PARALLEL PROCESSING AND VLSI DESIGN:
SOLVING LARGE-SCALE LINEAR SYSTEMS

A Thesis Presented to
The Faculty of the College of Engineering and Technology
Ohio University

In Partial Fulfillment
of the Requirements for the Degree of
Master of Science

by

Eric M. Schwarz
November 1984

Ohio University
Library
Abstract

This thesis discusses techniques of solving large-scale linear systems. In particular, parallel processing methods with a VLSI design of a machine that can do this are discussed. Two architectures described in this paper are original. Also, techniques for evaluating architectures and algorithms have been improved, and with this evaluation technique, a comprehensive review of architectures for all types of systems of linear equations is given.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abstract</td>
<td>i</td>
</tr>
<tr>
<td>Table of Contents</td>
<td>ii-iii</td>
</tr>
<tr>
<td>Acknowledgements</td>
<td>iv</td>
</tr>
<tr>
<td>List of Figures</td>
<td>v</td>
</tr>
<tr>
<td>List of Tables</td>
<td>vi</td>
</tr>
<tr>
<td>Nomenclature</td>
<td>vii</td>
</tr>
<tr>
<td>I. Introduction</td>
<td>1-2</td>
</tr>
<tr>
<td>1.1 State-of-the-Art Techniques</td>
<td>2-3</td>
</tr>
<tr>
<td>1.2 Conventional Techniques</td>
<td>4</td>
</tr>
<tr>
<td>1.3 Why Parallel?</td>
<td>4</td>
</tr>
<tr>
<td>1.4 Possibilities of Asynchronousity</td>
<td>5</td>
</tr>
<tr>
<td>1.5 Direct Versus Indirect Methods</td>
<td>5-6</td>
</tr>
<tr>
<td>1.6 General System</td>
<td>6</td>
</tr>
<tr>
<td>II. LU Decomposition and Gauss Elimination</td>
<td>7-8</td>
</tr>
<tr>
<td>2.1 Gauss Elimination</td>
<td>8-9</td>
</tr>
<tr>
<td>2.2 Variations of LU Decomposition</td>
<td>9</td>
</tr>
<tr>
<td>2.3 Algorithm</td>
<td>10-12</td>
</tr>
<tr>
<td>2.4 Architecture</td>
<td>12-15</td>
</tr>
<tr>
<td>2.5 Performance Evaluation</td>
<td>15-18</td>
</tr>
<tr>
<td>III. QR Method</td>
<td>19</td>
</tr>
<tr>
<td>3.1 Algorithm</td>
<td>19-20</td>
</tr>
<tr>
<td>3.2 Overdetermined Systems (Least Squares Soln.)</td>
<td>21</td>
</tr>
<tr>
<td>3.3 Given Rotations vs. Householders Transformations</td>
<td>21-23</td>
</tr>
<tr>
<td>3.4 Comparison of Two Parallel Architectures</td>
<td>23-27</td>
</tr>
</tbody>
</table>
3.5 Underdetermined Systems........................... 27

IV. Parallel Asynchronous Jacobi Method....................... 28
   4.1 Algorithm........................................... 29-32
   4.2 Simulation and Example.............................. 32-40
   4.3 Architecture....................................... 40-46
   4.4 Similar Methods..................................... 46
       4.4.1 Parallel Conjugate Gradient Method......... 47-48
       4.4.2 JOR Method.................................... 48-49
   4.5 Performance Evaluation............................. 50-55

V. Multiplication - Array Multiplication...................... 56
   5.1 Binary Tree......................................... 57-59
   5.2 CSA................................................ 59-61
   5.3 3-D CSA............................................ 61-65
   5.4 Performance........................................ 65-66

VI. Conclusion............................................... 67
   6.1 Decision of System Type............................. 67
   6.2 Decision of Speed vs. Hardware....................... 68
   6.3 Decision of How Much Adaptability vs. Speed....... 68-69

References................................................... 70-71

Appendix A Parallel Asynchronous Jacobi Simulation Program........................ Al-All

