Copyright © 1989, by the author(s).
All rights reserved.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission.

PARALLEL DIRECT-METHOD SOLUTION OF A SPARSE LINEAR SYSTEM OF EQUATIONS

by

Abhijit Ghosh

Memorandum No. UCB/ERL M89/89

31 July 1989

COVER PAGE

PARALLEL DIRECT-METHOD SOLUTION OF A SPARSE LINEAR SYSTEM OF EQUATIONS

by

Abhijit Ghosh

Memorandum No. UCB/ERL M89/89

31 July 1989

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

PARALLEL DIRECT-METHOD SOLUTION OF A SPARSE LINEAR SYSTEM OF EQUATIONS

by

Abhijit Ghosh

Memorandum No. UCB/ERL M89/89

31 July 1989

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

Abstract

Circuit simulation programs are very important computer-aided design tools for analyzing the electrical performance of circuits. As integrated circuits (IC's) get larger and more complicated, it is not always economically feasible to fabricate them and test them before production and so the IC designer must rely on a circuit simulator for predicting the performance of circuits prior to fabrication. Unfortunately, circuit simulation speeds have not kept up with the growing demands of the IC designer. Conventional circuit simulators were designed for the cost-effective analysis of a few hundred transistors or less and programs like SPICE3 can take a long time to simulate a circuit with a thousand transistors on conventional uniprocessors.

With the continuing decline in the cost of hardware, it is becoming economically feasible to build computers with many processors. In coming years, one can expect to see computers with a large number of processors. This expectation has motivated the application of parallel algorithms to various steps in the transient analysis of circuits. The subject of this report is the parallelization of the solution of a system of linear equations, an important step in the transient analysis of electrical circuits. New scheduling and reordering algorithms have been developed for this purpose and results indicate that appreciable speedup can be obtained.

Acknowledgements

I greatly acknowledge my research advisor, Professor Richard Newton, for his inspiration, encouragement and advice during the course of this project. I would also like to thank Geroge Jacob for answering numerous questions about PDSPLICE. I would like to thank Brian O'Krafka for suffering through an initial draft of this report, and also for many interesting discussions on parallel architectures and algorithms.

I would like to thank Professor D. Pederson for his helpful comments and suggestions concerning the project and this report. I would also like to thank Tom Quarles for his help with SPICE, Don Webber for helping in finding the test examples, and Andrea Casotto for his help with the graphics package.

This project was supported in part by Defense Advanced Research Project Agency (under contract N00039-87-C00182), Digital Equipment Corporation, AT&T Bell Laboratories and Semiconductor Research Corporation. Their support is gratefully acknowledged.

Contents

Table of Contents i

List of Figures iii

List of Tables iv

1 INTRODUCTION 1

- 1.1 Circuit Simulation 2
- 1.2 Parallelizing the LOAD Phase 6
- 1.3 Organization of this Report 7

2 APPROACHES TO PARALLEL LINEAR EQUATION SOLUTION 8

- 2.1 Task Granularity 9
- 2.2 Scheduling 12
  - 2.2.1 Hu's Levelized Scheduling Algorithm 13
  - 2.2.2 Other work 14
- 2.3 Reordering 15
- 2.4 Hardware-based Acceleration Techniques 16
  - 2.4.1 Vector Processors 16
  - 2.4.2 Special-purpose Hardware for Parallel Circuit Simulation 17
- 2.5 Use of MIMD Multiprocessors 18
- 2.6 Conclusion 19

3 TASK GRANULARITY 20

- 3.1 Task Graph 20
- 3.2 Fine Grain Parallelism 21
  - 3.2.1 Maximum Speedup Obtainable from Fine Grain Parallelism 27
- 3.3 Medium Grain Parallelism 27
- 3.4 Large Grain Parallelism 29
- 3.5 Implementation of Medium Grain Approach 31
- 3.6 Results 32
- 3.7 Conclusions 33

4 SCHEDULING 34
4.1 Terminology 35
4.2 Dynamic Scheduling 37
4.3 Static Scheduling 38
4.4 Scheduling Algorithm 39
4.4.1 Algorithm 39
4.4.2 Implementation 42
4.5 Interprocessor Communication 43
4.5.1 Reducing synchronization overhead 44
4.5.2 Implementation 46
4.6 Results 47
4.7 Conclusion 49

5 REORDERING 50
5.1 Graph Model of Matrix 53
5.1.1 Fillin Generation 54
5.1.2 Gaussian Elimination 55
5.1.3 LU Decomposition 56
5.2 Reordering Algorithm 56
5.3 Results 63
5.4 Conclusion 65

6 CONCLUSION 66

Bibliography 69

List of Figures

1.1 Structure of a Transient Simulator 4
1.2 SOLVE and LOAD Times as a Function of Circuit Size 5

3.1 Example Matrix 21
3.2 Task graph for fine grain parallelism 23
3.3 Task graph for fine grain parallelism 25
3.4 Task graph for medium grain parallelism 29
3.5 Task graph for large grain parallelism 30
3.6 Computation tree for element a_{44} 31

4.1 Scheduling algorithm 40
4.2 Load on processors at some point during scheduling 41
4.3 Synchronization algorithm 45

5.1 Example matrix 50
5.2 Task graph for matrix shown in Figure 5.1 51
5.3 Reordered matrix 52
5.4 Task graph for matrix in Figure 5.3 52
5.5 Example matrix #2 53
5.6 Associated graph for matrix shown in Figure 5.5 54
5.7 Graph for matrix after row and column interchange 54
5.8 Graph for matrix after fillin generation for elimination of first row 55
5.9 Example matrix #3 58
5.10 Partial Graph of matrix shown in Figure 5.9 59
5.11 Reordering Algorithm 61
5.12 Choosing Algorithm 62
5.13 Algorithm for correcting problem 63

List of Tables

3.1 Time to solve for various matrices 32

4.1 Characteristics of the circuit matrices 47
4.2 Performance predicted by scheduling algorithm 47
4.3 Speedup achievable without synchronization 48
4.4 Speedup achievable with synchronization 48
4.5 Speedup comparison 49

5.1 Effect of reordering on T_c 64
5.2 Effect of reordering on actual execution time 64

Chapter 1

INTRODUCTION

Circuit simulation programs like SPICE [39] and ASTAP [56], have proven to be important tools for the analysis of the electrical performance of circuits. These programs can perform a variety of analyses, including ac, dc, and transient analysis of circuits containing a wide range of non-linear active devices. The phrase that best characterizes the circuits and systems of today is *very large scale.* Designers today plan to put a whole system involving a million or more transistors on a single chip. This high degree of integration requires the use of various computer-aided design tools. With the advent of ASIC's (Application Specific Integrated Circuits), IC vendors are aiming for the quickest possible turn-around time and low non-recurring engineering costs (NRE). Thus fast and accurate computer-aided design tools have assumed a new significance.

As circuits get larger and more complicated, it is not always possible to fabricate them and test them before production. The IC designer must rely on a circuit simulator for evaluating the performance of circuits. Over the past decade, the level of integration has grown almost exponentially and the number of devices on a chip have touched the million mark. Although the designer typically simulates only small parts of a chip, sometimes it is necessary to simulate a whole system, which may consist of ten thousand or more transistors. Unfortunately, circuit simulation speeds have not kept up with the growing demands of the VLSI designer. Conventional circuit simulators were designed initially for the cost effective analysis of a circuit containing few hundred transistors or less. Programs like SPICE3 [41] can take a long time to simulate a circuit with a thousand transistors, and the effect of a minor change to a circuit may take hours to verify. Obviously, designers need faster circuit simulators for simulating large circuits. Before going into the details of

improving the performance of simulators, in the following section is a brief review of the basic algorithm of a direct method circuit simulator.

1.1 Circuit Simulation

One of the most common analyses performed by circuit simulators, especially in the design of digital circuits, is time-domain transient analysis. With transient analyses, precise electrical waveform information can be obtained if the device model and parasitics are characterized properly. A simulator reads an input description detailing circuit connectivity and element parameters and assembles a set of non-linear ordinary differential equations (ODE's) to represent the circuit behavior. There are two common ways of assembling these equations — Sparse Tableau Analysis (STA) [56] or Modified Nodal Analysis (MNA) [39]. Sparse Tableau Analysis involves the writing of the Kirchoff's Current and Voltage Law (KCL and KVL) equations and appending the branch equations to them. The resulting coefficient matrix is very sparse, but it requires sophisticated programming techniques to exploit the sparsity of the matrix [23]. Modified Nodal Analysis is a generalization of Nodal Analysis method where the KCL equations are written first and then the currents of all possible branches are eliminated using branch equations. Finally, all node voltages are substituted for branch voltages and a set of equations involving node voltages is obtained, together with some branch equations. The MNA coefficient matrix is also very sparse, though not as sparse as the STA matrix. All coefficient matrices can be assembled by inspection of the circuit.

For circuits containing non-linear elements, a set of non-linear simultaneous equations has to be solved. In classical simulators, the equations are solved numerically using the Newton-Raphson (NR) iteration method, and the coefficient matrix corresponds to a Jacobian matrix. The original circuit can be interpreted as a Sparse Tableau of a linear circuit whose elements are specified by the linearized branch equations. This linear circuit is called the companion network [47]. A circuit simulator, when analyzing a circuit, can first linearize the branch equations of the non-linear elements and then assemble and solve the circuit equations for the companion network.

For dynamic circuits (circuits containing capacitors or inductors and other non-linear elements), a set of non-linear ordinary differential equations has to be solved. Circuit simulators such as SPICE and ASTAP, when analyzing the transient behavior of dynamic

circuits, discretize the branch equations by replacing the derivatives of the circuit variables with a stiff-stable integration formula. After the branch equations have been discretized, a dc solution problem has to be solved using the NR-iteration technique. The process of discretization, linearizing, and assembling the final coefficient matrix for solution is often called the *LOAD* (refer to Figure 1.1) phase of simulation and it is linear in complexity [47] (the time taken for loading the coefficients grows linearly with the number of elements in the circuit).

Assuming that the circuit equations have been assembled, they have to be solved in the most efficient way. Algorithms for solving a system of linear equations are discussed in [3]. There are two classes of methods for solving the linear equations — direct methods and iterative methods. Direct method algorithms can find the exact solution of the system of linear equations in a finite and known number of steps, if computations are carried out with infinite precision. Most circuit simulators use direct methods because of their robustness and predictability. This report is restricted to the use of direct methods for the same reason. Among the direct methods available, Cramer's rule is very well known but it is too costly to consider for a computer solution. If n is the number of equations in the linear system, Cramer's Rule would require O(n!) operations to solve a dense system. Though circuit simulation problems give rise to very sparse system of linear equations, the operations required for Cramer's Rule would result in substantial fillins during the elimination process [47], thus loosing the advantage offered by a sparse system. Other methods like Gaussian Elimination (GE) [3] and LU decomposition [3] are used in practice. Strassen's algorithm [52] is the optimal way of solving a system of linear equations, but the algorithm is very complicated. Ease of programming makes GE the most efficient algorithm available for solving a system of linear equations. This phase of the simulation is called the *SOLVE* phase. For GE, or LU decomposition, the complexity is $O(n^3)$ for a full matrix of size $n \times n$. For sparse matrices it has been observed that the complexity of the solution algorithm is $O(n^\beta)$ where β is between 1.2 and 1.8, depending on the sparsity of the matrix [47]. Thus exploiting the sparsity of the matrices is of great importance for minimizing both storage and execution time.

A flow chart depicting the structure of a transient simulator is shown in Figure 1.1. The two important phases, *LOAD* and *SOLVE*, are shown. As the number of circuit elements in the analysis grows, the time required to formulate the nodal equations grows

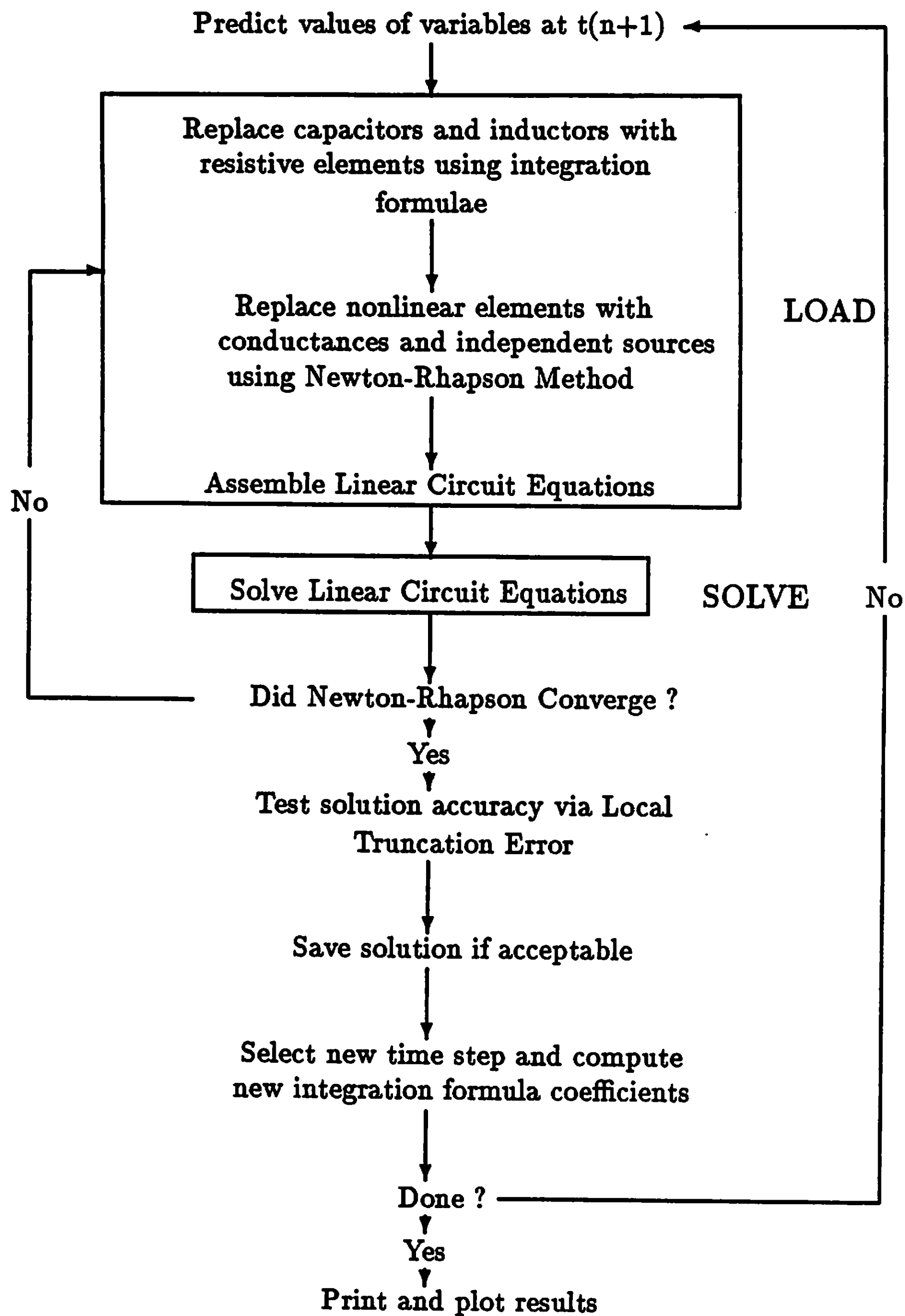

Figure 1.1: Structure of a Transient Simulator

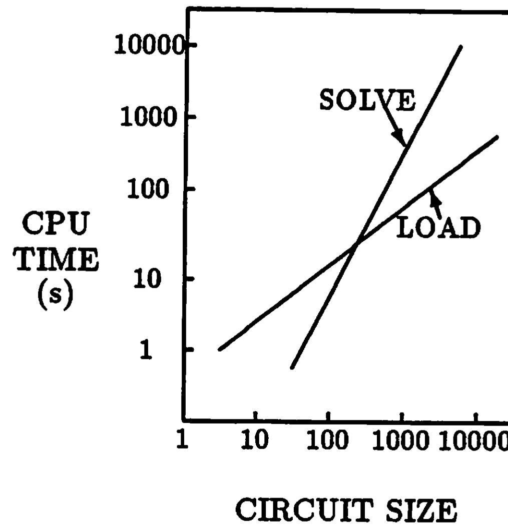

Figure 1.2: SOLVE and LOAD Times as a Function of Circuit Size

linearly with circuit size but the time required to solve the circuit equations increases at a faster rate and rapidly becomes the dominant cost of the analysis. For the circuit simulator SPICE2 [39], the solution time is about 10% of the total time for a circuit of less than thirty transistors, but reaches half the total CPU time for a circuit containing three thousand transistors. These two phases are the two major bottlenecks which limit the speed of a circuit simulator, with the SOLVE phase being the more critical of the two. In Figure 1.2 the time to form the equations and solve them is plotted against the number of circuit nodes for a family of MOS circuits. This result implies that for large circuits, where SOLVE time dominates the total run time, an improvement in SOLVE time would have a large impact on overall simulator throughput. The figure shows that the crossover point is around 500 nodes.

