July 1990

UILU-ENG-90-2227
DAC-23

(2)

**COORDINATED SCIENCE LABORATORY**
*College of Engineering*



AD-A225 377

# PARALLEL SOLUTION OF SPARSE LINEAR SYSTEMS ON A VECTOR MULTIPROCESSOR COMPUTER



**Pi-Yu Chung**

UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN

SECURITY CLASSIFICATION OF THIS PAGE

# REPORT DOCUMENTATION PAGE

Form Approved OMB No. 0704-0188

| | |
|---|---|
| 1a. REPORT SECURITY CLASSIFICATION: Unclassified | 1b. RESTRICTIVE MARKINGS: None |
| 2a. SECURITY CLASSIFICATION AUTHORITY | 3. DISTRIBUTION/AVAILABILITY OF REPORT: Approved for public release; distribution unlimited |
| 2b. DECLASSIFICATION/DOWNGRADING SCHEDULE | |
| 4. PERFORMING ORGANIZATION REPORT NUMBER(S): UILU-ENG-90-2227 (DAC-23) | 5. MONITORING ORGANIZATION REPORT NUMBER(S) |
| 6a. NAME OF PERFORMING ORGANIZATION: Coordinated Science Lab, University of Illinois; 6b. OFFICE SYMBOL (If applicable): N/A | 7a. NAME OF MONITORING ORGANIZATION: Office of Naval Research |
| 6c. ADDRESS (City, State, and ZIP Code): 1101 W. Springfield Ave., Urbana, IL 61801 | 7b. ADDRESS (City, State, and ZIP Code): Arlington, VA 22217 |
| 8a. NAME OF FUNDING/SPONSORING ORGANIZATION: Joint Services Electronics Program; 8b. OFFICE SYMBOL (If applicable) | 9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER: N00014-84-C-0149 |
| 8c. ADDRESS (City, State, and ZIP Code): Arlington, VA 22217 | 10. SOURCE OF FUNDING NUMBERS: PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO. / WORK UNIT ACCESSION NO. |

11. TITLE (Include Security Classification)

Parallel Solution of Sparse Linear Systems on a Vector Multiprocessor Computer

12. PERSONAL AUTHOR(S)

Chung, Pi-Yu

| 13a. TYPE OF REPORT | 13b. TIME COVERED | 14. DATE OF REPORT (Year, Month, Day) | 15. PAGE COUNT |
|---|---|---|---|
| Technical | FROM 08/88 TO 08/90 | 1990 July 30 | 81 |

16. SUPPLEMENTARY NOTATION

| 17. COSATI CODES | | | 18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number) |
|---|---|---|---|
| FIELD | GROUP | SUB-GROUP | |
| | | | Sparse systems, partitioning, parallel solution, multiprocessing, vectorization, LU factorization, multilevel nodetearing |

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

This paper describes an efficient approach for solving sparse linear systems using direct method on a shared-memory vector multiprocessor computer. The direct method is divided into three steps: LU factorization, forward substitution and backward substitution. If the size of the linear system is large, LU factorization is a very time-consuming step, so that concurrency and vectorization are exploited to reduce execution time. Parallelism of LU factorization is obtained by partitioning the matrix using multilevel node-tearing techniques. The partitioned matrix is reordered into a NBBD (Nested Bordered-Block Diagonal) form. A nested-block data structure is used to store the sparse matrix, enabling the use of vectorization as well as multiprocessing to achieve high performance. This approach is suitable for many applications that require the repeated direct solution of sparse linear systems with identical matrix structure, such as circuit simulation. The approach has been implemented in a program that runs on an ALLIANT FX/8 vector multiprocessor with shared memory. Speedups in execution time compared to conventional serial computation with no vectorization are up to 20 using eight processors.

| 20. DISTRIBUTION/AVAILABILITY OF ABSTRACT | 21. ABSTRACT SECURITY CLASSIFICATION |
|---|---|
| ☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT. ☐ DTIC USERS | Unclassified |
| 22a. NAME OF RESPONSIBLE INDIVIDUAL | 22b. TELEPHONE (Include Area Code) / 22c. OFFICE SYMBOL |

DD Form 1473, JUN 86 — Previous editions are obsolete. — SECURITY CLASSIFICATION OF THIS PAGE

UNCLASSIFIED

# PARALLEL SOLUTION OF SPARSE LINEAR SYSTEMS ON A VECTOR MULTIPROCESSOR COMPUTER

BY

PI-YU CHUNG

B.S., National Taiwan University, 1986

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Electrical Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 1990

Urbana, Illinois

# ABSTRACT

This thesis describes an efficient approach for solving sparse linear systems using the direct method on a shared-memory vector multiprocessor computer. The direct method is divided into three steps: LU factorization, forward substitution and backward substitution. If the size of the linear system is large, LU factorization is a very time-consuming step, so that concurrency and vectorization are exploited to reduce execution time. Parallelism of LU factorization is obtained by partitioning the matrix using multilevel node-tearing techniques. The partitioned matrix is reordered into a NBBD (Nested Bordered-Block-Diagonal) form. A nested-block data structure is used to store the sparse matrix, enabling the use of vectorization as well as multiprocessing to achieve high performance. This approach is suitable for many applications that require the repeated direct solution of sparse linear systems with identical matrix structure, such as circuit simulation. The approach has been implemented in a program that runs on an ALLIANT FX/8 vector multiprocessor with shared memory. Speedups in execution time compared to conventional serial computation with no vectorization are up to 20 using eight processors.

# ACKNOWLEDGEMENTS

I would like to express my sincere appreciation to my advisor, Professor Ibrahim N. Hajj, for his consistent support, valuable discussions, and constant encouragement.

I would also like to express my gratitude to Dr. Mi-Chang Chang of Texas Instruments, Inc., for his helpful guidance and support. Thanks also go to the Center for Supercomputing Research and Development for providing access to the ALLIANT FX/8 Computer and to Professor Resve A. Saleh, Dr. Kyle A. Gallivan and Dr. Gung-Chung Yang at CSRD for helpful information on programming the ALLIANT FX/8 and accessing examples. I would also like to thank all the members of the Digital and Analog Circuits Group of the Coordinated Science Laboratory, especially Yun-Cheng Ju, for their assistance that made this thesis possible.

Finally, special thanks go to my father, Chung-Ming Chung, my mother, Fen-Fang Hsu, and my husband, Yi-Min Wang, for their love, support, understanding and encouragement.

This research was supported by the Joint Services Electronics Program, contract number N00014-84-C-0149.

# TABLE OF CONTENTS

# LIST OF TABLES

# LIST OF FIGURES

# CHAPTER 1

# INTRODUCTION

It is known that the most time-consuming task in computer simulation of large systems is solving large sparse linear systems. Many efforts have been made to use the power of parallel/vector computers in speeding up sparse matrix computations [1-9]. In this thesis we consider the solution of linear sets of equations by the direct method as opposed to relaxation methods. The direct method is used when relaxation methods are expected to be too slow or nonconvergent.

Consider the direct solution of

$$\mathbf{A}\,\mathbf{x} = \mathbf{b} \tag{1.1}$$

where A is a real n x n sparse matrix, b is the right-hand side vector, and x is the unknown vector. The solution is usually divided into three steps: LU factorization, forward substitution and backward substitution. The time complexity of LU factorization is one order greater than that of forward and backward substitutions. When the system is large, LU factorization dominates the solution time. Given that $n$ is the number of equations in the linear system, it is known that the time complexity for LU factorization is $O(n^3)$ for a full matrix. However, the exploitation of sparsity can save enormous computation time. It has been observed that the complexity of the solution algorithm is between $O(n^{1.2})$ and $O(n^{1.8})$, depending on the sparsity of the matrix.

The purpose of our research is to find an efficient solution on the vector multiprocessor computer for those applications that require the repeated direct solution of sparse linear systems with an identical matrix structure, such as circuit simulation. The main problem is that most good sequential LU factorization algorithms for sparse matrices are not suitable to be parallel-

ized and vectorized directly because of high operation dependency and data sparsity. Different methods must be found for the vector parallel computer architecture.

We will concentrate our research on speeding up the LU factorization; nevertheless, speed-ups can also be achieved for forward and backward substitutions using similar methods. The LU factorization involves two types of operations:

(1) Normalization operations, which involve dividing the nonzero elements of a row by the diagonal element.

(2) Update operations, which involve the addition of a multiple of the elements of a source row to the corresponding elements of a target row.

To parallelize the LU factorization one needs to break up the set of operations into a number of tasks in order to identify those which can be performed in parallel at any given step in the solution procedure. In Chapter 2, recent approaches implemented on MIMD (Multiple Instruction stream-Multiple Data stream) computers are reviewed. The parallel solution of sparse linear systems includes three subproblems: (1) determine task size, (2) matrix ordering, and (3) task scheduling. The existing algorithms for these three topics will be briefly described and discussed.

In our research, we find that the data storage scheme is very critical for vectorization. The data sparsity and vector length will determine the efficiency of vectorization. In order to exploit the maximum degree of vectorization, we introduce a special data storage scheme-- nested-block structure in Chapter 3, which is especially suitable for vector multiprocessor computer architecture. We first reorder the sparse matrix into nested Bordered-Block-Diagonal (NBBD) form and store the matrix according to the NBBD form. A detailed example is given to explain how to construct this data structure. A number of advantages are listed which simplify the solution

algorithm and accelerate the execution time.

In Chapter 4, a standard sequential algorithm and our parallel algorithm are presented. Two levels of parallelism are exploited:

(1) multiprocessing concurrency (coarse-grain): by multilevel partitioning

(2) vector concurrency (fine-grain) : by vectorization

The parallelization and vectorization methods are given in detail. Task description and task scheduling are also contained in this chapter.

In Chapter 5, we discuss implementation issues. Our algorithm has been implemented on the ALLIANT FX/8, which is a shared-memory multiprocessor computer with eight vector processors. Speedups compared to a sequential algorithm are given. Moreover, the relationship between the levels of partitioning and solution time is studied. Promising results are obtained: speedups of 6 to 20 can be achieved as compared to those for the conventional sequential approach.

In Chapter 6, we apply our approach to circuit simulation. It is found that the nested-block structure is an excellent choice for parallel circuit simulation. The programming issues for circuit simulation are discussed. The results are compared to those obtained by a sequential circuit simulator.

Chapter 7 concludes this thesis and introduces several tasks planned for the future.

# CHAPTER 2

# OVERVIEW OF PREVIOUS PARALLEL ALGORITHMS

## 2.1. Introduction

This chapter will survey recent approaches to solve sparse linear systems on MIMD computers. We will concentrate on the parallel algorithms of LU factorization for large sparse matrices using MIMD computers, specially those approaches suitable for application domains such as circuit simulation that require the repeated direct solution of sparse linear systems of equations with identical zero-nonzero structure.

Consider the direct solution of

$$\mathbf{A}\,\mathbf{x} = \mathbf{b} \tag{2.1}$$

in a parallel processing system, where A is n x n, sparse, large, and nonsingular. It is known that the time complexity for LU factorization is $O(n^3)$ for a full matrix. For sparse matrices it has been observed that the complexity of the solution algorithm is between $O(n^{1.2})$ and $O(n^{1.8})$, depending on the sparsity of the matrix. Thus exploiting the sparsity of the matrices is of great importance for minimizing both storage and execution time.

Parallelizing LU factorization of sparse matrices has three subproblems:

(1) determine task size,

(2) matrix ordering and

(3) task scheduling.

The amount of parallelism available depends on the size of the tasks, or task granularity. There are three levels of granularity: fine-grain, medium-grain and large-grain. In Section 2.2, we will

use a simple example to discuss several approaches using different levels of task granularity. In Section 2.3, we will describe how to increase the degree of parallelism and decrease the number of operations by ordering. Different reordering and partitioning techniques are described. In Section 2.4, different scheduling algorithms based on different assumptions are described. Conclusions are then given in Section 2.5.

## 2.2. Task Granularity

### 2.2.1. Fine-grain parallelism

Fine-grain parallelism is the parallelism exploited when the size of each task is a single operation. The LU factorization involves two types of operations.

(1) Normalization operations: dividing the nonzero elements of a row by the pivot.

(2) Update operations : addition of a multiple of the elements of a source row to the corresponding elements of a target row.

Wing and Huang used these two types of operations as individual tasks [1], [2]. Consider the example shown in Figure 2.1. The list of operations needed to LU decompose the matrix is given below.

1. $a_{13} = a_{13} / a_{11}$
2. $a_{33} = a_{33} - a_{13}a_{31}$
3. $a_{23} = a_{23} / a_{22}$
4. $a_{26} = a_{26} / a_{22}$
5. $a_{33} = a_{33} - a_{23}a_{32}$

$$\begin{vmatrix} a_{11} & & a_{13} & & & \\ & a_{22} & a_{23} & & & a_{26} \\ a_{31} & a_{32} & a_{33} & & & a_{36} \\ & & & a_{44} & a_{45} & \\ & & & a_{54} & a_{55} & a_{56} \\ & a_{62} & a_{63} & & a_{65} & a_{66} \end{vmatrix}$$

Figure 2.1. Example 2.1

6. $a_{36} = -a_{32}a_{26}$

7. $a_{63} = -a_{62}a_{23}$

8. $a_{66} = a_{66} - a_{26}a_{62}$

9. $a_{36} = a_{36} / a_{33}$

10. $a_{66} = a_{66} - a_{36}a_{63}$

11. $a_{45} = a_{45} / a_{44}$

12. $a_{55} = a_{55} - a_{45}a_{54}$

13. $a_{65} = a_{65} / a_{55}$

14. $a_{66} = a_{66} - a_{65}a_{56}$

If we assume that each task takes one unit of time, then it takes 14 units of time to complete the LU factorization of the matrix, using a sequential algorithm. The levelized task graph for these operations is shown in Figure 2.2. The numbers in the nodes correspond to the numbers of