Appendix B Parallel Gauss Elimination Simulation Program.. Bl-B5

Appendix C Parallel Conjugate Gradient Simulation Program.C1-C6
Acknowledgements

I would like to thank Dr. Janusz Starzyk for his help, guidance and encouragement. I would also like to thank my father for his help in editing this paper.
List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fig. 2.1</td>
<td>Architecture for L-U Decomposition</td>
<td>14</td>
</tr>
<tr>
<td>Fig. 3.1</td>
<td>Computational Array for Given's Rotations</td>
<td>23</td>
</tr>
<tr>
<td>Fig. 4.1</td>
<td>Parallel Asynchronous Jacobi Method Pictorial</td>
<td>30</td>
</tr>
<tr>
<td>Fig. 4.2</td>
<td>Sequential Jacobi Method Algorithm and Pictorial</td>
<td>31</td>
</tr>
<tr>
<td>Fig. 4.3</td>
<td>Gauss-Seidel's Method Algorithm and Pictorial</td>
<td>32</td>
</tr>
<tr>
<td>Fig. 4.4</td>
<td>Timing of Iterates for Parallel Asynchronous Jacobi Method on Example Problem 1</td>
<td>34</td>
</tr>
<tr>
<td>Fig. 4.5</td>
<td>Hierarchy of the Simulation Program for the Parallel Asynchronous Jacobi Method</td>
<td>36</td>
</tr>
<tr>
<td>Fig. 4.6</td>
<td>Example Problem 1: Unsymmetric, Unstructured, Sparse System</td>
<td>38</td>
</tr>
<tr>
<td>Fig. 4.7</td>
<td>Row i Processor for the Parallel Asynchronous Jacobi Architecture</td>
<td>42</td>
</tr>
<tr>
<td>Fig. 4.8</td>
<td>Example Problem 2</td>
<td>50</td>
</tr>
<tr>
<td>Fig. 4.9</td>
<td>Example Problem 3</td>
<td>51</td>
</tr>
<tr>
<td>Fig. 4.10</td>
<td>Passive Circuit Used in Example 4</td>
<td>53</td>
</tr>
<tr>
<td>Fig. 4.11</td>
<td>Matrix Representation of Example 4</td>
<td>54</td>
</tr>
<tr>
<td>Fig. 5.1</td>
<td>Array Multiplication</td>
<td>57</td>
</tr>
<tr>
<td>Fig. 5.2</td>
<td>Binary Tree of Parallel Adders</td>
<td>57</td>
</tr>
<tr>
<td>Fig. 5.3</td>
<td>Binary Tree Multiplier Unit</td>
<td>59</td>
</tr>
<tr>
<td>Fig. 5.4</td>
<td>Addition of Product Terms by the Use of Carry-Save Adders</td>
<td>60</td>
</tr>
<tr>
<td>Fig. 5.5</td>
<td>Three Adders: a)Binary Tree Adder, b)CSA, and c)3-D CSA</td>
<td>61</td>
</tr>
<tr>
<td>Fig. 5.6</td>
<td>Example of How Carry Bits and Sum Bits Are Determined in 3-D CSA</td>
<td>62</td>
</tr>
<tr>
<td>Fig. 5.7</td>
<td>15 Term Adder Used In an Array Multiplier with 3-D CSA</td>
<td>63</td>
</tr>
</tbody>
</table>
List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Comparison of Time, Hardware, and Efficiency of Four Direct Methods' Architectures</td>
</tr>
<tr>
<td>3.1</td>
<td>Comparison of Time, Hardware, and Efficiency of Q-R Method Architectures</td>
</tr>
<tr>
<td>4.1</td>
<td>Timing Simulation of the Parallel Asynchronous Jacobi Method on Example Problem 1</td>
</tr>
<tr>
<td>4.2</td>
<td>Required Hardware for the Parallel Asynchronous Jacobi Method</td>
</tr>
<tr>
<td>4.3</td>
<td>Comparison of L-U Decomposition and the Parallel Asynchronous Jacobi Architecture's Hardware</td>
</tr>
<tr>
<td>4.4</td>
<td>Results of Example Problem 1</td>
</tr>
<tr>
<td>4.5</td>
<td>Results of Example Problem 2</td>
</tr>
<tr>
<td>4.6</td>
<td>Results of Example Problem 3</td>
</tr>
<tr>
<td>4.7</td>
<td>Results of Example Problem 4</td>
</tr>
<tr>
<td>5.1</td>
<td>Comparison of the Number of Addition Stages Required to the Number of Product Terms in a Binary Tree Multiplier</td>
</tr>
<tr>
<td>5.2</td>
<td>Comparison of the Number of Addition Stages Required to the Number of Product Terms in a CSA Multiplier</td>
</tr>
<tr>
<td>5.3</td>
<td>Comparison of the Number of Addition Stages Required to the Number of Product Terms in a 3-D CSA Multiplier</td>
</tr>
<tr>
<td>5.4</td>
<td>Comparison of Binary Tree, CSA, and 3-D CSA Stage to the Maximum Number of Terms that can be Summed</td>
</tr>
<tr>
<td>5.5</td>
<td>Area Required for Different Adders</td>
</tr>
</tbody>
</table>
Nomenclature

A - Matrix, Known
b - Vector, Known
L - Lower Triangular Matrix
M - Dimension of Underdetermined Matrix (MxN)
N - Dimension of Matrix (NxN)
p - Number of Rows In the First Column of a Band Matrix
P - Dimension of Overdetermined Matrix (Pxn)
q - Number of Columns In the First Row of a Band Matrix
Q - Orthogonal Matrix Used in the Q-R Method
r - Residual Vector (b - Ax)
R - Upper Triangular-like Matrix Used in the Q-R Method
U - Upper Triangular Matrix
x - Solution Vector, Unknown
\( \lambda^2 \) - 1 Micron^2 - Critical Design Length Squared
w - JOR Method's Coefficient
\( \Theta \) - Angle Used In Given's Rotations
When an engineer or scientist studies a solution, he or she must first see if there was ever a problem. The engineer is not like the businessman who takes a solution and creates business by creating the problem too, so as to have a demand for the solution. Several solutions are to be discussed in this thesis which have been developed and studied by engineers. So, what is the problem they solve? The problem stems from the delays in processing large quantities of data and from the question of how to perform several operations on the data in the most efficient manner, balancing the use of additional hardware to the reduction of time to complete the operations. The specific algorithms and architectures to be discussed solve large-scale linear systems. These types of systems have massive quantities of data to be processed; currently, most facilities to process these systems are sequential computers, which are slow in comparison to more advanced parallel techniques. This is not to say that parallel techniques are not used today, but that these techniques have not yet been exploited to their fullest strengths in reducing processing time. One may think of making sequential computers more efficient by using parallel algorithms. For example, the solution of a system that contains $N$ linear equations with $N$ unknowns takes on the order of $N^3$ units of time for a sequential computer, while a parallel system can solve it on the order of $N$ units of time. One possible approach is to apply parallel techniques to the designing of a machine that can be interfaced to the computer, which can be accessed whenever a large-scale linear system must be solved. Hopefully, to decrease the cost of producing it and the complexity of interfacing with the computer, this
machine will fit on one VLSI chip. This is a general goal to a
genral problem. The problem will be specified more fully and a solu-
tion to fit that particular problem will be discussed.

So, the problem is how to solve large-scale linear systems.
Several algorithms will be discussed and a few architectures will be
given. These architectures are designed to fit on, preferably and
possibly, a few VLSI chips. The reason for this limitation is to make
it easy and inexpensive to interface the device to any type of com-
puter, from micro-computer to mainframe. The main algorithms to be
discussed are L-U decomposition in Chapter 2, the Q-R method in
Chapter 3, and the Parallel Asynchronous Jacobi method in Chapter 4.
This study is done to find and show very fast algorithms to solve
large-scale linear systems with a limited amount of hardware and to
give a design of this hardware. In addition to the study of the
architectures of these algorithms, the basic element of the architec-
ture, the multiplier, will be studied to compare and to see fast
algorithms and architectures in Chapter 5. And, finally, in Chapter
6, there will be discussion on the decision of what algorithm and
architecture is best for the given circumstances. But, first, some
other state-of-the-art techniques will be discussed.