With the continuing decline in the cost of hardware it is becoming economically feasible to build computers with many processors. In coming years, one can expect to see MIMD (Multiple Instruction Stream Multiple Data Stream) computers, with hundreds of processors. This expectation has motivated the application of parallel algorithms to various steps in the transient analysis process. The LOAD phase can be easily parallelized and is the topic of the next section. Parallelizing the SOLVE phase is a difficult problem and is the subject of this report. Methods to speed up the execution of the SOLVE phase using a MIMD multiprocessor and a new set of algorithms are presented.

1.2 Parallelizing the LOAD Phase

Since circuit model evaluation is expensive, a number of techniques have been employed to reduce the time taken for the linearization phase. It has been shown that table lookup reduces the model evaluation time by eliminating the need for repeated calculation of device coefficients [21,29,50]. Some circuit simulation programs [57] do not update the Jacobian at each NR-iteration, on the argument that the Jacobian matrix is inaccurate when distant from the solution and when near the solution its accuracy makes frequent updates unnecessary. It has also been shown [40] that an *element-bypass* technique, where models are evaluated only if their input voltages or currents have changed by more than a certain amount, can result in significant savings. Special-purpose hardware [21] have been built to enable fast parallel model evaluation for MOSFET's. In [55] a vectorized scheme for model evaluation is discussed. Vectorization is achieved whenever all devices reference the same model parameters or when all homologous devices in different instances of the same sub-circuit definition reference the same model parameters. Model evaluation is split into a gather-scatter and a computation phase. There is no clear advantage with vectorization, mainly because of the gather-scatter bottleneck. However, with gather-scatter hardware, device model evaluation time can be significantly improved [59].

For a multiprocessor, the LOAD operation can be parallelized easily. The LOAD operation consists of different parts, each with a certain degree of parallelism. Only a small portion of the work has to be done sequentially. The bulk of the computation consists of evaluating the linearized companion elements of various devices and storing ("stamping") them into the final coefficient matrix. This is usually done by device type. The resistors are evaluated and stamped on to the Jacobian, then the capacitors, and then the MOSFET's and other devices. If there are enough processors, all devices can be evaluated in parallel. There are two approaches to storing the final value in the Jacobian — one that uses locks and another that does not. In the first approach stamping the value on to the Jacobian must be atomically locked so that no two processors can add to the same location in the matrix at the same time. This is a *lock-synchronized parallel loop.*

One example of a lockless approach would be the use of multidimensional matrices. Here auxiliary arrays are used to extend the matrix into the third dimension with the depth in this direction at any point being equal to the number of devices that stamp on to the corresponding matrix location. Each contribution to a particular location is then assigned a

unique location in the auxiliary array. With such a structure, the evaluation and stamping of devices can be done completely without locks. After all stamping is done, the three-dimensional matrix is collapsed into a two-dimensional matrix by summing the contributions that go into each location. This part can also be done in parallel as each location can be added independently. Though faster, this approach uses much more memory and there is overhead in keeping track of the matrix contributions at each location.

The two approaches outlined above are at the two extremes. A unified approach would use multiple locks. The basic idea is to use different locks for different regions of the memory so that a processor would have to idle only if another processor is writing into the same region that it needs to write into. The number of locks to be used and the set of circuit nodes that should be assigned to each of these locks depends on the number of processors available.

Significant research [44] has proved that parallelizing the LOAD phase is not a very difficult problem. Many researchers [44,62,6] have reported linear speedup in loading time with up to 16 processors. Efficient parallelization of the SOLVE phase is less straightforward.

1.3 Organization of this Report

This report is organized as follows. Chapter 2 is a survey of work already done in this area. In Chapter 3 we take a look at the amount of parallelism inherent in the SOLVE phase. Three levels of task granularity are described and the reasons for choosing medium grain parallelism are given. In Chapter 4 the problem of optimum scheduling of computation graphs on a finite number of processors is presented and a heuristic solution algorithm is given. Chapter 5 contains the description and a heuristic solution of the problem of reordering a matrix for minimizing parallel computation time. Conclusions are presented in Chapter 6.

Chapter 2

APPROACHES TO PARALLEL LINEAR EQUATION SOLUTION

Parallelizing linear equation solution has been a subject of research for a long time. Approaches to the problem can be found in [30,53,36,9,13,6,27]. On multiprocessor systems, it has been shown [53] that Gaussian elimination can be efficiently parallelized when solving dense, large-scale linear systems. Nested dissection techniques [36] allow efficient parallel solution of well-ordered sparse linear systems. However, circuit matrices, in addition to their extreme sparsity (three or four non-zero elements per row or column), have an irregular structure that makes most decomposition techniques, including nested dissection, difficult to use [49]. Since most circuits give rise to sparse, irregular matrices, from now on only approaches suitable for such systems are described.

In [30] a parallel algorithm based on LU-decomposition, called *Segmented Partial Pivoting* (SPP), was implemented on the BBN Butterfly Parallel Processor as a first step towards parallelizing the solution of irregular, sparse linear systems. In the SPP algorithm, each processor is assigned a block of rows from the system matrix ; for each column in the matrix, all the processors work within their assigned blocks to determine local pivot candidates and then to eliminate other non-zero elements in the column. Next the processors compete to set a lock. The local pivot of the processor that sets the lock becomes the global pivot for the given column, and the processors use the global pivot to eliminate their own

local pivots. This process is repeated until the system is triangular. Forward elimination and backward substitution is also done in parallel. These operations employ parallel multiplication and subtraction after an unknown has been solved for, as well as parallel division, which is made possible by the sparsity of the matrix. Experimental results using the SPP algorithm indicate high speedup using about ten processors, but no improvement beyond that. Further, the competetive global pivot selection scheme results in a large number of fillins, indicating that even with the parallelism exploited, parallel solution may not be faster that efficient uniprocessor solution. This study thus leads to two conclusions : first, there is a relatively low degree of parallelism in the parallel linear equation solution for circuit matrices; secondly, schemes that attempt to exploit as much parallelism as possible may result in excessive fillins, thereby nullifying their advantage.

The approach taken in [26] is to break up the task of solution of a sparse system of equations into a number of subtasks, some of which can be executed in parallel. Then the problem of parallel equation solution has three separate sub-problems. They are :

- Scheduling
- Reducing interprocessor communication
- Reordering to minimize parallel computation time

In this chapter approaches adopted by other researchers for solving these problems are briefly reviewed.

2.1 Task Granularity

The amount of parallelism achievable depends on the size of the individual tasks to be executed in parallel. The size of the tasks is what we call task granularity. There are three levels of granularity : fine grain, medium grain and coarse grain. Fine grain parallelism is the parallelism exploited when the size of each task is a single floating-point operation. The number of tasks is large and scheduling becomes a difficult problem. As the name suggests, medium grain parallelism uses tasks that are larger than single operations. This is achieved by combining a group of tasks in fine grain parallelism into a single task. Large grain parallelism uses even larger tasks, typically complete row eliminations. The main issues in deciding on task granularity are :

- The amount of parallelism available.
- The complexity of the scheduling problem.
- The amount of interprocessor communication needed.
- Suitability of the computation method to the underlying computer architecture.

Cox [12] describes three approaches based on task size. Homogeneous multi-tasking is similar to fine grain task granularity. All tasks to be executed in parallel are approximately of the same size and there are a large number of such tasks. Precedence constraints between tasks makes the problem of scheduling and reducing interprocessor communication harder. Cox observes that synchronization overheads more than offset the gains from parallel processing for this approach. Cox's coarse grain parallelism involves partitioning the circuit into sub-circuits and solving each of them individually on separate processors. Partitioning a matrix into submatrices results in over 50% increase in total number of matrix operations which must be performed. This increase in matrix operations is a result of restricted ordering within each of the sub-circuit matrices. Within these matrices, nodes that have connections to other subcircuits through the interconnect must be ordered last. This restriction produces many more fillin terms during LU decomposition than would occur without partitioning. The results indicate that hardly any improvement is achievable on a small number of processors. A mixture of these two methods yield the heterogeneous macro-tasking approach. Circuits are partitioned into functional units following the natural hierarchy of the specification. Moreover, the method treats closely coupled devices as a unit and minimizes the number of off-diagonal terms used to describe the block interconnection. All tasks such as loading of matrices, LU factorization, etc. are performed as data and resources become available, and as they are required for additional processing. Scheduling of tasks is done according to their relative priority. Cox reports that this approach gives the greatest speedup in execution time using parallel processing.

Wing and Huang [58] describe a computation model for the parallel solution of linear equations. They define a task graph that models the computation of the LU factors of a matrix, as well as forward and backward substitution. They use elementary divide and update operations as individual tasks – giving fine grain task granularity. They derive the lower bound on the number of processors required for completing the evaluation of the task graph in the minimum possible time and also a lower bound on the computation time for a

given number of processors.

The sensitivity of the performance of a parallel algorithm to the underlying architecture is shown in the paper by Saad [43]. He analyzes the behavior of two different implementations of Gaussian elimination in a hypercube based architecture. Suppose that rows (or columns) of a matrix are distributed in some way among the processors. At each step j of the first implementation, called the broadcast algorithm, the elimination row is sent to all processors that hold at least one of the rows $i, i+1, i+2, \ldots, N$. This data movement is a broadcast operation : one processor sends data to all or a subset of the others. Once broadcasting is completed, the arithmetic corresponding to the ith step of Gaussian elimination is performed.

The second approach differs only in its organization. Processors are no longer required to perform the jth step of Gaussian elimination at the same time. A processor waits for the jth row, passes it to the next processor as soon as it arrives and then proceeds with the arithmetic. The next processor will in turn pass the row to a neighbor and proceed with the arithmetic. This is called pipelining.

The second approach requires only a one dimensional or a two dimensional grid of processors for its implementation. The question arises : can we expect the broadcast method that takes advantage of the complicated hypercube topology to outperform the pipelined methods? Saad concludes that if a matrix is mapped by stripes, i.e., a few rows or columns per processor, there is no compelling reason for using a broadcast algorithm. If a matrix is split into square blocks and mapped to the nodes of a processor grid embedded in the hypercube, then pipelined techniques are better.

An interesting account of the effect of communication cost on parallel Gaussian elimination can be found in [37]. The target architecture is assumed to be an MIMD computer. The time for accessing a data from the shared memory is taken to be equal to the time for performing an arithmetic operation. They consider two forms of Gaussian elimination and show that the communication costs in one are significantly smaller than the other. Therefore communication costs also depend on the way computation is done.

Sadayappan and Visvanathan [44] discusses three levels of task granularity in parallel Gaussian elimination. Their coarse grain task granularity uses the target-row directed formulation of Gaussian elimination. Here a single task is the elimination of a complete row of a matrix. They conclude that parallelism is very limited. Their fine grain approach uses tasks that are individual update or divide operations, as in [26]. Using a task cluster

scheduling algorithm and a vector processor they were able to get appreciable speedups. They conclude that the real overhead in a fine grain approach is not scheduling but operand access. Their medium grain approach tries to alleviate the problem of operand access by replacing the repeated scatter-gather operations on the target-row by using an indirect destination pointer. They claim that this approach yields better speedup than the other two. A similar treatment of task granularity can be found in [10]. Arnold [4] shows that for parallel forward and backward elimination on an MIMD computer with a single bus, contention for the bus is not a major time consuming factor. This shows that operand access time is not a major bottleneck in the substitution phase.

2.2 Scheduling

Once tasks that can be executed in parallel are identified, it is necessary to schedule them on processing elements. One very important aspect in multiprocessor system design is the analysis of algorithms for scheduling parallel tasks on independent processors. Associated with each task is a non-negative real number representing the execution time of the task on any of the processors. A valid schedule is an assignment of the tasks to the processors so that none of the precedence relationships between the tasks are violated, and no more than one processor is assigned to a task at any instant. An additional assumption that is often made is that the execution of each task must be completed once it is begun, i.e., the schedule is non-preemptive. The objective is to find a schedule that minimizes the finishing time of the computation graph. Scheduling is a very important part of any parallel triangularization algorithm and a significant amount of research has been done in this area. The main issues in scheduling are :

- Static or Dynamic Scheduling.
- Scheduling to minimize the computation time.
- Scheduling to minimize the overhead in communication.

Finding an optimal schedule for a computation graph for more than two processors is an NP-complete problem [38]. Only heuristic solutions are obtainable. In Chapter 4 a heuristic solution to the scheduling and interprocessor communication problem is presented. In the following section the simple yet effective scheduling algorithm due to Hu is given.

2.2.1 Hu's Levelized Scheduling Algorithm

Many scheduling or sequencing problems can be formulated as follows. Given n tasks with known times to perform each tasks and with ordering relations among these tasks, two problems posed are the following :

- Assume that all tasks must be completed by time T. Arrange a schedule that requires the minimum number of processors.
- If m processors are available, arrange a schedule that completes all tasks at the earliest times.

Hu [25] proposed an algorithm for solving the above problems.

The precedence constraints between tasks can be represented by a directed acyclic graph (DAG) consisting of n nodes representing the tasks and directed arcs representing the precedence constraints. The graph must be acyclic for a schedule to be found. We shall write $N_i > N_j$ if node N_i must precede node N_j. The nodes in the graph form a partial order. A node N_k is called a final node if there does not exist a node N_i in the graph so that $N_k > N_i$. A node N_j is called a starting node if there does not exist a node N_i such that $N_i > N_j$. A node N_i is labelled with $\alpha_i = x_i + 1$ if x_i is the length of the longest path from N_i to the final node in graph G. The final node by the above rule has a label 1. We now give the algorithm for finishing the tasks at the earliest time with a given number of processors.

Algorithm

Preliminary : Label all nodes with $\alpha_i = x_i + 1$ where x_i is the length of the longest path from N_i to the final node in G.

Rule : If the total number of starting nodes is less than or equal to m where m is the total number of processors available, then choose all starting nodes for processing.

If the total number of starting nodes is greater than m , choose m starting nodes with values of α_i not less than those not chosen. In the case of a tie, the choice is arbitrary.

Remove completed tasks from the graph. Then repeat the rule for the remaining graph.

Hu shows that the above algorithm completes all tasks at the earliest possible times if the graph is a tree and all tasks are homogeneous. In most cases this assumption

is not valid, and the algorithm does not find the optimum schedule. This strategy is used by Huang and Wing in their optimal matrix triangularization algorithm [26].

2.2.2 Other work

Thomas et al. [2] compares several scheduling algorithms for parallel processing systems. Cases where task execution times are deterministic and others in which execution times are random variables are analyzed. It is shown that different algorithms suggested in the literature vary significantly in execution time. They also present a dynamic programming solution for the case in which the execution times are random variables. Despite having fixed number of operations in each task, the memory access times makes the time taken for each task a random variable. A scheduling algorithm that takes this fact into account and gives a schedule that works optimally in the average case is very important. Srinivas [51] gives an optimal scheduling algorithm for a dense Gaussian elimination DAG. He establishes a lower bound for the schedule length and proves that for dense matrices, these bounds are achieved. His algorithms, however, are not optimal for sparse systems.

Kohler [31] discusses the critical path method for scheduling tasks on multiprocessor systems. After a review of known theoretical results, a general branch and bound algorithm for finding optimal schedules is presented. The critical path heuristic is very similar to the heuristic used by Hu [25]. The schedule is generated by placing task T_i before T_j whenever the critical path of T_i exceeds that of T_j. The critical path of T_j is defined as the length of the longest path from T_j to a terminal node. The schedules produced by a simple critical path priority method are shown to be near optimal for randomly generated computation graphs.

Dekel and Sahni [15] discuss parallel scheduling algorithms. Their algorithms are targeted for Single Instruction Stream Multiple Data Stream (SIMD) computers and can be easily extended to MIMD computers. They develop a design technique for parallel algorithms based on binary computation trees [14] and consider various scheduling problems and obtain fast parallel algorithms for each. Rote [42] discusses a solution to the different but related problem of scheduling n tasks with different processing times on one processor in order to maximize the number of tasks that are scheduled in time. He uses the techniques of Dekel and Sahni [14].

Two classes of scheduling algorithms are possible. Casotto [8] uses dynamic

scheduling in the parallelization of a sparse matrix package. With dynamic scheduling there is a high overhead involved. Generalized static scheduling is discussed in the paper by Lee [18]. Chen and Hu [11] describe a task clustering algorithm which is static. Each processor has its own list of tasks and the algorithm used for assigning tasks to processors is very simple. They use an elemental task model and tasks that have the same row or column number are assigned to the same processor. Unfortunately, this algorithm can create load imbalance.