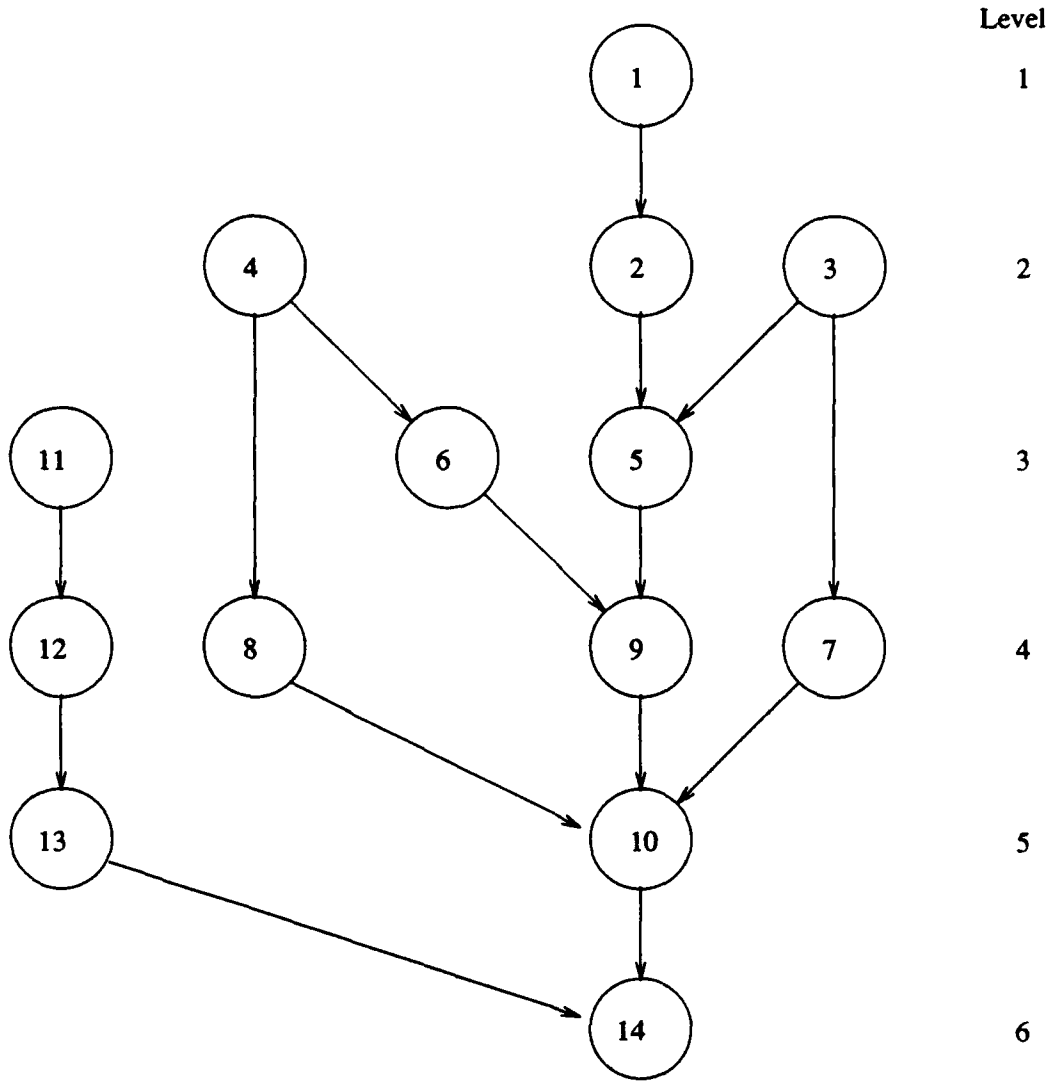



Figure 2.2. Fine-grain task tree of example 2.1

the operations in the list above. The arrows are the edges depicting dependencies. The maximum number of tasks that has to be completed at any level is 4 (at level 4), and with four processors, LU factorization can be completed in 6 units of time instead of 14.

The most important feature for fine-grain parallelism is:

The maximum amount of parallelism between operations can be exploited because tasks cannot be divided further.

However, there are some problems:

(1) The number of processors required to decompose a large system in minimum time is large. It might not be possible to have shared memory MIMD computers that have that many processors. Thus there is another optimal scheduling problem if the number of processors is insufficient.

(2) There is a large overhead required for storing tasks and all temporary results.

We will see that large-grain approaches are free from these two problems.

### 2.2.2. Large-grain parallelism

A large-grain approach used by Chang is based on a multilevel partitioning technique [5]. According to the partitioning, the matrix is reordered into a nested bordered-block-diagonal (NBBD) form. The LU factorization process for the whole matrix can then be divided into several tasks, where each task consists of the LU factorization of the submatrices in diagonal blocks.

The example in Figure 2.1 is in NBBD form. The block representation is shown in Figure 2.3 and its corresponding task graph is shown in Figure 2.4. The set of operations associated with the LU factorization of each submatrix is given below.

$B_{11}$: $a_{13} = a_{13} / a_{11}, a_{33} = a_{33} - a_{13}a_{31}$

$B_{22}$: $a_{23} = a_{23} / a_{22}, a_{26} = a_{26} / a_{22}, a_{33} = a_{33} - a_{23}a_{32}$

$a_{36} = -a_{32}a_{26}, a_{63} = -a_{62}a_{23}, a_{66} = a_{66} - a_{26}a_{62}$

$B_{33}$: $a_{36} = a_{36} / a_{33}, a_{66} = a_{66} - a_{36}a_{63}$

$B_{44}$: $a_{45} = a_{45} / a_{44}, a_{55} = a_{55} - a_{45}a_{54}, a_{65} = a_{65} / a_{55}, a_{66} = a_{66} - a_{65}a_{56}$

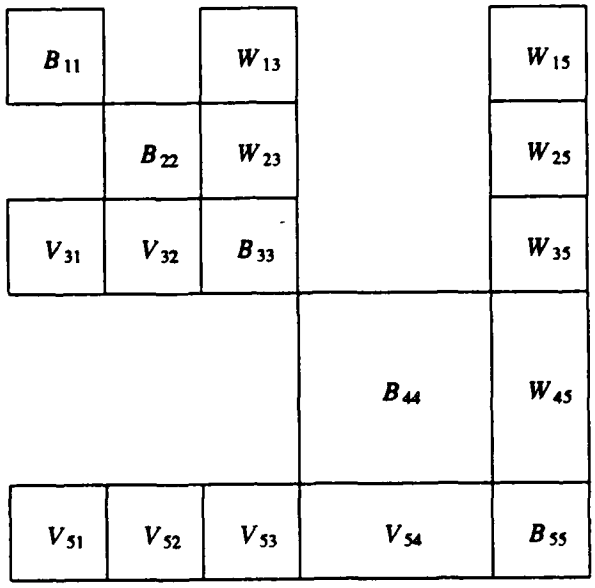



Figure 2.3. NBBD blocks of example 2.1

$B_{55}$: none

For this approach, the task graph is always a tree. The task tree for example 2.1 has three levels: the first level-$B_{55}$ is the root of the tree, the second level-$B_{33}$ and $B_{44}$, are two children of $B_{55}$, and the third level-$B_{11}$ and $B_{22}$, are children of $B_{33}$. Thus two processors are enough to achieve maximum parallelism.

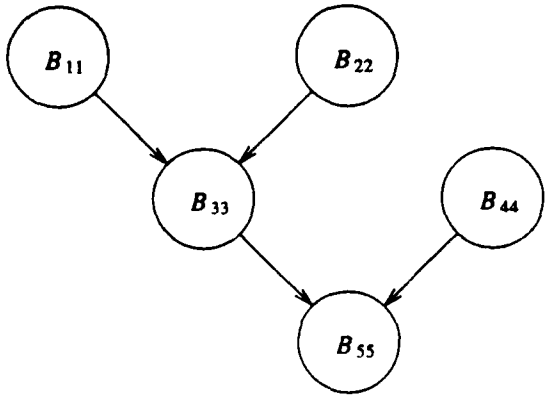



Figure 2.4. Large-grain task tree

Large-grain parallelism has a major drawback. It exploits only a limited amount of parallelism. Chang solved this problem by further partitioning the submatrices into smaller ones. Thus parallelism will increase. But because the tasks do not necessarily take the same amount of execution time, optimal scheduling becomes a more difficult problem.

### 2.2.3. Medium-grain parallelism

Medium-grain parallelism uses tasks consisting of more than one operation. This is achieved by combining a set of nodes in the fine-grain task graph into a single node. Most approaches use a vector operation as a task. Thus they always exploit two levels of parallelism, namely,

(1) The concurrent processing of tasks and

(2) The pipeline processing inside tasks.

There are various ways of implementing the above two levels of parallelism; consequently, there are different kinds of medium-grain parallelism. In the following we describe two such approaches.

Approach 1

Sadayappan and Visvanathan proposed a method for parallel vector machines [6]. Because the matrix elements are stored row by row in compressed vector form, there are one data vector and one index vector corresponding to each row. The set of operations at each task in Figure 2.1 is given below and the task graph is shown in Figure 2.5.

1. $a_{13} = a_{13} / a_{11}$

2. $a_{33} = a_{33} - a_{13}a_{31}$

3. $a_{23} = a_{23} / a_{22}, a_{26} = a_{26} / a_{22}$

4. $a_{33} = a_{33} - a_{32}a_{23}, a_{36} = -a_{32}a_{26}$

5. $a_{63} = -a_{62}a_{23}, a_{66} = a_{66} - a_{62}a_{26}$

6. $a_{36} = a_{36} / a_{33}$

7. $a_{66} = a_{66} - a_{36}a_{63}$

8. $a_{45} = a_{45} / a_{44}$

9. $a_{55} = a_{55} - a_{45}a_{54}$

10. $a_{65} = a_{65} / a_{55}$

11. $a_{66} = a_{66} - a_{65}a_{56}$

The problem is, for update operations, matching source-row elements with the appropriate elements of the various target rows will require scattering and gathering target rows or scanning the target rows to locate the corresponding elements. This requires either large memory or time overhead. In [6] this problem is solved by explicitly enumerating the target elements involved in each operation during a symbolic analysis phase. The indices are then stored in a Target-Indirection-Vector to facilitate source-target element matching at run time.

Approach 2

Chen and Hu proposed a different computation model [7]. They also reordered the matrix into an NBBD form so that a high degree of concurrency can be obtained. The matrix elements are stored row by row in compressed vector form for the upper triangular part and column by column for the lower triangular part, as shown in Figure 2.6. A normalization task of stage $k$ is referred to as $T_k^k$. A row-column updating task at stage $k$ is referred to as $T_j^k$, where $j$ is the number of the row-column pair being updated. The task graph is shown in Figure 2.7. The list

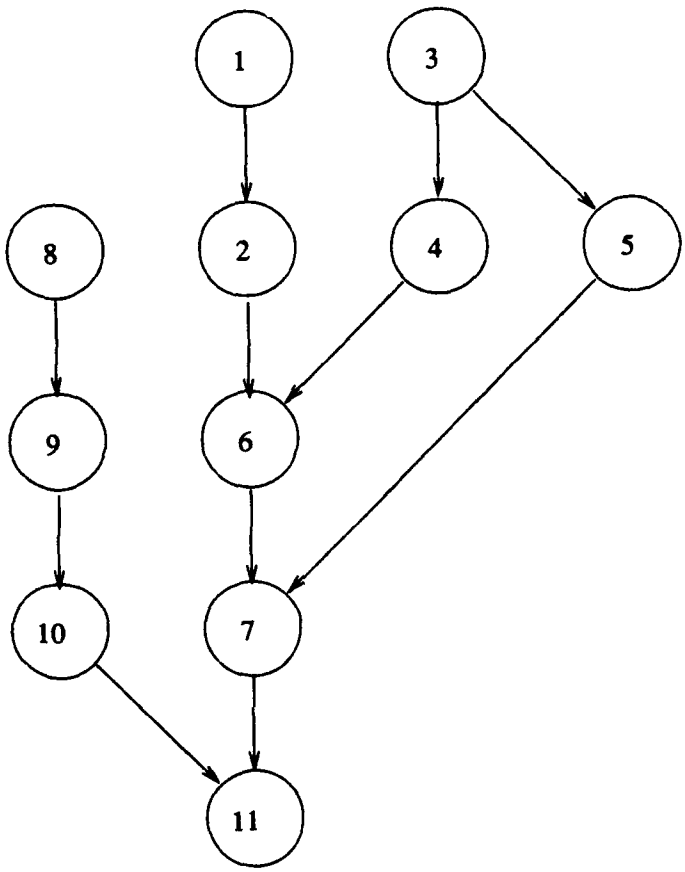



Figure 2.5. Medium-grain task graph of approach 1

of operations necessary for LU factorization is shown below.

$T_1^1 : a_{13} = a_{13} / a_{11}$

$T_1^3 : a_{33} = a_{33} - a_{13} a_{31}$

$T_2^2 : a_{23} = a_{23} / a_{22}, a_{26} = a_{26} / a_{22}$

$T_2^3 : a_{33} = a_{33} - a_{23} a_{32}, a_{36} = -a_{32} a_{26}, a_{63} = -a_{62} a_{23}$

$T_2^6 : a_{66} = a_{66} - a_{26} a_{62}$

$T_3^3 : a_{36} = a_{36} / a_{33}$

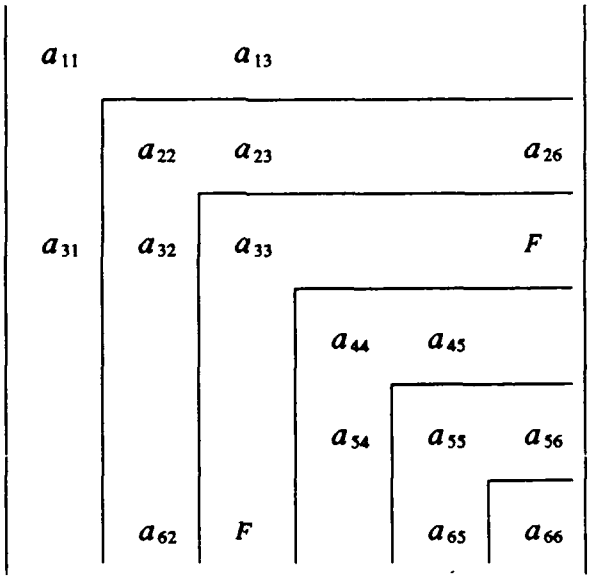



Figure 2.6. Data storage scheme for Chen and Hu's approach

$T_3^6 : a_{66} = a_{66} - a_{36}a_{63}$

$T_4^4 : a_{45} = a_{45} / a_{44}$

$T_4^5 : a_{55} = a_{55} - a_{45}a_{54}$

$T_5^5 : a_{65} = a_{65} / a_{55}$

$T_5^6 : a_{66} = a_{66} - a_{65}a_{56}$

This method has more short vector operations than the previous method does, which makes it very inefficient for applying it on a vector machine because of the short vector operations, but the higher degree of concurrency still can result in speedups.

## 2.3. Ordering

The purpose of ordering is to increase the degree of parallelism and to decrease the number of operations. In sparse matrix techniques, the number of operations depends on the order in which the rows and columns are arranged because of fill-ins during the factorization process.