1.1 State-of-the-Art Techniques

This thesis is too short to give details on all methods known,
but the references of some very recent methods and state-of-the-art
techniques will be given. Though this paper is limited to design of
VLSI chips, recent articles have discussed these same algorithms in
detail, but for use on vector computers and arrays of microprocessors. One such paper that discusses "Iterative Algorithms for Large Sparse Linear Systems on Parallel Computers" is by Loyce M. Adams[2]. This paper has a discussion of many iterative algorithms; one method discussed, Jacobi's method, will be given in detail in Chapter 4 with an architecture developed by the author of this paper to be implemented on a VLSI chip. Loyce Adams discusses only architectures for parallel computers, but she does give a detailed algorithm and a good bibliography. A short paper of note is by Thomas Crockett and Adams; this paper describes a machine that can test the execution time of parallel algorithms[1]. This could be of use to compare several methods for timing on actual examples. In section 4.5 of this thesis, several methods are compared for timing by another technique, which calculates the timing as if the algorithms were actually done in parallel, though they're simulated on a sequential computer. Some other state-of-the-art techniques are using a 3-D array and using optical communication. In a paper by Jan Grinberg et al., a three dimensional array that could be used to solve linear systems is discussed[6]. This makes the design more compact and also seems to aid in reducing communication lines. Communication lines are dealt with by using a state-of-the-art technique of optical interconnections as developed by Goodman et al.[5]. Optic waves can be compacted more than electricity because there is no interference between crossing waves; in the future, communication lines within a VLSI chip will probably be by optics, though, currently, the cost of optics doesn't outweigh the advantage of dense communication lines. These are some of the new advances in the area of solving linear systems.
1.2 Conventional Techniques

Methods of solving large-scale linear systems have been known for years. From these old sequential methods, the new parallel methods have been developed. So, to study and develop new methods, an in-depth study of sequential methods should be made. Several texts are available on the subject. Most noteworthy is the book Numerical Methods by Germund Dahlquist[3]. Other good books are A Survey of Numerical Mathematics by David Young[17], Iterative Solution of Large Linear Systems by Young[18], and Applied Iterative Methods by Louis A. Hageman and Young[7]. These texts, especially Numerical Methods, will provide an understanding of well-known sequential methods. From their algorithms, the parallel methods have been developed.

1.3 Why Parallel?

Why use parallel methods? The idea of parallelism is to study a sequential algorithm, find parts that are independent, and solve the independent parts concurrently. This makes parallel methods inherently faster and inherently requiring more hardware. These two factors determine how parallel a user wants his algorithm to be. In other words, the user must decide how much he wants to spend on hardware for a further reduction in time. The current progression is to use more hardware to gain the speed for two reasons: year after year less area is required and it is less expensive to fabricate hardware, and the need for speed to solve larger and larger equations is increasing. The more parallel operations are, the shorter the execution time. This is why parallelism is the trend in solving linear systems.
1.4 Possibilities of Asynchronosity

Another topic very similar to parallelism is the topic of asynchronosity. If the algorithm can be separated into parts which have very little dependency on each other, each operation can be run in parallel and at its own rate of speed. If parts can be found that don't need sequencing, then the algorithm can be run asynchronously. This will usually increase the speed over a sequential algorithm because time is not spent waiting for another parallel processing element to complete its task. For example, if an addition operation had to be paralleled with a square root operation and both had to use each other's results before proceeding, there would be wasted time by the adder. This seems very difficult, to find an algorithm which could be paralleled asynchronously, but, as will be seen in Chapter 4, iterative methods for solving large-scale linear systems have operations which can be paralleled in this way. Thus, paralleling in an asynchronous way usually increases speed.

1.5 Direct Versus Indirect Methods

Speed and accuracy are the main objectives in choosing an algorithm. Two classes of algorithms are (1) direct, and (2) indirect methods. A direct method is one that takes a fixed number of steps to solve a particular size system of equations; it is an exact method, with errors due only to rounding, as long as the system is not ill-conditioned. So, it is a fixed procedure that systematically solves for the unknown vector. An indirect method approximates, or guesses, at solutions and, hopefully, converges on the correct solution. The
indirect method is usually used when the operator thinks that his set of equations are simple to be guessed at, but too large for direct methods which would require a fixed amount of time to solve his large system. This usually is true in a sparse system of equations because the solution vector's elements are dependent on only a few other elements. Thus, guessing would more likely converge on a solution. The more dependent the equations, the more the iterates will jump from one value to the next value. This is why, in general, direct methods are used for dense systems and "hard to solve" systems, and indirect methods are used for sparse systems.

1.6 General Systems

To make this thesis easier to read, a standard system of linear equations is used. The system is equation 1.1 where \( A \) is the \( N \times N \) matrix that is given, \( b \) is the given \( N \) dimensional vector, and \( x \) is the \( N \) dimensional vector to be solved.

\[
Ax = b \quad (1.1)
\]

This will be used for all methods discussed, unless noted otherwise.
II. L-U DECOMPOSITION AND GAUSS ELIMINATION

The first parallel method to be studied is L-U decomposition. A study of L-U decomposition, however, would not be complete without a discussion of Gauss elimination. So, Gauss elimination will be studied in detail too. Both these methods are direct methods. They go through a step by step routine of reducing the original matrix $A$ to simplify the work necessary to find the solution vector, $x$. Thus, the error in the final solution is due only to rounding errors, unless the matrix is ill-conditioned see [3]. Both methods work well with dense matrices and banded matrices, especially compared to iterative methods, since only rounding errors are a factor in finding a solution. These two methods are not complete in themselves; they require forward and/or backward substitution to find the solution vector. So, when L-U decomposition or Gauss elimination are referred to in solving of linear systems, it is assumed that a standard substitution method is used in conjunction. The difference between these two methods is that Gauss elimination operates on both the matrix $A$ and vector $b$, while L-U decomposition just operates on matrix $A$. Gauss elimination produces one triangular system that must be solved, while L-U decomposition produces two triangular systems. Though, sequentially Gauss elimination takes additional steps to manipulate vector $b$ which balances with the number of steps it takes for the sequential L-U decomposition and one triangular system to be solved, in parallel one additional set of processors can be used on vector $b$. Therefore, Gauss elimination takes less steps than L-U decomposition. But, since L-U decomposition only decomposes matrix $A$, it is easier to use in systems where vector
b varies and x must be solved for several times. With Gauss elimination, if either A or b varies, the whole method must be repeated. L-U decomposition requires only the solving of two triangular systems if b changes, since L and U are already known. L-U decomposition and Gauss elimination are similar, but they do have distinct differences that make them particularly advantageous for specific problems.

2.1 Gauss Elimination

The specific problem that Gauss elimination works best on is dense or banded systems where the matrix elements continually change from problem to problem. No information on the original matrix need be stored. So, if this is the type of system, just use Gauss elimination. But, which type of Gauss elimination? Gauss elimination differs in pivoting strategies. The different types of pivoting strategies are no pivoting, neighbor pivoting, row or column pivoting, and complete pivoting. Complete and row or column pivoting are very stable methods of solving linear systems, but they require global communication among processing elements. Global communication slows the process greatly and is a major hardware cost; therefore, this method is used only on ill-conditioned matrices. Neighbor pivoting (described in [4]) should be used if the matrix is not positive definite and diagonally dominant; otherwise, no pivoting is the best. Both require very little hardware. Neighbor pivoting communicates with the row above and below to see if an exchange is necessary. It can handle zero pivot elements by exchanging with the next row. It also creates less arithmetic errors due to small pivot elements, but
is not as good as complete pivoting. So, if pivoting is to be used, it is suggested to only use neighbor pivoting.

