

Space and Time Efficient Implementations of Parallel Nested Dissection

Deganit Armon
University of California, Riverside

John Reif *
Duke University

Abstract This paper is concerned with algorithms for the efficient parallel solution of sparse, symmetric $n \times n$ linear systems via direct factorization. The algorithms make use of nested dissection orderings which are used to impose an order of elimination on the variables, so to reduce the storage and arithmetic requirements. Parallel nested dissection (PND) algorithms, which perform this variable elimination in parallel, have been developed for various parallel machine architectures, including the parallel random access machine (PRAM) model and for grid architectures. In this paper, we describe the following improvements to the known PND algorithms:

1. FAST PND: a fast PND algorithm for the PRAM model, which reduces the time bound by a factor of $O(\log n)$, without significantly increasing the processor bound.
2. COMPACT PND: a PND algorithm for a mesh-connected processor array, applicable to matrices representable by a grid graph, that uses $O(n)$ processors and takes $O(\sqrt{n})$ time, and reduces the space bounds to a constant factor of the size of the input matrix, without significant increase in time or processor bounds.
3. GENERALIZED COMPACT PND: a PND algorithm for a mesh-connected processor array, that uses $O(n)$ processors and takes $O(\sqrt{n})$ time for the more general case of matrices representable by a graph of constant degree and separator size \sqrt{n} (as required for many 2D PDE applications).

In addition to these practical results, we show, using known theoretical results about parallel matrix multiplication, that it is theoretically possible to achieve tighter bounds on the amount of work performed by PND algorithms. We also show how our algorithms generalize to solve all-pairs minimal-cost path problems, within the same complexity.

*Supported by National Science Foundation Grants CCF-0432038, CCF-0432047, ITR 0326157, EIA-0218376, EIA-0218359, and EIA-0086015.

1 Introduction

The problem of solving very large sparse linear systems arises often in scientific computing. Much research has been devoted to finding efficient solutions to this problem. Researchers have been able to develop fast sequential algorithms for solving linear systems by exploiting their sparsity and structure.

In this paper we will use the term *nested dissection (ND)* (and *parallel nested dissection (PND)*, respectively) to denote the use of nested dissection graph theoretic methods in the direct (parallel, respectively) solution of sparse linear systems, and we will cost the methods both for the decomposition as well as the resources to solve the sparse linear system.

ND methods often yield highly efficient algorithms, particularly for the class of sparse linear systems that arise from finite element problems in two and three dimensions.

The original work on the use of nested dissection for matrices representable by grid graphs is due to George [5], and was extended to general separable graphs by Lipton, Rose, and Tarjan [11]. The tightest and most comprehensive analysis of nested dissection is due to Rose and Whitten [25].

With the advent of parallel computing, research focus has shifted toward finding parallel algorithms, and especially parallel versions of known sequential algorithms (see [9] for a review and extended bibliography). In particular, considerable research has been devoted to developing parallel nested dissection (PND) algorithms. The goal of these research efforts is to find algorithms that are both fast and work efficient.

Birkhoff and George predicted that a parallel implementation for a sparse matrix representable by a grid graph would be able to achieve an $O(\sqrt{n})$ parallel time bound on an n processor mesh-connected machine [2]. Indeed, several parallel versions of the nested dissection algorithm have been suggested, taking into account different parallel models [18] and architectures [27, 15, 26, 13, 14]. More recently, work has been done on parallel Choleski factorization for dense [6] and sparse [8] matrices. The best known algorithm for the PRAM model, [18] uses $O(n^{3/2})$ processors, takes $O(\log^3 n)$ time, and is applicable to a general class of $O(\sqrt{n})$ separable graphs.

Throughout this paper we will refer to $P(n)$, the number of processors necessary to multiply two $n \times n$ matrices in $O(\log n)$ parallel steps on the PRAM model. $P(n) = n^\omega$, where $2 < \omega < 3$. Currently, the best known value for ω is 2.376... [S 86, 3], however a practical bound for ω is at best 2.81. This is the processor bound used by several parallel nested dissection algorithms on the PRAM model.

1.1 Open Problems with Existing Algorithms

Time Reductions

Research efforts have concentrated mostly on improving time bounds and reducing the processor count of each version of parallel nested dissection (PND), thereby reducing the overall work performed. Even so, a major open problem remains:

- On the PRAM model, reducing the work bound by dropping the time bound to $O(\log^2 n)$ while maintaining a processor bound of $P(n)$.

This parallel bound is known to hold for computing dense matrix inverse [16, 17], using a reduction to the computation of the characteristic polynomial or a related form. However, while theoretically possible, this method is numerically unstable, and thus of little practical value. The

problem of developing a FAST PND attaining these bounds is an open problem dating to the work of [18]. Using a pipelining technique, Pan and Reif [24] achieved a time bound of $O(\log^2 n)$ for solving minimum path problems using PND, but this speedup does not hold for linear systems.

Storage Reductions

Despite these problems, parallel nested dissection methods can be highly efficient for general 2D and 3D problems (however, for restricted 2D and 3D problems, such as constant degree linear PDE, parallel multigrid can be more time efficient; see [23]). However, there is an additional parameter involved when analyzing the practical aspects of an algorithm, which is the amount of storage necessary. This is particularly true when one considers the memory constraints of existing parallel machines.

The storage required for direct factorization (which is not PND) is completely determined by the nonzero structure of the matrix and the order in which the factorization (elimination of rows/columns) occurs, and this storage requirement can be precomputed exactly.

Whereas iterative methods generally need only a constant factor of additional storage, the known PND implementations are not space efficient and require at least a logarithmic factor of additional storage over the input matrix. For moderate size sparse matrices, with 1,000 variables, this relative factor is already 10. This leads to the following problem:

- Reducing the space bounds to a constant factor of the input matrix, while maintaining the time and processor bounds.

We propose an interesting modification of PND algorithm, which we call COMPACT PND, which uses only a small constant factor of extra storage (beyond the matrix input) without compromising the asymptotic time bounds of the algorithm. This result holds for implementations on a grid- or mesh-connected machine when the graph of the matrix has bounded degree and an $O(\sqrt{n})$ separator set.

Specialized Matrix Classes

A third issue, in addition to reducing work and space bounds, is the class of matrices that can be solved using PND. On the mesh connected processor array, all existing implementations have concentrated on matrices whose underlying graphs are grids. This leaves the following open problem:

- On a mesh-connected processor array, implementing PND using $O(n)$ processors and taking $O(\sqrt{n})$ time for the important and more general case where the graph of the matrix is of constant degree and separator size \sqrt{n} .

This requires us to extend the previous work of Birkhoff and George [2], which is restricted to matrices with grid graphs, and also of Pan and Reif [18], which is inefficient with respect to work by a polylog factor.