2.3 Reordering

Circuit matrices can be reordered to achieve various objectives, including numerical accuracy, stability, and maintaining sparsity. The familiar Markowitz [24] method is used to minimize the number of fillins generated. Ping Yang [61] discusses a simple partitioning and reordering algorithm that guarantees that no zero-valued pivot will be generated for a matrix formulated by the modified nodal approach. The issue of minimizing the number of fillins is also addressed. For nested dissection there exist algorithms for reordering a matrix into a block-bordered diagonal form [48]. The Cuthill-McKee [19] reordering algorithm is used to reduce the bandwidth of a symmetric sparse matrix. The minimum degree ordering algorithm [54] is a very popular fill-reducing algorithm. It reduces to the Markowitz criterion for asymmetric matrices. Betancourt [5] describes a minimum depth reordering algorithm that is based on choosing the vertex to be eliminated to be one with least depth in the computation graph. He also considers a combination of the minimum degree and minimum depth algorithm and concludes that an algorithm that picks the next vertex on the basis of depth and breaks ties on the basis of degree produces an ordering with the smallest amount of computation. A survey of various reordering techniques for sparse symmetric matrices can be found in [20]. Zmijewski [63] suggested an algorithm for ordering and partitioning that not only allows for more parallelism but also reduces the communication overhead. His work is based on the Kernighan-Lin partitioning algorithm [7].

Most of the above algorithms are targeted for reducing uniprocessor solution time and can be used for sparse, symmetric, positive definite matrices. None of these algorithms address the issue of reducing computation time in a parallel processing environment. While solving large system of equations, we are interested in solving the system in the least possible time, but not necessarily with the least amount of computation. It may be possible to trade

off fillins for better completion time. The following algorithms were aimed at minimizing the parallel computation time. For sparse, symmetric, positive definite matrices, Jess and Kees [28] provide a strategy for reordering to minimize the height of the computation graph. In a related paper, Liu and Mirazian [34] give an efficient algorithm for this reordering strategy which is linear with respect to the number of non-zeros in the filled graph. Liu [33] proposed a variant of the Jess and Kees algorithm based on Elimination tree transformations. These algorithms work only for symmetric, positive definite matrices and cannot be used for most matrices seen in circuit simulation. Huang and Wing [26] proposed an algorithm that works for all kinds of circuit matrices but for large systems, the time taken to find the operation sets and the depth sets can be quite large. Also, this algorithm assumes fine grain task granularity.

Rose [17] showed that the problem of finding a permutation matrix P for a given matrix A, such that the matrix PAP^T has the minimum number of non-zeros after elimination is NP-complete. There is almost no hope of finding a polynomial time algorithm for minimizing the number of fillins produced. This is only a sub-problem of the complete reordering problem. We are interested in permutation matrices that minimize the parallel computation time. It is my conjecture that this problem too is NP-complete and only heuristic solutions can be attempted.

2.4 Hardware-based Acceleration Techniques

While a number of number of algorithmic techniques [45,40,32,57,46,29,60] have improved the speed of circuit simulators, large-scale improvement in the speed still depends on hardware-based approaches. The first parallel computers were the SIMD (Single Instruction Stream Multiple Data Stream) computers, like vector and array processors. In the following section, work on vector and array processors is described briefly, revealing that a high degree of parallelism is available in the LOAD phase. Subsequently, special-purpose hardware approaches to parallel circuit simulation are presented.

2.4.1 Vector Processors

The first parallel processors available were vector processors and pioneering work on developing parallel circuit simulators was preformed on vector and array processors. The first parallel circuit simulator CLASSIE [55], was implemented on the CRAY-1, CYBER

205 and FPS-164. CLASSIE employed user defined node tearing [49] to decompose a circuit into cells, according to functional and structural hierarchy, at the non-linear equation level. Identical cells are analyzed in parallel using vector operations. CLASSIE provides a number of modifications to standard circuit simulation algorithms. Firstly, it orders devices according to device type thereby saving time gathering model parameters. Furthermore, by decomposing the circuit into smaller subcircuits, it circumvents the linear equation solution problem. Also, it employs code generation to reduce linear equation solution time. For sparse matrix solution, a speedup of 12 was obtained by efficient vectorization. Overall circuit simulatior run time was decreased by a factor of 4. Although CLASSIE in its vector mode is faster than its scalar mode and speedup increases as circuit size grows, its performance is not encouraging on the whole. This is primarily due to the data gather-scatter overhead in vector operations.

It was shown in [59] that vectorized LU-decomposition can result in significant gains in simulation speeds if the vector processor has special purpose hardware for data gather-scatter. Further, while CLASSIE requires identical subcircuits in parallel, the vectorization algorithm used in [59] automatically detects parallelism within the irregular structure of the matrix. The Block Vectorization Algorithm (BVA) and the Maximal Vectorization Algorithm (MVA) used in [59] give very good performance for certain examples. However, speedup figures are not too good for small vectors and, also, the speedup depends on the decomposition of the matrix. If a good decomposition cannot be found, then the speedup obtained is mediocre. In addition, vector processors, being SIMD machines, require static scheduling and this forces all elemental operations to be of the same length and the same type. A general-purpose parallel processor using an MIMD architecture, allows more flexibility in scheduling and performing individual elemental operations according to their respective requirements.

2.4.2 Special-purpose Hardware for Parallel Circuit Simulation

As a result of the increasing sophistication and the reduced cost of VLSI, it is now economical to synthesize hardware for the purpose of performing a certain set of operations that were previously performed using software. To this end, numerous designs have been proposed that replace computation intensive sections of circuit simulation programs with special-purpose hardware. A special-purpose attached processor MMAP (MOS-Model At-

tached processor) that evaluates the dc-MOS transistor equations has been designed and evaluated in [21]. The special-purpose processor is attached to an IBM PC-XT personal computer, and the circuit simulator used is BIASC [22], a subset of SPICE written in the C programming language and designed to run on the IBM PC. A prototype MMAP utilizes pipelining and local memory to evaluate linear models for four MOSFET devices simultaneously. Experiments with the prototype MMAP shows that it incurs a 20% communication overhead, while simulations indicate that with efficient operation of the system, the overhead may be as high as 60%. In addition it was observed that 40% of MMAP's time was spent in transferring data between its processing and memory unit which were on separate chips.

BLOSSOM [30] is a special-purpose architecture system proposed for parallel, sparse linear equation solution. It consists of a reconfigurable systolic array connected to a host computer. BLOSSOM uses its own memory, data bus and executive control unit to operate independently once the system to be solved has been loaded into its local memory. Submatrix operations performed by the processors are micro-coded. Results from a software simulation of the BLOSSOM system indicate that most of the partitions are 2×2 submatrices. For a 1957×1957 matrix it is seen that a 2×25 processor array solves the system 9.25 times faster than a 2×2 array, indicating 74% efficiency. Given the high efficiency of BLOSSOM, it is evident that such hardware-based approach shows promise for rapid parallel sparse linear equation solution. However, the hardware required is rather expensive, and its nature prevents any other use of it. From the economic standpoint, it is better to use a multiprocessor which can be used for solving a wide range of problems.

2.5 Use of MIMD Multiprocessors

Most recent work on parallel sparse matrix solution [36,62,27,6,9] involve the use of MIMD (Multiple Instruction Stream Multiple Data Stream) general purpose multiprocessors. Such an architecture allows more flexibility in scheduling and performing operations according to their respective requirements. The work of Jacob [27] deserves special mention as it is the starting point of this research.

In [27] tasks that can be executed in parallel are identified in two ways. In the first method, all pivots that can be eliminated independently are considered for parallel elimination. Each processor does all operations necessary for the elimination of the pivot assigned

to it. The study of different pivot selection algorithms and their respective concurrency potentials was made using the Pivot Dependency Graph (PDG). In the second method, all row operations during the elimination of a pivot, that are independent are considered for parallel execution. For this method, a Row Dependency Graph (RDG) is used. The RDG-based method is able to exploit more parallelism than the PDG-based method. This method can be made more general, and using the Task Graph approach described in Chapter 3, more parallelism can be exploited. Both dynamic and static scheduling was studied in [27]. The scheduling scheme used is quasi-static. It exploits static scheduling by using the known PDG or RDG, yet dynamically prevents high-priority tasks to preempt less important ones. Using a quasi-static scheduling algorithm Jacob was able to avoid most of the overheads inherent in dynamic scheduling when the number of tasks is large, as in the case of a large system of equations. A new pipelined algorithm was also presented, that simultaneously addressed the problems of poor linear equation parallelism and the inter-phase bottleneck between the LOAD and the SOLVE phase, and the factorization and forward and backward elimination phase. For a 155-node, 416-MOSFET transistor digital-to-analog circuit, simulation speedup using the algorithms mentioned was 5.18 on a 8-processor Sequent. Though it is possible to get substantial speedup with this approach, reordering rows and columns of a matrix to decrease parallel solution time can have a significant impact on the performance of a parallel simulator.

2.6 Conclusion

The three aspects of parallel solution of a sparse system of linear equations were discussed in this chapter. All the major problems encountered are NP-complete. Some of the heuristics used by other researchers have been described. In the following chapters each of these problems are discussed in more detail and our solutions to these problems are presented.

Chapter 3

TASK GRANULARITY

The efficient solution of dense matrices on parallel computers is fairly well understood [51,35]. However, the compact representation of sparse matrices, combined with the structural irregularity of the sequence of operations, makes parallelization of solution of a sparse matrix a challenging problem. For this purpose, the whole task has to be broken up into independent sub-tasks, which can then be solved in parallel. The amount of parallelism available depends on the size of the tasks, or task granularity. Standard algorithms for triangularization show significant amount of parallelism when each task corresponds to individual divide and update operations. This is the fine grain parallelism approach. Single operations can be combined to form larger tasks. Such tasks might correspond to computing the value of an element in the triangularized matrix. This is the medium grain approach. When all tasks correspond to operations on a row or a column, we have the large grain approach. In this chapter three levels of task granularity are examined with the goal of identifying the one most suitable for implementation on a shared memory MIMD computer.

3.1 Task Graph

Introduced by Huang and Wing [26], this graph is a representation of the operations and dependencies in the solution of a linear system of equations. The triangularization process consists of a set of operations on which a set of precedence relations exist. The process can therefore be represented by a directed acyclic graph $G(V, E)$ where V is a set of nodes representing the operations and E is a set of edges between nodes i and j if the result

$$\begin{bmatrix} a_{11} & & & a_{14} & & \\ & a_{22} & & & a_{25} & \\ & & a_{33} & a_{34} & & a_{36} \\ a_{41} & & a_{43} & a_{44} & & a_{46} \\ & a_{52} & & & a_{55} & a_{56} \\ & & a_{63} & a_{64} & a_{65} & a_{66} \end{bmatrix}$$

Figure 3.1: Example Matrix

of node i is used by node j. The graph is acyclic because there are no cyclic precedence relations in linear equation solution. For the matrix of Figure 3.1, the task graph for LU decomposition of the matrix is shown in Figure 3.2.

3.2 Fine Grain Parallelism

Consider the matrix shown in Figure 3.1. The list of operations needed to LU decompose the matrix is given below.

1. $a_{14} = a_{14}/a_{11}$
2. $a_{44} = a_{44} - a_{41} * a_{14}$
3. $a_{25} = a_{25}/a_{22}$
4. $a_{55} = a_{55} - a_{25} * a_{52}$
5. $a_{34} = a_{34}/a_{33}$
6. $a_{36} = a_{36}/a_{33}$
7. $a_{44} = a_{44} - a_{34} * a_{43}$
8. $a_{46} = a_{46} - a_{36} * a_{43}$
9. $a_{64} = a_{64} - a_{63} * a_{34}$
10. $a_{66} = a_{66} - a_{63} * a_{36}$
11. $a_{46} = a_{46}/a_{44}$

12. $a_{66} = a_{66} - a_{64} * a_{46}$

13. $a_{56} = a_{56}/a_{55}$

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

Let us assume that all floating-point operations take one unit of time. Then it takes 22 units of time to complete the LU decomposition of the matrix, using a sequential algorithm. The levelized task graph for these operations is shown in Figure 3.2. The numbers in the nodes correspond to number of the operations in the list above. The arrows are the edges depicting dependencies. A node is called *simple* if only one operation is done at that node. If more than one operation is done at a node, it is called *complex*. In Figure 3.2, nodes 1, 3, 5, 6, 11 and 13 are simple nodes while 2, 4, 7, 8, 9, 10, 12 and 14 are complex nodes. In this graph, simple nodes take one unit of time, while complex nodes take 2 units of time. Levels 1 and 4 take one unit of time to complete, all other levels take two units of time. The total time to complete the execution of the operations in the task graph, given a sufficient number of processors is 10 units. The maximum number of tasks that has to be completed at any level is 5 (at level 2), and with five processors, triangularization can be completed in the minimum possible time. This is an upper bound on the number of processors needed to complete the tasks, but not a tight upper bound because execution can be completed in 10 units of time using only four processors.

Complex nodes in the task graph can be broken up so that the graph only has simple nodes. The set of operations, considering only simple operations at each node, is given below.

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

2. $a_{34} = a_{34}/a_{33}$

3. $a_{14} = a_{14}/a_{11}$

4. $a_{25} = a_{25}/a_{22}$

5. $T_4 = a_{36} * a_{63}$

6. $T_3 = a_{36} * a_{43}$

7. $T_6 = a_{34} * a_{63}$

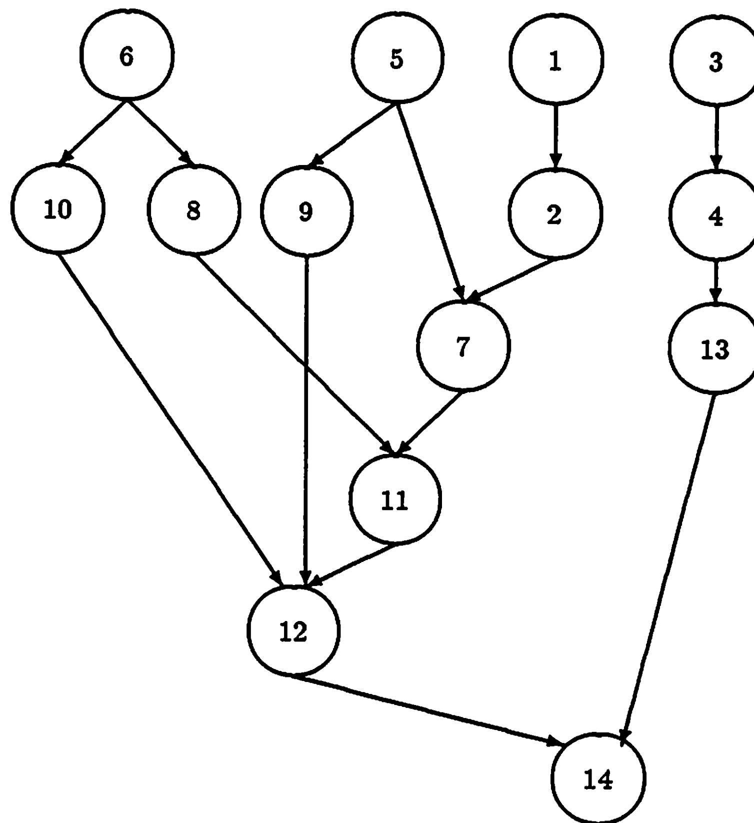

Figure 3.2: Task graph for fine grain parallelism

8. $T_5 = a_{34} * a_{43}$

9. $T_1 = a_{41} * a_{14}$

10. $T_2 = a_{25} * a_{52}$

11. $a_{66} = a_{66} - T_4$

12. $a_{46} = a_{46} - T_3$

13. $a_{64} = a_{64} - T_6$

14. $a_{44} = a_{44} - T_1$

15. $a_{55} = a_{55} - T_2$

16. $a_{44} = a_{44} - T_5$

17. $a_{56} = a_{56}/a_{55}$

18. $a_{46} = a_{46}/a_{44}$

19. $T_8 = a_{65} * a_{56}$

20. $T_7 = a_{64} * a_{46}$

21. $a_{66} = a_{66} - T_7$

22. $a_{66} = a_{66} - T_8$

The modified task graph is shown in Figure 3.3. There are eight levels in the graph, and each level takes one unit of time to complete execution. The total time to complete the execution of the task graph, given sufficient number of processors is 8 units. This time the maximum number of nodes at a level is 6 and thus a maximum of 6 processors are needed to complete the tasks in the minimum possible time. Once again, 6 is the upper bound on the number of processors needed to complete triangularization as it is possible to complete it in the minimum possible time using only four processors. Since tasks cannot be subdivided any further, this is the finest level of task granularity. Temporary variables are necessary for storing intermediate results. Such temporary variables are shown as T_i's in the above list.