2.3 Variations of L-U Decomposition

L-U decomposition has several variations as well. The variations in pivoting are the same as for Gauss elimination. Basically, L-U decomposition breaks matrix $A$ into two matrices, $L$ and $U$, as is shown in equation 2.1.

$$A = LU$$

$L$ is a lower triangular matrix and $U$ is an upper triangular matrix. Since $L$ and $U$ are not unique for a given matrix, there are several possible combinations of $L$ and $U$ factors. Because of this, there are several methods of decomposing matrix $A$, not just limited to different methods of pivoting. Three methods that have been paralleled are Doolittle's, Crout's, and Choleski's method\[10\]. These methods originated as sequential methods and their sequential algorithms are given in Numerical Methods see [3]. Doolittle's method creates a lower triangular matrix with its diagonal elements equal to one. Crout's method creates an upper triangular matrix with its diagonal elements equal to one. Choleski's method approaches the decomposition differently, and creates an upper triangular matrix that is equal to the transpose of the lower triangular matrix. The three algorithms are similar; therefore, only one algorithm will be studied in detail in section 2.3 noting the major differences in the algorithms. These variations are described by Dahlquist as being compact schemes of
Gaussian elimination, but since Gaussian elimination and L-U decomposition are so similar and because these schemes are types of L-U decomposition schemes, they have been described as L-U decomposition variations. These variations of L-U decomposition and Gauss elimination will be compared in section 2.5. But, first to better understand these methods, Doolittle's algorithm will be given in detail.

2.3 Algorithm

The algorithm for Doolittle's form of L-U decomposition has been described in many papers and books; for this paper the form is taken from Introduction to VLSI Systems[14]. As mentioned before, L-U decomposition works best for band and dense matrices, since a dense matrix is just a matrix with a band width equal to 2*N-1, where N is the dimension, the method will be shown for a band matrix. This form, like the other forms of L-U decomposition, produces two matrices, L and U, which simplify the system of equations to two triangular sets of equations. These sets of equations are equations 2.2 and 2.3.

\[
L \cdot y = b \quad (2.2)
\]

\[
U \cdot x = y \quad (2.3)
\]

In equation 2.2, L is known from the decomposition of matrix A, vector b is given from the original equation, and y is to be solved. y is then used to solve for x since U is also known from decomposition. Both systems are triangular; the first equation requires forward substitution and the second equation requires back substitution. The decomposition process requires the most time since substitution is
performed fast, taking $2N + q$ units of time for each substitution, where $q$ is the width of the triangular system\[l4\]. These two triangular systems must be solved in all three variations of L-U decomposition.

The decomposition varies for all three methods. The scheme to be described is for Doolittle's method. One $N \times N$ matrix is needed for the storage of $A$, $L$, and $U$, since $L$ and $U$ can be stored in one matrix remembering that the diagonal elements of $L$ are all equal to one. $A$ is transformed during the process into a smaller matrix in the lower right-hand corner of the original matrix until, finally, $L$ and $U$ take its place. The process is as follows:

\[
\begin{align*}
    a_{ij}^{(1)} &= a_{ij} \\
    \text{For } k = 1, 2, \ldots, N \\
    u_{kj}^{(k)} &= a_{kj}^{(k)} \text{ for } k < j \\
    l_{ik} = a_{ik} u_{kk} &- 1 \text{ for } i = k \\
    a_{ij}^{(k+1)} &= a_{ij}^{(k)} + l_{ik} (-u_{kj}^{(k)}) \text{ for } i = k \text{ and } j > k
\end{align*}
\]

Remember that the lower triangle of matrix $U$ is zero, except for the diagonal, and that the upper triangle of matrix $L$ is zero, except for the diagonal, which is equal to one. A row of matrix $U$ is calculated, then a column of matrix $L$, and then matrix $A$ is transformed. This is repeated for all rows of $U$ and all columns of $L$.

One way of paralleling this method is to calculate all the elements in one column of $U$ concurrently, then calculate all the elements in one row of $L$ concurrently, and then calculate all the elements of the $k+1$ iteration of $A$ concurrently. This is repeated $N$ times, so the
total amount of time is of the order of $3N$ units of time. But, what are the units of time? This is hard to answer unless a specific architecture is discussed; it depends on how the algorithm is paralleled and how it is pipelined. For instance, if the algorithm is pipelined with these three steps, which can be reduced to two steps if only one matrix of memory is used (look at the calculation of columns of $U$), then each step takes the length of the maximum step, which is the time it takes to calculate a reciprocal and perform a multiplication. For Crout’s method, the equations are very similar and the timing is the same, but with Choleski's method, a square root operation is needed which slows calculations tremendously if the algorithm is pipelined. Thus, the differences in paralleling and pipelining architectures can drastically affect the timing.

2.4 Architecture

Several architectures have been developed for L-U decomposition [10,11, and 12]. BLOSSOM is rather interesting since it uses block L-U decomposition, which makes each element in the previous discussed algorithm equal to a sub-matrix. The reciprocal of a scalar element corresponds to the inverse of a sub-matrix. This article provides a table of different methods and timing, but it does not compare each method with the same time units. It is difficult to compare BLOSSOM to other methods since all its internal pipelining for each step in the algorithm are not discussed. In “VLSI Algorithms for Doolittle's, Crout's, and Choleski's Methods” and in Introduction to VLSI Systems, arrays of processing elements are pipelined in two dimensions. In C. Pottle’s article, an elaborate form of memories,
both RAMs and CAMs, is used to try to deal with an unstructured sparse system, which is not advised by the author of this thesis to be done with L-U decomposition. The reason is that either many extra processing elements are needed, or a simulation run is needed to determine which elements of the matrix will be transformed from zero to nonzero; the unstructured sparse case will be dealt with in Chapter 4. Now, another architecture will be presented that eliminates some of the problems in these other architectures, but has some of its own problems.