We also show that FAST PND and COMPACT PND generalize to solve all pairs minimal cost path problems within the same complexity.

1.2 Organization of the Paper

Section 2 provides an overview of nested dissection and parallel nested dissection. Section 3 describes the FAST-PND PRAM algorithm. Section 4 describes implementations of the PND algorithm on

a grid-connected architectures (meshes). In Section 5 we describe COMPACT PND, an algorithm for reducing the space bounds of PND implementations on meshes. In Section 6 we generalize COMPACT PND to linear systems with small separators. Section 7 extends PND to path problems. Open problems are posed in Section 8.

2 Nested Dissection

This section provides an overview of sequential and parallel nested dissection.

2.1 Definitions

Consider a system of linear equations

$$Ax = b$$

where A is an $n \times n$ symmetric positive definite matrix, b is a vector and x is the unknown solution vector to be computed.

2.1.1 The graph of a matrix

We define the graph of A to be an undirected graph $G(A) = (V, E(A))$, where $V = \{1, 2 \dots n\}$ and $E(A) = \{(i, j) | a_{i,j} \neq 0\}$. Each variable (row or column in the matrix) is a vertex in the graph, and a non-zero entry in the matrix corresponds to an edge. For example when solving n second order elliptic partial differential equations by using a finite element mesh, the graph of the matrix is a $\sqrt{n} \times \sqrt{n}$ grid. In much of this paper, we will refer to the matrix and its graph interchangeably.

2.1.2 Solving a System of Equations

Finding a solution to a system of equations $Ax = b$ by Cholesky (also written as Choleski) factorization consists of two steps. First, factor the matrix A into the product of three matrices, $A = LDL^T$ where L is lower triangular and D is diagonal. Next, solve the simplified system $LDL^T x = b$ by solving three simpler problems: $Lz = b$, $Dy = z$ and $L^T x = y$.

2.1.3 Fill-In

When A is sparse, we would like to take advantage of its sparsity when solving the system. When factoring A into LDL^T , a key issue is to limit *fill-in* – the creation of many more non-zero entries in L than were in A . If L is not sparse, the backsolve step becomes more complex. The higher the fill-in, the less we can take advantage of the sparsity of A , and the more time and space will be necessary for factoring and solving the system.

A sparse matrix is represented by a graph with few edges. Elimination of a variable corresponds to the removal of a vertex and its incident edges from the graph, and adding edges between its neighbors. When new edges are added to the graph, fill-in occurs. As more edges are added, the graph will no longer be sparse. Nested dissection is a way of ordering the variables in such a way that will limit the amount of fill-in in the matrix, and consequently the number of edges in its graph.

2.2 Sequential Nested Dissection

The idea of nested dissection (ND) was first introduced by George [5]. He showed that a system of n equations whose matrix had an underlying $\sqrt{n} \times \sqrt{n}$ grid structure could be solved in $O(n^{3/2})$ sequential time using $O(n \log n)$ space.

Without loss of generality, throughout this paper we assume \sqrt{n} is of the form $2^k - 1$ for some integer k .

In the factoring stage of the nested dissection algorithm, matrix A is actually factored into a product of $n - 1$ lower triangular matrices, a diagonal matrix and the transposes of the triangular matrices, as follows:

$$A = L_1 \dots L_{n-1} D L_{n-1}^T \dots L_1.$$

This is performed in steps. First, A is factored into

$$A = L_1 A_1 L_1^T$$

where

$$A_1 = \begin{bmatrix} d_1 & 0 \\ 0 & B_1 \end{bmatrix}$$

and B_1 is an $(n - 1) \times (n - 1)$ matrix.

Next, A_1 is factored into

$$A_1 = L_2 A_2 L_2^T$$

where

$$A_2 = \begin{bmatrix} d_1 & 0 & 0 \\ 0 & d_2 & 0 \\ 0 & 0 & B_1 \end{bmatrix}$$

and B_1 is an $(n - 2) \times (n - 2)$ matrix.

In the i^{th} factorization step, A_{i-1} is factored into

$$A_{i-1} = L_i A_i L_i^T$$

where

$$A_i = \begin{bmatrix} D_i & 0 \\ 0 & B_i \end{bmatrix},$$

D_i is a diagonal $i \times i$ matrix, and B_i is an $(n - i) \times (n - i)$ matrix yet to be factored.

Finally

$$A_{n-1} = D.$$

The k^{th} factorization step is called the *elimination of variable x_k* .

When referring to $G(A)$, the graph of A , the elimination of the variable x_k is equivalent to removing vertex k and its incident edges from the graph and adding an edge between each pair of its neighbors. For every pair of vertices u, v which are neighbors of k , we add an edge (uv) to the graph if it does not already exist. This process essentially “short-circuits” k out of the graph. The vertex is removed and its neighbors now form a *clique* – a subgraph in which there is an edge between every two vertices. Each added edge corresponds to a new non-zero entry in one of the factorization matrices – the fill-in which we seek to minimize.

George presented an ordering scheme for eliminating the variables, which bounds the fill-in in the L matrices to $O(n \log n)$. For a positive, symmetric definite matrix representable by a grid graph, this scheme involves partitioning the $\sqrt{n} \times \sqrt{n}$ grid into four $\frac{\sqrt{n}}{2} \times \frac{\sqrt{n}}{2}$ subgrids by using a “cross” through the central row and column of the grid as a separator. Let the set of vertices in this separator cross be called S_1 . If we remove the vertices of S_1 and their incident edges from the graph, we are left with four separate subgrids. The subgrids are *independent*, that is, there are no edges between them, and each one is a “stand-alone” grid. Each of the four subgrids is now partitioned into four again, using four separator crosses, one in each subgrid. Let S_2 be the set of vertices in all four separator crosses. We continue this process of subdividing the graph until we have independent subgrids consisting of at most one vertex. The last set of separators is S_{d-1} , and the remaining vertices of the graph make up S_d . This process partitions the vertices of the grid into d subsets. Figure 1 shows the partition of a 7x7 grid graph.

2.2.1 Separator Tree

It is useful to think of the vertices of $G(A)$ as placed in a data structure called a *separator tree* (also known as the *elimination tree* in this context). At the root of the tree are the vertices of S_1 . The root has four children. Each subtree contains the vertices of a subgrid, and recursively follows the same structure, i.e., the separator vertices are in its root and the vertices of each independent subgrid are in a subtree. Each level of the separator tree contains the vertices of one of the S_i , with the S_1 vertices at the root and the S_d vertices in the leaves.