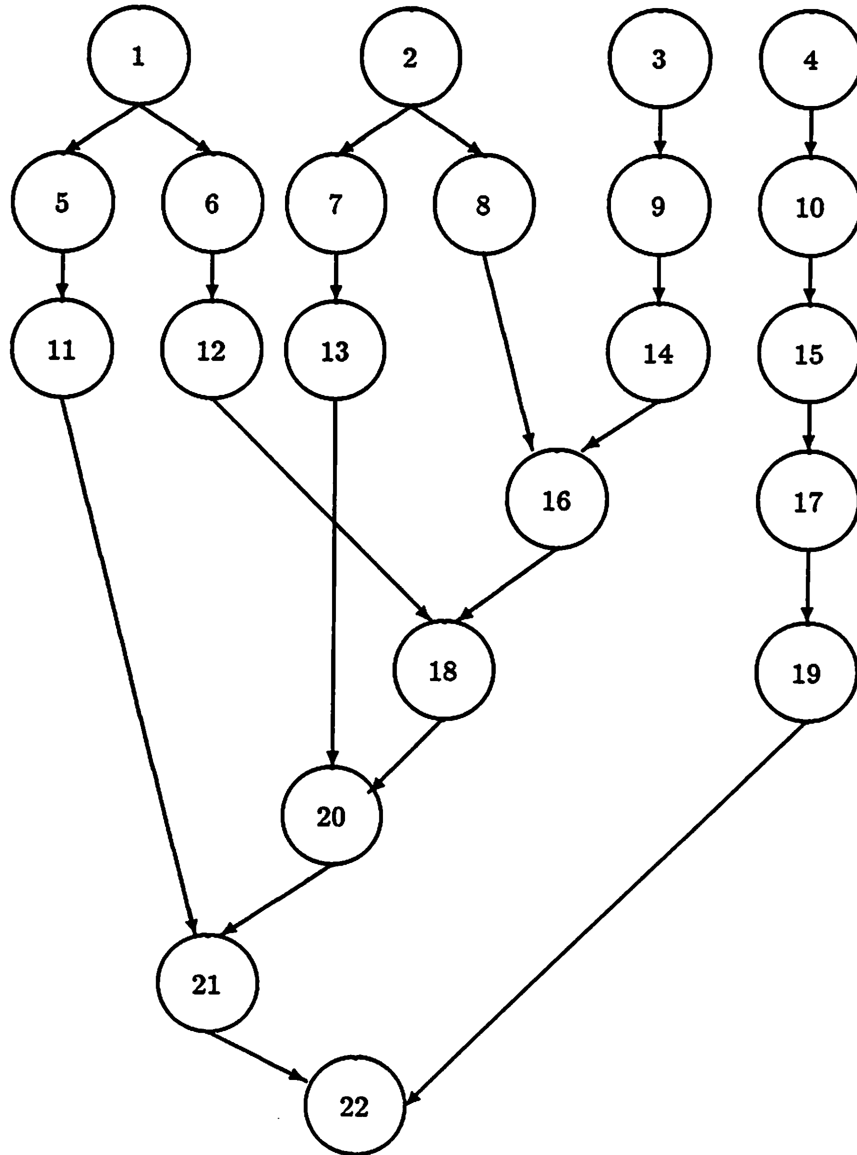

Figure 3.3: Task graph for fine grain parallelism

Fine grain parallelism is the parallelism exploited when the nodes in the task graph are single operations on elements of a matrix or on temporary variables. Tasks cannot be divided any further and all operations are represented as individual nodes in the Task Graph. Thus the maximum amount of parallelism between operations can be exploited using this approach. Though this approach extracts the maximum parallelism that is present in the solution of a sparse system of equations, there are some problems in the implementation of such a method of solution. The number of processors required to triangularize a large system in minimum time is large. It might not be possible to have shared memory MIMD computers that have that many processors. Working with a smaller number of processors than the required number, there is the additional problem of optimum scheduling of the nodes to the processors so that execution is completed in the minimum possible amount of time. This is an NP-complete problem [38] and only heuristic solutions are available, some of which have been discussed in the previous chapter. Also, there is the overhead of storing all the temporary results.

Assuming there is an adequate number of processors, it is still not possible to complete execution in the minimum possible time. This is due to the overhead incurred in synchronization and inter-processor communication. Consider the task graph of Figure 3.3, and assume there are six processors. For a processor to execute the operations at a node, it is necessary that the operands have their correct values. (e.g., Node 8 cannot be computed before node 2). A simple algorithm that ensures correct evaluation of the task graph requires all processors to work on nodes of the same level, and check in at a barrier before proceeding to the next level. Such a barrier is often implemented using *atomic locks* [1]. Say the time required for an atomic lock operation is x units. With n processors checking in at a barrier, the time taken at each barrier is nx units. If there are m barriers, then the total time taken for barrier synchronization is nxm units. On the Sequent Balance 8000, a lock-unlock operation takes twice as long as a floating-point division (i.e. two units of time). With six processors checking in at a barrier, the time required at each barrier is 12 units. Since six synchronization points are required for the graph of Figure 3.3, the total time taken for barrier synchronization is 72 units. This is far greater than the time for solving the system sequentially (22 units). It will be seen later that this problem can be alleviated using special techniques, but there is no way to do away with interprocessor communication overhead, and sophisticated programming techniques are required to exploit the amount of parallelism available.

3.2.1 Maximum Speedup Obtainable from Fine Grain Parallelism

Let the number of operations required to do Gaussian elimination or LU decomposition be $n^{1+\delta}$ for a $n \times n$ sparse matrix. There are n pivots to be eliminated. Given a sufficient number of processors, all the work necessary to eliminate any pivot can be done in a fixed amount of time. For example, for the matrix shown in Figure 3.1, if row 1 is being eliminated, the operations on all elements affected by row 1 can be done in parallel as they are independent of each other. For each element, two floating-point operations are necessary in order to update their value and therefore the elimination of a row can be done in 2 units of time. The total parallel triangularization time is $2n$. The maximum speedup achievable is :

$$Speedup = n^{\delta}/2$$

For a sparse matrix, pivots can be eliminated in parallel as often they are mutually independent. If a pivot dependency graph (PDG) [27] is used, then the time taken for parallel triangularization is $2 \times (Height\ of\ PDG)$. If the height of the PDG is represented as $n^{1-\gamma}$, then the maximum speedup achievable is :

$$Speedup = n^{\delta+\gamma}/2$$

This analysis assumes that interprocessor communication time is negligible.

3.3 Medium Grain Parallelism

As the name suggests, medium grain parallelism uses tasks that are larger than single operations. This is achieved by combining a set of nodes in the fine grain task graph into a single node. There are various ways of doing this, and consequently there are different kinds of medium grain parallelism. Sadayappan and Visvanathan [44] uses a source-row driven parallel formulation that uses the source-row directed LU decomposition algorithm and an indirection vector for each source-target row pair. The innermost loop of the operation is a vector operation through the indirection vector, which cuts down on the number of indirect accesses to the memory.

In our medium grain approach, each node represents an element in a matrix, and all operations needed to obtain the value of the element in the triangularized matrix is represented at that node. This idea is best illustrated with an example. Consider the

matrix of Figure 3.1. The list of operations necessary for LU decomposition is shown below. Each formula gives the operations necessary to compute the value of the element in the triangularized matrix. Thus each element appears on the left hand side only once.

1. $a_{14} = a_{14}/a_{11}$
2. $a_{36} = a_{36}/a_{33}$
3. $a_{34} = a_{34}/a_{33}$
4. $a_{25} = a_{25}/a_{22}$
5. $a_{44} = a_{44} - a_{14} * a_{41} - a_{34} * a_{43}$
6. $a_{55} = a_{55} - a_{52} * a_{25}$
7. $a_{46} = (a_{46} - a_{43} * a_{36})/a_{44}$
8. $a_{64} = a_{64} - a_{63} * a_{34}$
9. $a_{56} = a_{56}/a_{55}$
10. $a_{66} = a_{66} - a_{63} * a_{36} - a_{64} * a_{46} - a_{65} * a_{56}$

The levelized task graph for these operations is shown in Figure 3.4. Level 1 takes one unit of time, Level 2 takes 4 units, Level 3 takes 3 units and Level 4 takes 6 units. Total solution time is 14 units, compared to the 22 units required for a purely sequential solution (and 8 units for a fine grain parallel solution). The amount of parallelism obtainable decreases, but now a smaller number of processors is necessary to achieve the minimum possible solution time. Since the number of processors required is smaller, the concomitant scheduling and synchronization problems are less severe.

The problems of scheduling and synchronization encountered in medium grain parallelism are more tractable than the corresponding ones in fine grain parallelism. Despite the loss of a significant amount of parallelism, there is a reduction in the amount of overhead. Using the same synchronization algorithm as used for fine grain parallelism, the total time spent in barrier synchronization is 24 units (8 units at each barrier and only three barriers are needed). This is much smaller compared to the 72 units of time spent in barrier synchronization for fine grain parallelism.

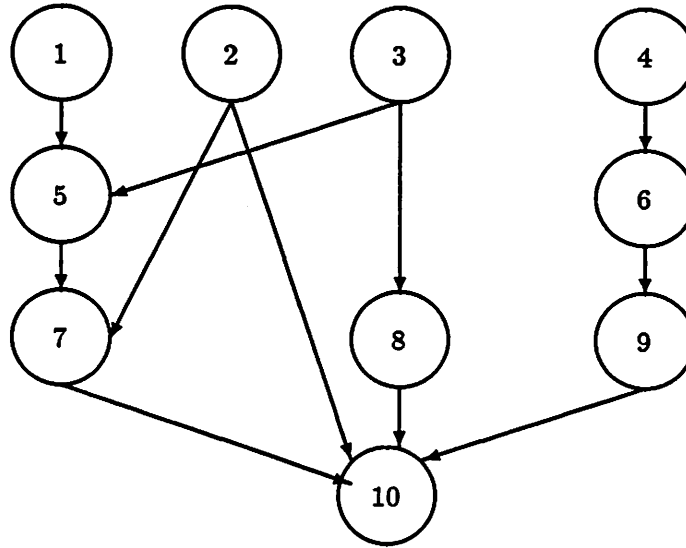

Figure 3.4: Task graph for medium grain parallelism

3.4 Large Grain Parallelism

Large grain parallelism uses tasks that are typically larger than single element update operations. Often, tasks are complete row eliminations. Sadayappan and Visvanathan [44] used the target-row directed approach, where given a pivot, all rows depending on the pivot are eliminated on a single processor. Jacob [27] uses the pivot dependency graph technique to obtain pivots that can be eliminated in parallel.

Consider the matrix of Figure 3.1. Considering each row and column at a time, the operations required for LU decomposition are given below.

1. $a_{14} = a_{14}/a_{11}$

2. $a_{25} = a_{25}/a_{22}$

3. $a_{34} = a_{34}/a_{33}$, $a_{36} = a_{36}/a_{33}$

4. $a_{44} = a_{44} - a_{14} * a_{41} - a_{34} * a_{43}$, $a_{64} = a_{64} - a_{63} * a_{34}$

5. $a_{46} = (a_{46} - a_{43} * a_{36})/a_{44}$

6. $a_{55} = a_{55} - a_{52} * a_{25}$

7. $a_{56} = a_{56}/a_{55}$

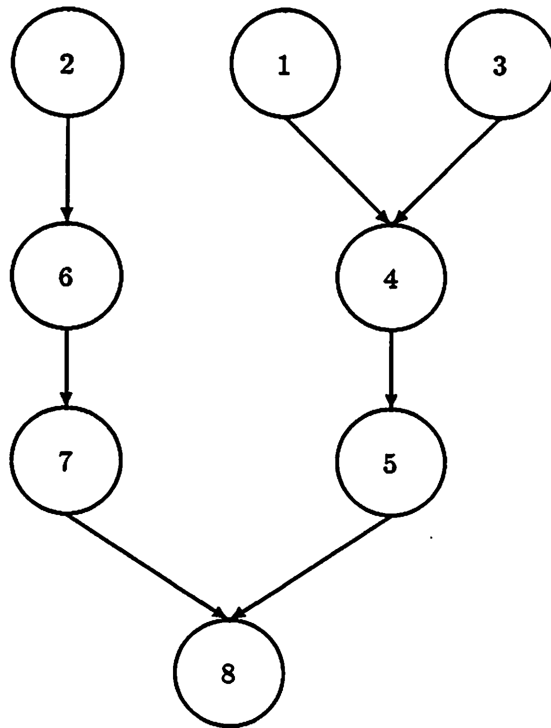

Figure 3.5: Task graph for large grain parallelism

8. $a_{66} = a_{66} - a_{63} * a_{36} - a_{64} * a_{46} - a_{65} * a_{56}$

The levelized task graph for these operations is shown in Figure 3.5. This graph has 4 levels and the total time required to solve it is 16 units (Level 1 takes 1 unit, 2 takes 6 units, 3 takes 3 units and 4 takes 6 units). The maximum number of processors required is three.

Large grain parallelism has a major drawback. It exploits only a limited amount of parallelism. For most applications the amount of parallelism achievable is so limited that though only a few processors are needed to achieve the maximum speedup, and the speedup is very small.

Considering the advantages and disadvantages of all levels of task granularity, we conclude that the medium grain parallelism offers the best compromise. It has an appreciable amount of parallelism and yet does not have severe synchronization and scheduling problems. This approach is used for our work.

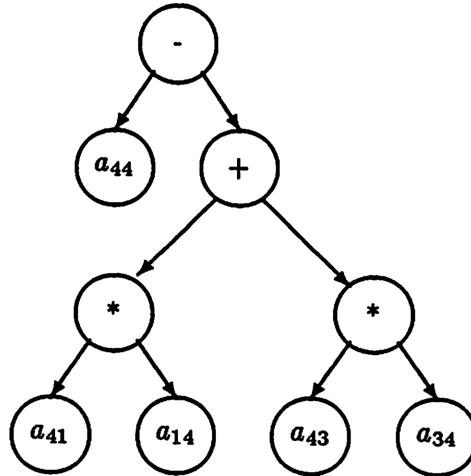

Figure 3.6: Computation tree for element a_{44}

3.5 Implementation of Medium Grain Approach

In this section the single processor implementation of the element based medium grain parallelism approach is described. At various stages, the matrix of Figure 3.1 will be used as an example.

Given a matrix, the objective is to find the value of each of its elements in the triangularized matrix. The algorithm used for triangularization gives a specific sequence of operations needed to obtain the value of an element in the triangularized matrix from its original value. This sequence of operations is represented symbolically at each element in the form of a *computation tree.* For example, element a_{44}, has the computation tree is shown in Figure 3.6. Internally, this tree is represented using linked lists. The computation tree of an element is *evaluated* to give the value of that element in the triangularized matrix. Before building computation trees, all possible fillin's are generated. The computation tree method of representing computation gives us great flexibility in trying various algorithms and evaluating their performance. In our implementation, the algorithm for triangularization is provided in a file using a pseudo-C syntax. This algorithm is then parsed and computation trees are built for each element. Fillin generation is also done according to the algorithm specified.

After getting computation trees for each element, fanout lists are built. The fanout list of an element a is a list of other elements in the matrix that need the value of element a in computing their values. For example, the fanout list of element a_{43} contains elements

Circuit	Matrix Size	No. of Elements	No. of Level 0 Elements	Time for Sequential Solution (s)	Time for One Proc. Solution (s)
IYOUNG	76	1222	272	0.50	0.62
PSPI	80	687	165	0.15	0.23
NAY	166	1202	489	0.11	0.17
A2D1	218	1630	673	0.17	0.23
BEN2K	402	2205	1438	0.61	0.19
ADDER	450	4360	1370	1.14	1.31
EPROM	687	4199	2111	0.81	0.44
XX1	1013	7145	2287	1.12	0.93

Table 3.1: Time to solve for various matrices

a_{44} and a_{46}. Fanout lists are built by looking at the computation tree of an element a and putting a in the fanout list of all elements that feature in its computation tree. These fanout lists are edges of the task graph, while the elements are the nodes. The work to be done at a node is represented by the computation tree.

After all fanout lists are made, the task graph is levelized. All elements that don't need any computation are marked as level zero elements and are also included as a part of the task graph. A simple levelizing algorithm is used to assign levels to other elements in the graph. Only elements of Level 1 and higher are evaluated. Triangularization proceeds by starting with Level 1 elements. All elements at a particular level are computed before proceeding to the next. Triangularization is complete when all elements have been evaluated.

3.6 Results

Table 3.1 contains results from triangularizing various matrices using our method. The time taken on a single processor by our method is compared to the time taken by the uniprocessor algorithm used in SPICE3.

We see that for some examples, the time taken by the sequential algorithm is more than the time taken by our method. This is because our method ignores all level zero elements during computation. The sequential algorithm deduces the same information after considerable searching. This indicates that uniprocessor triangularization algorithms can be improved by identifying level zero elements during the first solution, and bypassing

them during subsequent solutions. In cases where the number of level zero elements are small, the sequential algorithm is faster. This is because doing any computation using a computation tree is slower than doing it with compiled code.

3.7 Conclusions

In this chapter three levels of task granularity for the parallel solution of a sparse system were examined. Fine grain parallelism exploits the maximum amount of parallelism but has large scheduling and synchronization overhead. Large grain parallelism, though relatively free from such problems, exploits very little parallelism. Medium grain parallelism is a good compromise between the two extremes. An implementation of the medium grain approach was described, and it was shown that for most large systems, a single processor solution was faster than a sequential algorithm. In the following chapter, we examine the issues involved in scheduling for this element based medium grain approach.