The architecture to be described requires \((N+1)*(p+q-1)-p*q\) processing elements, where \(N\) is the dimension, \(p\) is the number of rows in the first column of the band, and \(q\) is the number of columns in the first row. This is true if \(N\leq 2*\max(p,q)\), which signifies the band is thin compared to the dimension of the matrix, which is usually the case. One processing element is used for each element of the original matrix within the band. The algorithm is modified to the algorithm below:

\[
\begin{align*}
\text{initialize} \\
D_{ij} &= a_{ij} \\
\text{For } k = 1, 2, \ldots, N \\
\text{Step 1} \quad &D_{ik} = D_{ik} * D_{kk}^{-1} \quad \text{for } i \neq k \text{ within band} \\
\text{Step 2} \quad &D_{ij} = D_{ij} - D_{ik} * D_{kj} \quad \text{for } i \neq k \text{ and } j \neq k \text{ within band}
\end{align*}
\]

This simplifies the algorithm to two steps, as mentioned in the algorithm section. Step 1 is executed and then step 2; this is repeated \(N\) times. The algorithm is the same as C. Pottle's algorithm, but it is used in a different form that shows the reasoning of the
architecture for this system better[11].

The arrangement of processors is shown in Fig. 2.1.

As can be seen, the D processors are connected to all the L processors below them so as to transmit $D_{kk}^{-1}$ to each L processor for step 2.

For step 2 of the algorithm, the U processors must be connected to $p-1$ processors below them, and L processors must be connected to $q-1$ processors to the right of them. So, there are some complications in
routing, and probably communication delay caused by long interconnections, if the band is wide. The D processors are able to perform reciprocals, multiply, and subtract. The L and U processors only need to be able to multiply or subtract, which is done in step 2.

The timing for this architecture is $N$ times the time for step 1 plus step 2. So, total time is the time necessary to load the processor array plus the time needed to evaluate $2N$ multiplications, $N$ reciprocals, and $N$ subtractions. Another architecture that is described by Mead and Conway[14] requires $3N + \min(p,q)$ units of time. The loading time is equal to the $\min(p,q)$ units of time. Since their architecture is pipelined, the units of time are equal to the time to compute a reciprocal. Thus, the total time for their architecture is load time plus the time it takes to compute $3N$ reciprocals, which is probably greater than the architecture just described.

2.5 Performance

The performance of this method is good. It is very fast, but it requires a large amount of processing elements for matrices with small bands. One way to compare the performance with other methods is by the use of an efficiency factor, $\varepsilon$. This factor is the inverse of the product of hardware and time. The time will be in units of time to complete reciprocals or divisions, multiplications, additions or subtractions, and square roots. To compare these units, the number of machine cycles each one takes on the IBM 360/91 is used[8]. This provides a rough comparison to meet our needs. Reciprocal and division time units are multiplied by 9, multiplication time units are
multiplied by 3, and additions and subtractions are multiplied by 1. Square root generators require additional area to fabricate and additional time to calculate and are not suggested to be used; the time factor is assumed to be large. Table 2.1 shows a comparison of four different architectures. The first method is the architecture described in section 2.4, which is the fastest of the L-U decomposition methods, but may require the most hardware. Note that both Doolittle's and Crout's algorithms are listed for both method 1 and method 2, because both methods can be executed with very similar architecture. Method 2 is by Mead and Conway and it requires little hardware. Method 3 requires even less hardware, but requires a square root generator. Method 4 is a paralleled Gaussian elimination scheme that is used in a simulator program given in Appendix B. It is the fastest method, especially if it is considered that it only requires one triangular system to be solved, while the other three methods require two triangular systems to be solved. But, as discussed before, if the matrix does not change for several problems to be solved, this method must recompute the solution completely from the beginning, while the other methods only need to solve the two triangular systems. The efficiency factor is not very useful unless it is compared on a specific system, though, it does seem, on most systems the second method will be the best. The efficiency factor does not take into account the necessity for speed. Probably the efficiency factor should have two variables, one for hardware and one for speed, that tell the relative importance of each factor. So, the table can be interpreted in many ways, and it should be noted that the times
<table>
<thead>
<tr>
<th>Architecture and Algorithm</th>
<th>Reciprocals or Divisions</th>
<th>Multiplications</th>
<th>Additions or Subtraction</th>
<th>Square Root</th>
<th>Number of Processors</th>
<th>Efficiency Factor, $\epsilon$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Method 1 Doolittle's or Crout's by method in Section 2.4</td>
<td>$N$</td>
<td>$2N$</td>
<td>$N$</td>
<td>0</td>
<td>$(N+1)(p+q-1)$</td>
<td>$1/(16N(N+1)^2 - pq)$</td>
</tr>
<tr>
<td>Method 2 Doolittle's or Crout's by Mead and Conway[14]</td>
<td>$3N$</td>
<td>$0$</td>
<td>$0$</td>
<td>0</td>
<td>$pq$</td>
<td>$1/(27Npq)$</td>
</tr>
<tr>
<td>Method 3 Choleski's by Lennart Johnsson[10]</td>
<td>$0$</td>
<td>$0$</td>
<td>$0$</td>
<td>$3N$</td>
<td>$1/2pq$</td>
<td>Very Small</td>
</tr>
<tr>
<td>Method 4 Gauss Elimination by Paralleled Method in Program in Appendix B</td>
<td>$N$</td>
<td>$N$</td>
<td>$N$</td>
<td>0</td>
<td>$N(N+1)$ or less</td>
<td>less than $1/(13N^2)$</td>
</tr>
</tbody>
</table>
that are given reflect the amount of time only for decomposition and not solution. The performance of the method described in section 2.4 is equal to other well-known methods, and the method chosen to solve a dense or banded matrix should be determined by the importance of small hardware and short execution time.
III. Q-R METHOD

Speed versus hardware is an important decision, but more important is whether the architecture can handle the given system of equations. As discussed previously, L-U decomposition and Gauss elimination work best on dense or banded matrices, but, can they solve underdetermined or overdetermined systems? For overdetermined systems, their architecture would allow the first $N$ equations to be input, and then not use the extra equations. But, by doing this, some information will be lost. If this information is important, it should be used. So, what method can solve systems that are under or overdetermined? The answer is the Q-R method.