Lemma 2.1 *The depth of the separator tree, d , is at most $\frac{1}{2} \log n$.*

Proof: Let r be the set of vertices in the top row of the mesh. The set r contains \sqrt{n} vertices. The first cross, S_1 , has one vertex in r , splitting r into two. S_2 contributes two vertices, splitting each of the halves, and so on. The contribution of all separators and remaining subgrids add up to the entire row. In other words, $\sum_{i=1}^d 2^i \leq \sqrt{n}$. Solving this inequality, we get $d < \frac{1}{2} \log n$. ■

2.2.2 Order of Elimination of Vertices

The elimination of variables proceeds up the separator tree, starting with the vertices at the leaves and ending with the vertices at the root. The first vertices to be eliminated are the vertices in S_d . Note that no two vertices of S_d share an edge; they are independent of each other. Because these variables are independent, the order in which they are eliminated is not important. The elimination of all the S_d vertices is called the first *elimination phase* of the factorization. During the second elimination phase, the vertices in S_{d-1} are eliminated. In the next phase we eliminate the vertices in S_{d-2} and so on until, in the last phase, the vertices in S_1 are eliminated. Figure 2 shows the graph in Figure 1 after the first elimination phase.

2.3 Parallel Implementation

The key to parallelizing the nested dissection algorithm is the realization that during factorization, vertex eliminations within each elimination phase are independent of each other, and thus can be performed in parallel. Several parallel versions of nested dissection have been suggested, some [27, 15, 26] achieving the $O(\sqrt{n})$ time bounds predicted by Birkhoff and George in [2].

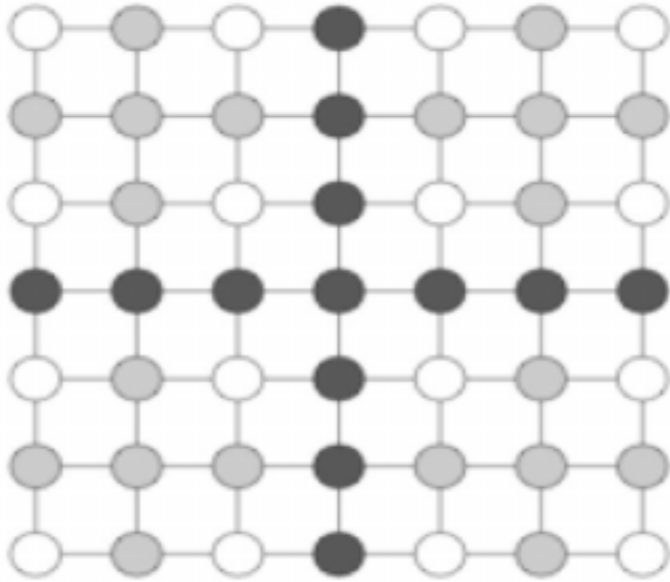


Figure 1. The 7 x 7 grid graph. Vertices eliminated in the same phase have the same shading.

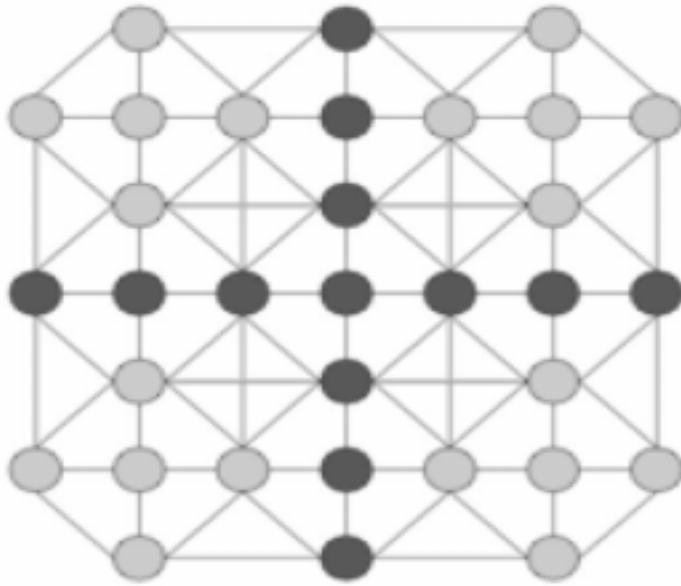


Figure 2. The 7 x 7 grid graph after the first elimination phase.

2.4 Generalized PND

The algorithm in [18], operating on a more general class of graphs, works on a PRAM model using $O(n^{3/2})$ processors and takes $O(\log^3 n)$ time. The separators in this class of graphs are not as straightforward to construct as in the case of grid graphs. However, the size of the separator set is still $O(\sqrt{n})$ and each induced subgraph has at most $2n/3$ vertices. As in the grid, the graph is separated into two independent subgraphs using, as a separator, a set of vertices S_1 , where $|S_1| \leq O(\sqrt{n})$. Each subgraph is separated again, and the set of vertices in all these separators makes up S_2 . The process continues until all that is left of the graph are independent subsets of at most one vertex. S_d is the set of all these subsets. The depth of the separator tree is $O(\log n)$. Throughout this paper, as in [18], we shall assume that the structure of the separator tree is known (calculated) in advance before the start of the nested dissection algorithm. This does not add to the complexity of the algorithm.

Since much of the work is done in parallel, the factorization is not done column by column as in the sequential case, but rather in blocks, where each small block is factored in turn. The smaller blocks are factored in parallel. Adopting the method used by Pan and Reif [18], we define the factorization recursively over a sequence of matrices A_0, A_1, \dots, A_d . For $h = 0, 1, \dots, d - 1$, each matrix has the structure

$$A_h = \begin{bmatrix} X_h & Y_h^T \\ Y_h & Z_h \end{bmatrix}$$

where the X_h matrix is block diagonal, $Z_h = A_{h+1} + Y_h X_h^{-1} Y^T$ is the part of the matrix yet to be factored, and each block in the X_h matrices is further factored in a later stage of the algorithm.

The factorization of each A_h can be written as an LDL^T factorization:

$$A_h = \begin{bmatrix} I & 0 \\ Y_h X_h^{-1} & I \end{bmatrix} \begin{bmatrix} X_h & 0 \\ 0 & A_{h+1} \end{bmatrix} \begin{bmatrix} I & X_h^{-1} Y_h^T \\ 0 & I \end{bmatrix}$$

Note that here, D is block diagonal.

It follows that

$$A_h^{-1} = \begin{bmatrix} I & -X_h^{-1} Y_h^T \\ 0 & I \end{bmatrix} \begin{bmatrix} X_h^{-1} & 0 \\ 0 & A_{h+1}^{-1} \end{bmatrix} \begin{bmatrix} I & 0 \\ -Y_h X_h^{-1} & I \end{bmatrix}$$

Thus the problems of factoring A_h and of inverting it for the backsolve are reduced to factoring and inverting much smaller block diagonal matrices.