Chapter 4

SCHEDULING

Once tasks that can be executed in parallel are identified, it is necessary to schedule them to processors so that none of the precedence constraints are violated. For any given task graph, there is a minimum time required to evaluate all the nodes of the graph. For a sequential algorithm, this is the sum of the times required for each node. Let this time be T_{seq}. Given a large number of processors (equal to or more than the required upper bound) a parallel algorithm will require a smaller amount of time. The time required is the sum of the times of the nodes on the longest path in the graph. Let this time be the *critical path time* denoted by the symbol T_c. It is not possible to evaluate the task graph in less than the critical path time even when more processors are added. The breadth of a task graph having n levels is defined to be :

$$breadth = \max_{i=1}^{n} \ (number\ of\ nodes\ at\ level\ i)$$

Clearly, the maximum number of processors needed to complete evaluation of the task graph within the critical path time is equal to the breadth of the task graph. This is the upper bound on the number of processors required. Whenever there are as many processors as the upper bound, scheduling is trivial. The algorithm for parallel evaluation is also simple. All processors work on nodes of a particular level at one time. Any processor can select any node. After all nodes at a particular level are computed, then all the processors move on to the next level. This is done until all nodes are computed.

For a large system of equations, the breadth of the task graph is much larger than the number of processors available. Under such circumstances, the simple algorithm given above fails to complete evaluation within T_c. Whenever the number of processors is

smaller than the upper bound, it might not be possible to complete the execution within T_c. However, for the given number of processors, there exists a minimum possible time for evaluating the task graph. Let this time be the ***minimum task graph evaluation time using n processors*** denoted by the symbol T_n. Our scheduling algorithm should be able to schedule nodes onto processors so that evaluation is complete within T_n. The simple scheduling algorithm rarely completes an evaluation within T_n. In fact, it can be shown that finding a schedule that completes evaluation within T_n is an NP-complete problem [38]. In most cases, we have to be content with near-optimal schedules found using heuristics.

The way scheduling is done places restrictions on how the processors should evaluate the nodes so that precedence relations of the task graph are not violated. In the simple example above, it was required that all processors work on nodes of the same level; if any processor finishes early, it waits for all others to finish before it goes to the next level. Interprocessor communication is necessary to ensure such a protocol. Obviously, there is some overhead involved in doing this and the overhead is dependent on the kind of scheduling algorithm used.

In general, there are two classes of scheduling algorithms — dynamic scheduling algorithms and static scheduling algorithms. In the next two sections these two techniques are discussed.

4.1 Terminology

Parent and Child Processes

In many parallel processing applications, programs start executing on a single processor as a single process. This process is called the parent process. When a portion of the task can be executed in parallel, the parent *forks* (creates) a number of new processes called child processes. A child process is a duplicate of the parent process, with the same data, register contents and program counter. If the parent has access to files and shared memory, the child has access to the same files and shared memory. However, they have different process identification numbers (PID's). On the Sequent Balance 8000 whenever a child is created, its PID is returned to the parent. From this point on, the parent and the child are separate entities. The child processes then start running on the other processors of the system. After the child processes have completed their work, the parent process

terminates them.

A UNIX fork operation is relatively expensive (about 55 milliseconds on the Balance 8000). A parallel application typically forks as many processes as it is likely to need at the beginning of a program and does not terminate them until the program is complete.

DYNIX (version of UNIX on the Sequent Balance 8000) allows any parent process to fork only as many child processes as there are processors, giving a one-to-one mapping between processes and processors. In this report, the terms are often used interchangeably. Their meanings should be clear from the context in which they are used.

Parent and Child Tasks

Precedence relations in a task graph are represented as edges between nodes. If there is an edge from node i to node j, then node i is the parent task of node j and node j is the child task of node i.

Locks

At any point in the execution of a parallel program, when two or more parallel processes can read or write the same data structure, then the results of the program depends on when a given process references that data structure. There are two basic types of dependencies : access dependencies and order dependencies. Access violations occur when two or more processes try to access a shared data at the same time. Order violations occur if two or more processes try to access the same shared data structure in the wrong order. To avoid these violations, processes must communicate with each other to execute the dependent sections one at a time. This communication is done using *locks.*

A lock can ensure that only one process accesses a shared data structure at a time. A lock has two values : locked and unlocked. Before attempting to access a shared data structure, a process waits until the shared data structure is unlocked. The process then locks the lock, accesses the shared data structure and unlocks the lock. When a process waits for a lock to become unlocked, it spins in a tight loop, doing no useful work. This spinning is referred to as *busy-wait.*

The hardware locks provided on the Sequent are referred to as *atomic locks* because the action required to lock are performed in a single atomic (indivisible) operation. Hence it is impossible for two processes to acquire the lock at the same time.

Barrier

A barrier is a synchronization point. A processor does the following when it reaches a barrier :

1. Mark myself present at the barrier.
2. Wait for the other processors to arrive.
3. Proceed.

4.2 Dynamic Scheduling

In dynamic scheduling, each processor schedules its own task at run time by checking a task queue. The task queue contains tasks that are waiting to be executed. In our case, all nodes requiring evaluation are kept in the queue. Tasks are put in the queue before hand, or are put in the queue by other processors. A processor takes the first task from this queue, and works on it. As soon as it completes its work, it takes the next available task from the queue. If no more tasks are available, then the processor idles, waiting for all other processors to finish.

A useful feature of dynamic scheduling is that it results in dynamic load balancing. All processes keep working as long as there is work to be done. Since the workload is distributed among the processes, the work can be completed sooner. Unfortunately, dynamic scheduling entails a significant amount of overhead. Suppose a task graph has k nodes and m levels. Let us assume that there is only one task queue, and all the nodes of the task graph are already put there and say there are n processors. It is necessary to use *atomic locks* to ensure that only one processor takes an element from the queue at a time. If locking takes two units of time, it requires $2k$ units of time to take elements out of the queue. In order to enforce the precedence relationships, no processor should be allowed to work on a task as long as its parent tasks are not completed. Whenever a processor takes a task from the queue, it has to verify that the parents of the task have been evaluated. This can be done in many ways. One way to do this would be to store explicit information about the parents at each node. Also, each node can have a flag to indicate whether it is computed or not. Whenever a node is to be computed, the processor makes sure that all parents have this flag set. When a processor finds that a parent task has not finished, it spends time idling in a wait loop.

Another way of enforcing the precedence relations is to make sure that all processors work on nodes of the same level at any time. A global level counter is used to keep track of the level of the nodes the processors are working on. If a processor gets a node with level different from the level indicated by the level counter, it checks in at a barrier to indicate that no more tasks for that level are left. As soon as all processors have checked in at the barrier, one of the processors increments the counter index and the processors start working on the nodes of the next level. This approach is equivalent to using multiple task queues, one for each level of the graph. Processors work on nodes at a level by taking a task from the queue corresponding to that level. When no more tasks are left, processors check in at a barrier before proceeding to the queue for the next level. Since there are n processors checking in at the barrier and there are m levels, $2n(m-1)$ units of time is required for barrier synchronization. Though the time spent in sychronization is higher here than in the previous algorithm, this algorithm uses less memory since explicit information about parent tasks don't have to be stored. This is a memory versus speed tradeoff. An estimate of the worst case overhead involved in dynamic scheduling is $2k + 2n(m-1)$ units of time.

4.3 Static Scheduling

In static scheduling, tasks are scheduled at run time, but are divided in some predetermined way, so that each processor has its own set of tasks to do. Given a set of tasks, the processors complete all their tasks and wait for other processors to finish. Since each processor has its own task queue, there is no need to lock before removing tasks from the queue, thereby eliminating the $2k$ units of time overhead involved in dynamic scheduling. The time required for initial scheduling is the only overhead. In most applications the task graph is evaluated many times and the cost of initial scheduling is spread over the number of times the solution of the system of equations is attempted. If the time taken to complete each task is known *a priori*, it is also possible to achieve proper load balancing.

Precedence relations and the constraints they impose on the way the processors evaluate the tasks have been ignored hitherto. The barrier synchronization algorithm described above can be used to ensure proper evaluation of each node. In that case, the overhead for synchronization is comparable to that of dynamic scheduling. It will be shown later that with some preprocessing, this overhead can be significantly reduced for static schedules. Static scheduling is more attractive since it has reduced overhead.

4.4 Scheduling Algorithm

In this section the scheduling algorithm used for our element based medium grain approach is described. Since static scheduling has significantly smaller overhead than dynamic scheduling, static scheduling is chosen. Since the the number and type of operations at each node is known *a priori*, it is possible to schedule the nodes so that load balancing is achieved and processors have minimum idle time.

We defer the discussion on synchronization and interprocessor communication until the next section. In this section, it is assumed that no interprocessor communication is necessary, and a scheduling algorithm is described that produces schedules that have minimal computation time.

The medium grain approach produces tasks that are not homogeneous in terms of the number of operations needed to complete each task. If all processors are working on nodes of the same level, it is not necessary that they all take the same time. In order to get an optimal schedule, it has to be ensured (to the maximum extent possible) that nodes that are scheduled on processors that finish earlier do not depend on the nodes that are going to take a longer time. This added degree of freedom makes the scheduling problem more difficult. The algorithm described below is a simple modification of Hu's levelized scheduling algorithm. Since tasks are not homogeneous, they are not scheduled only on the basis of their level in the graph, but processor idle time and load balancing is also taken into account.

4.4.1 Algorithm

While scheduling nodes to processors we try to achieve proper load balancing and seek the minimum possible execution time. Firstly, all the nodes on the critical path are assigned to a single processor. If all the other nodes can be scheduled on the other processors so that none of them have more load than the processor that evaluates the critical path, then the schedule is the best possible one. The scheduling algorithm starts at the topmost level of the task graph. While scheduling nodes at a particular level, all nodes at that level are sorted in a way that will be described shortly. A node is taken from this list and assigned to the processor that until then had the smallest load. After all nodes at a particular level have been scheduled, the same procedure is repeated for nodes at the next level, until the bottom most level is reached. The algorithm is given in Figure 4.1.

```
build_schedules(task_graph, num_procs)
{
   /*Input : Task graph and Number of Processors.
   Output : An ordered list of nodes for each processor. */

   /* First get the critical path through the graph */
   cpath = critical_path(task_graph);
   level = 1 ;

   while (level <= max_level){
      node_list = all nodes in the task graph with level = level;
      /* Sort the node list in descending order of time required to
      evaluate the node */
      sort_node_list (node_list);
      /* Put the critical path node on Processor 0 */
      add_to_list(listP[0], cpathnode);
      while (node_list){
         procnum = choose_processor(node_list, time_array, num_procs)
         add_to_list(listP[procnum], node);
         time_array[procnum] = time_array[procnum] + time_of_node(node);
      }
      level = level + 1;
   }
}
```

Figure 4.1: Scheduling algorithm

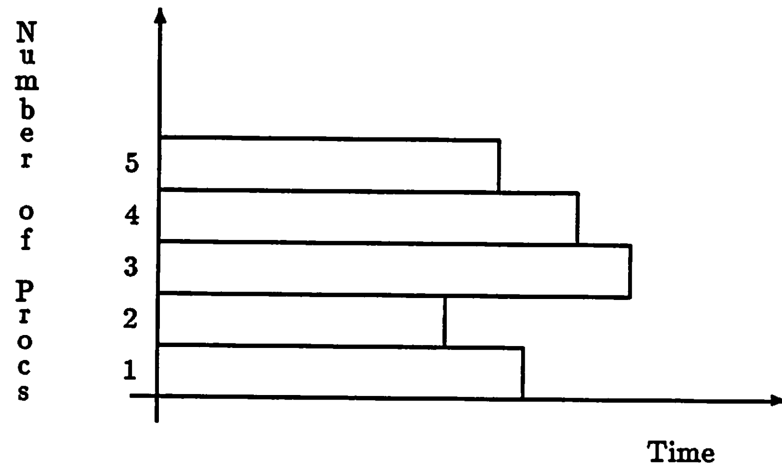

Figure 4.2: Load on processors at some point during scheduling

The routine **critical path()** gives us the critical path through the graph. All nodes at a particular level are sorted into a list called the *node_list*. The routine **add_to_list()** adds a node to the list of a particular processor. As an index of the load on the processors, an array called *time_array* is maintained. At any point this array contains the total amount of work done by that processor in terms of the amount of time it will need to compute all the nodes scheduled for it. The routine **choose_processor()** selects the processor with the minimum possible load. Thus nodes are always scheduled on the least loaded processor.

Consider the situation depicted in Figure 4.2. The figure shows the load on each processor in terms of the time required to complete all the tasks scheduled for it. If a node is now scheduled, it will be assigned to processor number 2. However, if this node depends on the node which processor 1 is working on, then processor 2 will spend time waiting for processor 1 to finish before it can start computing. If there are tasks available that are ready, then processor two can start working on them. To eliminate this wait time the time when a task is ready has to be estimated.

Given a task graph, the number of operations needed at each node is an indication of the amount of time taken to complete the evaluation of that node. If a node has n operations, we say that it takes n units of time to evaluate the node. The symbol t_e is used to denote this evaluation time. Since all computation trees are known *a priori*, evaluation times can be assigned to each node. Assuming that all computations start at time $t = 0$, then the time of completion (t_c) of a node is the time when some processor completes the

evaluation of that node. Assume that nodes of level k are being scheduled. Then all parents of the nodes have already been scheduled. It is possible to find the t_c for all parents of any node that is about to be scheduled. Assign to all nodes a number $T_{c_{max}}$ defined as :

$$T_{c_{max}} = \max \{ t_c \textit{ of all parents of the node } \}$$

Then $T_{c_{max}}$ for a node is the time when the node becomes ready for computation. Nodes are sorted in ascending order according to the value of their $T_{c_{max}}$. Nodes are scheduled by picking up the head of the list and scheduling it onto the least loaded processor.

For some levels of the task graph, there might be n nodes at that level, while there are k ($k > n$) processors available. If subsequent levels in the graph also have n or smaller number of nodes, then only n processors are necessary for the rest of the triangularization. Whenever the number of nodes at any level becomes less than the number of processors available, then only the required number of processors is used for computation. For such situations, some processors are deliberately alloted more tasks than others in order to minimize the interprocessor communication overhead.

4.4.2 Implementation

The above algorithm is very easy to implement. We have built into its implementation some additional features that help us evaluate the quality of our schedules. Given a task graph, the number of operations needed at each node is an indication of the amount of time taken to complete the evaluation of that node. If a node has k operations, then its t_e is equal to k. Since all computation trees are known *a priori*, each node can be assigned an evaluation time. Assuming that all computations started at time $t = 0$, then the time of completion (t_c) of a node is the time when some processor completes the evaluation of the node. Assume there is an adequate number of processors, so that when a node is ready for evaluation, there is a free processor that can evaluate it. Then the time of completion for a node is :

$$t_c = \max_{i \epsilon A} (t_{c_i} : i \epsilon A) + t_e$$

The set A is the set of all parents of the node. and t_{c_i} is the t_c of the ith parent. The time of completion for all nodes at level one is equal to their evaluation times. The minimum possible time of evaluation of the task graph is the maximum of the times of completion of all nodes. It is easy to compute the time of completion of each node in the task graph and

find their maximum. This is the minimum possible time of evaluation of the task graph, assuming no synchronization overheads. This time is called T_{min}.

Once a schedule is made, it is of interest to find out the time required by processors using the schedule to complete the execution of the task graph. Each processor has a list of nodes that it has to evaluate. For each node, the wait time t_w (using previous notation) is defined as :

$$t_w = \max_{i \epsilon A} (t_{c_i} : i \epsilon A) - T_{prev}$$

T_{prev} is the time the processor completes the execution of the previous node in its list. All negative wait times are set to zero. Then the time of completion of any node is defined to be :

$$t_c = T_{prev} + t_w + t_e$$

All level zero nodes have zero t_w. We can compute the t_c of each node in a list and the t_c of the last node in each list gives us the time taken to evaluate the list. The maximum of completion times for all lists gives us the time taken to complete the evaluation of the task graph. This time is called T_{sched}. The sum of the wait times on individual processors is also an indication of the total idle time.

4.5 Interprocessor Communication

In order to enforce the precedence relations of the task graph, it is necessary that nodes be evaluated only after their parents have been evaluated. In the previous section, the concept of wait time was introduced. Wait time can be estimated but cannot be accurately known. This is because floating-point operations are not the only factors that determine the time of evaluation. Non-deterministic memory access times play a major role in the evaluation time. This non-deterministic nature of the evaluation time of each node prevents guaranteeing that waiting for a predetermined time for the parents to complete will always ensure that the parent tasks are complete. Thus some form of interprocessor communication is necessary.

