamin/Cummings Publishing Company, Redwood City, CA, 1994.

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

[32] J. W.-H. Liu. *Reordering sparse matrices for parallel elimination. Parallel Computing*, 11:73–91, 1989.

[33] J. W.-H. Liu. *The Role of Elimination Trees in Sparse Factorization. SIAM Journal on Matrix Analysis and Applications*, 11:134–172, 1990.

[34] J. W. H. Liu. *Equivalent Sparse Matrix Reorderings by Elimination Tree Rotations. Siam Journal on Scientific and Statistical Computing*, 9(3):424–444, 1988.

[35] J. W. H. Liu. *A Graph Partitioning Algorithm by Node Separators. ACM Transactions on Mathematical Software*, 15(3):198–219, September 1989.

[36] J. W. H. Liu. *The Minimum Degree Ordering With Constraints. SIAM J. Sci. Stat. Comput.*, 10(8):1136–1145, November 1989.

[37] J. W. H. Liu. *The Role of Elimination Trees in Sparse Factorization. SIAM J. Matrix Analysis and Applications*, 11(1):134–172, January 1990.

[38] J. W. H. Liu. *The Multifrontal Method for Sparse Matrix Solution: Theory and Practice. SIAM Review*, 34(1):82–109, 1992.

[39] R. F. Lucas. *Solving planar systems of equations on distributed-memory multiprocessors*. PhD thesis, Department of Electrical Engineering, Stanford University, Palo Alto, CA, 1987. Also see *IEEE Transactions on Computer Aided Design*, 6:981–991, 1987.

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

[41] I. J. Lustig, R. E. Marssten, and D. F. Shanno. *Interior Point Methods for Linear Programming: Computational State of the Art. ORSA Journal on Computing*, 6(1):1–14, Winter 1994.

[42] I. J. Lustig, R. E. Marsten, and D. F. Shanno. *Computational experience with a primal-dual interior-point method for linear programming. Journal of Linear Algebra and Its Applications*, 152:191–222, 1991.

[43] I. J. Lustig, R. E. Marsten, and D. F. Shanno. *On Implementing Mehrotra's Predictor-Corrector Interior-Point Method for Linear Programming. SIAM J. Optimization*, 2(3):435–449, August 1992.

[44] R. E. Marsten, M. J. Saltzman, D. F. Shanno, G. S. Pierce, and J. F. Ballintizn. *Implementation of a Dual Affine Interior Point Algorithm for Linear Programming. ORSA Journal on Computing*, 1(4):287–297, 1989.

[45] K. A. McShane, C. L. Monma, and D. Shanno. *An Implementation of a Primal-Dual Interior Point Method for Linear Programming. ORSA Journal on Computing*, 1(2):70–83, Spring 1989.

[46] S. Mehrotra. *On the Implementation of a Primal-Dual Interior Point Method. SIAM J. Optimization*, 2(4):575–601, November 1992.

[47] M. Mu and J. R. Rice. *A grid-based subtree-subcube assignment strategy for solving partial differential equations on hypercubes. SIAM Journal on Scientific and Statistical Computing*, 13(3):826–839, May 1992.

[48] K. Onaga and H. Nagayasu. *A Wavefront-Driven Algorithm for Linear Programming on Dataflow Processor-Arrays.* In *Proceedings of International Computer Symposium*, pages 739–746, 1984.

[49] C. H. Papadimitriou and K. Steiglitz. *Combinatorial Optimization.* Prentice Hall, Englewood Cliffs, NJ, 1982.

[50] B. N. Parlett, H. Simon, and L. M. Stringer. *On Estimating the Largest Eigenvalue With the Lanczos Algorithm. Mathematics of Computation*, 38(157):153–165, January 1982.

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

[52] A. Pothen and C.-J. Fan. *Computing the block triangular form of a sparse matrix. ACM Transactions on Mathematical Software*, 1990.

[53] A. Pothen, H. D. Simon, and K.-P. Liou. *Partitioning Sparse Matrices With Eigenvectors of Graphs. SIAM J. on Matrix Analysis and Applications*, 11(3):430–452, 1990.