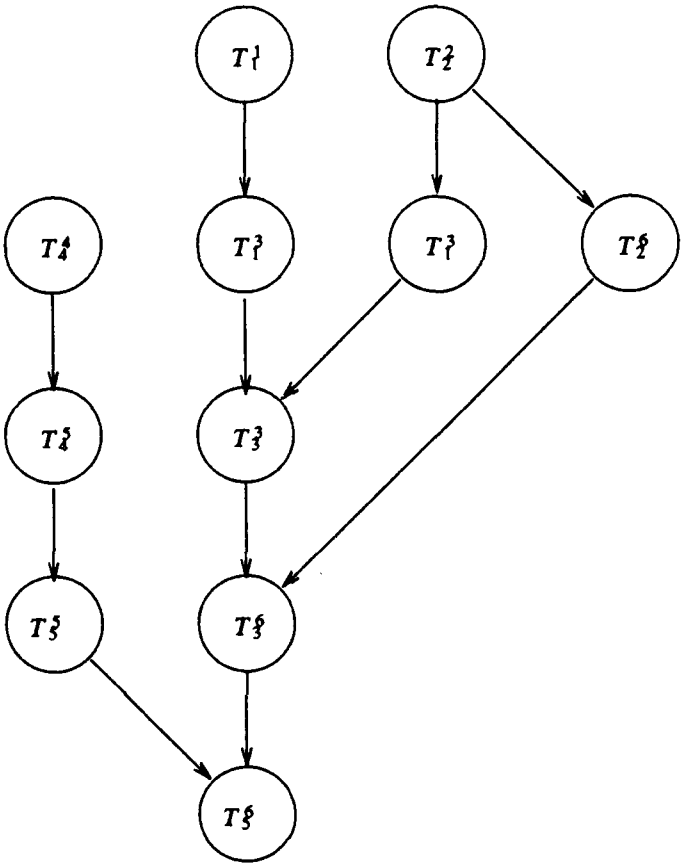


Figure 2.7. Medium-grain task tree (2)

Also, for parallel processing, the degree of parallelism and the minimum completion time are closely related to the matrix ordering. However, the goals of minimizing the completion time and minimizing the fill-ins are conflicting. This makes ordering a difficult problem. There are two type of approaches to finding an appropriate ordering:

(1) reordering, i.e., select the variables that are ordered first first.

(2) partitioning, i.e., select the variables that are ordered last first.

But, so far there is no exact solution to optimal ordering.

### 2.3.1. Reordering

Huang and Wing proposed a heuristic reordering algorithm which chooses the next pivot based on a comparison among all diagonal elements [1]. The algorithm computes two parameters for every unordered pivot each time:

(1) the number of operations required for further decomposition and

(2) the depth that the task graph is expected to grow into.

Because we want to minimize both, the pivot that generates the minimum weighted sum of these two parameters will be picked as the next pivot. The disadvantage of this algorithm is that it requires a time-consuming procedure for monitoring the growth of the task graph depth while choosing pivots.

Another way of doing this reordering is by pivot independency. Smart and White proposed an algorithm called large independent set reordering [10]. In this algorithm, any pivot $i$ can be included in an independent set only if $a_{ij}$ and $a_{ji}$ are both zero for any pivot j already in that set. The basic idea is that at a certain step in the elimination process, a set of candidate pivots is constructed from those pivots with low Markowitz counts. From the set, a large independent set is extracted. All the pivots in an independent set can be processed concurrently with no conflicts, except that more than one pivot may contribute a term to the same update destination. Thus, a certain degree of parallelism is obtained. Test results show that Huang's and Smart's reordering methods obtain approximately the same degree of parallelism [10].

### 2.3.2. Partitioning

Partitioning can be viewed as a graph approach to reordering the matrix. The induced graph for a matrix is constructed as follows: Each row/column corresponds to a vertex in the the

graph. Vertex $i$ and vertex $j$ are connected if and only if $a_{ij}$ is nonzero. To partition the graph, a separator set in the induced graph is found and removed. The remaining graph will have two or more disconnected components. The pivots in each component are ordered first and the pivots in the separator set are ordered at the end of the matrix; thus, the matrix becomes a bordered-block-diagonal form. If we further partition each component and order the pivots inside that component using the same rules, we can obtain an NBBD form ordering.

We already saw that the block dependency in NBBD form has a tree structure. The pivots in different blocks at the same level can be processed concurrently. But the pivots in a parent block (border or separator block) can only be processed after all pivots in the children's blocks are done. In order to minimize the total factorization time, the goal of partitioning is

(1) to minimize the separator sets

(2) to minimize the size of largest submatrices (this is equivalent to finding a balancing partitioning).

There are a number of partitioning algorithms available, but no optimal solution has been found, and the performance of each algorithm depends on the graph structure. We consider three existing general algorithms: nested dissection method [11] , Kernighan and Lin's algorithm [12] and RESP (Restricted Exhaustive Search Partitioning) algorithm by Chang [5].

The nested dissection method proposed by George and Liu is a popular partitioning algorithm. It starts with an initial vertex which is assigned to level 1. Its neighboring vertices are then assigned to level 2, and so on. The set of vertices at level $\frac{L}{2}$ which connects to level $\frac{L}{2}+1$ is then selected as the separator set, where L is the maximum number of levels assigned. This algorithm is fast, but may result in a large separator set and unbalanced partitioning (the sizes of

components could vary a lot).

Kernighan and Lin's graph partitioning algorithm starts with some random partitioning and then tries to exchange subsets of vertices between different subgraphs. Only those exchanges which lead to smaller separator sets are actually performed. The algorithm stops when there are no more exchanges that produce smaller separator sets. This method has been shown to give a near optimal solution for balanced partitioning.

The basis of Chang's RESP algorithm is to check whether there is a separator set with only one node. If so, then the separator clearly has minimum size; otherwise, a node with maximum degree (degree of a node is the number of neighbors of the node) or minimum radius (radius of a node is the maximum of the distance between the node and all other nodes) will be deleted, and the checking for one-node separator continues. The process repeats until a one-node separator set is found. The node together with the already deleted nodes form a separator for the original graph.

The RESP algorithm usually produced unbalanced partitioning but with a smaller separator set. For parallel processing, a small separator set is preferred because the pivots in the separator set are always factorized sequentially after the diagonal blocks are factorized, which seriously decrease parallelism. Thus we choose to apply RESP as our partitioning algorithm.

## 2.4. Scheduling

Scheduling is to assign tasks to a given number of processors such that the constraints in the task graph are followed and total execution time is minimized. Static scheduling is used in almost all approaches; that is, the task assignment is determined before computation.

### 2.4.1. Optimal scheduling

An optimal scheduling solution can be found if the following two conditions are met: (1) the number of processors is infinite and (2) each task takes the same amount of time. Sadayappan and Visvanathan proposed an algorithm to obtain an optimal scheduling [13]. It can be used in both fine-grain and medium-grain approaches as long as the above two conditions stand. They used Minimally Constrained Task Graphs (MCTGs) instead of Directed Acyclic Graphs (DAGs) as in Figure 2.2. The MCTGs contain both directed edges and undirected edges. Directed edges are used only to represent strict temporal dependences, while undirected edges model constraints on the non-simultaneity of execution of multiple updates to a common matrix element. Figure 2.8 shows the MCTG of example 2.1. According to the MCTG, the greedy level assignment algorithm is used to assign tasks to an unbounded number of processors. The greedy level assignment algorithm assigns positive integer level numbers to the nodes of the MCTG so that:

(1) each node has a level number that is higher than that of any of its predecessor nodes.

(2) no two sibling nodes (connected by an undirected edge) are assigned the same level, and

(3) the highest assigned level number is as small as possible.

This algorithm was evaluated and shown to provide up to fifty percent improvement over conventional approaches, but the refinements required to accommodate the characteristics of practical finite-processor systems for their effective scheduling are still open questions for further research.

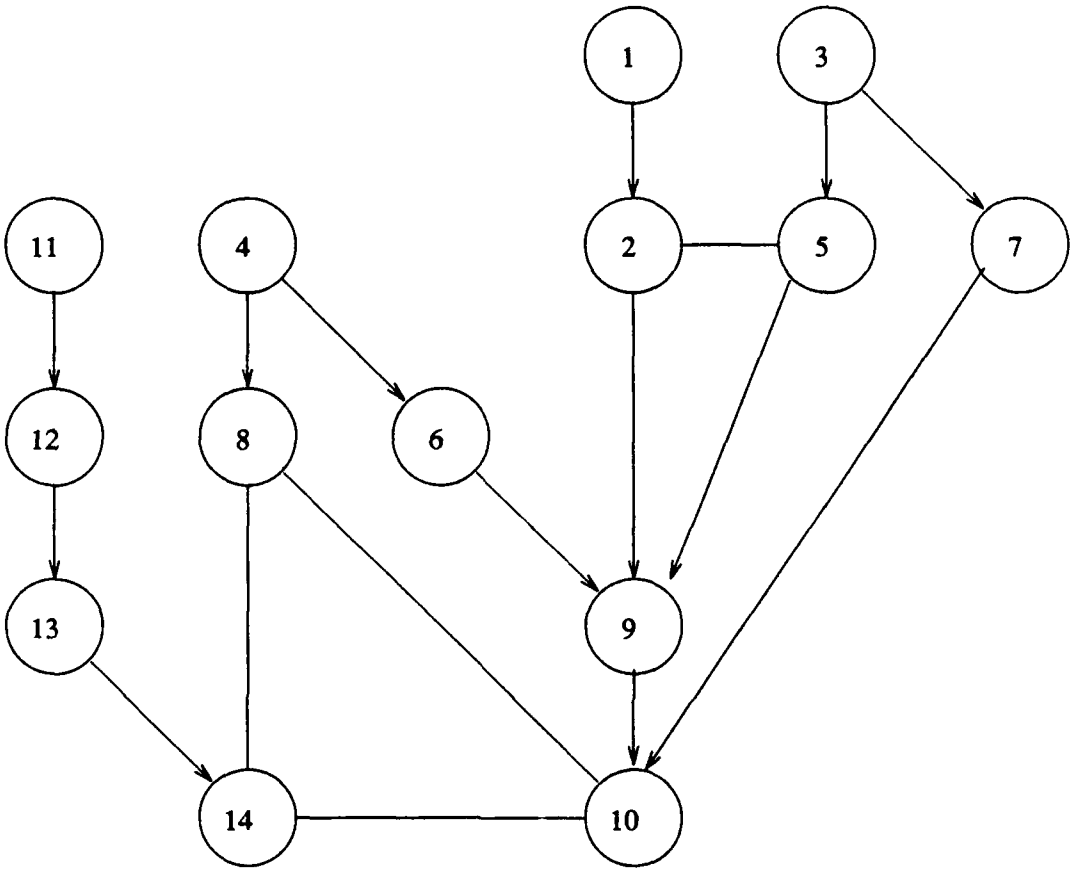



Figure 2.8. The MCTG of example 2.1

### 2.4.2. The levelized scheduling heuristic

If the number of processors is finite and each task takes the same amount of time, Hu's levelized algorithm is usually used [1]. Given a task graph, a node is called a final node if there does not exist another node in the graph which must be executed after it. Conversely, a node is called a starting node if there does not exist another node in the graph which must be executed before it. Let $m$ be the number of processors; the algorithm is described as follows:

(1) Label all nodes with $x+1$, where $x$ is the length of the longest path from the node to the final node in task graph.

(2) If the total number of starting nodes is not greater than $m$, then choose all starting nodes for processing. If it is greater than $m$, choose $m$ starting nodes with values not less than those not chosen.

(3) Remove completed tasks from the graph, and repeat the rule for the remaining graph.

## 2.5. Conclusions

In this chapter, we have discussed three aspects of the parallel solution of sparse linear systems, namely, task granularity, ordering and scheduling.

The approaches corresponding to the three levels of task granularity are described. Although fine-grain parallelism exploits the maximum amount of parallelism, it has high scheduling and memory overhead. Large-grain parallelism, although relatively free from such problems, exploits very little parallelism. Medium-grain parallelism is seen to be a good compromise between the two extremes.

The issue of ordering for increasing parallelism and decreasing the number of operations was also addressed. We saw that the two goals were conflicting and there was no optimal solution for this problem right now. For small-grain approaches, Huang and Wing's method compromises these two factors during reordering. Smart and White's method tries to obtain a large independent set of pivots and at the same time keep the Markowitz sum small. For large-grain approaches, partitioning is always used to obtain nested bordered-block-diagonal form. Minimizing separator sets and balancing submatrices are two objectives of partitioning. Nested dissection recursively cuts graphs in the middle. It is fast, but the results may not be good for certain structures. Kernighan and Lin's algorithm which iteratively exchanges vertices in different components seems to produce better partitioning.

Scheduling is an important aspect of any parallel algorithm. The optimal solution can be found if the number of processors is infinite and the tasks are homogeneous. For small-grain approaches, Hu's levelized scheduling heuristic is used if the number of processors is finite.

Chen and Hu ran their algorithm on the Sequent Balance 21000. Sadayappan and Visvanathan implemented their medium-grain approach on a Cray X-MP using overlap-scatter data structure. It is not surprising that their medium-grain approach exploiting vector processing gives the most promising speedups.

# CHAPTER 3

# A DATA STRUCTURE FOR LARGE SPARSE MATRICES

## 3.1. Introduction

A good data structure for large sparse matrices is very critical to the speed of the LU factorization process, especially for solving large sparse matrices efficiently on a vector multiprocessor computer. The important characteristics of a data structure for sparse matrices on a vector machine are the following :

(1) It must preserve the sparsity of the matrices to save memory as well as number of operations.

(2) The matrix elements should be stored in a vector form in order to be accessed fast and be suitable for vector operations.

(3) The matrix elements are always stored row by row each in a single vector; therefore, the data structure should provide an efficient way for matching two rows with different element distributions.

To exploit both parallelism and vectorization, we derive a new storage scheme--nested-block structure, which has the three features listed above. The basic idea is obtained from the nested Bordered-Block Diagonal form for sparse matrices. We will explain the details in later sections.

In Section 3.2, we will discuss some popular storage schemes currently used in most applications. In Section 3.3, we will introduce the nested Bordered-Block Diagonal form for a sparse matrix. In Section 3.4, nested-block structure will be described.

### 3.2. Existing Data Structure for Sparse Matrices

In this section, we will discuss existing data structures used for storing sparse matrices and explain why they are not suitable for applications on parallel vector machines.

The conventional orthogonal linked list structure [14] does preserve the sparsity of matrices, but it is impractical for efficient operand access and vector operations. Another disadvantage is in greater storage demands because of having to hold the links (pointers) for each elements.

Another alternative is to store each row as a packed sparse vector [15], also called scatter-gather approach [6]. Because compressed vectors are used to store matrix elements, explicit scattering and gathering of vectors to match source and target rows are required. The operations involving indirect addressing are not efficient for application on parallel vector machines.

Recently, Sadayappan and Visvanathan suggested a new data structure, namely, overlap-scatter representation of sparse matrices [16], in which they put every row of the matrix into one long vector without compression. Because the matrix is sparse, the rows may overlap one another to save space as long as no two nonzero elements occupy the same location. Although this method saves the scattering and gathering operations, the fitting strategy itself requires time-consuming overhead. Also, it may need twice as much storage as the scatter-gather approach.

### 3.3. Nested Bordered Block Diagonal Form

Our approach to solving large sparse linear systems is to use a nested Bordered-Block-Diagonal (NBBD) form. The NBBD matrix can be obtained by multilevel node tearing techniques [5] , [17], which partition the graph representation of the matrix recursively, then order the matrix according to the partitioning. The LU factorization process for an NBBD form atrix

can be divided into several tasks. Each task factorizes a block.