The barrier synchronization method investigated in the section on static scheduling has large overhead when the number of processors and the number of levels in the task graph are large. We propose a different method of synchronization which is based on the following observation. To reach a particular node in the list, the processor must have completed the

evaluation of all nodes before that node in the list. Thus all parents of that node that are scheduled on the same processor have already been computed. It is only necessary to make sure that the other parents of the node have been computed before computing that node. The other parents of the node have been assigned to other processors, sometimes more than one parent to a processor. Since the position of the parents in the lists of the other processors is known, it is only necessary to see if the other processors have processed beyond that particular element in their list. In that case all parents have been computed and the node can be evaluated. In case there is more than one parent on a single processor, it is only necessary to consider the parent furthest down in the list. All nodes in the list have an index, which is their position in the list. Say we have an array called the Processor Index Array (PIA), which for each processor holds the index of the element it is working on. Associated with each node in the list is a Required Index Array (RIA) which holds the minimum index value of each processor before the element can be computed. When corresponding elements of the Processor Index Array are greater than the Required Index Array, the element can be computed. The algorithm for evaluating all the nodes in the node list for a particular processor is given in Figure 4.3.

4.5.1 Reducing synchronization overhead

The problem of scheduling nodes of a task graph onto k processors is similar to the problem of partitioning the nodes of a graph into k subsets. Then the problem of reducing inter-processor communication is equivalent to finding partitions no larger than a given maximum size, so as to minimize the total cost of the edges cut. If all edges are assigned the same cost, then this is equivalent to finding partitions with smallest number of edges between them. Since edges between partitions imply inter-processor communication, a partition like the one above will require the minimum interprocessor communication.

There are various approaches to solving this problem. Since the problem is NP-complete, finding optimum partitions by exhaustive search takes an inordinate amount of computation. Max flow-min cut and clustering are the more commonly used methods. However, these methods do not in general include any provision for satisfying constraints on the sizes of the subsets. Kernighan and Lin [7] developed an efficient heuristic procedure for partitioning graphs. This algorithm is used to improve the schedules so that interprocessor communication is minimized.

```
solve(node_list)
{
   /* Input : A node list
   Output : All nodes evaluated */

   while (node_list not exhausted){
      node = next_in_list(node_list);
      until (PIA > RIA)
         spin;
      evaluate (node);
      increment_processor_index();
   }
}
```

Figure 4.3: Synchronization algorithm

Algorithm

The partitions generated by the algorithm of Figure 4.1 constitute the starting partitions. The essential idea is to start with these partitions and then by repeated application of the two way partitioning procedure make the partitions as close as possible to pairwise optimal. Of course pairwise optimality is only a necessary condition for global optimality. There may be situations where some complex interchange of three or more items from three or more subsets is required to reduce the pairwise optimal solution to the global optimum. However, there is no efficient procedure to do this. The algorithm selects (i, j) as the next pair of sets to be optimized where either i or j has been changed since the last time the pair (i, j) was considered.

During pairwise optimization, the following rules have to be observed while interchanging nodes between processors. These rules ensure that the resulting schedule won't lead to a deadlock or cause serious load imbalance.

- Only two elements of the same level can be interchanged with each other.
- The nodes selected should roughly have the same t_e. Heuristically, their t_e's should not differ by more than 10%.

4.5.2 Implementation

It is necessary to do some preprocessing to build up the Required Index Array at each node in the list. The computation tree of each node gives us the parents of the node. Since during scheduling the processor assignment for each node can be stored, it is easy to find the processors on which the parents are evaluated. If a parent of a node is evaluated by the same processor as the node, it is ignored. For parents on other processors, the Required Index Array holds the index of the node. For two parent nodes on the same processor, the higher index number is stored in the Required Index Array. This preprocessing is done once at the beginning of the solution process. Since the task graph is evaluated many times, the cost of preprocessing is spread over the number of times it is evaluated. Each processor writes to different locations in the Processor Index Array, while reading from all others. Therefore locks are not needed. Elimination of locks saves a considerable amount of time since locking and unlocking are costly operations.

Circuit	Size	δ	γ	Max. Speedup Fine Grain	Breadth of Task Graph	T_c	T_{seq}	Max Speedup Med. Grain
IYOUNG	76	1.13	0.18	145	55	1695	28689	17
PSPI	80	0.87	0.26	70	40	690	10056	15
NAY	166	0.54	0.36	49	194	987	6249	6
A2D1	218	0.51	0.52	128	251	573	8253	14
BEN2K	402	0.35	0.58	132	411	1842	6888	4
ADDER	450	0.63	0.38	239	536	3468	56049	16
EPROM	687	0.35	0.58	217	696	1923	14874	8
XX1	1013	0.37	0.23	32	381	2754	31686	12

Table 4.1: Characteristics of the circuit matrices

Circuit	Number of Processors							
	4		6		8		11	
	T_{sched}	S	T_{sched}	S	T_{sched}	S	T_{sched}	S
IYOUNG	7377	3.88	5019	5.71	3930	7.30	3276	8.75
PSPI	2529	3.97	1719	5.84	1356	7.41	1149	8.75
NAY	1734	3.60	1392	4.48	1266	4.93	1176	5.31
A2D1	2199	3.75	1608	5.13	1299	6.35	1125	7.34
BEN2K	2802	2.45	2424	2.84	2247	3.06	2154	3.19
ADDER	14505	3.86	10035	5.58	7785	7.19	6555	8.55
EPROM	4650	3.19	3588	4.14	3099	4.79	2814	5.28
XX1	9462	3.34	7818	4.05	7053	4.49	6720	4.71

Table 4.2: Performance predicted by scheduling algorithm

4.6 Results

In this section the results for our scheduling and synchronization algorithms are presented. The first set of results show the effectiveness of the scheduling algorithm. The meanings of most of the terms are explained in the corresponding sections. Table 4.1 gives the characteristics of the circuit matrices and gives the speedups theoretically attainable from the fine grain and the medium grain approach. The δ and γ of the matrices discussed here is the same as that in chapter 3. The symbol S denotes speedup. The speedup formula for the fine grain approach is obtained by using the formula in chapter 3. For the medium grain approach, the maximum possible speedup is obtained using the formula :

$$Speedup = T_{seq}/T_c$$

It is seen that in most cases, the scheduling algorithm finds a schedule that can

Circuit	T_{seq}	Number of Processors							
		4		6		8		11	
		T_{sol}	S	T_{sol}	S	T_{sol}	S	T_{sol}	S
IYOUNG	0.62	0.16	3.83	0.11	5.54	0.08	7.03	0.06	8.98
PSPI	0.23	0.06	3.84	0.04	5.85	0.03	7.44	0.02	9.39
NAY	0.17	0.05	3.57	0.04	4.63	0.03	5.26	0.02	5.90
A2D1	0.23	0.07	3.68	0.05	5.13	0.04	6.46	0.03	8.26
BEN2K	0.19	0.07	2.55	0.06	3.09	0.05	3.41	0.05	3.67
ADDER	1.31	0.34	3.79	0.24	5.45	0.19	6.97	0.14	8.88
EPROM	0.44	0.13	3.37	0.10	4.47	0.08	5.37	0.07	6.16
XX1	0.93	0.30	3.10	0.25	3.73	0.23	4.02	0.19	4.85

Table 4.3: Speedup achievable without synchronization

Circuit	T_{seq}	Number of Processors							
		4		6		8		11	
		T_{sol}	S	T_{sol}	S	T_{sol}	S	T_{sol}	S
IYOUNG	0.62	0.17	3.66	0.12	5.25	0.09	6.58	0.07	8.31
PSPI	0.23	0.06	3.55	0.05	5.21	0.04	5.82	0.03	8.46
NAY	0.17	0.05	3.50	0.04	4.39	0.03	5.26	0.02	6.13
A2D1	0.23	0.07	3.37	0.05	4.61	0.04	5.69	0.03	7.34
BEN2K	0.19	0.07	2.58	0.06	3.19	0.05	3.56	0.04	4.04
ADDER	1.31	0.36	3.56	0.25	5.06	0.20	6.35	0.16	8.01
EPROM	0.44	0.14	3.02	0.11	3.98	0.09	4.79	0.07	5.69
XX1	0.93	0.31	3.03	0.24	3.79	0.21	4.35	0.19	4.80

Table 4.4: Speedup achievable with synchronization

complete execution within a time close to the minimum, for a smaller number of processors than the upper bound.

Table 4.3 and Table 4.4 gives the actual execution time of various task graphs with varying number of processors. To estimate the overhead involved, each task graph is solved in two ways, once using waiting, the other one using our synchronization algorithm. Waiting is done by using the wait time derived as a by-product of finding T_c. If the wait time is n units, then n dummy floating-point divisions are done before proceeding to the next node in the list. This algorithm doesn't evaluate the task graph correctly most of the times, but gives us a good estimate of the time that solution would take if there were no synchronization costs. Table 4.3 gives the solution times using this waiting algorithm. Table 4.4 gives the solution times using our synchronization method.

We note that the overhead introduced by our scheduling algorithm is quite small.

Circuit	Number of Processors	Speedup Jacob	Speedup Our algo
DECPLA	1	1.00	1.00
	2	2.61	1.87
	4	2.41	3.38
	6	2.83	4.89
	8	2.51	6.01
DAC	1	1.00	1.00
	2	1.84	1.92
	4	3.15	3.61
	6	3.85	5.10
	8	4.47	6.37

Table 4.5: Speedup comparison

A comparison is made between the speedups obtained by Jacob [27] against those obtained by the approach presented here, for examples for which the same matrices were available. It is seen that the approach presented here provides significantly better speedups.

4.7 Conclusion

Scheduling is an important aspect of any parallel algorithm for solving a system of linear equations. Using our medium grain approach, scheduling is made difficult because of the heterogeneity of the tasks. However, in most cases, our scheduling algorithm is able to find near minimum time schedules for a smaller number of processors.

A different synchronization algorithm was given and it was shown that it was better than barrier synchronization. The overhead incurred is quite small, and in most cases speedups close to the maximum possible were obtained. Comparison of the approach presented here with that of Jacob [27] showed that the approach presented here was better.

Chapter 5

REORDERING

It is well known in sparse matrix techniques that the number of divide and update operations depends on the order in which the rows and columns are arranged. For a very large system of equations, we are interested in minimizing the amount of time taken to complete the evaluation of the task graph. For a sequential algorithm, this is equivalent to minimizing the number of operations needed in evaluating the task graph and also minimizing the number of fillins generated. Consider the matrix shown in Figure 5.1. A sequential algorithm will take 19 units of time (where a unit of time is the time taken for a single floating-point operation) to complete the evaluation of the task graph shown in Figure 5.2. The numbers in brackets inside nodes is the t_e of each node. Given a sufficient number of processors, a parallel algorithm will take 11 units of time to evaluate this graph (T_c for the graph is 11).

If the matrix is reordered to give the matrix shown in Figure 5.3, the sequential algorithm takes 15 units of time to complete the evaluation of the corresponding task graph

$$\begin{bmatrix} x & & & x & & \\ & x & & & x & \\ & & x & x & & \\ x & & x & x & & \theta \\ & x & & & x & x \\ & & x & \theta & x & x \end{bmatrix} \quad \theta \text{ is a fill-in}$$

Figure 5.1: Example matrix

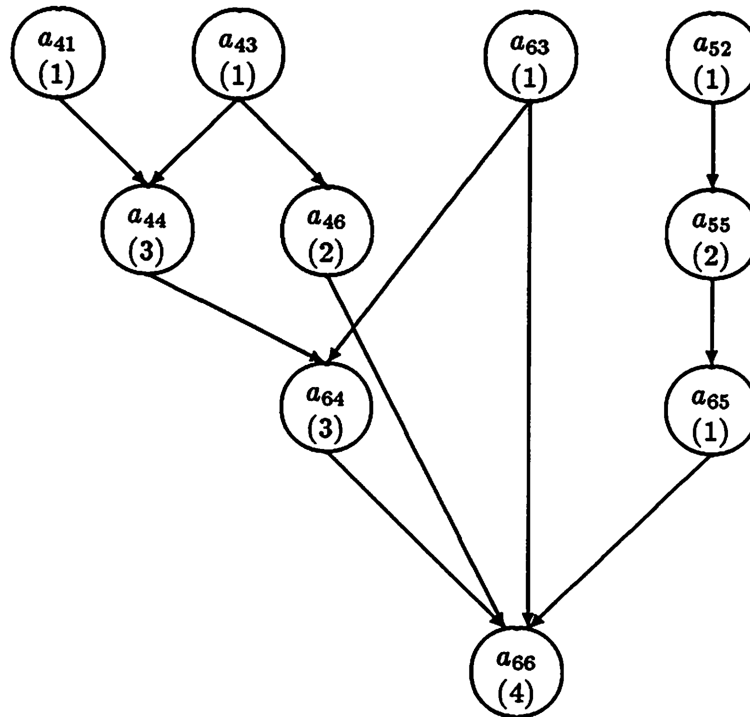

Figure 5.2: Task graph for matrix shown in Figure 5.1

shown in Figure 5.4. The number of operations needed to triangularize the graph have decreased, and the new ordering does not produce any fillins. The parallel algorithm now takes 15 units of time to complete the evaluation of the new task graph. It is evident that in trying to decrease the number of operations required for triangularization we have limited the amount of parallelism available. For a parallel algorithm, the objective of reordering to minimize fillins and reducing the number of operations may compete with the objective of reordering to minimize computation time. It is evident that in trying to minimize the fillins, there is a reduction in the degree of parallelism that might exist among operations, and consequently the parallel completion time goes up. On the other hand, if parallelism is maximized by generating more fillins, the completion time might increase due to the increased number of operations. Also, more fillins means more dependencies and therefore a larger communication overhead.

The problem of minimizing completion time can now be stated as follows. Given any matrix A, find an ordering of the rows and columns of A so that the completion time of the triangularization process using the medium grain parallelism approach is minimized.

$$\begin{bmatrix} x & x & & & & \\ x & x & x & & & \\ & x & x & x & & \\ & & x & x & x & \\ & & & x & x & x \\ & & & & x & x \end{bmatrix}$$

Figure 5.3: Reordered matrix

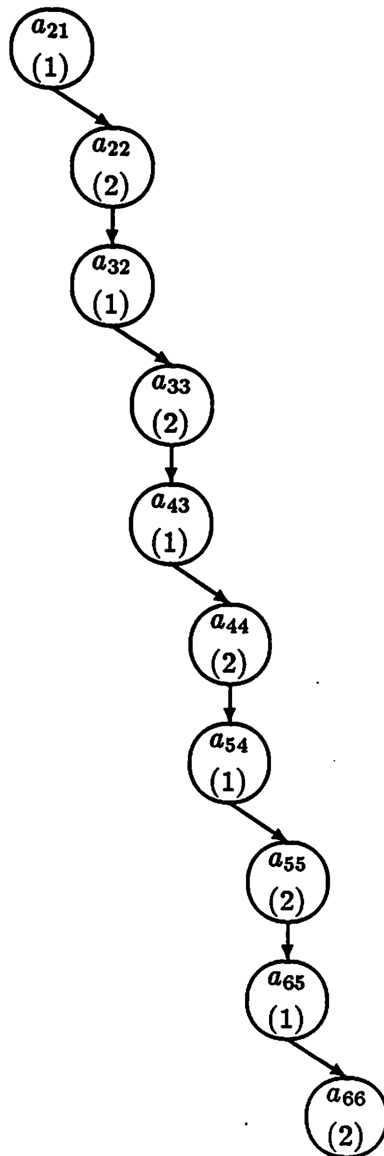

Figure 5.4: Task graph for matrix in Figure 5.3

$$\begin{bmatrix} x & x & x & \\ x & x & x & \\ & x & x & \\ x & & & x \end{bmatrix}$$

Figure 5.5: Example matrix #2

In other words, find an ordering of the rows and columns such that the critical path time T_c is minimized. This is a difficult optimization problem [17]. A heuristic algorithm based on local minimization of completion time is presented in the following sections.

5.1 Graph Model of Matrix

It is convenient to use a graph model of a matrix in the description of the algorithm. In this section the graph model is developed and its use is illustrated.

For any given matrix A of size $n \times n$ the associated graph G(A) is defined as follows. It is a bipartite graph consisting of n row nodes, one for each row, and n column nodes, one for each column. The row and column nodes are numbered according to the number of the row or column they represent. Edges are only allowed between row nodes and column nodes. An edge between row node i and column node j exists if and only if a_{ij} is a non-zero element in the matrix A. For the matrix shown in Figure 5.5 the associated graph is shown in Figure 5.6. All elements in a row are represented as edges from the corresponding row node, and all elements in a column are represented as edges from the corresponding column node. Two nodes are said to be connected if there exists an edge between them. A path in the graph is a sequence of alternating row and column nodes such that any two adjacent nodes in the sequence are connected.