[54] A. Pothen and C. Sun. *Distributed multifrontal factorization using clique trees.* In *Proceedings of the Fifth SIAM Conference on Parallel Processing for Scientific Computing*, pages 34–40, 1991.

[55] R. Pozo and S. L. Smith. *Performance evaluation of the parallel multifrontal method in a distributed-memory environment.* In *Proceedings of the Sixth SIAM Conference on Parallel Processing for Scientific Computing*, pages 453–456, 1993.

[56] D. Rose. *A graph-theoretic study of numerical solution of parse symmetric positive definite systems of linear equations.* In *Graph Theory and Computing.* Academic Press, 1972.

[57] E. Rothberg and A. Gupta. *An efficient block-oriented approach to parallel sparse Cholesky factorization.* In *Supercomputing '92 Proceedings*, 1992.

[58] M. J. Saltzman. *Implementation of an Interior Point LP Algorithm on a Shared-Memory Vector Multiprocessor.* In R. Sharda, O. Balci, and S. A. Zenios, editors, *Computer Science and Operations Research: New Developments in their Interface.* Pergamon Press, 1992.

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

[60] W. Shu and M.-Y. Wu. *Sparse Implementation of Revised Simplex Algorithm on Parallel Computers.* In *SIAM Conference on Parallel Processing for Scientific Computing*, March 1993.

[61] C. B. Stunkel and D. A. Reed. *Hypercube Implementation of the Simplex Algorithm.* In *Proceedings of the Third Conference on Hypercube Concurrent Computers and Applications*, volume ——, pages 1473–1482, 1988.

[62] C. Sun. *Efficient parallel solutions of large sparse SPD systems on distributed-memory multiprocessors.* Technical report, Department of Computer Science, Cornell University, Ithaca, NY, 1993.

[63] R. J. Vanderbei. *LOQO User's Manual.* Princeton University, Princeton, NJ, November 1992.

[64] R. J. Vanderbei. *ALPO: Another Linear Program Optimizer. ORSA Journal on Computing*, 5(2):134–146, Spring 1993.

[65] S. Venugopal and V. K. Naik. *Effects of partitioning and scheduling sparse matrix factorization on communication and load balance.* In *Supercomputing '91 Proceedings*, pages 866–875, 1991.

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

Appendix A Orderings

The algorithms for ordering sparse matrices work on the undirected graphs that are constructed from the coefficient matrix $M = (m_{i,j})$. A given $m \times m$ symmetric matrix, can be structurally represented by a graph $G_M = (V_M, E_M)$, where the nodes (vertices) in V_M correspond to rows of M and there is an edge $(i,j) \in E_M$ if the element $m_{i,j}$ is nonzero [56, 33]. In view of this graph model, the process of computing a permutation matrix P can be viewed as selecting the order in which nodes of G_M can be eliminated during the factorization process.

A.1 Minimum Degree

The most successful and widely applicable algorithm developed to date for limiting fill-in in sparse Cholesky factorization is the minimum degree algorithm [16, 20, 17]. Let G be the undirected graph associated with a symmetric matrix M and let u be a node of G. Let $Adj_G(u)$ be the set of nodes adjacent to u in G. The degree of the node u in G will be denoted by $degree_G(u)$, which is simply $|Adj_G(u)|$. The basic minimum degree algorithm can be best described in terms of elimination graphs [56, 33]. Let G_u be the elimination graph obtained after the elimination of the node u from G. The graph G_u can be obtained by deleting the node u and its incident edges from G and then adding edges to make the nodes that were adjacent to u into a clique. This last step essentially adds the fill-in generated by eliminating node u into the graph G_u. These edges are called fill edges.

At each step of the elimination process, the minimum degree algorithm selects as the next node to be eliminated a node of minimum degree in the current elimination graph. After that, a new elimination graph is constructed to reflect the elimination of the node with minimum degree. For more details about the minimum degree algorithm the reader should refer to [17, 30]. Over the years, a variety of modifications have been developed for the minimum degree algorithm that greatly improved its performance [17].

Even though, the minimum degree algorithm does well in minimizing the fill-in, the shape of the generated elimination tree is highly unstructured. This is because, being a bottom-up approach in nature, it lacks information to balance the elimination tree. Algorithms based on elimination tree reorderings have been proposed to balance the shape of the elimination tree generated by minimum degree [24, 34]. These algorithms reduce the height of the elimination tree without increasing the fill-in the the Cholesky factor L.