Figure 3.1 shows a graph representation of a network which is a subcircuit extracted from a bus layout. By multilevel node tearing techniques, the graph is first separated into two parts by deleting node 13, then into four parts by deleting nodes 6 and 12. The subgraph 4-5-3 can be further partitioned into two parts by deleting node 5. According to the partitioning, the NBBD form of the matrix is shown in Figure 3.2. Figure 3.3 shows the ten blocks in this example, which are A, B1-B3, C1-C4, D1 and D2.

The LU factorization for this matrix can be divided into ten tasks, corresponding to the ten blocks. The dependency relationship of the tasks (task tree) is shown in Figure 3.4. The constraint is that a block cannot be factorized until all of its child blocks are done. A number of tasks can be executed concurrently as long as they have no ancestor-descendent relationship.

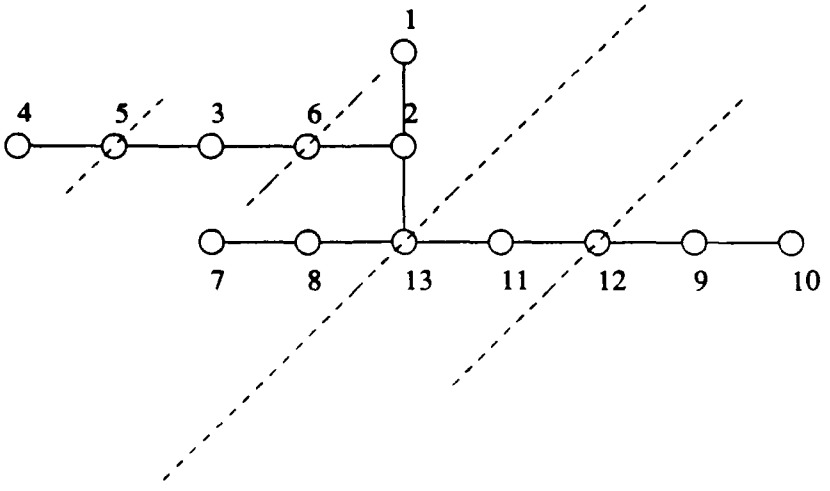



Figure 3.1. Example of multilevel node tearing

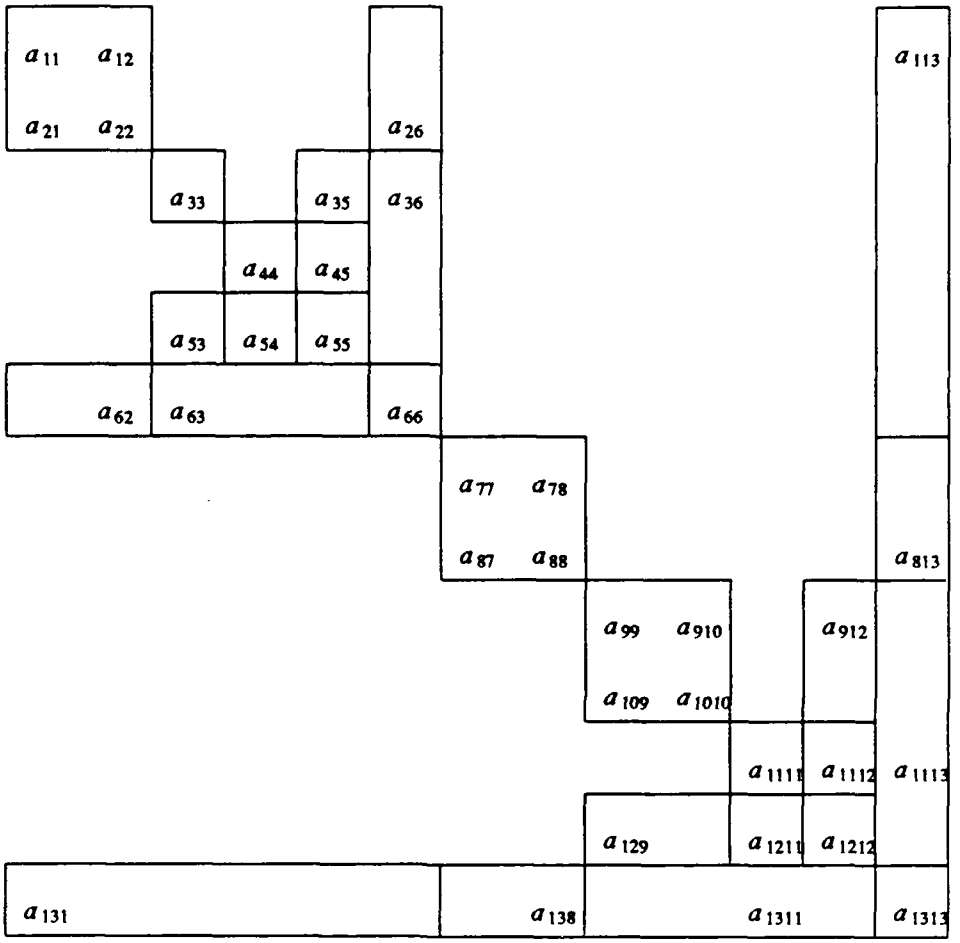



Figure 3.2. The NBBD matrix of example 3.1

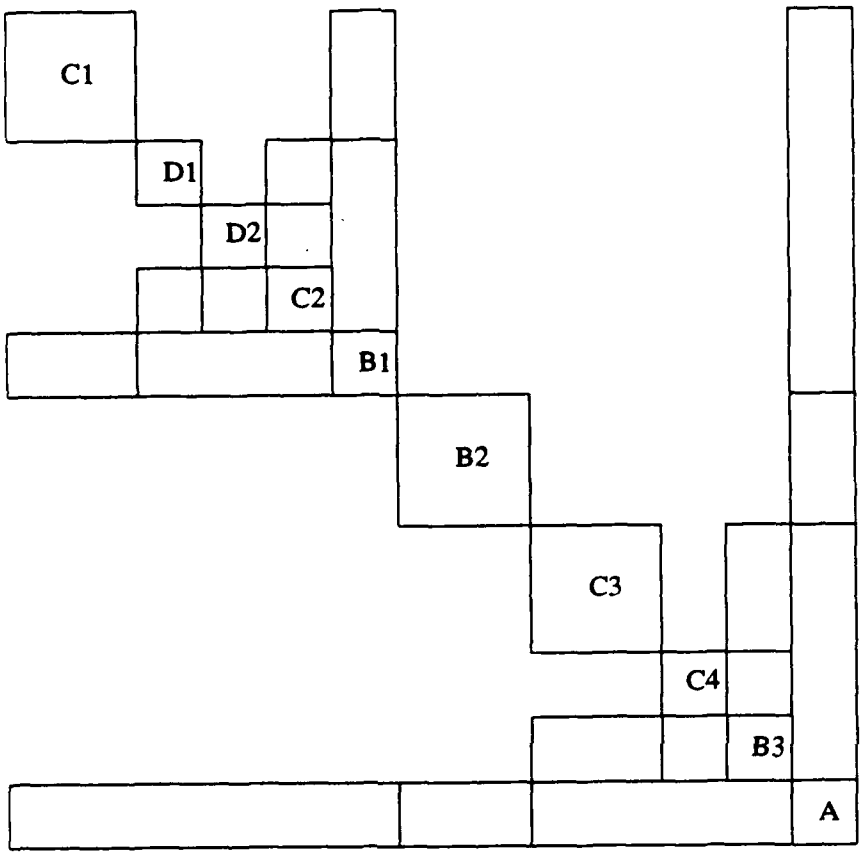



Figure 3.3. The blocks of the NBBD matrix in Figure 3.2

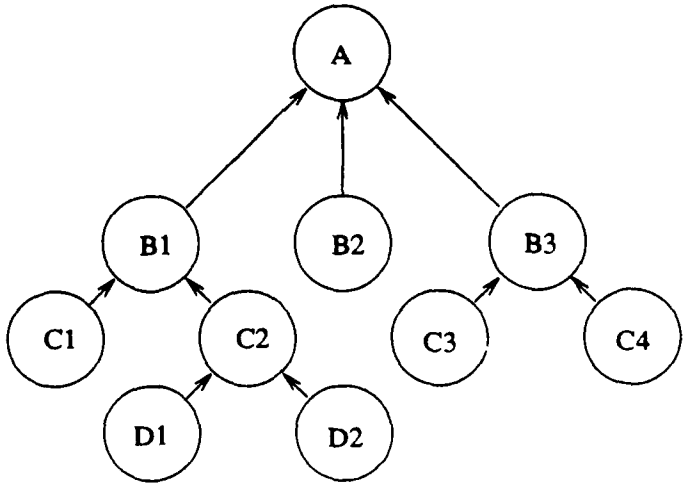



Figure 3.4. The task tree of the NBBD matrix in Figure 3.2

### 3.4. Nested-block Structure for NBBD Matrices

The basic storage unit for nested-block structure is a "block." We define a diagonal submatrix plus its border as a block. The blocks are linked in the task tree structure, see Figure 3.5.

The nested-block structure requires data storage only for the lowest-level blocks. In Figure 3.5, C1, D1, D2, B2, C3 and C4 are the lowest-level blocks; thus, only the storage for these six blocks is needed. The matrix elements of each block are stored row by row in one long vector. There is an address vector for each block which stores the beginning address of each row, so the operands can be accessed directly as in a two-dimensional array. Because the upper-level blocks

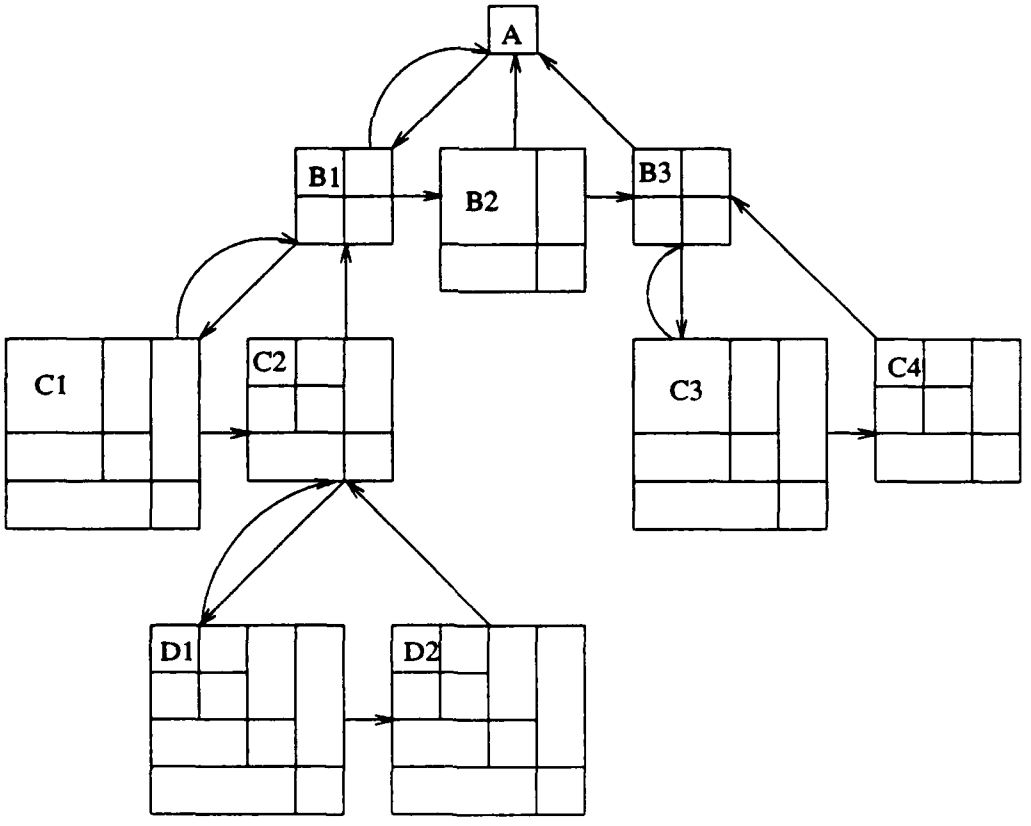



Figure 3.5. Nested-block structure for the example in Figure 3.2

(A, B1, C2 and B3 in this example) are subblocks of the lowest-level blocks, we can make the address vector of these blocks point only to the corresponding addresses in the lowest-level block without requiring any additional storage.

Figure 3.6 shows that the six lowest-level blocks cover all nonempty parts of the entire matrix and upper-level blocks can just be located inside them. For example, block A is inside

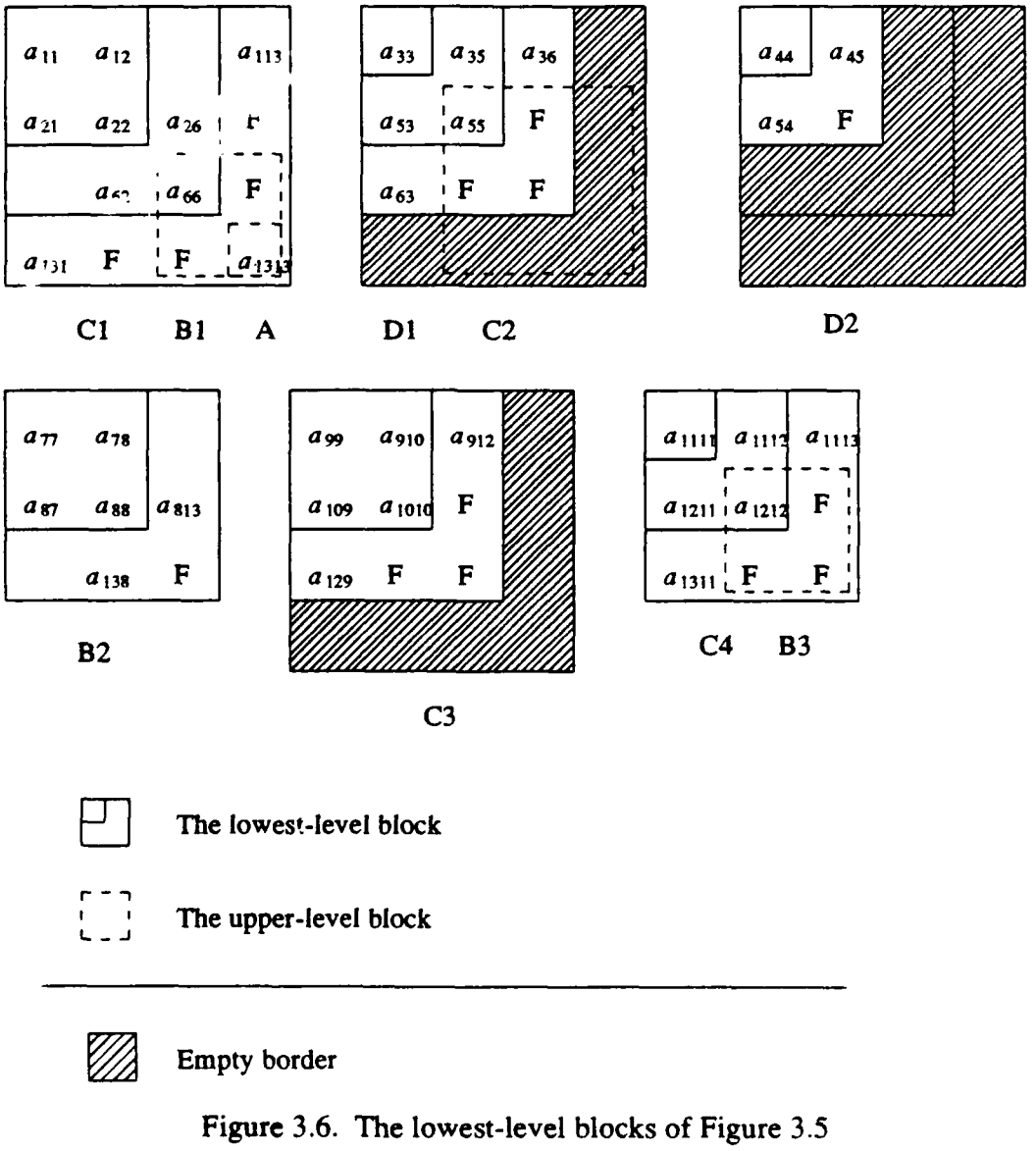



Figure 3.6. The lowest-level blocks of Figure 3.5

block B1 and block B1 is inside block C1. This is the reason we name this storage scheme nested-block structure. The data of the lowest-level blocks are stored one by one as a row-oriented two-dimensional array. The memory arrangement is illustrated in Figure 3.7 and the address vector for each block is shown as in Figure 3.8.