Given a matrix A and its associated graph G(A), for any matrix B which is obtained by interchanging rows and column of A, G(B) can be easily obtained. If rows (columns) a and b in A are interchanged the new associated graph is obtained by renumbering row (column) node a to b and row (column) node b to a. If columns 1 and 4 are interchanged in the matrix of Figure 5.5 then the new associated graph is obtained by interchanging the numbers of the column nodes 1 and 4 in G(A). If now the rows 1 and 4 are interchanged, the new graph G(A) is obtained by interchanging the number of the row

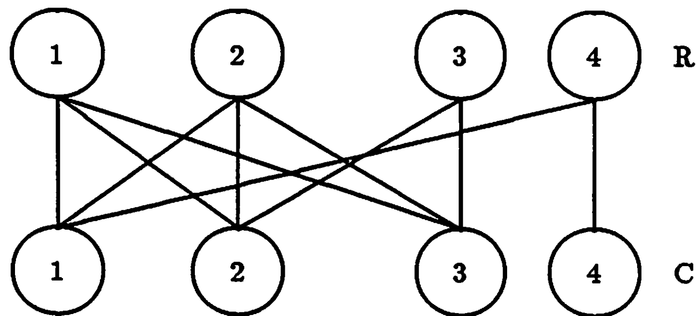

Figure 5.6: Associated graph for matrix shown in Figure 5.5

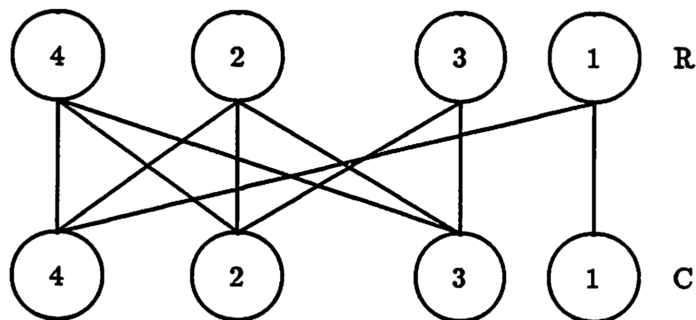

Figure 5.7: Graph for matrix after row and column interchange

nodes 1 and 4. The resulting graph is shown in Figure 5.7. Note that we have done pivot reordering, bringing a_{44} to a_{11}, and the resulting graph is obtained by renumbering a pair of row and column nodes.

This graph model can be used to interpret the process of Gaussian elimination and LU decomposition. Before giving an interpretation of Gaussian elimination and LU decomposition, we need an algorithm that helps us generate all the fillin elements. Since the fillins generated are the same for Gaussian elimination and LU decomposition, only one fillin generation algorithm is necessary.

5.1.1 Fillin Generation

The fillin generation algorithm is as follows. Suppose that the kth row is being eliminated. Form the set E_k given by :

$$E_k = \{ r : r \textit{ is a row node and } r \textit{ is connected to column node } k \textit{ and } r > k\}$$

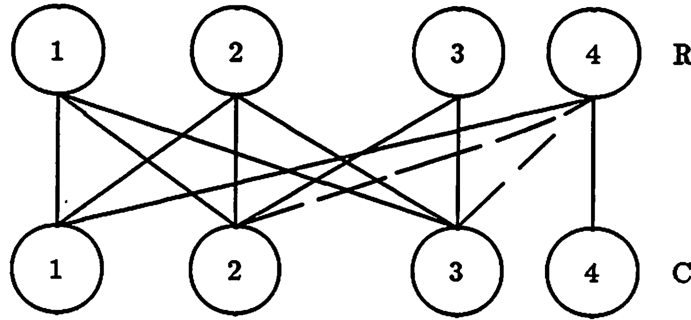

Figure 5.8: Graph for matrix after fillin generation for elimination of first row

Form the set F_k given by :

$$F_k = \{ c : c \text{ is a column node and } c \text{ is connected to row node } k \text{ and } c > k\}$$

Form the cartesian product of E_k and F_k which is defined as :

$$E_k \times F_k = \{(r,c) : r \in E_k \text{ and } c \in F_k\}$$

The ordered pairs $(r,c) \in E_k \times F_k$ are edges in the graph that represent elements that are affected by elimination of the kth pivot. All edges that are in $E_k \times F_k$ and not in the graph already, represent fillin elements. Fillin generation corresponds to adding these edges in the graph and marking them as fillin edges. For the matrix shown in Figure 5.5, considering the first row as the pivot row, we get $E_k = \{2,4\}$ and $F_k = \{2,3\}$. The fillin edges are (4,2) and (4,3). The resulting graph is shown in Figure 5.8 where the fillin edges are shown by broken lines.

We now give an interpretation of Gaussian elimination and LU decomposition for this graph.

5.1.2 Gaussian Elimination

At the kth stage of Gaussian elimination, element a_{kk} is considered as the pivot. There must exist an edge between row node k and column node k. If not, by a suitable set of row and column interchanges, an element can be brought into position (k,k). Now generate all the fillins due to the elimination of this pivot using the fillin generation algorithm. Mark row node k and column node k as eliminated. Also mark all edges connected to row node k and column node k as eliminated.

5.1.3 LU Decomposition

At first all the fillin elements are generated using the fillin generation algorithm for each pivot. At the kth stage of LU decomposition, mark row node k and column node k as eliminated and also mark all edges connected to them as eliminated.

5.2 Reordering Algorithm

Given a matrix, our objective is to reorder the matrix in such a manner that the completion time is minimized. To achieve this objective, the amount of parallelism have to be increased, inter-processor communication and also the amount of computation has to be reduced. As has been shown earlier, the goals of increasing parallelism and reducing the amount of computation are conflicting. It seems that to increase the amount of parallelism, the number of fillins have to be large. However, with more fillins there are more edges in the task graph, and higher overhead in execution of tasks. Thus the goals of reducing inter-processor communication and increasing parallelism may conflict.

If a sparse matrix can be reordered into an upper or a lower triangular form, then there are no intertask dependencies. More precisely, there is nothing to be done in the triangularization process in such a case. However, most practical matrices cannot be reordered to an upper or lower triangular form. If a matrix is close to this form, the inter-task dependencies are fewer and the amount of computation is smaller. Our objective would be to find an ordering that takes a matrix closer to an upper or lower triangular form from its given structure. This is our reordering heuristic.

The algorithm starts by making a choice for the first row of a matrix. The row selected to be the first row has to have an element in position $(1,1)$. From all rows having elements at position $(1,1)$, the row that has the minimum number of elements in it is chosen. This ensures that the fillins generated by the first row will be a minimum. If there is a tie, the row with the elements furthest away form the diagonal is selected. The reason for this is as follows. Any element after the diagonal will generate primary fillins in the column in which it is present. The primary fillins can produce more fillins. These are called secondary fillins. In order to keep the number of fillins down, we try to generate fillins as late as possible in the elimination process. Selecting a row with elements furthest from the diagonal helps to do this. Once a choice is made, the selected row is swapped with Row 1,

which is equivalent to interchanging the numbers of the row nodes in the associated graph. All edges connected to row node 1 and column node 1 are then marked eliminated.

For each subsequent row k, a choice is made from among the remaining set of rows the row that helps keep the matrix closest to the upper triangular form. The selected row is then brought into position for row k. First of all a set of rows called the qualifying set is formed. For any row k, the qualifying set Q_k is the set of all rows below it having an element at (k, k). In terms of G(A), Q_k is given as :

$$Q_k = \{ r : r \text{ is a row node connected to column node } k \text{ and } r \geq k \}$$

From this set of rows, the row that takes the matrix closest to the upper triangular form is chosen. The following heuristics are applied.

- If a row in the qualifying set with no elements before position (k, k) is found, then it is the best choice.
- If there is more than one such row, the row with the least number of elements after the diagonal is chosen.
- If there is still a tie, the row whose elements after the diagonal are farthest from the diagonal is chosen.

In terms of G(A), this is equivalent to finding a row node in Q_k with the smallest number of edges marked eliminated. The best choice is one with no eliminated edges connected to it. If more than one such node exists, then the one with lesser degree is chosen. If there is still a tie, the row node connected to higher numbered column nodes is chosen.

If there is no row with no elements before (k, k), then the row that minimizes the cost function C_k given below is chosen.

$$C_k = W_a N_F + W_b N_b$$

In the formula above, N_F is the number of fillin elements generated by choosing that row as the pivot, and N_b is the number of elements before the diagonal, while W_a and W_b are given weights. After each pivot row is chosen and the corresponding row node numbers are interchanged, the row and column nodes and all edges connected to them are marked eliminated.

$$\begin{bmatrix} & \cdot & & & & \\ & & \cdot & & & \\ Rowb & & x & x & & \\ Rowc & & & x & x & \\ Rowk & & x & & & \\ & & & & & \cdot \\ & & & & & & \cdot \end{bmatrix}$$

Figure 5.9: Example matrix #3

Consider the matrix shown in Figure 5.5. For row 1, the best choice is row 4. If rows 1 and 4 are swapped, the qualifying set for row 4 becomes empty. This is because the current row 1 does not have any element beyond column 3, and should only be swapped with rows with number 3 or less. Let C_{max} be the column number of largest column node connected to row node k. Then the new definition of the qualifying set Q_k is :

$$Q_k \;=\; \{\; r \;:\; r \textit{ is a row node connected to column node } k \textit{ and } k \leq r \leq C_{max} \;\}$$

Once the qualifying set is found, the heuristics mentioned above may be applied to choose the best candidate row. Before the heuristics can be applied, we have to make sure that there is no row in Q_k that does not have any element after (k, k). In such a case, even if that row has elements before the diagonal, it has to be chosen, because it cannot be used for any row below row k. In terms of the graph G(A), the algorithm ensures that if there is a row node in Q_k with only one non-eliminated edge then that node is chosen. This is called the preemptive rule, which has to be applied before the choice heuristics. Note that if the original matrix had all non-zero diagonal elements then only one such row can exist in the qualifying set. Moreover, if such a row exists, it will be the current row k.

Suppose for a certain row p, row k is the best choice. Then a_{kp} is non-zero but we don't know whether a_{pk} is non-zero or not. If a_{pk} is zero, then after the swap, row p is left with a zero diagonal element. If there are no rows below row k with an element in column k, then Q_k will be empty. This happens because all rows with elements on column k have been chosen for rows above. The only way to continue with the algorithm is to swap the current row k with some row above (or make a composite set of row swaps) so that an element is brought in at position (k, k) without introducing a zero element at any of the diagonals above. If the original matrix had all non-zero diagonal elements, then it is easy

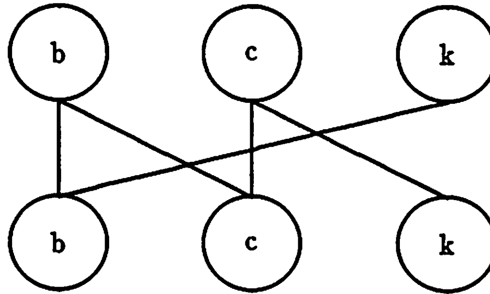

Figure 5.10: Partial Graph of matrix shown in Figure 5.9

to see that the current row will have elements before position (k, k). Say that current row has an element (k, a) where $a < k$. Then if row a has element (a, k), then we can bring in a non-zero element at position (k, k) without introducing any zero's in other diagonal positions by swapping rows a and k. In terms of the graph G(A), this is equivalent to doing the following : for every column node a connected to row node k such that $a < k$, if row node a is connected to column node k, then swap row nodes a and k. This might not always be possible. Consider part of a matrix shown in Figure 5.9. Obviously the condition stated above is not satisfied. However, the situation can still be remedied by swapping rows c and b and then swap rows b and k. In terms of the graph G(A), this is equivalent to finding a path from column node k to row node k such that the following constraints are satisfied :

- If a column node is visited, the corresponding row node is visited
- Only eliminated edges are traversed

If the set of row nodes in the path is maintained in the order visited, then swapping the rows in that order will bring in a non-zero element at (k, k). It can be easily shown that such a path will always be found if the original matrix had non-zero diagonal elements. This is illustrated with the example of Figure 5.9. The graph for the corresponding matrix is shown in Figure 5.10. The path from column node k to the row node k can be easily seen, and the order in which nodes are visited is $\{c, b, k\}$. By swapping nodes c and b and then b and k, the problem is solved. This procedure also works for the first case where only a simple swap was necessary.

Whenever Q_k is empty, the set of row swaps takes the matrix away from the upper diagonal form. To ensure that this is happens infrequently, a range limiting heuristic is applied. This heuristic ensures that for a particular row k under consideration, only those

rows which are within a certain range R of row k should be considered in the qualifying set. Define R_{max} as :

$$R_{max} = \min \{C_{max}, k + R\}$$

Then the definition of the qualifying set is :

$$Q_k = \{ r : r \text{ is a row node connected to column node } k \text{ and } k \leq r \leq R_{max} \}$$

The main reordering algorithm is shown in Figure 5.11. The routine **get_qualifying_set** computes the qualifying set Q_k (as defined above) for a given row k. If the qualifying set is non-empty, then the choosing heuristics can be used to find the best candidate row. The choosing heuristics are implemented in the routine **get_choice.** If the qualifying set is empty, then the routine **correct_problem** is called to find a set of row swaps to take care of the situation. After the rows have been swapped, the appropriate fillins have to be generated.

The algorithm for choosing the best candidate row is given in Figure 5.12. At first the preemptive rule is applied to Q_k. If there is an element that preempts the application of the heuristics, then it is chosen. Otherwise, the heuristics are applied in choosing the best row. The routine **get_best** returns all rows in Q_k that do not have an eliminated edge in G(A). If this set is not null, then the routine **choose_best_row** applies the other two heuristics to choose the best row. If there are no rows without elements before (k, k) then the cost function is used to determine the best choice. By varying the values of the weights W_a and W_b, we can reorder to minimize fillins or minimize the number of elements before the diagonal.

The algorithm used in the routine **correct_problem** is given in Figure 5.13. All old fillin edges are removed from the graph, as some eliminated rows are going to be interchanged. Then the routine **find_path** finds the shortest path from the column node k to the row node k satisfying the required conditions. Once the path is found the routine **swap_rows** performs the set of swaps.

The problem of finding the shortest path from the row node k to the column node k satisfying the constraints can be reformulated to make it easier. If row nodes and column nodes are thought of as one node for all nodes less than k, then the problem of finding the shortest path satisfying the constraints reduces to finding the shortest path in the modified graph between row node k and column node k. This can be easily done using Dijkstra's

```
reorder_matrix(matrix)
{
   /*Input : Matrix to be reordered
   Output : Reordered matrix */

   size = size_of(matrix);
   range = 0.2 * size ;
   for (k=1; k<size ; k++){
      Q_k = get_qualifying_set(k);
      if (Q_k not empty){
         best_choice = get_choice(Q_k);
      }
      else{
         correct_problem(k);
         best_choice = 0;
      }
      if (first_choice > 0)
         swap_rows(k, best_choice);
      generate_fillins(k);
   }
   return (matrix);
}
```

Figure 5.11: Reordering Algorithm

```
get_choice(Q_k)
{
   /*Input : Q_k for a particular row k
   Output : The best choice for row k */

   /* Apply the pre-emptive rule first */
   if ( row k is in Q_k and does not have element beyond (k, k))
      return (k);
   else {
      choice_set = get_best(Q_k);
      if (choice_set is not null){
         /* We choose the best amongst these */
         choice = choose_best_row(Q_k);
         return (choice);
      }
      else{
         /* No rows with no elements before (k, k) */
         foreach (row j in Q_k) {
            C(j) = W_a N_F + W_b N_b;
         }
         choice = j* such that C(j*) = min_j C(j);
         return (choice);
      }
   }
}
```

Figure 5.12: Choosing Algorithm

```
correct_problem(k)
{
   /* Input : The number of the row for which Q_k is empty

   remove_all_old_fillins();
   path = find_path();
   swap_rows(path);
   generate_new_fillins();
   return;
   }
}
```

Figure 5.13: Algorithm for correcting problem

shortest-path algorithm [16].

As is evident, this algorithm transforms any matrix as close as possible to an upper triangular form. If a matrix is already close to a lower triangular form, it may be easier and better to transform it into a lower triangular form. To transform a matrix to a lower triangular form, the algorithm can be applied to the transpose of the matrix. The reordering algorithm is applied to both the original matrix and its transpose. The one that gives a smaller computation time is selected.

5.3 Results

Table 5.1 gives the results of reordering on a set of test matrices. The comparison is made on the basis of T_c obtained from the scheduling algorithm. We see that for most of the examples considered, the heuristics were able to find an ordering that had smaller T_c.

In the table, N_F is the number of fillins, N_E is the number of elements and N_L is the number of levels in the task graph. P.I. indicates percentage improvement. We note that in one case the algorithm produced no improvement. This is probably because the matrix cannot be reordered to give a better matrix. We note that in some cases the number

Circuit	Size	Before Reorder				After Reorder				P.I.	Time taken
		N_F	N_E	N_L	T_c	N_F	N_E	N_L	T_c		
IYOUNG	76	800	1222	34	1695	230	652	19	588	65	1.21
PSPI	80	409	687	24	690	376	654	24	675	2.17	1.50
NAY	166	519	1202	25	987	807	1490	31	882	11	2.28
A2D1	218	738	1630	16	573	743	1635	16	573	0	5.44
BEN2K	402	205	2205	11	1842	194	2194	11	1839	0.16	20.76
ADDER	450	2316	4360	43	3468	1829	3873	38	1311	62	5.82
EPROM	687	1122	4199	25	1923	1234	4311	27	1911	0.62	50.31
XX1	1013	3644	7145	200	2754	4947	8448	205	2298	17	26.68

Table 5.1: Effect of reordering on T_c

Circuit	T_{sol} Before Reorder	T_{sol} After Reorder	P.I.
IYOUNG	0.075	0.018	75
PSPI	0.027	0.020	25
NAY	0.028	0.027	3
A2D1	0.032	0.027	15
BEN2K	0.047	0.037	21
ADDER	0.163	0.082	50
EPROM	0.077	0.076	1
XX1	0.192	0.106	45

Table 5.2: Effect of reordering on actual execution time

of fillins went up but still the value of T_c went down, indicating that increasing parallelism may require more fillins.