A.2 Minimum Degree-Based Nested Dissection

Liu [35] proposed a nested dissection ordering algorithm that selects the separators using the minimum degree algorithm. We refer to this algorithm as *minimum degree-based nested dissection* or MND. The algorithm selects a separator as follows: A minimum degree ordering is obtained for G and let $x_1, x_2, \ldots, x_n$ be the ordering obtained from the minimum degree algorithm. Consider a node x_j in this sequence. Let C_j be the connected component in the subgraph $\{x_1, x_2, \ldots, x_j\}$ that contains the node x_j. From [33] we have that the adjacent set of x_j at elimination is given by $Adj_G(C_j)$. If the component C_j satisfies $|C_j| + |Adj_G(C_j)| < n$, then $Adj_G(C_j)$ forms a separator.

Unless the graph is dense, many separators of this form can be found from a minimum degree ordering. Furthermore, the node separator $Adj_G(C_j)$ partitions the graph in such a way so one of the subgraphs is C_j.

The question is which node x_j from the minimum degree sequence to select in order to construct the node separator. A number of criteria were proposed in [35]. In the experiments reported in Section 8 we used the following two criteria.

1. A node x_j is selected if

$$|C_j| = \max\{|C_j| : |C_j| <= n/2 - deg(x_j)\}.$$

   Since C_j forms one of the connected components, the above criterion guarantees that one component contains less than half of the nodes remaining after removing the $Adj_G(C_j)$. Because, $|C_j|$ is maximized over all nodes x_j, this methods selects a separator whose removal creates a component as close to half the number of nodes as possible.

2. A node x_j is selected if the absolute value of the difference between $|C_j|$ and $n - |C_j| - |Adj_G(C_j)|$ is minimized. This has the desirable effect of maintaining some balance in size between the two partitions.

Since a minimum degree ordering is used to select the separators, MND leads selects small separators which leads to low fill-in. Our experiments have shown that the fill-in of MND is comparable to that of MD. However, as the results in Section 8 show MND suffers from elimination tree imbalances.

A.3 Spectral Nested Dissection

Pothen *et al.* [53] have developed an ordering algorithm based on a graph partitioning strategy that uses a specific eigenvector of the Laplacian matrix of G. This algorithm is called *spectral nested dissection* SND.

The SND algorithm computes a separator by finding the eigenvector associated with the 2nd largest eigenvalue λ_2 of the Laplacian matrix $Q = D - V$, where V is the adjacency matrix of G with $b_{i,i} = 0$ and D is a diagonal matrix with $d_{i,i}$ being the degree of node v_i in G.

Let y be the eigenvector corresponding to λ_2. This vector can be used to partition the graph G as follows. Let r be a real number such that $\min_i\{y_i\} < r < \max_i\{y_i\}$. Define $V_1 = \{v_i | y_i < r\}$ and $V_2 = \{v_i | y_i > r\}$. The nodes v_i of G with $y_i = r$ are placed in either V_1 or V_2 so that the absolute value of $|V_1 - |V_2|$ is minimized. Having constructed the sets V_1 and V_2, then the separator is computed from the edge separator between V_1 and V_2.

The (node) separator is computed from the edge separator E_1 as follows. Let V_1' be the set of vertices of V_1 adjacent to some vertex in V_2, and let V_2' be the set of vertices in V_2 adjacent to some vertex in V_1. Let $H = (V_1', V_2', E_1)$ be the bipartite graph induced by the vertex sets V_1' and V_2'. A minimum vertex cover S of H is found by a maximal matching algorithm [52]. Let $V_1^s \subseteq V_1'$ and $V_2^s \subseteq V_2'$ be the vertices in the minimum set cover. Then the node separator S is $S = V_1^s \cup V_2^s$ that separates G into the subgraphs $V_1 \setminus V_1^s$ and $V_2 \setminus V_2^s$.