A block could possibly have empty borders at some levels, because a subgraph may not be connected to the cut vertices at all of its upper levels. For example, in Figure 3.1, block D1 (node 3) is not connected to node 13 (cut vertex at first level) and block D2 (node 4) is not connected to node 6 (cut vertex at second level) and node 13. The empty borders are indicated by the shadow area in Figure 3.6. If the empty border is at the end of the block, we can get rid of the empty border so that the block size as well as memory space are reduced. In this example, the size of D1 is reduced from 4 x 4 to 3 x 3 and the size of D2 is reduced from 4 x 4 to 2 x 2. If the empty border is at the middle of the block, we keep the block size unchanged, but put a null in the address vector for a completely empty row, as in Figure 3.9. In this case, the memory space reduces from 4 x 4 to 3 x 4. The reason why we keep the block size unchanged is that when considering LU factorization, we have to keep the columns aligned for different levels of blocks so that no scatter-gather operations will be necessary.

We have implemented this data structure in C language. The following record is used to define a block:

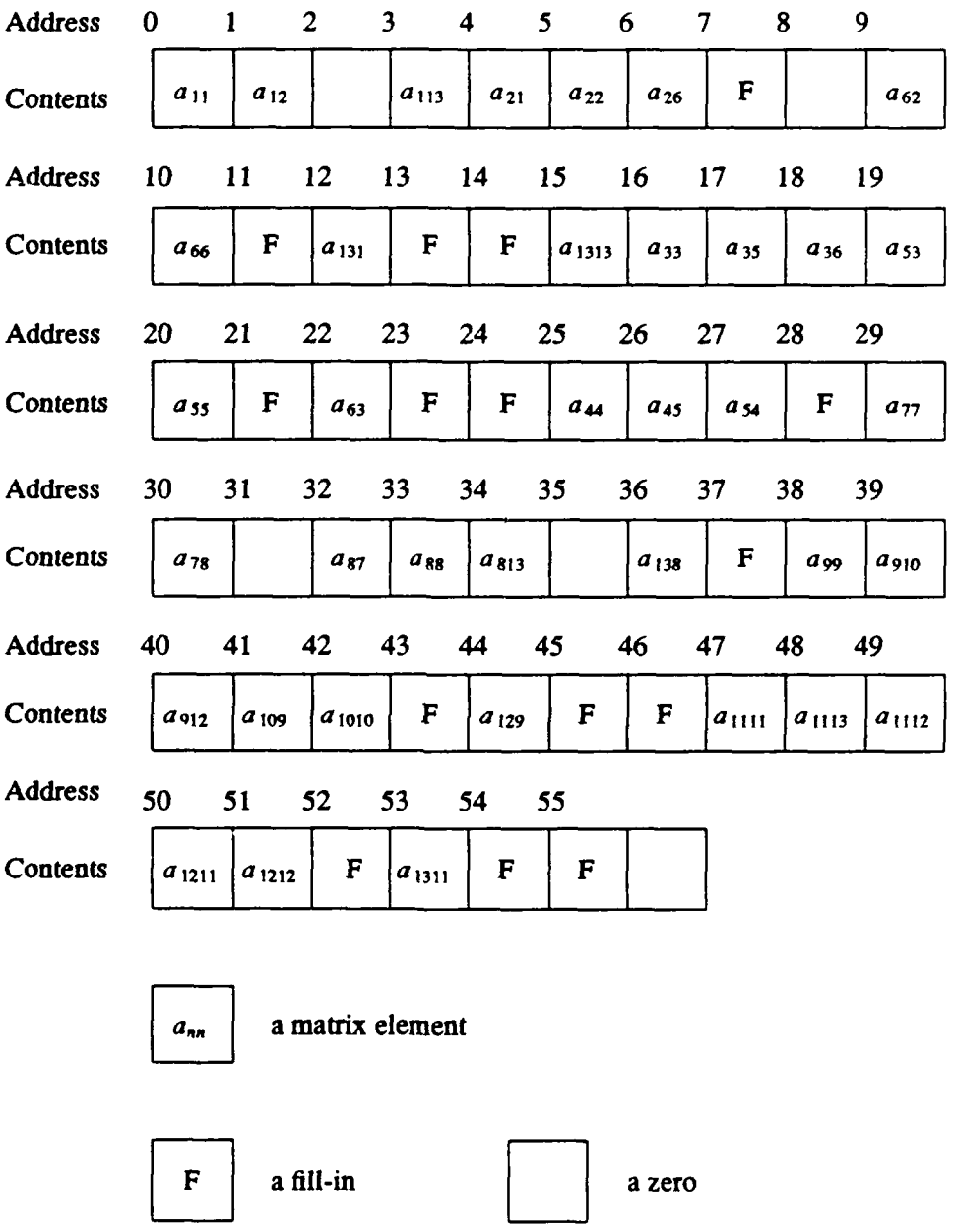



Figure 3.7. The vector which stores the matrix elements

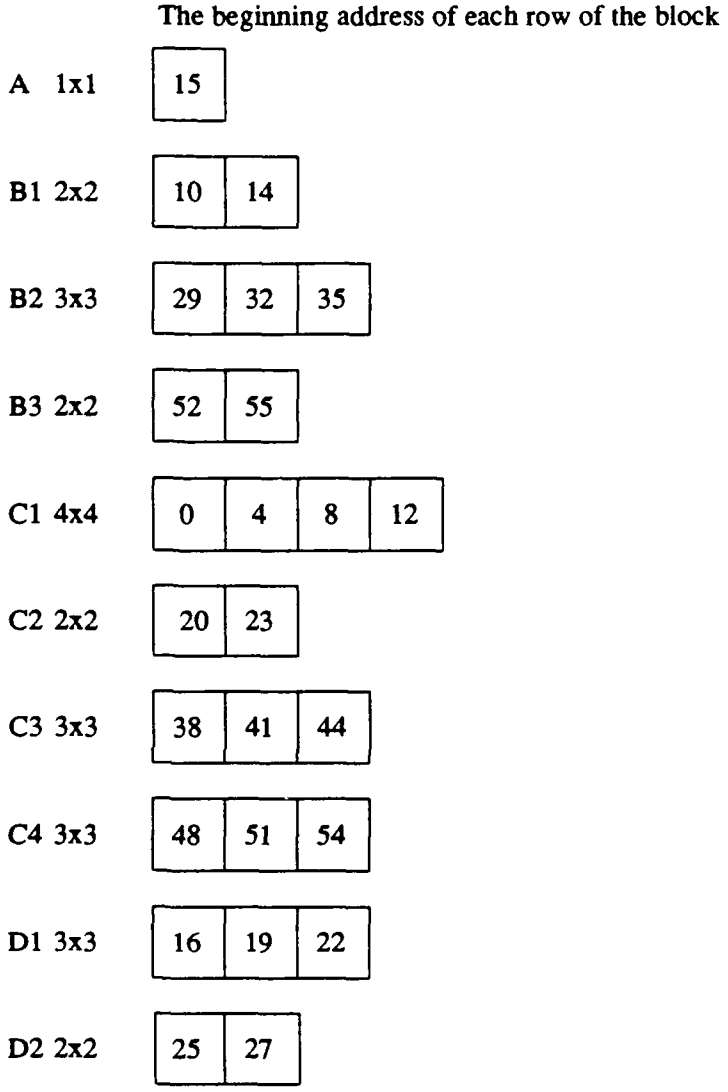



Figure 3.8. Address vectors for each block

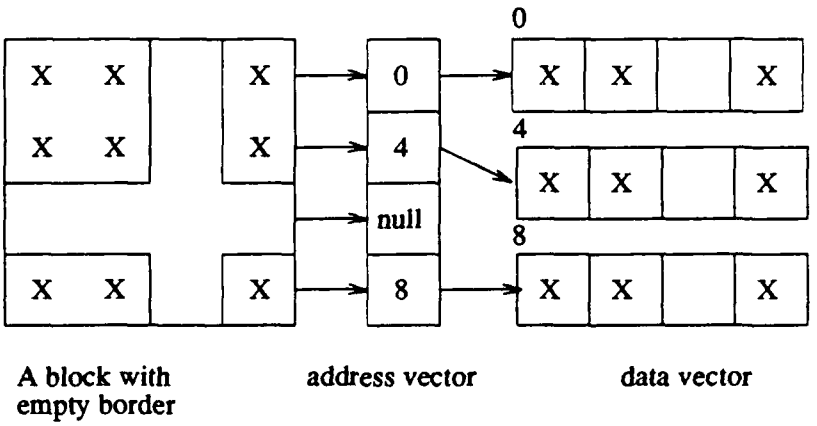



Figure 3.9. The data storage for a block with empty border

```
typedef struct blkr {
    int inode;      /* number of internal nodes or
                       the size of diagonal submatrix */
    int tnode;      /* the size of block */
    struct blkr *son;      /* pointer to son block */
    struct blkr *par;      /* pointer to parent block */
    struct blkr *sib;      /* pointer to sibling block */
    double *address[];     /* address vector */
} blkrec;
```

For example, for the B1 block in Figure 3.5, the inode is 1, tnode is 2, son is pointing to C1, par is pointing to A , sib is pointing to B2 and address is a array of length 2 storing the beginning address of the two rows as in Figure 3.8. This record will be referred to later many times when we describe the algorithms.

# CHAPTER 4

# AN EFFICIENT PARALLEL SOLUTION ALGORITHM

## 4.1. Introduction

In this chapter, we will introduce an efficient parallel sparse linear system solver on a vector multiprocessor computer. We consider the solution of linear sets of equations by the direct method as opposed to relaxation methods. The direct method for solving a linear set of equations can be divided into an LU factorization step and forward substitution and backward substitution steps. When the system is large, LU factorization dominates the solution time.

Our LU factorization algorithm basically follows Gauss' algorithm, also known as Source-Row Directed form. Gauss algorithm involves two types of operations:

(1) Normalization operations, which involve dividing the nonzero elements of a row by the diagonal element.

(2) Update operations, which involve the addition of a multiple of the elements of a source row to the corresponding elements of a target row.

To parallelize the LU factorization of sparse matrices one needs to break up the set of operations into a number of tasks in order to identify those which can be performed in parallel at any given step in the solution procedure. In our approach, both fine-grain (done by vectorization) and coarse-grain (done by partitioning) parallelisms are adopted. Both forward and backward substitution can also be parallelized in a similar way.

In Section 4.2, a standard sequential algorithm will be reviewed. In Section 4.3, we will discuss our parallel algorithms in detail.

## 4.2. A Standard Sequential Algorithm

In this section, we review a sequential algorithm for the direct solution of general sparse linear systems. Let

$$A\,x = b \tag{4.1}$$

where A is a real, N x N sparse matrix; b is the right-hand side vector; and x is the unknown vector where both b and x are of dimension N. The solution to (4.1) is usually carried out in two steps:

(1) LU factorization:

$$A = L\,U \tag{4.2}$$

(2) Forward and backward substitutions:

$$x = U^{-1}\,L^{-1}\,b \tag{4.3}$$

where L is a lower triangular matrix with nonzero diagonal elements and U is an upper triangular matrix with ones on the diagonal. The algorithms for these two steps are listed in Figure 4.1.

## 4.3. Parallel Algorithms

Our approach to solving (4.1) can be listed in the following steps:

(1) Partition the linear system by multilevel node tearing techniques.

(2) Reorder the matrix into nested-bordered-block diagonal form and store matrix elements in nested-block structure.

(3) Schedule the blocks according to block dependency for parallel processing.

(4) Distribute jobs to different vector processors to perform LU factorization and forward and backward substitutions.

*ALGORITHM* LUFACT

```
for k=1 to N-1 do
begin
   forall ( j>k and A_kj≠0 ) do
      A_kj=A_kj/A_kk;
      forall ( i>k and A_ik≠0 ) do
         A_ij=A_ij-A_ik*A_kj;
      endforall
   endforall
endfor
```

*ALGORITHM* FORSUB

```
for k=1 to N do
   forall ( j<k and L_kj≠0 ) do
      b_k=b_k-L_kj*b_j;
   endforall
   b_k=b_k/L_kk;
endfor
```

*ALGORITHM* BACKSUB

```
for k=N-1 to 1 do
   forall ( j<k and U_kj≠0 ) do
      x_k=b_k-U_kj*x_j;
   endforall
endfor
```

Figure 4.1. Sequential algorithms for direct methods

We already considered steps (1) and (2) in Chapters 2 and 3. In this chapter, we will discuss the parallelization in our approach.

### 4.3.1. Task description

The procedures of LU factorization and forward/backward substitutions can be divided into a number of tasks. Each task operates on a block. The task graph of the forward/backward substitutions is the same as that of LU factorization, only the constraint for backward substitution is reversed. The codes for a typical task, block_lufac, block_forsub and block_backsub are listed in Figure 4.2 and 4.3. The data structure used here is given in Section 3.4.

The algorithms listed in Figure 4.2 and 4.3 are very efficient. The speed is gained because of the following reasons:

(1) All operations can be directly applied on the matrix elements without making another copy first.

(2) The operands can be accessed efficiently without any scatter-gather process or tracing a long linked-list.

(3) Because of the arranged column alignment, the updating operations in LU factorization can be done simply by adding multiples of one vector to another vector.

(4) The factorization of a block will update its parent block. If its parent block is inside it, the operation is done implicitly. Otherwise, add the corresponding vectors to the parent block.