3 Improved Time Bounds: FAST PND

In this section, we describe a $\log n$ factor improvement to the time bounds of the PRAM PND algorithm. We achieve this by grouping together several elimination phases. Looking at the separator tree, we see that at each level we break up the graph into two roughly equal parts, using a separator of size \sqrt{n} . This gives a tree of depth $O(\log n)$. The algorithm recursively works on each level of the tree. We can therefore conceive of a *recursion tree*, reflecting the separator tree, which describes the work of the algorithm. The depth of this recursion tree is also $O(\log n)$. The time necessary to perform PND with this algorithm is determined by the number of recursive calls and by the time to perform each one.

Gazit and Miller [7] first suggested reducing the depth of the recursion tree by grouping together several levels of the tree and performing the work on a group of levels together. They were able to limit the depth of the tree to $O(\log \log n)$. Our idea is similar in that we, too, group levels together into *multilevels*, and perform the work on each multilevel of the tree all at once (see Figure 3).

We introduce a new parameter ϵ , an arbitrarily small constant $0 < \epsilon < 1/2$. The number of multilevels in our recursion tree will be bounded by $\log \frac{1}{\epsilon}$, and the number of processors used will be $O(n^{\omega/2+\epsilon})$. The key idea, which allows us to bound the number of multilevels by a constant, is that *the size of each multilevel is not fixed; rather, the size of each multilevel is a function of n , ϵ , and the size of the multilevels above it in the tree.*

Since the structure of the separator tree is known in advance, work performed at each multilevel involves only the elimination of separator vertices from several levels. More work is being done at each multilevel than at any single level in the original tree. However, since the subproblems are independent, we can, in the PRAM model, spread the extra work among more processors. Specifically, we require a factor of n^ϵ more processors to perform the extra work.

Theorem 3.1 *Nested dissection on a PRAM model can be performed in $O(\log^2 n)$ time using $O(n^{\omega/2+\epsilon})$ processors, where the constant factor in the time bound is a logarithmic function of $1/\epsilon$.*

Proof: The depth of the recursion tree is initially $\log n$. We label the levels of the tree from 1 (root) to $\log n$ (leaves). Let $k_0 \dots k_m$ be a sequence of strictly increasing integers over $[0 \dots \log n]$, where $k_0 = 0$ and $k_m = \log n$. The levels of the tree are grouped into m multilevels, where k_j is the deepest tree level in multilevel j . We define multilevel K_j of the recursion tree to be the aggregate of levels $[k_{j-1} + 1 \dots k_j]$ of the tree. The work in multilevel K_j consists of eliminating all vertices of the separators in these levels. The number of levels in each multilevel is a function of the size of the level above it.

To clarify the analysis that follows, let us first examine what happens in the algorithm if there is no grouping of levels. When working on a single level of the recursion tree, we eliminate the separator vertices of several subgraphs in parallel. In level i , the number of subtrees is $g_i = 2^{i-1}$. Let v_i be the number of separator vertices in each subgraph. Since the separators are of size \sqrt{n} , $v_i \leq \frac{\sqrt{n}}{2^{i-2}}$. The work involved in eliminating the separator vertices in level i is the work necessary to eliminate all separator vertices in each subgraph. This is dominated by the work necessary to invert g_i matrices of v_i variables each. Since inverting a matrix of n variables takes $O(n^\omega)$ work (with $2 < \omega < 3$), the total work for eliminating the separator vertices in level i is $g_i O(v_i^\omega) \leq 2^{i-1} O((\sqrt{n}/2^{i-2})^\omega) \leq 2^{i-1} O(\sqrt{n}^{\omega/2})$.

When working on a multilevel, separator vertices from several levels are eliminated in parallel. Since a multilevel is defined to be the aggregate of several adjacent levels, eliminating vertices in a multilevel results in a graph identical to the graph resulting from eliminating the separator vertices in each of the levels sequentially. The number of subtrees in a multilevel is the number of subtrees in the highest level of the multilevel, and the number of separator vertices in each subtree is the total of all separator vertices in all levels of the subtree within the multilevel.

Looking at multilevel K_j , which is the aggregate of levels $[k_{j-1} + 1 \dots k_j]$, each subgraph is a subtree of subgraphs. The number of subgraphs is the number of roots in the highest level, or $g_{k_{j-1}+1} = 2^{k_{j-1}}$. The number of separator vertices in each subgraph is the total number of separator vertices in all levels of the multilevel, or $\sum_{i=k_{j-1}+1}^{k_j} 2^{i-k_{j-1}-1} \sqrt{\frac{n}{2^{i-1}}} = \sum_{i=0}^{k_j-k_{j-1}-1} 2^i \sqrt{\frac{n}{2^{i+k_{j-1}}}}$.

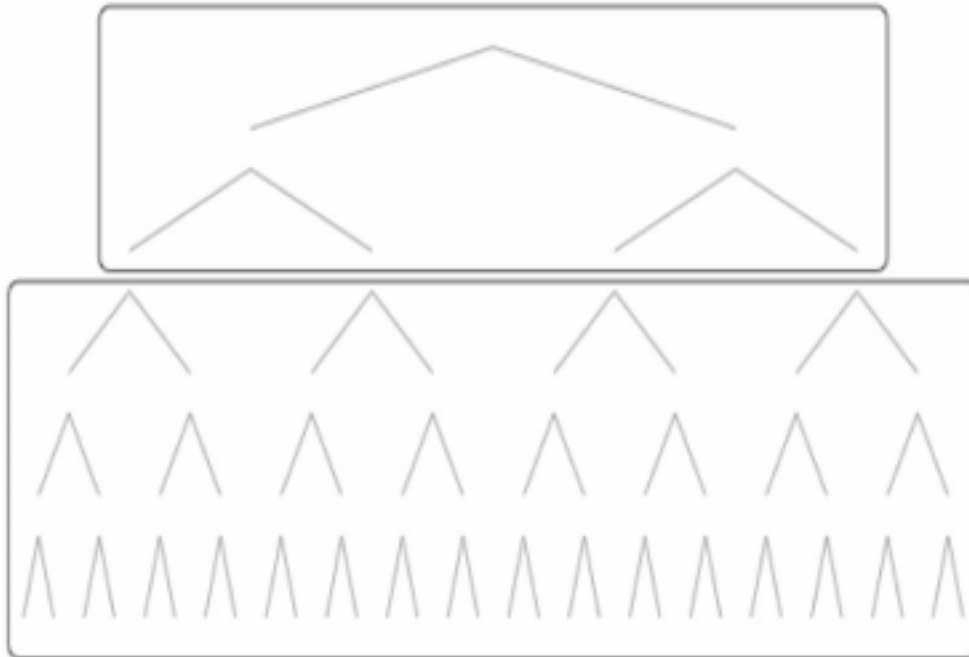


Figure 3. The multilevel grouping derived from a separator tree.