Table 5.2 gives us the actual times of evaluation of the task graphs of each of the matrices, using eleven processors. We compare the percentage improvement values in the above table to those from table 5.1 to arrive at conclusions about the amount of sychronization overhead involved. We see that in most cases, the overhead introduced by more fillins is very small. In fact we get better percentage improvement in speedup than what the schedule forecasts. This is due to the smaller synchronization overhead incurred after reordering.

5.4 Conclusion

An algorithm for reordering matrices to minimize the completion time was given. It was observed that the goals of minimizing completion time and minimizing fillins are conflicting. For some cases, to increase the amount of parallelism available, the number of fillins have to be increased.

Results indicate that our algorithm does find an ordering which requires smaller amount of time for parallel completion of evaluation. In most cases, it is not necessary to reorder a matrix once it has been reordered. This reordering can be done once at the beginning of computation, and can be used for the rest of the simulation. The cost of reordering, like that of scheduling, is spread out over a number of solutions.

Chapter 6

CONCLUSION

We have discussed three aspects of parallel solution of a sparse system of linear equations, namely, scheduling, interprocessor communication and reordering to minimize the computation time. Finding a schedule that minimizes the interprocessor communication and the total computation time is known to be NP-complete. Our conjecture is that the problem of reordering is NP-complete too. Some of the heuristics used by other researchers in solving the above problems were outlined.

Three levels of task granularity for the parallel solution of a sparse system were examined. Though fine grain parallelism exploits the maximum amount of parallelism it has high scheduling and synchronization overhead. Large grain parallelism, though relatively free from such problems, exploits very little parallelism. Medium grain parallelism is seen to be a good compromise between the two extremes. The computation tree model introduced by Huang and Wing [26] to represent tasks to be done in parallel and their inter-dependencies was adopted for our work. Tasks were represented as computation trees. An implementation of the medium grain approach was described, and it was found that, for most large systems, a single processor solution was faster than a sequential algorithm.

Scheduling is an important aspect of any parallel algorithm for solving a system of linear equations. Since the problem of finding an optimum schedule is NP-complete, only heuristic solutions can be attempted. We argued against the use of dynamic scheduling techniques and showed how static schedules can produce faster run times. The scheduling algorithm developed tries to assign jobs to the processors so that completion time is minimized. For the medium grain approach, scheduling is more difficult because of unequal task size. However, in most cases, the scheduling algorithm was able to find near minimum

time schedules with a smaller number of processors than the maximum required. For many implementations of the fine grain approach, interprocessor communication time dominated the total solution time. A different synchronization algorithm was proposed and was shown to be better than lock-based barrier synchronization. The synchronization overhead was thus significantly reduced. In most cases we were able to obtain speedups close to the maximum possible. Comparison of the approach developed in this report work with that of Jacob [27] showed that the algorithms in this report performed better in terms of speedup achievable.

The issue of reordering the rows and columns of the matrix for minimizing the computation time was also addressed. We saw that the goals of minimizing completion time and minimizing fillins are conflicting. Some heuristics were developed for solving the reordering problem. An algorithm based on these heuristics was presented. For some cases, it was observed that in order to minimize the computation time the number of fillins had to be increased. Results indicate that our algorithm does find an ordering which takes minimal time for solution. In most cases, it is not necessary to reorder a matrix once it has been reordered. This reordering can be done once at the beginning of computation, and can be used for the rest of the simulation. The cost of reordering, like that of scheduling is spread out over a number of solutions.

For large examples, the number of processors required to complete the execution in the minimum possible time is very large. Most multiprocessors today do not support a very large number of processors. If the matrix can be reordered so that the breadth of the task graph is equal to the number of processors, scheduling becomes a trivial task. This problem is slightly different from the reordering problem treated here, and needs to be solved. Though static scheduling gives better performance than dynamic scheduling we have seen that synchronization overheads are not insignificant. Reducing synchronization overhead by reordering or using specialized hardware has to be investigated. We have used the standard Doolittle's LU decomposition algorithm as the algorithm for this research. There is a possibility that different solution algorithms may show varying performance. It might be possible to find a solution algorithm that for all kinds of matrices produces the best average result. Identification of such an algorithm is very important for a viable parallel circuit simulator. Digital circuits show a significant amount of temporal latency. Thus all entries in the Jacobian do not change every time it is loaded. An event-driven solution algorithm would detect all the changed values in the new Jacobian and recompute only

those elements whose values are affected by this change. Since all LU factors don't have to be computed, the amount of time required for solution may decrease. However, there might be some overhead involved in this method. The viability of an event-driven sparse system solver also has to be investigated. Lastly all the algorithms need to be incorporated in a working circuit simulator like SPICE3.

Bibliography

[1] *Balance 8000/21000 Parallel Programming.* Sequent Computer Systems, Inc., November 1986.

[2] Thomas Adams, K. Chandy, and J.R. Dickson. A Comparison of List Schedules for Parallel Processing Systems. *Communications of the ACM*, 17(12):685–690, December 1974.

[3] A.V. Aho and J.D. Ullman. *Design and Analysis of Computer Algorithms.* McGraw Hill, 1980.

[4] Christopher P. Arnold, Michael I. Parr, and Michael B. Dewe. An Efficient Parallel Algorithm for the Solution of Large Sparse Linear Matrix Equations. *IEEE Transactions on Computers*, C-32(3):265–272, March 1983.

[5] R. Betancourt. Efficient Parallel Processing Technique for Inverting Matrices with Random Sparsity. *IEE Proceedings*, 133(4):235–240, July 1986.

[6] Gabriel Bischoff and Steven Greenberg. CAYENNE : A Parallel Implementation of the Circuit Simulator Spice. In *Proceedings of the ICCAD*, 1986.

[7] B.W.Kernighan and S.Lin. An Efficient Heuristic Procedure for Partitioning Graphs. *The Bell System Technical Journal*, 49:291–307, 1970.

[8] Andrea Casotto. Parallel Algorithm for Sparse Matrices : LU Decomposition. 1985. Class Project at UCB.

[9] Mi-Chang Chang and I.N. Hajj. iPRIDE : A Parallel Integrated Circuit Simulator Using Direct Method. In *Proceedings of the ICCAD*, pages 304–307, 1988.

[10] Chien-Chih Chen and Yu-Hen Hu. Parallel LU Factorization for Circuit Simulation on MIMD Computer. In *Proceedings of ICCAD*, 1988.

[11] Chien-Chih Chen and Yu-Hen Hu. A Practical Scheduling Algorithm for Parallel LU Factorization of Circuit Simulation. In *Proceedings of ICCAD*, 1988.

[12] Paul Cox, Richard Burch, and Berton Epler. Circuit Partitioning for Parallel Processing. In *Proceeding of the ICCAD*, 1986.

[13] Paul Cox, Richard Burch, Dale Hocevar, and Ping Yang. SUPPLE : Simulator Utilizing Parallel Processing and Latency Exploitation. In *Proceeding of the ICCAD*, pages 368–371, 1987.

[14] Eliezer Dekel and Sartaj Sahni. Binary Trees and Parallel Scheduling Algorithms. *IEEE Transactions on Computers*, C-32(3):307–315, March 1983.

[15] Eliezer Dekel and Sartaj Sahni. Parallel Scheduling Algorithms. *Operations Research*, 31(1):24–49, January-February 1983.

[16] E.W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959.

[17] D.J.Rose and R.E.Tarjan. Algorithmic aspects of vertex elimination. In *Proceedings of the 7th Annual Symposium on the Theory of Computing*, 1975.

[18] E.A.Lee and D.G.Messerschmitt. Static Scheduling of Synchronous Data Flow Programs for Digital Signal Processing. *IEEE Transactions on Computers*, C-36(1):24–35, January 1987.

[19] E.CutHill and J.McKee. Reducing the bandwidth of sparse symmetric matrices. In *24th National Conference of the ACM*, 1969.

[20] Alan George and Joseph W-H Liu. *Computer Solution of Large Sparse Positive Definite System*. Prentice Hall, Inc, 1981.

[21] R.S. Gyurcsik. *An Attached Processor for MOS-transistor Model Evaluation*. PhD thesis, University of California at Berkeley, 1986. Memorandum No. UCB/ERL M86/82.

[22] R.S. Gyurcsik. BIASC : A Circuit Simulation Program for the IBM PC. Wescon/85 Professional Program Session Record 32/4, San Francisco, CA, Nov. 1985.

[23] Gary Hachtel, Robert K. Brayton, and Fred G. Gustavson. The Sparse Tableau Approach to Network Analysis and Design. *IEEE Transactions on Circuit Theory*, CT-18(1):101–113, January 1971.

[24] H.M.Markowitz. The Elimination form of the Inverse and its Application to Linear Programming. *Management Science*, 3:255–269, April 1957.

[25] T.C. Hu. Parallel Sequencing and Assembly Line Problems. *Operations Research*, 9:841–848, 1961.

[26] John W. Huang and Omar Wing. Optimum Parallel Triangulation of a Sparse Matrix. *IEEE Transactions on Circuits and Systems*, CAS-26(9):726–732, September 1979.

[27] George K. Jacob. *Direct Methods in Circuit Simulation Using Multiprocessors*. PhD thesis, University of California at Berkeley, 1987. Memorandum No. UCB/ERL M87/67.

[28] J.A.G.Jess and H.G.M.Kees. A data structure for parallel LU decomposition. *IEEE Transaction on Computers*, C-31:231–239, 1982.

[29] A.R. Newton J.L. Burns and D.O. Pederson. Active Device Table Look-up Models for Circuit Simulation. In *IEEE International Symposium on Circuits and Systems*, pages 250–253, Newport Beach, CA, May 1983.

[30] H.F. Ko. *A Special Purpose Architecture and Parallel Algorithms on a Multiprocessor System for the Solution of Large Scale Linear Systems of Equations*. PhD thesis, University of California at Berkeley, 1986. ERL memo.

[31] Walter H. Kohler. A Preliminary Evaluation of the Critical Path Method for Scheduling Tasks on Multiprocessor Systems. *IEEE Transactions on Computers*, :1235–1238, December 1975.

[32] E. Lelarasmee. *The Waveform Relaxation Method for Time Domain Analysis of Large Scale Integrated Circuits : Theory and Applications*. PhD thesis, University of California at Berkeley, 1982. Memo Number ERL-M82/40.

[33] Joseph W. H. Liu. *Reordering Sparse Matrices for Parallel Elimination*. Technical Report CS-87-01, York University, Department of Computer Science, January 1987.

[34] J.W.H. Liu and A. Mirazian. *A linear reordering algorithm for parallel pivoting of chordal graphs.* Technical Report, Department of Computer Science, York University, 1987.

[35] R.E. Lord, J.S. Kowalik, and S.P. Kumar. Solving Linear Algebraic Equations on an MIMD Computer. *Journal of the ACM*, 30(1):103–117, January 1983.

[36] Robert Lucas, Tom Blank, and Jerome Tiemann. A Parallel Solution Method for Large Sparse System of Equations. In *Proceeding of the ICCAD*, 1986.

[37] M.Consard, J.M.Muller, Y.Robert, and D. Trystram. Communication costs versus computation costs in parallel Gaussian elimination. In *Parallel Algorithm and Architectures*, pages 19–29, Elsevier Science Publishers B.V.(North Holland), 1986.

[38] M.R.Garey and D.S.Johnson. *Computers and Intractability — A Guide to the Theory of NP-completeness.* Freeman : San Francisco, 1979. Problem SS9.

[39] L. W. Nagel. *SPICE2 : A Computer Program to Simulate Semiconductor Circuits.* PhD thesis, University of California at Berkeley, 1975. Memo Number ERL-M520.

[40] A.R. Newton. *The Simulation of Large Scale Integrated Circuits.* PhD thesis, University of California at Berkeley, 1978. Memo Number ERL-M78/52.

[41] T. Quarles. Spice3 User Guide. January 1988.

[42] Gunter Rote. A Parallel Scheduling Algorithm for Minimizing the Number of Unscheduled Jobs. In *Parallel Algorithm and Architectures*, pages 99–108, Elsevier Science Publishers B.V.(North Holland), 1986.

[43] Yousef Saad. Gaussian Elimination on Hypercubes. In *Parallel Algorithm and Architectures*, pages 5–17, Elsevier Science Publishers B.V.(North Holland), 1986.

[44] P. Sadayappan and V.Visvanathan. *Parallelization and Performance Evaluation of Circuit Simulation on a Shared Memory Multiprocessor.* Technical Report, AT&T Bell Laboratories, 1988.

[45] K. Sakallah and S.W. Director. An Activity Directed Circuit Simulation Algorithm. In *IEEE International Conference on Circuits and Computers*, pages 1032–1035, October 1980.

[46] R.A. Saleh. *Nonlinear Relaxation Algorithms for Circuit Simulation.* PhD thesis, University of California at Berkeley, 1987. Memo Number ERL-M87/21.

[47] A. Sangiovanni-Vincentelli. *Computer Design Aids for VLSI Circuits*, chapter Circuit Simulation, pages 19–113. Groningen, The Netherlands : Sijthoff and Noordhoff, 1981.

[48] A. Sangiovanni-Vincentelli, Li-Kuan Chen, and L.O. Chua. An Efficient Heuristic Cluster Algorithm for Tearing Large Scale Networks. *IEEE Transactions on Circuits and Systems*, CAS-24(12), December 1977.

[49] A. Sangiovanni-Vincentelli, L.K. Chen, and L.O. Chua. A New Tearing Approach — Node-Tearing Nodal Analysis. In *Proceedings of IEEE International Symposium on Circuits and Systems*, pages 143–147, 1977.

[50] T. Shima, T. Sugawara, S. Moriyama, and H. Yamada. Three-dimensional Table Look-up MOSFET Model for Precise Circuit Simulation. *IEEE Journal of Solid-State Circuits*, SC-17:449–454, June 1982.

[51] Mandayam A. Srinivas. Optimal Parallel Scheduling of Gaussian Elimination DAG's. *IEEE Transactions on Computers*, C-32(12):1109–1117, December 1983.

[52] V. Strassen. Gaussian Elimination is not optimal. *Numerische Mathematik*, 13:354–356, 1969.

[53] R. Thomas. *Using the Butterfly to Solve Simultaneous Linear Equations.* Technical Report, BBN Labs Memorandum, Cambridge, MA, March 1985.

[54] W.F. Tinney. *Comments on using sparsity techniques for power system problems.* Technical Report, *Sparse Matrix Proceedings*, IBM Research Report, 1969.

[55] A. Vladimirescu. *LSI Circuit Simulation on Vector Computers.* PhD thesis, University of California at Berkeley, 1982. Memorandum No. UCB/ERL M82/75.

[56] W.T. Weeks et al. Algorithms for ASTAP — A Network Analysis Program. *IEEE Transactions on Circuit Theory*, CT-20:628–634, November 1973.

[57] J.K. White. *The Multirate Integration Properties of Waveform Relaxation, with Applications to Circuit Simulation and Parallel Computation.* PhD thesis, University of California at Berkeley, 1985. Memo Number ERL-M85/90.

[58] Omar Wing and John W. Huang. A Computation Model of Parallel Solution of Linear Equations. *IEEE Transaction on Computers*, C-29(7):632–638, July 1980.

[59] F. Yamamoto and S. Takahashi. Vectorized LU Decomposition Algorithms for Large Scale Circuit Simulation. *IEEE Transactions on Computer-Aided Design*, CAD-4(3):232–239, July 1985.

[60] P. Yang. *An Investigation of Ordering, Tearing and Latency Algorithms for the Time-Domain Simulation of Large Circuits.* Technical Report, *Report R-891, Coordinated Science Lab.*, University of Illinois, Urbana, August 1969.

[61] Ping Yang. An Efficient Ordering Algorithm in the Modified Nodal Approach for VLSI Circuit Simulations. In *Proceeding of the ICCAD*, 1985.

[62] Chen-Ping Yuan, Robert Lucas, Philip Chan, and Robert Dutton. Parallel Electronic Circuit Simulation on the iPSC System. In *IEEE Custom Integrated Circuits Conference*, 1988.

[63] Earl Edward Zmijewski. *Sparse Cholesky Factorization on a Multiprocessor.* PhD thesis, Cornell University, 1987.