(5) Instructions V1, V2, V3, V4 and V5 are vector operations. They can be vectorized to increase throughput.

```
block_lufac(B : block)
begin
    for i=1 to B.inode do              /* Normalization */
    begin
      source_row[ ]←B.address[i]+i+1;
      pivot←B.address[i]+i;
      length←B.tnode−i−1;
      for j=1 to length do
      begin
          source_row[j]←source_row[j] / pivot;    /* V1 */
      endfor
    endfor

    for k=i+1 to B.tnode do            /* Updating */
    begin
      target_row[ ]←B.address[k]+i+1;
      factor←B.address[k]+i;
      for j=1 to length
      begin
          target_row[j]←target_row[j] + factor * source_row[j];    /* V2 */
      endfor
    endfor

    if (B's parent block A is not inside B)          /* Updating parent blocks */
    begin
      length←B.tnode−B.inode;
      for i=1 to length;
      begin
          source_row[ ]←B.address[i+B.inode]+B.inode;
          target_row[ ]←A.address[i];
          for j=1 to length
          begin
            target_row[j]←target_row[j]+source_row[j];    /* V3 */
          endfor
      endfor
    endif
end
```

Figure 4.2. The LU factorization of a block

```
block_forsub(B : block)
begin
    for i=1 to B.tnode do
    begin
      length ←min(i ,B.inode );
      source_row [ ]←B.address [i];
      for j=1 to length do
      begin
          rhs [i]←rhs [i]–rhs [j] * source_row [j];   /* V4 */
      endfor
      if ( i≤B.inode ) rhs [i]←rhs [i] / pivot;
    endfor
end

block_backsub(B : block)
begin
    for i=B.tnode–1 to 1 do
    begin
      source_row [ ]←B.address [i];
      for j=i to B.tnode do
      begin
          rhs [i]←rhs [i]–rhs [j] * source_row [j];   /* V5 */
      endfor
    endfor
end
```

Figure 4.3. Forward/backward substitutions of a block

### 4.3.2. A sequential algorithm

A sequential algorithm for solving a matrix in NBBD form using nested-block structure is listed in Figure 4.4. The subroutines block_lufac, block_forsub and block_backsub are

described in Section 4.3.1. Because the dependency graph is a tree, the task constraint could be obeyed by transversing the tree.

### 4.3.3. Scheduling

Given a number of processors, a schedule assigns tasks to the processors according to a specified order. In our approach, the task graph is determined before processing and the execution time of each task can be confidently estimated, so static scheduling is used. An asynchronous static scheduling heuristic proposed by Chang [5], [17] is implemented which yields near optimal results. It is briefly described as follows:

Assign the starting time in a topdown manner; the root of the tree is first assigned to a processor; when it is done, its sons can all be available for processing. Each processor keeps a task queue. The heuristic will choose an available task and assign it to some processor such that the maximum processing time of all task queues is minimum.

Due to vectorization, we can estimate the execution time by the number of vector operations. The abstract execution time for LU factorization of a block **B** is given by

$$T_{fac}(B) = \sum_{k=1}^{inode}(tnode - k - 1) = tnode * inode - inode * \frac{(inode-1)}{2} \qquad (4.4)$$

There is one exception : the execution time of root block **A** is given by

$$T_{fac}(A) = \sum_{k=2}^{inode} k = inode * \frac{(inode+1)}{2} - 1 \qquad (4.5)$$

The scheduling algorithm is listed in Figure 4.5. The scheduling parameters of example 3.1 for three processors are listed in Table 4.1 and the results are shown in Figure 4.6.

In this example, it will take 31 units of time if run on a single processor. By the scheduling algorithm, it takes 12 units of time, which gives a speedup of 2.6 for three processors.

```
ALGORITHM SOLVE_SQ
  begin
       lufac (root_block);       /* LU factorization */
       forsub (root_block);      /* Forward substitution */
       backsub (root_block);     /* Backward substitution */
  end

  lufac(B : block)
  begin
    forall son_blocks Ci of B
      lufac (Ci);
    block_lufac (B);
  end

  forsub(B : block)
  begin
    forall son_blocks Ci of B
        forsub (Ci);
      block_forsub (B);
  end

  backsub(B : block)
  begin
       block_backsub (B);
    forall son_blocks Ci of B
        backsub (B);
  end
```

Figure 4.4. A sequential algorithm for solving NBBD matrices

The same schedule can be used in both LU factorization and forward substitution. In addition, the schedule can be reversed for use in backward substitution.

Table 4.1. Scheduling parameters of example 3.1

| Task | A | B1 | B2 | B3 | C1 | C2 | C3 | C4 | D1 | D2 |
|---|---|---|---|---|---|---|---|---|---|---|
| fac | 0 | 2 | 5 | 2 | 7 | 2 | 5 | 3 | 3 | 2 |
| acfac | 9 | 9 | 5 | 7 | 7 | 5 | 5 | 3 | 3 | 2 |
| tf | 0 | 0 | 0 | 0 | 2 | 2 | 2 | 2 | 7 | 7 |

*ALGORITHM* SCHEDULING

```
Φ = { A };     /* A is the root block */
A.tf = 0;
for i=0 to noproc do     /* noproc is the number of processors */
begin
      queue[i]=nil;
      ptime[i]=0;
endfor
while Φ is not empty do
begin
      get a task T from Φ with minimum tf;
      if there is a tie, choose the one with maximum acfac;
      get a proc i with minimum ptime.
      push task T to queue[i];
      if (T.tf > ptime[i]) ptime[i] = T.tf;
      ptime[i] = ptime[i] + T.fac;
      forall son blocks X_j of T do
      begin
        X_j.tf = ptime[i];
        Φ = Φ ∪ X_j;
      endforall
endwhile
```

*Definition*

queue[i] : task queue for processor i;

ptime[i] : processing time for processor i;

T.tf : processing time needed after T is finished;

T.fac : the factorization time of T, given by eq (4.4-5);

T.acfac : the accumulative factorization time of T,

$$T.acfac = T.fac + \max \{ X_j.acfac \}$$

where $X_j$ are son block of T;

Figure 4.5. Scheduling algorithm

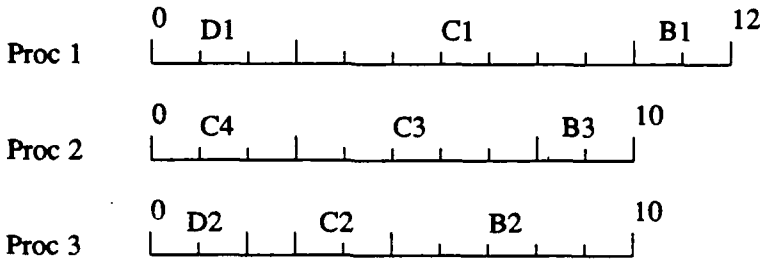



Figure 4.6. Schedule of example 3.1

# CHAPTER 5

# IMPLEMENTATION AND RESULTS

## 5.1. Introduction

The proposed algorithm has been implemented as a linear system solver on a shared memory vector multiprocessor computer ALLIANT FX/8. It is written in C language for the flexibility of experimenting with different data structures for the sparse matrix techniques. Also, the dynamic memory allocation is easy to implement in C. One disadvantage of using C on the Alliant FX/8 is that vectorization must be done explicitly by the programmer, while vectorization of Fortran is done automatically by the compiler [18].

In Section 5.2, we discuss implementation issues on the ALLIANT. In Section 5.3, we compare the results of this algorithm with those of a sequential solver using linked-list structure. In Section 5.4, the relations between speedups and partitioning levels are discussed.

## 5.2. Implementation

The most important features of our approach are that both concurrency and vectorization are adopted in the sparse solver. On the ALLIANT FX/8, we can use up to eight vector processors. Concurrent execution of a procedure is done via the system call *concurrent_call*. The procedure to be executed in parallel is a parameter of the system call. Each processor then receives a copy of the procedure (a task) and executes the code in parallel. The global list scheme is used to pass the data to parallel tasks. Each processor must then lock the pointer, access the pointed parameter, update the pointer, and then unlock the pointer such that other processors can gain access to it.

The library on the ALLIANT contains a large number of routines that perform operations in vector- concurrent mode. The name of a vector routine is of the following form:

vec_*type name[_opn] (arguments)*

*Type* specifies the type of data involved (byte, word, single or double). *Name* is the name of the operation, add or move. *Operation specifies the scope of the operation; for example, vvs means* vector-vector-scalar in a triadic operation.

There are three routines used to implement vector operations (V1-V5 in Figure 4.2 and 4.3). They are

vec_ddiv_vs(*result_vec, operand_vec, divisor, vsize*)

Divides *operand_vec* by *divisor* and stores the result in *result_vec*. This routine is applied to normalization.

vec_dma_vsv(*result_vec, op1_vec, multiplier, op2_vec, vsize*)

Multiplies *op1_vec* with *multiplier*, adds *op2_vec* to the product and stores the results in *result_vec*. This routine is applied to updating.

vec_ddot (*result, op1_vec, op2_vec, vsize*)

Finds the dot product of *op1_vec* and *op2_vec* and stores the product in *result*. This routine is applied to forward and backward substitutions.

Vector routines are called only if the vector length is greater than 6 to prevent overhead of vector startup time. It is found that the vector routines on the ALLIANT take almost the same amount of time for different vector lengths of 1 up to 128. Figure 5.1 shows the cpu time required for sequential codes and vector routines vec_div and vec_ma. Vec_div is not faster than the sequential codes until the vector length is greater than 11 while vec_ma gains speedup as long as vector length is greater than 6. Table 5.1 shows the relationship between speedups

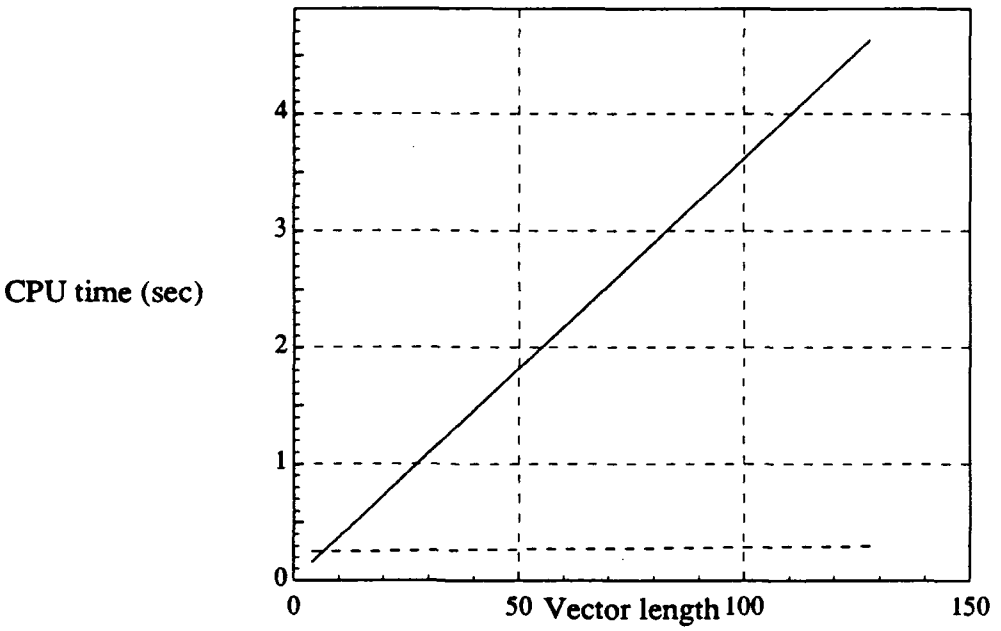
and vector lengths.

The source program of the large sparse linear system solver **solve_p** is listed in the Appendix.

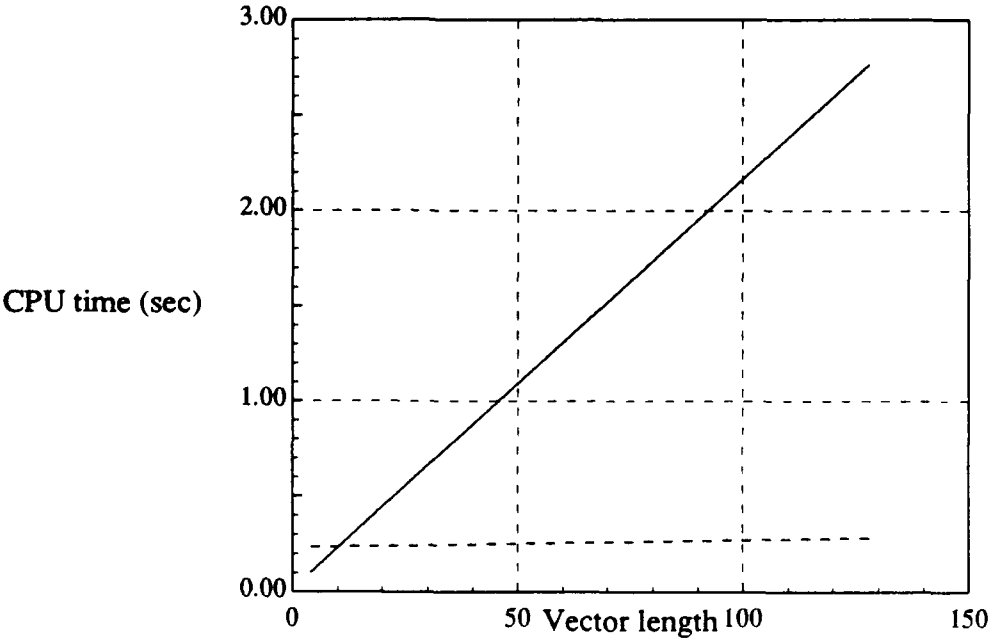
## 5.3. Results

We show the results of solving two sparse matrix examples. Tables 5.2 and 5.3 show the speedups of LU factorization at different partitioning levels using the vectorized algorithm with one processor and eight processors. The results are compared to a sequential solver, also written in C language using the algorithm described in Section 4.2, with a linked-list data structure and the Markowitz reordering technique [14]. Tables 5.4 and 5.5 show the storage requirements for these two different data structures. In linked-list structure, there are 16 bytes required for one nonzero element (1 real number, 2 integers as indices and 2 pointers).

The best results for both examples are obtained with five-level partitioning, where the speedups are maximum and the memory size required is minimum. It is found that the memory size at the optimum level is close to that used in the linked-list structure. It is interesting that the NBBD form of the first matrix has a large dense block; therefore, it does not get much speedup from the coarse-grain parallelism. But the large dense block can be processed efficiently by vectorization and yields a total speedup of more than 20. On the other hand, because the second matrix is well partitioned into small borders and subblocks of equal size, most of the speedups are achieved by the coarse-grain parallelism rather than vectorization. From the results of these two examples, we can see that the proposed algorithm is suitable for factorizing matrices of differing arbitrary structures.