The key question is how the value of r is determined in order to create the sets V_1 and V_2. Pothen *et al.* [53] suggests that r can be the median value of the components of the eigenvector y. In that case, the sizes of the sets V_1 and V_2 are equal (off by one element at most). Thus, leading to an ordering whose elimination tree is quite balanced in terms of nodes and has small height.

However, even if the sizes of the connected components generated after a single dissection have the same number of nodes, they may represent significantly different amounts of computation.

In the experiments reported in Section 8 we have also implemented a different heuristic for selecting r. This value is selected so that the rows of A belonging to V_1 and V_2 have the same number of non-zero elements. The motivation behind this heuristic is that by balancing the number of non-zeros in the two sets might lead to an overall balancing in the computation.

The eigenvector y is computed using the Lanczos algorithm [50]. This algorithm is iterative and the number of iterations required depends on the desired accuracy. In our experiment we observed that a two-to-three digit accuracy was sufficient in obtaining good separators, and increasing the accuracy increased the number of iterations without significant reduction in the size of the separator.

A.4 Kernighan-Lin Nested Dissection

The Kernighan-Lin algorithm [29] is a graph partitioning algorithm that partitions the graph by finding small edge separators. The Kernighan-Lin algorithm works as follows: Suppose the vertices of G are initially partitioned into two equal-sized sets, $\mathcal{A}$ and $\mathcal{B}$, in some manner. The edges between vertices in $\mathcal{A}$ and in $\mathcal{B}$ are called *external* edges, and all the other edges are called *internal* edges. Let T be the number of external edges. Kernighan-Lin's algorithm reduces T by repeatedly swapping equal-sized subsets of $\mathcal{A}$ and $\mathcal{B}$. It selects these subsets to guarantee that T decreases at each iteration of the algorithm. The subsets to be swapped are selected as follows. Let v be a vertex, and define the *external cost* E_v of v to be the number of its incident external edges. Similarly, define the *internal cost* I_v of a vertex v to be the number of its incident internal edges. Let $D_v = E_v - I_v$.

If we swap $v \in \mathcal{A}$ and $u \in \mathcal{B}$, then we can update T by subtracting from it the quantity

$$g = D_v + D_u - 2a_{u,v}$$

where $a_{u,v} = 1$ if there is an edge between v and u, and zero otherwise. The quantity g is called the *gain* in swapping v and u. Swapping u and v may alter the D values of other vertices incident on v and u. These D values can be recalculated as follows.

$$\begin{aligned} D'_x &= D_x + 2a_{x,v} - 2a_{x,u}, \quad x \in \mathcal{A} - \{v\} \\ D'_y &= D_y + 2a_{y,v} - 2a_{y,u}, \quad y \in \mathcal{B} - \{u\} \end{aligned}$$

Using these definitions we can state the Kernighan-Lin algorithm as follows. First unmark all the vertices of G and compute their initial D values with respect to the current partition, $\mathcal{A}$ and $\mathcal{B}$. Then locate two unmarked vertices $v \in \mathcal{A}$ and $u \in \mathcal{B}$, that would produce the largest gain if swapped. Do not swap these vertices, but simply mark them and update the D values of the unmarked vertices. Repeat this process until no unmarked vertices remain. The result is a sequence of pairs (v_i, u_i) of vertices and their associated gains g_i. Note that the gains g_i can be positive or negative. Finally, determine which vertices of $\mathcal{A}$ and $\mathcal{B}$ to swap by finding the smallest k that maximizes $G = \sum_{i=1}^{k} g_i$. If $G > 0$, swap vertices $v_1, \ldots, v_k$ of $\mathcal{A}$ with $b_1, \ldots, b_k$ of $\mathcal{B}$ and repeat this entire process. If $G = 0$, no further improvements are possible using this approach, and the algorithm terminates.

A key step in the Kernighan-Lin algorithm is the selection of the initial sets $\mathcal{A}$ and $\mathcal{B}$. We implemented to different heuristics. The first just assigns the first $n/2$ nodes to $\mathcal{A}$ and the rest to $\mathcal{B}$, while the second heuristic assigns nodes in $\mathcal{A}$ and $\mathcal{B}$ such that the total number of nonzeros corresponding to row in $\mathcal{A}$ is the same as that in $\mathcal{B}$. The motivation behind this heuristic is to try to minimize the work load balance.