3.1 Algorithm

The Q-R method is another decomposition method. The matrix $A$, which in discussion of this method only is $P \times N$, is decomposed into two matrices, $Q$ and $R$, as is shown in equation 3.1.

$$A_{P \times N} = Q_{P \times P}^* R_{P \times N}$$  (3.1)

The problems occur mainly with overdetermined systems, as will be discussed in section 3.2, and this system is assumed not to be underdetermined, which will be discussed in section 3.5. Therefore, $P$ can be greater than or equal to $N$. $N$ is the number of variables and $P$ is the number of equations. $Q$ is an orthogonal matrix and $R$ is an upper triangular matrix, which is usually chosen with its diagonal equal to one. If this is true, then $Q$ and $R$ are unique, unlike $L$ and $U$ in L-U decomposition[3].
If $Q$ and $R$ can be found, how can they be used to solve the set of linear equations? Equation 3.2, being an altered form of equation 1.1, can be transformed into equation 3.3 if the inverse of $Q$ is calculated.

$$A_{P\times N} x_N = b_P \quad (3.2)$$

$$R_{P\times N} x_N = (Q_{P\times P})^{-1} b_P \quad (3.3)$$

But, rather than calculate the inverse of $Q$, usually $b$ is transformed into a vector $y$, which is equal to the product of $Q$ inverse and $b$, which is shown in equation 3.4.

$$(Q_{P\times P})^{-1} b_P = y_P \quad (3.4)$$

So, equation 3.3 simplifies to equation 3.5.

$$R_{P\times N} x_N = y_P \quad (3.5)$$

This shows a system very much like Gauss elimination, since $A$ is transformed into an upper triangular matrix and $b$ changes to $y$. The only differences are that $P$ is not always equal to $N$, and the transformation is done with unitary matrices to cause less errors. Stability is increased when $Q$ is unitary. $y$ and $R$ can be formed by two well-known methods, Househoulder's transformations and Given's rotations, which will be compared in section 3.3. After they are calculated, $x$ can be obtained by solving one triangular system with back substitution. Thus, the algorithm is very similar to Gauss elimination, but with $Q$ chosen to make the decomposition of the matrix more stable.
3.2 Overdetermined Systems - Least Squares Solution

A general idea has been given of how the Q-R method solves an overdetermined system, but what does this solution mean? If \( N \) equations determine an exact solution which exactly agreed with the additional equations, then, why use these additional equations? The answer is they usually never agree exactly. A comparison can be made to an experiment where every extra data point helps to determine the relationship. So, all the equations will help to close in on solution. The solution in this case is said to be a least squares solution. As is shown in equation 3.6, the residual vector, \( r \), is a factor which shows how close the solution vector is to an exact answer.

\[
r = b - Ax \quad (3.6)
\]

A least squares solution is when the length of the residual vector is minimum, which is when \( \|r\|_2 \) is a minimum. The Q-R method achieves a least squares solution; for a proof see [3]. So, overdetermined systems should be solved with a method that finds the minimum residue, which means the solution that best agrees with the data and the Q-R method does this.

3.3 Householder's Transformations Versus Given's Rotations

The Q-R method does this with similarity transformations. There are two types of similarity transformations that are usually used; they are called Householder's transformations (or reflections) and Given's rotations. Both methods are orthogonal transformations, and that's how they differ from Gauss elimination. These two orthogonal
transformation methods differ from each other by the way they choose a matrix to multiply the given matrix \( A \) is chosen.

The plane rotation method (Given's rotations) uses a matrix \( T \), which is equal to the identity matrix, except for the elements

\[
\begin{align*}
t_{jj} &= t_{kk} = \cos \theta, & t_{jk} = -t_{kj} = \sin \theta & (3.7)
\end{align*}
\]

Only columns \( j \) and \( k \) of matrix \( A \) will change when multiplied by this matrix. \( \theta \) is chosen to reduce element \( a_{jk} \) to zero. Then, element by element in the lower triangle of matrix \( A \) is reduced to zero.

The method of reflections uses a matrix \( P \) that is equal to

\[
P(w) = I - 2ww^T, \quad w^Tw = 1.
\]

\( w \) is chosen to reduce a particular column so that all elements below the diagonal are zeros. This is done by setting the first \( j-1 \) elements equal to zero, if the \( j \)-th column is to be reduced, and then scaling the other elements to make the system stable.

Both methods reduce the matrix \( A \) into a triangular matrix while transforming vector \( b \). These methods are very stable, since they use orthogonal matrices. In actuality, neither method uses a matrix multiplication between the chosen orthogonal matrix and the original matrix \( A \). Instead, it transforms the matrix \( A \) as though it were multiplied by this imaginary orthogonal matrix. The question is which method causes less errors? According to L. Johnsson, the limit on errors with the reflections method is less than with the rotations method [3, see also 4]. So, do the speed and the hardware requirements for rotations, as compared to reflections, justify the
loss in accuracy inherent in the algorithm? This will be studied in
the next section, which compares two architectures for the Q-R method.

3.4 Comparison of Two Parallel Architectures

Two architectures that are to be compared are Lennart Johnsson's
architecture for Householder's reflections[9], and W.M. Gentleman's
and H.T. Kung's architecture for Given's rotations[4]. No original
architecture for this algorithm has been developed in this thesis.
Though, for completeness, a comparison is given between two architec-
tures developed by a few widely published scientists in this field.
This will show the progress in parallel architectures for both
algorithms. For further details on the architectures see [4 and 9].

W.M. Gentleman's and H.T. Kung's architecture is a systolic array
that transforms matrix A into an upper triangular form row by row.
The array is pipelined; matrix A and vector b are fed into the array
in an off-skew manner, as is depicted in Fig. 3.1.

Fig. 3.1 Computational Array for Given's Rotations[4]
The array is triangular and has \( \frac{1}{2}(p+1)^2 \) processing elements, where lower case \( p \) represents the number of rows in the first column of the band. As can be seen from Fig. 3.1, it takes \( P + 2 \times \min(p,q) \) units of time to feed all the elements of \( A \) and \( b \) through the array, where \( q \) represents the number of columns in the first row of the band, and capital \( P \) represents the number of equations in matrix \( A \). A study of the algorithm indicates that the units of time are the time needed to execute an addition, multiplication, division, and a square root in sequence. Kung and Gentleman state, but do not describe, an architecture that doesn't require calculation of square roots. The architecture for the array is said to be similar, and the time units are equal to the time necessary to execute five multiplications and one reciprocal. This type of architecture would save time over calculating square roots. The time for execution and the required hardware have been calculated for Given's rotations.

A calculation is needed for the time and hardware required for Householder's reflections. The architecture used for this comparison is Lennart Johnsson's, as mentioned earlier. This method is more complicated, and more intricate design is given in his paper. His algorithm transforms the matrix, column by column, into an upper triangular matrix \( R \). It operates on a computational window at a time where the window is \( (p) \times (p+q-1) \). Inasmuch as the array is very complicated, it is suggested that one see his paper for a detailed diagram of it[9]. The timing of the architecture requires \( P(p+2) \) multiplications, \( P(p+2) \) additions, and \( P \) square roots. Hardware is required to calculate the square roots. This architecture, because of
its intricate design, raises the question of how hardware should be compared. Should it be added up multiplier by multiplier, or should a general number of processing elements, which have the same area as a multiplier and an adder, be compared? This study will use general numbers to evaluate hardware involved, since some architecture are not detailed enough to compare them on a device to device basis. So, after a detailed analysis of the architecture, the hardware required is

\[ \text{Hardware} = 3p^2 + 3pq + 2p. \]

Thus, the time and hardware requirements have been given for Householder's transformations, as designed by Lennart Johnsson.