Vector Multiplication and Addition




Vector Division

solid line : sequential codes
dashed line : vector routines

Figure 5.1. CPU time for sequential codes and vector routines (100 iterations)

Table 5.1. Speedups and vector length

| vector length | vec_div | vec_ma |
|---|---|---|
| 8 | 0.859 | 1.267 |
| 16 | 1.467 | 2.188 |
| 32 | 2.800 | 4.733 |
| 64 | 5.188 | 8.750 |
| 128 | 9.765 | 15.444 |

Table 5.2. Speedups

| matrix size : 100 x 100<br>no. of nonzeros : 347 | | |
|---|---|---|
| level | 1 processor | 8 processors |
| 1 | 9.32 | 8.91 |
| 2 | 9.01 | 8.45 |
| 3 | 11.42 | 21.34 |
| 4 | 11.26 | 21.34 |
| 5 | 10.96 | 21.34 |
| 6 | 9.65 | 21.34 |
| 7 | 8.91 | 20.28 |

Table 5.3. Speedups

| matrix size : 237 x 237<br>no. of nonzeros : 647 | | |
|---|---|---|
| level | 1 processor | 8 processors |
| 1 | 0.38 | 0.37 |
| 2 | 0.88 | 2.28 |
| 3 | 1.13 | 4.65 |
| 4 | 1.31 | 6.05 |
| 5 | 1.28 | 6.67 |
| 6 | 1.00 | 5.00 |
| 7 | 0.82 | 3.57 |

Table 5.4. Memory size

| matrix size : 100 x 100 | | | |
|---|---|---|---|
| level | no. of nonzeros (include fill-ins) | linked list (bytes) | nested-block (bytes) |
| 1 | 1081 | 17.3K | 40K |
| 2 | 1087 | 17.4K | 39.5K |
| 3 | 1091 | 17.5K | 23.7K |
| 4 | 1216 | 19.5K | 19.7K |
| 5 | 1215 | 19.4K | 19.7K |
| 6 | 1177 | 18.8K | 20.4K |
| 7 | 1169 | 18.7K | 23.6K |

Table 5.5. Memory size

| matrix size : 237 x 237 | | | |
|---|---|---|---|
| level | no. of nonzeros (include fill-ins) | linked list (bytes) | nested-block (bytes) |
| 1 | 860 | 13.8K | 224.7K |
| 2 | 1019 | 16.3K | 77.4K |
| 3 | 1255 | 20.1K | 41.0K |
| 4 | 1419 | 22.7K | 24.7K |
| 5 | 1477 | 23.6K | 17.0K |
| 6 | 1487 | 23.8K | 17.1K |
| 7 | 1441 | 23.1K | 18.1K |

## 5.4. The Optimal Partitioning Level

In our approach, the total parallel factorization time may vary with different partitioning levels for a given number of processors. As the number of levels in the NBBD form increases, the number of tasks increases, but the size of each task decreases. The total CPU time would seem to decrease because smaller task granularity has higher parallelism. On the other hand, because the size of each block decreases, the vector length decreases; thus, the speedups gained by vectorization are lost. Also, as the number of levels in NBBD form increases, the data storage will initially decrease because more zero elements are discarded, then increase because of too many copies of border blocks. Also as the levels increase more "block fills" are created.

We have observed that the best level of partitioning that results in minimum cpu time usually requires the least block storage. To determine the optimal level of partitioning, the original matrix structure is partitioned into the maximum hierarchy, then flattened to a certain level by combining the submatrices at lower levels to their ancestor at that level. The CPU time and storage can be estimated for each level. The level corresponding to minimum run time is then used in solving the linear equations.

# CHAPTER 6

# APPLICATION IN CIRCUIT SIMULATION

## 6.1. Introduction

Circuit simulation is a very time-consuming and numerically intensive process, especially when the problem size is large as in the case of VLSI circuits. The standard approach to solving the circuit equations is commonly referred to as the direct method and is used in the SPICE2 program. The simulation process includes the following steps:

(1) The circuit problem is described by a system of ODE equations using the modified nodal approach [19].

(2) An implicit integration method is used to convert the differential equations into a sequence of systems of nonlinear algebraic equations.

(3) Newton-Raphson's method is used to transform the nonlinear algebraic equations into linear equations.

(4) The resulting sparse linear equations are solved using LU factorization.

Circuit simulation requires the repeated direct solution of sparse linear systems with identical matrix structures, as in step (4). The linear system solver we discussed in previous chapters is especially suitable for an application such as circuit simulation. In addition, because the hierarchical description for circuits is used for almost every designer, multilevel node tearing is achieved naturally by the specified hierarchy. The program iPride written by Mi-Chang Chang is a hierarchical direct-method parallel circuit simulator [17]. Our algorithm has been implemented in iPride and called iPride_V for solving linear systems more efficiently by introducing

vectorization.

## 6.2. Circuit Storage Scheme

In terms of circuit structure, either the multilevel node tearing technique or user-specified hierarchy produces a tree of subcircuits where each diagonal submatrix represents the internal nodes of a subcircuit and the border submatrix represents the interconnection of the terminal nodes of the subcircuits. The tree of subcircuits matches the tree of blocks in the NBBD form. Thus, in our approach, the subcircuits are stored in a similar way as in the nested-bl( ck data structure. The difference is that a subcircuit record has to keep information of element connections, nodal voltage of previous time points, and so on. A typical subcircuit record is listed in Figure 6.1.

Since the circuits are stored in nested-block structure, the algorithm in Chapter 4 can be carried out to solve circuit equations with the same high efficiency. There is another advantage of the nested-block structure over other data structures: the values of the network variables, "stamps" [19], are loaded into circuit matrices efficiently. For example, a resistor with conductance $g$ between node $i$ and node $j$ can be loaded into the matrix by the following codes:

```
address[i][i] = address[i][i] + g;
address[i][j] = address[i][j] - g;
address[j][i] = address[j][i] - g;
address[j][j] = address[j][j] + g;
```

The loading and updating of operands can be accessed in the same way as in a two-dimensional array.

```
typedef struct subckt {

    /* The nested-block data */
    int inode;      /* number of internal nodes */
    int tnode;      /* number of total nodes */
    struct subcktrec *son;      /* pointer to son block */
    struct subcktrec *par;      /* pointer to parent block */
    struct subcktrec *sib;      /* pointer to sibling block */
    double *address[];      /* address vector */

    /* The circuit data */
    struct noderec *nodelist;       /* node list */
    struct termrec *termlist;       /* terminal node list */
    struct resrec *reslist;         /* resistor list */
    struct caprec *caplist;         /* capacitor list */
    struct mosrec *moslist;                 /* mos transistor list */
    struct vscrec *vsclist;         /* voltage source list */
            .
            .
            .
} subcktrec;
```

Figure 6.1 Example of subcircuit record

### 6.3. Results

The speedups of iPride_V are compared with the original iPride run on one processor, because the original iPride program basically use the same algorithms and data structures as SPICE2 for one processor. Tables 6.1 and 6.2 list the speedups of two examples, and Tables 6.3 and 6.4 compare the memory size between two different data structures (linked list and nested-block).

Table 6.1. Speedups

| Circuit size : 642 nodes<br>no. of nonzeros : 1919 | | |
|---|---|---|
| level | 1 processor | 8 processors |
| 5 | 0.44 | 2.75 |
| 6 | 0.51 | 3.67 |
| 7 | 0.50 | 4.13 |
| 8 | 0.49 | 3.01 |
| 9 | 0.45 | 2.75 |

Table 6.2. Speedups

| Circuit size : 237 nodes<br>no. of nonzeros : 1175 | | |
|---|---|---|
| level | 1 processor | 8 processors |
| 3 | 0.77 | 3.03 |
| 4 | 1.03 | 4.71 |
| 5 | 1.15 | 5.30 |
| 6 | 1.06 | 6.06 |
| 7 | 1.03 | 6.06 |

Table 6.3. Memory size

| Circuit size : 642 nodes | | | |
|---|---|---|---|
| level | no. of nonzeros (include fill-ins) | linked list (bytes) | nested-block (bytes) |
| 1 | 1926 | 30.8K | - |
| 5 | 2496 | 39.9K | 51.9K |
| 6 | 2718 | 43.5K | 30.0K |
| 7 | 2852 | 45.6K | 21.4K |
| 8 | 2934 | 46.9K | 19.7K |
| 9 | 2938 | 47.0K | 19.8K |

Table 6.4. Memory size

| Circuit size : 237 nodes | | | |
|---|---|---|---|
| level | no. of nonzeros (include fill-ins) | linked list (bytes) | nested-block (bytes) |
| 1 | 1175 | 18.8K | - |
| 3 | 1509 | 24.1K | 40.1K |
| 4 | 1781 | 28.5K | 22.5K |
| 5 | 1901 | 30.4K | 14.5K |
| 6 | 1837 | 29.4K | 13.6K |
| 7 | 1791 | 28.7K | 14.1K |

# CHAPTER 7

# CONCLUSIONS AND FUTURE WORK

This thesis presents an approach to efficiently solve a large sparse linear system by exploiting parallel vector processing. In order to gain maximum speedups from the parallel vector computer structure, a new data storage scheme, namely, the nested-block, was proposed.

The nested-block is a structure designed especially for the multilevel node tearing technique. It stores the matrices in nested Bordered-Block Diagonal form block by block in one long vector. The storage scheme preserves the sparsity of the matrices, facilitates both parallelism and vectorization, and simplifies vector operations.

Our algorithm has been implemented in the ALLIANT FX/8, which is a supercomputer with eight vector processors. The speedups obtained in solving a large linear systems using eight processors range from 6 to 20 compared to the run time of a sequential program using Markowitz ordering and linked-list structure.

The approach is also applied to circuit simulation for VLSI design. The NBBD structure essentially matches the hierarchical circuit description used at the design phase, or it can be obtained by any partitioning techniques. Because the circuit is stored subcircuit by subcircuit, this data structure is also very suitable for subcircuit latency exploitation or mix-mode circuit simulation.

Our future works include:

(1) Implement the algorithm on the Cedar multiprocessor computer where the architecture is characterized by a hierarchical organization of both its computational capabilities and memory system. It consists of multiple clusters; each cluster is an Alliant FX/8 comprising

eight vector processors. Thus parallelism can be exploited at one more level.

(2) Construct a circuit simulator that has a hierarchical storage structure which does not make full copies for the same subcircuits only creates storage for changeable data of each subcircuit instance. The idea is similar to the hierarchical circuit description language, which uses subcircuit macros or device models to save duplicate information.

(3) Implement multilevel subcircuit latency in the circuit simulator. In transient analysis, each subcircuit can take different time steps. Those subcircuits that have long time steps and are assumed to be latent can be evaluated by a linear circuit model [20].

We will continue to develop techniques to improve the performance of the sparse linear system solver, circuit simulator and other CAD tools.

# APPENDIX

## PROGRAM SOLVE_P LISTING

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/times.h>
#include <cncall.h>
#define MAX 250
#define nil 0
#define true 1
#define false 0
#define VEC 4

typedef struct blkr{
        char name[80],lock;
        int oldest,size,tsize,nosub,col,wts,acwt,tf,fnshf;
        struct blkr *subblk,*sib,*par;
        float **mtrx;
} blkrec;

/*
 name: name of the block record
 size: the dimension of the block
 tsize: the dimension include parent's size
 nosub: the no. of subblocks
 col: the first col no. in the whole matrix
 subblk: pointer the first subblock
 sib: pointer to next block at the same level
 par: pointer to the parent block
 mtrx: pointer to the pointer array which point to row arrays that contains
     matrix elements
 wts:
 acwt:
 tf:
 fnshf:flag that indicate the block has been processed
*/
tvpedef struct qr {
        blkrec *ptr;
        struct qr *next;
} qrec;

blkrec *toplev;
char prtlock,string[80],name[20];
```

```
float full[MAX][MAX];
float *rhs;
int nonode,ptime[8];
qrec *que,*qind,*pq[8];

blkrec *readtr()
{
        int i,j,k,tag,tag1;
        float x;
        blkrec *p1,*p2,*p3;

        p1=(blkrec *)malloc(sizeof(blkrec));
        scanf("%d %d %s",&p1->size,&p1->nosub,p1->name);
/*      printf("%d %d %s0,p1->size,p1->nosub,p1->name); */
        p1->subblk=nil;
        p1->sib=nil;
        p1->par=nil;
        p1->oldest=0;
        p3=nil;
        for (k=0;k<p1->nosub;++k){
                p2=readtr();
                p2->par=p1;
                if (p3==nil) {
                   p1->subblk=p2;
                   p2->oldest=1;
                }
                else p3->sib=p2;
                p3=p2;
        }

        tag=true;
        tag1=true;
        while (tag){
                scanf("%d %d %f",&i,&j,&x);
                if (tag1){
                   p1->col=i-1;
                   tag1=false;
                }
                if (i==0) tag=false;
                else if (j==0) *(rhs+i-1)=x;
                else full[i-1][j-1]=x;
             if (tag) nonode=i;
        }
        return(p1);
}
```

```
formax(p1)
blkrec *p1;
{
        int i,j,size,offset;
        blkrec *p2,*p3;
        float a,*b,**row,**p2m;


        p1->tsize=p1->size;
        p3=p1->par;
        if (p3 != nil) p1->tsize += p3->tsize;

        p2=p1->subblk;
        while (p2!=nil){
           formax(p2);
           p2=p2->sib;
        }

        /* printf ("formax %s0,p1->name); */

        size=p1->tsize;
        row=p1->mtrx=(float **)malloc(size*sizeof(b));

        p2=p1->subblk;
        if (p2!=nil){
           offset=p2->size;
           p2m=p2->mtrx;
           for(i=0;i<size;++i)
                *(row+i) = *(p2m+offset+i) + offset;
        }


        else{
           for (i=0;i<size;++i)
              *(row+i) = (float *)malloc(size*sizeof(a));
        }

        for (i=0;i<p1->size;++i)
           for (j=0; j<p1->size; ++j)
              *( *(row+i) + j) = full[p1->col+i][p1->col+j];

        p3=p1->par;
        offset=p1->size;
        while (p3 !=nil){
           for (i=0;i<p1->size;++i)
```

```
            for (j=0;j<p3->size;++j) {
               *( *(row+i) +j+offset) = full[p1->col+i][p3->col+j];
                  *( *(row+offset+j) +i) = full[p3->col+j][p1->col+i];
               }

         offset += p3->size;
         p3 = p3->par;
       }
}


formax1(p1)
blkrec *p1;
{
       int i,j,offset;
       blkrec *p2,*p3;
       float a,*b,**row;



       p2=p1->subblk;
       while (p2!=nil){
          formax1(p2);
          p2=p2->sib;
       }