The total work performed in multilevel K_j is bounded by the work necessary to eliminate all the separator vertices in the multilevel, multiplied by the number of subgraphs in the multilevel. This is

$$2^{k_{j-1}} O \left(\left(\sum_{i=0}^{k_j - k_{j-1} - 1} 2^i \sqrt{\frac{n}{2^{i+k_{j-1}}}} \right)^\omega \right). \quad (1)$$

Evaluating the summation inside the power term, we get

$$\begin{aligned} \sum_{i=0}^{k_j - k_{j-1} - 1} 2^i \sqrt{\frac{n}{2^{i+k_{j-1}}}} &= \sqrt{\frac{n}{2^{k_{j-1}}}} \sum_{i=0}^{k_j - k_{j-1} - 1} 2^{i/2} = \\ &= \sqrt{\frac{n}{2^{k_{j-1}}}} (2^{(k_j - k_{j-1})/2} - 1)(\sqrt{2} + 1) = (\sqrt{2} + 1) \sqrt{n} 2^{(k_j - 2k_{j-1})/2}. \end{aligned}$$

Inserting this back into equation 1, we get an equation for the total work performed in multilevel K_j , which is, for a constant c :

$$2^{k_{j-1}} c \left[\sqrt{n} 2^{(k_j - 2k_{j-1})/2} \right]^\omega = c n^{\omega/2} 2^{\omega k_j / 2 - (\omega - 1)k_{j-1}} = n^{\omega/2} 2^{\omega k_j / 2 - (\omega - 1)k_{j-1} + \log c}.$$

We can spread this work over $n^{\omega/2+\epsilon}$ processors for any $0 < \epsilon < 1/2$, getting

$$n^{\omega/2+\epsilon} = n^{\omega/2} 2^{\omega k_j / 2 - (\omega - 1)k_{j-1} + \log c}$$

and thus

$$\epsilon \log n = \omega k_j / 2 - (\omega - 1)k_{j-1} + \log c.$$

Solving this equation for k_j , we get:

$$k_j = \frac{2}{\omega} (\epsilon \log n + (\omega - 1)k_{j-1} - \log c).$$

Substituting $c' = \frac{2}{\omega} \log c$, and solving this recurrence to get k_j in terms of n and ϵ , we get

$$k_j = \left(\frac{2\epsilon}{\omega - 2} \log n - \frac{c'\omega}{(\omega - 2)} \right) \left(\left(\frac{2(\omega - 1)}{\omega} \right)^j - 1 \right). \quad (2)$$

Recall that k_j is the deepest level in multilevel K_j . From this, we can determine the size of the multilevel K_j , which is $k_j - k_{j-1}$.

We can now use the information on the size of the multilevels to calculate how many multilevels there are in the recursion tree. Recall that there are m multilevels in the tree, with k_m being the deepest level of the deepest multilevel. This is also the deepest level of the recursion tree, which is level $\log n$. From equation 2 above we have

$$\log n = k_m = \left(\frac{2\epsilon}{\omega - 2} \log n - \frac{c'\omega}{(\omega - 2)} \right) \left(\left(\frac{2(\omega - 1)}{\omega} \right)^m - 1 \right).$$

Solving for m to find the number of multilevels in the tree, we get

$$\left(\frac{2(\omega - 1)}{\omega}\right)^m = \frac{\log n + \frac{2\epsilon}{\omega-2} \log n - \frac{c'\omega}{\omega-2}}{\frac{2\epsilon}{\omega-2} \log n - \frac{c'\omega}{\omega-2}} = \frac{\omega - 2 + 2\epsilon - \frac{c'\omega}{\log n}}{2\epsilon - \frac{c'\omega}{\log n}}$$

and thus

$$m = \log_{\frac{2(\omega-1)}{\omega}} \left[\frac{\omega - 2 + 2\epsilon - \frac{c'\omega}{\log n}}{2\epsilon - \frac{c'\omega}{\log n}} \right].$$

Noting that $\omega < 3$ and $2\epsilon < 1$, we get

$$m < \log_{\frac{2(\omega-1)}{\omega}} \left[\frac{2 - \frac{c'\omega}{\log n}}{2\epsilon - \frac{c'\omega}{\log n}} \right] < \log_{\frac{2(\omega-1)}{\omega}} \frac{1}{\epsilon}.$$

Thus the number of multilevels in the tree is bounded by a constant.

Since we have only a constant number of multilevels, rather than a logarithmic number of levels, the time complexity for nested dissection on a PRAM can be reduced by a factor of $\log n$ to $O(\log^2 n)$. This completes the proof of the theorem. ■

4 Mesh Model Implementations of Parallel Nested Dissection

We now return to using nested dissection to solve linear systems representable by grid graphs. Of the existing parallel architectures, the natural one to use when implementing a solution for a problem involving a grid graph would be a grid- or mesh-connected processor array. Memory mapping is trivial for the input matrix, each variable being assigned to a processor on the grid. After some elimination takes place, there are idle processors evenly distributed throughout the grid. If a naive approach is used in the implementation, communication time between active processors grows with the number of idle processors in the grid. It is possible however, to redo the memory mapping after each elimination phase by conceptually “folding” the grid to maintain a compact and efficient structure. Worley and Schreiber [27] described an implementation on a mesh connected processor array of n processors which works in $O(\sqrt{n})$ parallel time. However, the storage requirements of their algorithm grow as \sqrt{n} , i.e. there is a *per processor* multiplicative factor of increase in the storage requirements of the algorithm. Opsahl and Reif also present an implementation [15], based on the algorithm described in Section 2. The storage requirements for their implementation grows as a logarithmic factor of the input matrix size.

Unfortunately, even state-of-the-art parallel machines still have limited processor storage. If we wish to use them for solving practical problems, we need to give as much consideration to space efficient algorithms as we do to designing time efficient ones.

In Section 5 we present a variant of the PND algorithm implementation on a grid architecture, which we call COMPACT PND. This algorithm maintains the $O(\sqrt{n})$ time bound but uses only a constant amount of storage per processor beyond the input, if there are n processors. More generally, given any number of processors, our algorithm uses only a constant factor more space beyond the input matrix size. First, let us establish the known space bounds.

4.1 Space Bounds

The PND algorithm described in Section 2 has two stages - factorization and backsolve. There are $O(\log n)$ factorization steps, and a new matrix is generated in each of these steps. As these matrices are later used in the backsolve stage, they must be saved in memory. In this section we show a simple modification to PND, as implemented on a mesh-connected machine, which allows us to save this $\log n$ space factor. This saving can be achieved because we do not need to save the information generated at each level of the recursion. Instead, we can recompute it.

Let $|L_i|$ be the number of non-zero entries in the matrix L_i which is generated at level i of the recursion. Let $|L| = \max(|L_i|)$. Thus the total space requirement for the PND algorithm on a grid graph is $\leq |L| \log n$.