So, how do these architectures compare? Table 3.1 gives a comparison of the two architectures, with two variations for Given's rotations. This comparison shows that the architecture is less for Given's rotations, and the time of execution for the Given's rotations, without square roots, is the smallest one. So, for both a hardware comparison and a speed comparison, Gentleman's and Kung's architecture for Given's rotations, without square roots, is the best. Though this architecture has been suggested, a detailed architecture has not been given. Also, as mentioned earlier, Given's rotations inherently has more numerical errors than Householder's transformations. This architecture is still recommended, but the user must evaluate his needs and decide whether these detriments are not too great. So, to solve an overdetermined linear system, the Q-R method, implemented with Given's rotations without square roots, on an archi-
Table 3.1 Comparison of Time, Hardware, and Efficiency of Q-R Method Architectures

<table>
<thead>
<tr>
<th>Architecture and Algorithm</th>
<th>Reciprocals or Divisions</th>
<th>Multiplications</th>
<th>Additions or Root Subtraction</th>
<th>Square Root</th>
<th>Number of Processors</th>
<th>Efficiency Factor, e</th>
</tr>
</thead>
<tbody>
<tr>
<td>Method 1</td>
<td>P + 2 min(p, q)</td>
<td>P + 2 min(p, q)</td>
<td>P + 2 min(p, q)</td>
<td></td>
<td></td>
<td>1/2(p+1)^2</td>
</tr>
<tr>
<td>Given's Rotations 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>?</td>
</tr>
<tr>
<td>by H.T. Kung and W.M.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Gentleman[4]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Method 2</td>
<td>P + 2 min(p, q)</td>
<td>5P + 10 min(p, q)</td>
<td>0</td>
<td>0</td>
<td></td>
<td>1/2(p+1)^2</td>
</tr>
<tr>
<td>Given's Rotations 2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>12(p+1)^2</td>
</tr>
<tr>
<td>by H.T. Kung and W.M.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>(P + 2 min(p, q))</td>
</tr>
<tr>
<td>Gentleman[4]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Method 3</td>
<td>0</td>
<td>P(p+2)</td>
<td>P(p+2)</td>
<td>P</td>
<td></td>
<td>3p^2 + 3pq + 2p</td>
</tr>
<tr>
<td>Householder's Transformation</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>?</td>
</tr>
<tr>
<td>by Lennart Johnsson[9]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
tecture suggested by W.M. Gentleman and H.T. Kung is the most recommended.

3.5 Underdetermined Systems

Now that overdetermined systems are understood and solvable, how do you solve an underdetermined system? An underdetermined system is one involving less equations than unknowns. In this system, matrix $A$ is $M \times N$ where $M < N$. There is more than one solution, though, usually, only one of these solutions is wanted. So, to simplify calculations, the solution vector elements $x_{M+1}$ to $x_N$ are set equal to zero. Then the system is solved as though it were $M \times M$. The Q-R method is a good method to use since it is more stable than L-U decomposition or Gauss elimination. After a solution has been found, the elements $x_{M+1}$ to $x_N$ are output as zeros with the calculated elements $x_1$ to $x_M$. Thus, underdetermined systems are easy to solve, and in general, they have many correct solutions.

Another approach to the underdetermined system problem is to solve for the general solution. An architecture should be developed that can output the system reduced to a $M$ by $N-M$ system, which shows the dependency of the first $M$ variables to the next $N-M$ variables. This would be a general solution. An architecture needs to be developed for this type of solution to the underdetermined system.

As has been shown, both underdetermined and overdetermined systems can be solved in parallel. The Q-R method spends time orthogonalizing the transformation to increase stability. This requires additional time and should only be used when necessary, or predicted to be necessary.
IV. PARALLEL ASYNCHRONOUS JACOBI METHOD

The methods discussed so far for solving large-scale linear systems were designed primarily for dense matrices, or overdetermined and undetermined matrices. Another type of matrix is the sparse matrix. Sparse matrices can be either a band matrix or unstructured matrix. For an unstructured matrix, L-U decomposition must treat the matrix as though it were a dense matrix by providing an array of processors wide enough to input a dense matrix. This is necessary since the nonzero elements can be in any column or row, and not just within a few elements of the diagonal. Because of this failure of L-U decomposition, iterative methods have been used. With iterative methods, less memory is required and, usually, less processing elements, since only the nonzero elements are of importance. In L-U decomposition, even with only the nonzero elements stored, one must run a simulation to determine where nonzero elements will be created during the process. This requires additional memory for the created nonzeros and additional time for the simulation run. Because iterative methods don't have these requirements, they are popular in solving sparse systems.

A few iterative methods will be examined in this chapter, with major emphasis on the Parallel Asynchronous Jacobi method. Other variations to be discussed in section 4.4 are the Parallel Conjugate Gradient method and the JOK method. But first, the Parallel Asynchronous Jacobi method will be discussed in detail, with the algorithm being discussed in section 4.1, an example simulation in section 4.2, the architecture in section 4.3, and a performance evaluation in section 4.4.
4.1 Algorithm

The Parallel Asynchronous Jacobi method has a relatively simple algorithm which has been suggested in a paper by Loyce Adams[2]. This method solves a system of linear equations in the form of equation 4.1, where \( A \) is the given matrix, \( b \) is the given vector, and \( x \) is the solution vector to be solved for.

\[
Ax = b \quad (4.1)
\]

The sparse system need not be banded. In fact, the algorithm has no preference for where the nonzero elements are within the matrix, as long as the matrix is positive definite and diagonally dominant. The algorithm is derived by transforming equation 4.1 into index notation, as is done in equation 4.2, and then solving for a particular row's solution vector element, as in equation 4.3.

\[
\sum_{j \neq i}^{N} a_{ij} x_j = b_i \quad (4.2)
\]

\[
x_i = \frac{(b_i - \sum_{j \neq i}^{N} a_{ij} x_j)}{a_{ii}} \quad (4.3)
\]

This shows why the diagonal element, \( a_{ii} \), must be nonzero; this avoids division by zero. Furthermore, the Jacobi Convergence theorem, as stated by David Young, says that the matrix must be positive definite and diagonally dominant to assure convergence[18]. To find the solution vector, equation 4.3 must be iterated as is shown by equation 4.4.

\[
x_i^{(k)} = \frac{(b_i - \sum_{j \neq i}^{N} a_{ij} x_j^{(c)})}{a_{ii}}, \quad (4.4)
\]

for \( k=1,2,\ldots,N \)
k stands for the number of iterations on that particular element of the solution vector. \( x^c \) stands for the most current iteration of each element of the solution vector. The algorithm is run with each row in parallel. Since it is also run asynchronously, each element of the solution vector may be on a different iteration, and this is the reason a different superscript is used on the solution vector on the right-hand side of equation 4.3.