       /* printf ("formax %s0,p1->name); */

       row = p1->mtrx;

       for (i=0;i<p1->size;++i)
          for (j=0; j<p1->size; ++j)
             *( *(row+i) + j) = full[p1->col+i][p1->col+j];

       p3=p1->par;
       offset=p1->size;
       while (p3 !=nil){
          for (i=0;i<p1->size;++i)
             for (j=0;j<p3->size;++j) {
                *( *(row+i) +j+offset) = full[p1->col+i][p3->col+j];
                   *( *(row+offset+j) +i) = full[p3->col+j][p1->col+i];
                }

          offset += p3->size;
          p3 = p3->par;
```

```
        }

        if (!(p1->oldest)){
           for (i=p1->size;i<p1->tsize;++i)
                 for (j=p1->size;j<p1->tsize;++j)
                    *( *(row+i) + j) =  0;
        }
}



prtmx(p1)
blkrec *p1;
{
        int i,j;
        blkrec *p2;

        p2=p1->subblk;
        while (p2 !=nil){
           prtmx(p2);
           p2=p2->sib;
        }
        printf("%s0,p1->name);

        for (i=0;i<p1->tsize;++i){
           for(j=0;j<p1->tsize;++j)
              printf("%f ", *( *(p1->mtrx +i) +j));
                 printf("0);

        }
}

lufac(id)
int id;
{
        int i,j,k,length,size,tsize;
        float fac,*src,*tar,*srcrow,*tarrow,**p1m,**p3m;
        blkrec *p1,*p2,*p3,*p4;
        qrec *q1;

        q1=pq[id];
        while (q1!=nil){
           p1=q1->ptr;
           size=p1->size;
           tsize=p1->tsize;
```

```
if (p1==toplev) size--;

p2=p1->subblk;
while(p2!=nil){
      while(p2->fnshf);
      p2=p2->sib;
}

lock(&p1->lock);
p3=p1;
while ((p3->par!=nil) && (p3->oldest)){
      lock(&p3->par->lock);
      p3=p3->par;
}
/* lock(&prtlock);
printf("proc %d %s0,id,p1->name);
unlock(&prtlock);  */
p1m=p1->mtrx;
for (i=0;i<size;++i){
      srcrow = *(p1m+i);
      tar = srcrow+i+1;
      fac = *(srcrow+i);
      length=tsize-i-1;
      if (length > VEC) vec_sdiv_vs(tar,tar,fac,length);
      else for(k=0;k<length;++k) *(tar+k) /= fac;
      src = tar;
      for (j=i+1;j<tsize;++j){
         tarrow = *(p1m+j);
         fac = *(tarrow+i);
         if (fac!=0){
         tar = tarrow+i+1;
         if (length > VEC) vec_sma_vsv(tar,src,-fac,tar,length);
         else for(k=0;k<length;++k) *(tar+k) += *(src+k) * -fac;
          }
      }
}

unlock(&p1->lock);
p3=p1;
while ((p3->par!=nil) && (p3->oldest)){
      unlock(&p3->par->lock);
      p3=p3->par;
}

p3=p1->par;
```

```
            if ((p3!=nil) && !(p1->oldest)){
                while((p3!=nil) && (p3->oldest)){
                    lock(&p3->lock);
                    p3=p3->par;
                }
                p3=p1->par;
              length = p3->tsize;
              p3m=p3->mtrx;
              for (i=0;i<length;++i){
                    src = *(p1m+size+i) + size ;
                tar = *(p3m+i);
                    if (length > VEC) vec_sadd_vv(tar,src,tar,length);
                    else for(k=0;k<length;++k) *(tar+k) += *(src+k);
              }
                while((p3!=nil) && (p3->oldest)){
                    unlock(&p3->lock);
                    p3=p3->par;
                }
            }
            p1->fnshf=0;
            q1=q1->next;
        }
}

vec_sdot(iproduct,v1,v2,length)
float *iproduct,*v1,*v2;
int length;
{
        int i;

        *iproduct = 0;
        for(i=0;i<length;++i)
          *iproduct += *(v1+i) * *(v2+i);
}

forsub(p1)
blkrec *p1;
{

        blkrec *p2,*p3;
        float *vec1,*vec2,temp;
        int offset,i,length;

        p2=p1->subblk;
        while (p2!=nil){
```

```
            forsub(p2);
            p2=p2->sib;
          }

          *(rhs+p1->col) /= **p1->mtrx;
          for (i=1;i<p1->size;++i){
            vec1 = rhs+p1->col;
            vec2 = *(p1->mtrx+i);
            length = i;
            vec_sdot (&temp,vec1,vec2,length);
            *(rhs+p1->col+i) -= temp;
            *(rhs+p1->col+i) /= *( *(p1->mtrx+i) +i);
          }

          p3=p1->par;
          length=p1->size;
          offset=p1->size;
          while (p3!=nil){
            for(i=0;i<p3->size;++i){
              vec1 = rhs+p1->col;
              vec2 = *(p1->mtrx+offset+i);
                vec_sdot (&temp,vec1,vec2,length);
                *(rhs+p3->col+i) -= temp;
            }
            offset += p3->size;
            p3=p3->par;
          }
}

backsub(p1)
blkrec *p1;
{
        blkrec *p2,*p3;
        int i,length,offset;
        float *vec1,*vec2,temp;

        p3=p1->par;
        offset=p1->size;
        while (p3 !=nil) {
          length=p3->size;
          for (i=0;i<p1->size;i++){
            vec1 = rhs+p3->col;
              vec2 = *(p1->mtrx+i) +offset;
              vec_sdot (&temp,vec1,vec2,length);
              *(rhs+p1->col+i) -= temp;
```

```
            }
            offset += p3->size;
            p3=p3->par;
        }

        for (i=p1->size-1;i>0;--i){
                vec1 = rhs+p1->col+i;
                vec2 = *(p1->mtrx+i-1) +i;
            length = p1->size - i;
            vec_sdot (&temp,vec1,vec2,length);
                *(rhs+p1->col+i-1) -= temp;
        }

        p2=p1->subblk;
        while(p2!=nil){
            backsub(p2);
            p2=p2->sib;
        }
}

prtrhs()
{
        int i;

        for (i=0;i<nonode;++i) printf(" %d %g0,i,*(rhs+i));
}


count(p1)
blkrec *p1;
{
        blkrec *p2;
        int ak;

        ak=0;
        p2=p1->subblk;
        while (p2!=nil){
            count(p2);
            if (p2->acwt>ak) ak=p2->acwt;
            p2=p2->sib;
        }
        p1->wts=p1->size;
        p1->acwt=p1->wts+ak;
        printf("%s %d0,p1->name,p1->size);
}
```

```
pushq(b1)
blkrec *b1;
{

   qrec *q1,*q2;

   q1=(qrec *)malloc(sizeof(qrec));
   q1->ptr=b1;
   q1->next=que;
   que=q1;
}

blkrec *popq()
{
   qrec *q1,*q2,*q3;
   int i,j,k;

   q1=que;
   if (q1==nil) return(nil);
   i=1e6;                  /*find a task with the minimum starting time*/
   k=0;
   q2=nil; q3=nil;
   while (q1!=nil){
        if ((q1->ptr->tf<i) || ((q1->ptr->tf==i) && (q1->ptr->acwt>k))) {

           q3=q2;
           i=q1->ptr->tf;
           k=q1->ptr->acwt;
        }
        q2=q1;
        q1=q1->next;
   }
   if (q3!=nil) {
        q1=q3->next;
        q3->next=q1->next;
   }
   else {
        q1=que;
        que=q1->next;
   }
   return(q1->ptr);
}

sched(proc,tag)
int proc,tag;
```

```
{
  int i,j,k;
  blkrec *b1,*b2;
  qrec *q1;

  if (tag) {
       for (i=0; i<proc; i++) pq[i]=nil;
  }
  b1=popq();
  while (b1!=nil) {
    k=1e6;    /* find idle proc */
    j=0;
    for (i=0; i<proc; i++)
         if (ptime[i]<k) {
           k=ptime[i];
           j=i;
         }
      if (ptime[j]<b1->tf) ptime[j]=b1->tf;
    ptime[j]=ptime[j]+b1->wts;
      if (tag) {
        q1=(qrec *)malloc(sizeof(qrec));
        q1->ptr=b1;
        q1->next=pq[j];
        pq[j]=q1;
      }
    b2=b1->subblk;
    while (b2!=nil) {
         b2->tf=ptime[j];
         pushq(b2);
         b2=b2->sib;
    }
      b1=popq();
  }
  k=0;
  for (i=0; i<proc; i++)
  if (ptime[i]>k) k=ptime[i];
  if (tag) {
       for (i=0; i<proc; i++) {
         printf("proc %d0,i);
         q1=pq[i];
         while (q1!=nil) {
              printf("  %s w=%d ts=%d0,
              q1->ptr->name,q1->ptr->wts,q1->ptr->tf);
              q1=q1->next;
         }
```

```
        }
    }
    return(k);
}


init(p1)
blkrec *p1;
{
        blkrec *p2;

        p2=p1->subblk;
        while (p2 != nil){
           p2->fnshf=1;
           init(p2);
           p2=p2->sib;
        }
}

cntlev(p1)
blkrec *p1;
{
        int i,j;
        blkrec *p2;

        p2=p1->subblk;
        if (p2==nil) return (1);
        else {
           i=0;
           while (p2 != nil){
                j=cntlev(p2);
                if (j>i) i=j;
                p2=p2->sib;
           }
           return (i+1);
        }
}

main(argc,argv)
int argc;
char *argv[];
{
        int i,j,k;
        struct tms *timelink;
        float a,t1,t2,t3,t4;
```

```
    k=1;
    if (argc>=2) {
       i=0;
       k=0;
       while (argv[1][i]!=' '){
          k=k*10+argv[1][i++]-'0';
       }
}

    gets(string);
    printf("%s0,string);
    gets(name);
    printf("%s0,name);
    for (i=0;i<MAX;++i) for (j=0;j<MAX;++j) full[i][j]=0;
    rhs=(float *)malloc(MAX*sizeof(a));
    toplev=readtr();
    toplev->oldest=1;
    printf("0);
    formax(toplev);

    que=nil;
    count(toplev);
    printf("scheduling0);

    i=1;
    while (i<=8){
       for (j=0; j<8; j++) ptime[j]=0;
       pushq(toplev);
       j=sched(i,false);
       printf("%11d",j);
       if (i==1) i=2;
       else i+=2;
    }

    printf("0);
    for (j=0; j<8; j++) ptime[j]=0;
    pushq(toplev);
    sched(7,true);
    timelink = (struct tms *)malloc(sizeof(struct tms));
    times(timelink);
    t3= (float)timelink->tms_utime;
    t4= (float)timelink->tms_stime;

    for (i=0;i<k;++i){
       formax1(toplev);
```

```
        }

        times(timelink);
        t3= (float)timelink->tms_utime-t3;
        t4= (float)timelink->tms_stime-t4;
        times(timelink);
        t1= (float)timelink->tms_utime;
        t2= (float)timelink->tms_stime;

        for (i=0;i<k;++i){
           formax1(toplev);
           init(toplev);
           concurrent_call (CNCALL_NUMPROC|CNCALL_NO_QUIT,lufac);
        }

        times(timelink);
        t1= (float)timelink->tms_utime-t1-t3;
        t2= (float)timelink->tms_stime-t2-t4;

        forsub(toplev);
        backsub(toplev);

/*      times(timelink);
        t1= (float)timelink->tms_utime-t1;
        t2= (float)timelink->tms_stime-t2; */

        prtrhs();

        i=cntlev(toplev);
        printf("l=%d iter=%d cpu time %8gs user %8gs sys0,i,k,t1/60/k,t2/60/k);
}
```

# REFERENCES


[1] J. W. Huang and O. Wing, "Optimal parallel triangulation of a sparse matrix," *IEEE Trans. Circuit Syst.*, vol. CAS-26, pp. 726-732, Sept. 1979.

[2] O. Wing and J. W. Huang, "A computation model of parallel solution of linear equations," *IEEE Trans. Computers*, vol. C-29, July 1980.

[3] D. P. Arnold, M. I. Parr, and M. B. Dewe, "An efficient parallel algorithm for the solution of large sparse linear matrix equations," *IEEE Trans. Computers*, vol. C-32, Mar. 1983.

[4] F. Yamamoto and S. Takahashi, "Vectorized LU decomposition algorithms for large-scale circuit simulation," *IEEE Trans. CAD*, vol. CAD-4, no. 3, pp. 232-239, July 1985.

[5] M. Chang and I. N. Hajj, "iPRIDE: A parallel integrated circuit simulator using direct method," *Proc. ICCAD'88*, pp. 304-307, 1988.

[6] P. Sadayappan and V. Visvanathan, "Circuit simulation on shared-memory multiprocessors," *IEEE Trans. Computers*, vol. 37, pp. 1634-1642 , Dec. 1988.

[7] C. C. Chen and Y. H. Hu, "Parallel LU factorization for circuit simulation on an MIMD computer," *Proc. ICCD'88*, pp. 129-132, 1988.

[8] R. E. Lord, J. S. Kowalik, and S. P. Kumar, "Solving linear algebraic equations on an MIMD computer," *J. ACM*, vol. 30, no. 1, pp. 103-117, Jan. 1983.

[9] J. A. G. Jess and J. G. M. Kees, "A data structure for parallel LU decomposition," *IEEE Trans. Computers*, vol. C-31, no. 3, pp. 231-239, March 1982.

[10] D. Smart and J. White, "Reducing the parallel solution time of sparse circuit matrices using reordered gaussian elimination and relaxation," *Proc. ISCAS'88*, pp. 627-630.

[11] A. George and J. W. H. Liu, "An automatic nested dissection algorithm for irregular finite element problems," *SIAM Numer. Anal.*, vol. 15, pp. 1053-1069, Oct. 1978.

[12] B. W. Kernighan and S. Lin, "An efficient heuristic procedure for partitioning graphs," *Bell Syst. J.*, vol. 49, pp. 291-307, 1970.

[13] P. Sadayappan and V. Visvanathan, "Modeling and optimal scheduling of parallel sparse gaussian elimination," Technical Report, AT&T Bell Laboratories, 1988.

[14] K. S. Kundert, "Sparse matrix techniques," in *Circuit Analysis, Simulation and Design, Part 1*, A. E. Ruehli, Ed., Amsterdam, North-Holland: Elsevier Science Publishers B.V., 1986, pp. 281-324.

[15] I. S. Duff, A. M. Erisman, and J. K. Reid, in *Direct Methods for Sparse Matrices*, New York: Oxford Science Publications, 1986, pp. 24-25.

[16] P. Sadayappan and V. Visvanathan, "Efficient sparse matrix factorization for circuit simulation on vector supercomputers," *IEEE Trans. CAD*, vol. 8, no. 12, pp. 1276-1285, Dec. 1989.

[17] M. Chang, "Efficient direct-method parallel circuit simulation using multilevel node tearing," *UILU-ENG-89-2201 DAC-13*, Coordinated Science Laboratory, Univ. of Illinois at Urbana-Champaign, 1989.

[18] *CONCENTRIX C Handbook,* Alliant Computer Systems Corp., Feb. 1987.

[19] I. N. Hajj, "Analysis of linear circuit," in *Fundamental Handbook of Electrical and Computer Engineering*, vol. 3, 1983.

[20] P. F. Cox, R. G. Burch, P. Yang, and D. E. Hocevar, "New implicit integration method for efficient latency exploitation in circuit simulation," *IEEE Trans. CAD*, vol. 8, no. 10, pp. 1051-1064, Oct. 1989.