By definition, the number of non-zero entries in a matrix L_i is the number of edges in the underlying graph G_i . Therefore, we can limit our analysis to the number of edges in the graph generated at each step. Before proceeding with the exact analysis, we present some intuition. The original grid graph has n vertices, each with edges to at most 4 neighbors. In other words, the original matrix has at most $4n = O(n)$ non-zero entries. At each elimination phase, we remove a fraction of the vertices and their incident edges from the graph, and add edges between the neighbors of the eliminated vertices, which become cliques. When the only vertices in the graph are the vertices in S_1 , the graph has $2\sqrt{n} - 1$ cliqued vertices, for a total of $2n - \sqrt{n} = O(n)$ edges, which is also $O(n)$. The careful analysis which follows shows that even though the degree of every vertex increases through the adding of new edges, the number of vertices in the graph is reduced and the total number of edges in the graph at any phase remains $O(n)$.

The following lemmas are introduced to establish bounds on $|L_i|$. Recall that S_k is the k th separator set and consists of the vertices eliminated in phase $d - k + 1$ of the factoring.

Lemma 4.1 *The size of the set S_k is $2^k(\sqrt{n} + 1) - 3 \cdot 4^{k-1}$.*

Proof: S_1 is a cross of one row and one column in the mesh. Each row and column are \sqrt{n} long, and one vertex is counted twice. S_i consists of 2^i rows and columns, but we need to subtract the middle vertex of each cross (of which there are 4^{i-1}), and all vertices that have already been counted in $S_j, j < i$. A simple counting argument yields the desired result. As further proof, observe that $\sum_{i=1}^d S_i = n$. ■

Lemma 4.2 *The number of vertices in the mesh after S_d, \dots, S_{k+1} have been eliminated is at most $(\sqrt{n} + 1)2^{k+1}$.*

Proof: After eliminating the vertices of S_d, \dots, S_{k+1} the vertices left in the mesh are those in S_1, \dots, S_k . The number of vertices in those sets is

$$\begin{aligned} & \sum_{i=1}^k [2^i(\sqrt{n} + 1) - 3 \cdot 4^{i-1}] \\ &= (\sqrt{n} + 1)(2^{k+1} - 2) - 2^{2k} < (\sqrt{n} + 1)2^{k+1}. \end{aligned}$$

■

Lemma 4.3 *The number of edges incident on each vertex of S_k , after S_d, \dots, S_{k+1} have been eliminated, is at most $\frac{7\sqrt{n}}{2^k}$.*

Proof: As we eliminate sets of vertices from the mesh, we get a larger mesh with the vertices bordering on the “holes” in the mesh cliqued together. After i phases, each hole is a square, with $2^i + 1$ vertices along a side (see Figure 4). The “corner” vertices are not part of the clique, so we have $2^i - 1$ cliqued vertices along a side of the box, giving us cliques of $4(2^i - 1)$ vertices, each with degree $4 \cdot 2^i - 5$.

A vertex may be part of at most two cliques (if it is on the border between two boxes). The total number of vertices in both these cliques is $7(2^i - 1)$ (border is counted only once), and thus border vertices have $7 \cdot 2^i - 8$ neighbors.

If S_d, \dots, S_{k+1} have been eliminated, it means that $d - k$ elimination phases have been performed. Substituting $i = d - k = \log \sqrt{n} - k$, we find that the number of neighbors a vertex has is at most

$$7 \cdot 2^{\log \sqrt{n} - k} - 8 = \frac{7 \cdot 2^{\log \sqrt{n}}}{2^k} - 8 < \frac{7\sqrt{n}}{2^k}.$$

■

Lemma 4.4 *The number of non-zero entries in any of the L matrices is at most $15n$.*

Proof: Combining the results of the previous two lemmas, the number of edges in the graph after k elimination phases (for any k) is at most $(\sqrt{n} + 1)2^{k+1} \times \frac{7\sqrt{n}}{2^k} = 14(n + \sqrt{n}) < 15n$. ■

Since the total storage per phase is linear, and since there are $\log n$ phases, the total storage needed by an implementation which stores all intermediate results is therefore $O(n \log n)$ or $O(\log n)$ per processor.

5 Improved Space Bounds: COMPACT PND

In this section we present a modification of the PND algorithm, which we call COMPACT PND, that requires only a small constant factor of storage over the input matrix size. In the case of an implementation on a 2D mesh connected machine and where the graph of the matrix is a 2D grid, our algorithm maintains the current known asymptotic time and processor bounds for the PND algorithm, but requires only a small constant factor of extra storage.

The issue of limiting the storage requirement associated with sequential Gaussian elimination was first raised by Eisenstadt, Schultz and Sherman [4]. They suggested, within the context of sequential algorithms, recomputing rather than storing intermediate results as a method of saving space. We will apply this idea to the parallel implementation of the PND algorithm on a grid architecture, and prove that the space requirement is just a constant factor of the input matrix. Moreover, the time bound remains asymptotically the same.

The first stage of COMPACT PND is identical to the elimination phase of PND. It factors the matrix A into

$$A = L_0 L_1 \dots L_d D L_d^T \dots L_1^T L_0^T$$

where $d = \log n$, the L matrices are lower triangular and the D matrix block diagonal. All D and L matrices have definite bounds on the size and number of the non-zero block submatrices.

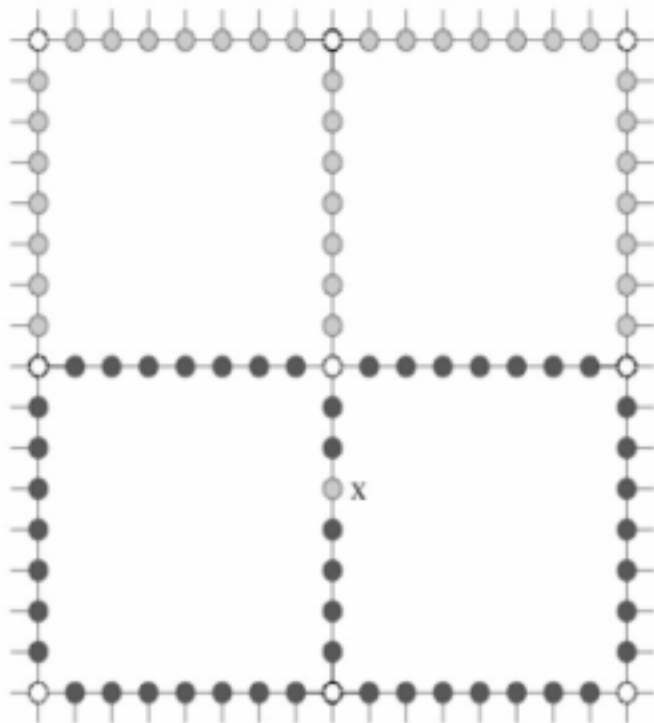


Figure 4. Cliques induced by elimination process.

In order to solve $Ax = b$, we actually perform $A^{-1}b = x$, but instead of inverting A we invert each of the L and D matrices. We then solve

$$(L_0^T)^{-1}(L_1^T)^{-1} \dots (L_d^T)^{-1} D^T L_d^{-1} \dots L_1^{-1} L_0^{-1} b = x$$

which, if performed from right to left, is a sequence of vector-matrix multiplications.

The L_i matrices are generated in increasing order of subscripts. Thus half of the multiplications can be performed during the first stage of the COMPACT PND algorithm by:

- calculating $A_i = L_i A_{i+1} L_i^T$
- inverting L_i , and
- computing a new vector $b_{i+1} = L_i^{-1} b_i$ (where $b_0 = b$).

As a new L_i is generated, the matrix is stored over L_{i-1} . Since only one L matrix is available at any point in the calculation, when we next want to multiply b_d by $(L_{d-1}^T)^{-1}$, we will need to recalculate L_{d-1} . Similarly we will need to recalculate all other L_i matrices. While it may seem that the extra work will hamper the performance of the COMPACT PND algorithm, as compared with PND, a careful analysis reveals that the asymptotic parallel time complexity of COMPACT PND remains the same.

Theorem 5.1 *The total time taken by COMPACT PND when recomputing rather than storing intermediate results is $O(\sqrt{n})$.*

Proof: The total time necessary to factor the full A matrix, which corresponds to a $\sqrt{n} \times \sqrt{n}$ grid, is $c\sqrt{n}$ for some constant c [27, 15]. When recalculating L_{d-1} , we perform all but the last factorization step, i.e. we are factoring four $\frac{\sqrt{n}}{2} \times \frac{\sqrt{n}}{2}$ grids *in parallel*, which requires $c\frac{\sqrt{n}}{2}$ parallel time. Similarly, recalculating L_i involves factoring the matrices of $\frac{\sqrt{n}}{2^{d-i}} \times \frac{\sqrt{n}}{2^{d-i}}$ grids, which takes $c\frac{\sqrt{n}}{2^{d-i}}$ parallel time. It follows that the total time needed by COMPACT PND is

$$\sum_{i=0}^{\log \sqrt{n}} c \frac{\sqrt{n}}{2^i} \leq 2c\sqrt{n}.$$

■

Despite the extra work performed, the asymptotic time bounds of COMPACT PND are the same as for PND. Moreover, the actual running time of COMPACT PND is only slowed down by a factor of less than two. The space savings of COMPACT PND, on the other hand, are substantial:

Theorem 5.2 *The COMPACT PND algorithm requires only a constant amount of storage per processor and the same time bounds of the PND algorithm, within a factor of 2.*

Proof: From lemma 4.4, we know that the number of non-zero entries in any matrix calculated by the algorithm is bounded by $15n$. On a mesh-connected $\sqrt{n} \times \sqrt{n}$ processor array, this amounts to constant amount of storage per processor. ■

6 GENERALIZED COMPACT PND

The previous sections concentrated on matrices whose underlying graph had a grid structure. For these matrices, we could exploit the advantages extended by using a grid architecture to solve the system of equations in parallel on the mesh. Because of the regular structure of the matrix, load balancing could be maintained throughout all stages of the algorithm. However, the nested dissection algorithm is applicable to other classes of graphs. Lipton, Rose and Tarjan [11] generalized the sequential algorithm to all matrices whose underlying graphs have “good separators”, where a “good” separator is of size $O(\sqrt{n})$. Several important classes of graphs have such separators, in particular, the class of all planar and nearly planar graphs [12]. Our space saving technique used in the COMPACT PND algorithm can be applied to develop a generalized algorithm when the underlying graph of the matrix is of bounded degree and has an $O(\sqrt{n})$ separator, as is true in many two dimensional PDE problems. We will call this generalized algorithm GENERALIZED COMPACT PND.

Theorem 6.1 *Let A be a matrix whose underlying graph G is of bounded degree and has an $O(\sqrt{n})$ size separator. COMPACT PND can be implemented on a grid- or mesh-connected machine in a way that maintains the current known asymptotic time bounds for the algorithm, and moreover, requires only a constant factor of extra storage.*

Proof: In order to be able to implement the COMPACT PND algorithm in the given time and space bounds, we must show that the size of the graphs generated at any step of the algorithm is bounded, and that we can balance the load among the processors. Consider the matrix

$$A_h = \begin{bmatrix} X_h & Y_h^T \\ Y_h & Z_h \end{bmatrix}$$

described in Section 2. By definition, X_h is a block diagonal matrix representing the vertices of the graph which are to be eliminated, Z_h represents the part of the matrix which is yet to be factored, and Y_h provides the connection between the two. Any unchecked growth in the size of the problem would occur only in Y_h , as X_h has bounded block size and Z_h is as yet unaffected by the algorithm. However, since the initial graph was of bounded degree, we can bound the size of the neighborhood of each eliminated vertex, and we can also bound the number of edges between this neighborhood and the rest of the graph. Consequently, we can bound the degree of each neighbor of an eliminated vertex, and thus the total size of the matrix remains bounded. ■

Note that this result does not generalize to all graphs to which nested dissection can be applied. If the input graph is not of constant degree or the algorithm is running in polylog time on a PRAM, adding the space saving modification will cause a slowdown of the algorithm by a factor of $\log n$. It would be of interest to see if this GENERALIZED COMPACT PND algorithm could be modified to be space efficient without compromising time bounds.

7 Extension of COMPACT PND to Path Problems

Pan and Reif [21] extended their PND algorithm to apply to path algebra computations, which have a wide range of applications. One particular application of importance is the problem of finding

the *all-pairs minimum cost path* in a graph. They extend the solution to apply to problems over a semi-ring (R, \oplus, \otimes) . The operations of a semi-ring are not necessarily invertible. Thus the inverse operation (A^{-1}) is replaced with the transitive closure operation (A^*) . We will again use $P(n)$ to denote the number of processors necessary to multiply two $n \times n$ matrices over this semi-ring in $O(\log n)$ parallel steps on the PRAM model. The transitive closure operation (A^*) on an $n \times n$ dense matrix A costs $P(n)$ processors and $O(\log^2 n)$.

Their algorithm solves all-pairs-shortest-path problems within the same complexity as PND for a sparse input matrix A with small separators. For example, it finds all-pairs-shortest-paths in the mesh network in $O(\sqrt{n})$ time using a linear number of processors.

The modification of our COMPACT PND to a path problems algorithm is similar to the modified PND algorithm described in Section 5. First, we factor the input matrix A into:

$$A = L_0 L_1 \dots L_d D L_d^T \dots L_1^T L_0^T$$

where $d = \log n$ and the L_i are appropriately defined with the new operations. We then wish to solve

$$(L_0^T)^* (L_1^T)^* \dots (L_d^T)^* D^T L_d^* \dots L_1^* L_0^* b = x$$

which, if performed from right to left, is a sequence of vector-matrix operations. Half of these can be performed during the factorization stage, and the factorization needs to be recalculated if each L_i is to be stored in memory over L_{i-1} in order to conserve storage.

The analysis of time and storage bounds is essentially identical to that given for COMPACT PND in Section 5. In particular, for each scalar operation \oplus, \otimes over the semi-ring (R, \oplus, \otimes) executed by the COMPACT PND path problem algorithm, there is a corresponding scalar operation $+, *$ respectively executed by the COMPACT PND in Section 5. Also, for each $n' \times n'$ matrix operation \oplus, \otimes , and transitive closure executed by the COMPACT PND path problem algorithm, there is a corresponding $n' \times n'$ matrix operation $+, *$, and matrix inverse respectively executed by the COMPACT PND in Section 5 within a constant factor of the same parallel cost in terms of time, space and processors. Hence, the cost of solving path problems is within a constant factor of the time, space and processor bounds of COMPACT PND for input matrices with the same class of graphs.

8 Summary and Open Problems

We have presented parallel algorithms for the solution of sparse linear systems. Our FAST PND algorithm speeds up by nearly a logarithmic factor previous PRAM PND algorithms. Our COMPACT PND significantly decreases the space bounds required for PND. We have also described the application of similar techniques to solve related combinatorial problems on graphs, such as path problems.

References

- [1] D. Armon and J.H. Reif. Space and Time Efficient Implementations of Parallel Nested Dissection, *Proceedings, 4th Annual ACM Symposium on Parallel Algorithms and Architectures*, 1992.

- [2] G. Birkhoff and J.A. George. Elimination by nested dissection. In *Complexity of Sequential and Parallel Numerical Algorithms* Traub JF, ed. Academic Press, New York, 1973.
- [3] D. Coppersmith and S. Winograd. Matrix multiplication via arithmetic progressions, em J Symbolic Comput 9(3):251–280, 1990.
- [4] S.C. Eisenstadt, M.H. Schultz and A.H. Sherman. Applications of an element model for Gaussian elimination. In *Sparse Matrix Computations* J.R. Bunch and D.J. Rose, eds., Academic Press, New York, 1976.
- [5] A. George. Nested dissection of a regular finite element mesh. *SIAM Journal of Numerical Analysis* 10:345–363, 1973.
- [6] A. George, M. Heath and J. Liu. Parallel Choleski factorization on a shared memory multiprocessor. *Lin Alg Appl* 77:165–187, 1986.
- [7] H. Gazit and G.M. Miller. Personal communication to JH Reif, 1992.
- [8] J. Gilbert and R. Schreiber. Highly parallel sparse Choleski factorization. *SIAM Symposium on Sparse Matrices*, 96–98, 1989.
- [9] M.T. Heath, E. Ng and B.W. Peyton. Parallel algorithms for sparse linear systems. *SIAM Review* 33:420–460, 1991.
- [10] C.E. Leiserson, J.P. Mesirov, L. Nekludova, S.M. Omohundro, J.H. Reif and W. Taylor. Solving sparse systems of linear equations on the connection machine. *Annual SIAM Conference*, A51, Boston, MA, July 1986.
- [11] R.J. Lipton, D.J. Rose and R.E. Tarjan. Generalized nested dissection. *SIAM Journal of Numerical Analysis* 16:346–358,1979.
- [12] R.J. Lipton and R.E. Tarjan. A separator theorem for planar graphs. *SIAM J. Applied Math.* 177–189, 1979.
- [13] J. Liu The Role of Elimination Trees in Sparse Factorization. *SIAM J. of Matrix Anal. and Appl.*, Vol 11, No. 1 pp 134-172 Jan 1990
- [14] J. Liu The Multifrontal Method for Sparse Matrix Solution: Theory and Practice. *SIAM Review*, Vol 34 No 1 pp 82-109. March 1992
- [15] T. Opsahl and J.H. Reif. Solving sparse systems of linear equations on the massive parallel machine. In *First Symposium on Frontiers of Scientific Computing, NASA, Goddard Space Flight Center, Greenbelt, MD*, 2241–2248, September 1986.
- [16] V. Pan. Fast and efficient parallel algorithms for the exact inversion of integer matrices. *Proceedings of the 5th Conference FST and TCS*, Lecture Notes in Computer Science, 504–521, 1985.
- [17] V. Pan. Complexity of parallel matrix computations. *Theoretical Computer Science* 54, 65–85, 1987.

- [18] V. Pan and J.H. Reif. Fast and efficient parallel solution of linear systems. *Proc. 17th ACM Symposium on the Theory of Computing*, 143–152, 1985. Also appeared as *Fast and efficient parallel solution of sparse linear systems*, *SIAM Journal on Computing*, 1992.
- [19] V. Pan and J.H. Reif, *Fast and efficient parallel solution of dense linear systems*, *Comp. Math. Applic.* 1481–1491, 1989.
- [20] V. Pan and J.H. Reif. Fast and efficient algorithms for linear programming and for the linear least squares problem. *Proc. 12th International Symposium on Mathematical Programming*, MIT, Cambridge, MA, August 1985; also in *Computers and Mathematics with Applications*, 12A(12):1217–1227, 1986.
- [21] V. Pan and J.H. Reif. Fast and Efficient Solution of Path Algebra Problems. *Journal of Computer and Systems Sciences* 38:494-510, June 1989.
- [22] V. Pan and J.H. Reif. Fast and efficient parallel solutions of dense linear systems, *Comp. Math. Applic.*, 1481–1491, 1989.
- [23] V. Pan and J.H. Reif. On the bit complexity of discrete approximations to PDEs . *International Colloquium on Automata, Languages, and Programming*, 612–625, 1990. Also additional results in Compact multigrids, *SIAM Journal of Scientific and Statistical Computing*, 1991.
- [24] V. Pan and J.H.Reif, *The parallel computation of minimum cost paths in graphs by stream contraction*, *Information Processing Letters*, 40:79–83, 1991.
- [25] D.J. Rose and G.F. Whitten. A Recursive analysis of dissection strategies. in *Sparse Matrix Computations*, Academic Press, New York, 1976.
- [S 86] V. Strassen, The asymptotic spectrum of tensors and the exponent of matrix multiplication, *Proc. 27th Annual IEEE Symposium on Foundations of Computer Science*, 49–54, 1986.
- [26] D. Wildenhain. Guide to parallel nested dissection on the MPP *Manuscript*, September, 1987.
- [27] P.H. Worley and R. Schreiber. Nested dissection on a mesh-connected processor array. *Proc ARO Workshop on New Computing Environments: Parallel, Vector and Systolic*, 1985.