Another way to view equation 4.3 is in a pictorial representation, as is given in Fig. 4.1.

![Fig 4.1 Parallel Asynchronous Jacobi Method Pictorial](image)

This shows a particular row's view of the process. Row i of matrix A is multiplied times vector x to give element i of vector b, which is known. \( x_i \) is solved for, since all other variables are known, by using the most current iterates of the solution vector.

This method may seem very familiar because it's just an asynchronous parallel variation of two sequential methods for solving sparse linear systems. These two sequential methods are Jacobi's method and Gauss-Seidel's method[3]. The sequential Jacobi's method is synchro-
nized in both calculating the next iterate for each row and for updating the solution vector. It does not update the solution vector, $x$, until after all the rows have computed new $x_i$'s, as is shown in the notation and in the diagram in Fig. 4.2.

$$x^{(k+1)}_i = \left( b_i - \sum_{j=1 \atop j \neq i}^N a_{ij} x^{(k)}_j \right) / a_{ii}; \ i = 1, 2, \ldots, N$$

![Diagram of Sequential Jacobi Method Algorithm and Pictorial](image)

Fig. 4.2 Sequential Jacobi Method Algorithm and Pictorial

The $k+1$ iterations of $x_i$'s are stored in temporary locations, and no updating is done until all $x_i$'s have been calculated. This could be accomplished in parallel by making the row processing elements wait until all of the rows have calculated the next iteration of $x_i$, where $i$ is the row.

Gauss-Seidel's method is another sequential method like Jacobi's, but it uses all the previously calculated information. As shown in Fig. 4.3, it continually updates the solution vector, $x$. Using this concept, the Parallel Asynchronous Jacobi method was developed. All the rows were isolated so they could be paralleled, and all the calculated information was immediately used to update the solution vector.
The Parallel Asynchronous Jacobi method's algorithm was developed from the study of sequential methods of solving large systems of linear equations.

4.2 Simulation and Example

The algorithm seems good, but how good? Since the algorithm is not sequential and synchronized, there is a very difficult mathematical problem in trying to find the rate of convergence. If an iteration of the solution vector converges on a value, then this value is the correct value, but how long did it take? The rate of convergence will depend on the system, in particular, by the inverse number of nonzeros in each row. This is due to the dependency of $x_i$ on the other row's $x$ that have nonzero elements. The more sparse, the faster the convergence. This is why this method is recommended only for sparse matrices. Since the mathematics involved is very difficult and changes for every matrix, a simulation program has been written which
can determine the time for convergence on a particular example. This simulation program for the Parallel Asynchronous Jacobi method, which was developed by the author, can be found in Appendix A. With this program, the time and the solution can be found as if the architecture to be described was actually used.

The simulation program is complicated because it must deal with many simulation problems. The program is run in FORTRAN on a sequential, synchronized computer, the VAX 11/750. To make a parallel simulation, time had to be expanded. The execution time of the simulation program is not equal to the execution time of the architecture. Instead, an internal clock is used, which calculates the number of additions, multiplications, and divisions that would use execution time if it were done in parallel. Another problem dealt with is asynchrony. An estimate had to be made of how fast each row is, and, from this information, an order had to be determined of when to iterate a row's solution vector. As can be seen from the algorithm, all rows operate at their own rate or speed, and this asynchronous operation had to be simulated. So, there are major problems in simulating the Parallel Asynchronous Jacobi method on a sequential computer.

Simulation of the Parallel Asynchronous Jacobi algorithm requires an internal clock in the program. To understand this need and to show how the internal clock works, Fig. 4.4 shows the different timings needed for different row processors to calculate an iterate of the solution vector, \( x \). This is the timing of an example problem that was actually run with this simulator program. The difference in calcu-
Fig. 4.4 Timing of Iterates for Parallel Asynchronous Jacobi Method on Example Problem 1

Calculating an iterate is caused by the difference in calculating in the number of nonzero elements in the example problem, which is an unstructured sparse matrix. From equation 4.4, the summation of $a_{ij}x_j$ must be calculated. If there are less elements in a row $i$ than row $j$, less additions are necessary in the calculation of $x_i$ than $x_j$. A parallel binary tree of adders is assumed for this example; so, it takes log base two of the number of nonzero elements in each row to accomplish this summation. Plus, it takes a multiplication step to calculate all $a_{ij}x_j$ products in parallel; also, to calculate an iterate for each row, it takes an addition of $-b$ and a division by the diagonal element. The internal clock records the number of additions, multiplications, and divisions separately, and uses the 1-3-9 system, as discussed in section 2.5, to compute a total time for comparison with other methods. This internal clock is used to find which row
should be iterated, and when this iterated information should be used to update the solution vector. Actually, the internal clock consists of several registers that record the number of additions, multiplications, and divisions for each row.

Other temporary registers are needed for the calculation of the iterates. Locations are needed to store the next iterate of each row, and when the relative clock of a row reaches a time that corresponds to the completion of an iteration, the next iterate is dumped from a temporary location into the solution vector location. This enables each row to use this iterate for the calculation of its next iterate. In actuality, two temporary locations are needed for each row's iterates. The second temporary location is used to help in the transferring of iterate information when several rows complete an iteration simultaneously. This is very common, since it occurs when two or more rows have the same number of nonzero elements. The algorithm, in this situation, would give all the rows the previously calculated solution vector, instead of giving any of these rows' last iterations. The simulation program does this, by giving each of the rows the same solution vector, and iterating each one, but it doesn't update the solution vector until after all these rows have used the previously calculated solution vector. These iterates are stored in the secondary temporary solution vector. So, temporary registers are required to simulate the parallelism and asynchronoussity of the Parallel Asynchronous Jacobi algorithm.

The simulation program's internal timing has been discussed, and now an overview of the program is needed. Fig. 4.5 shows the
hierarchy of the simulation program.

Fig. 4.5 Hierarchy of the Simulation Program for the Parallel Asynchronous Jacobi Method

The main program calls the BUFFER subroutine, which calls the SOLVER subroutine, which calls SEARCH and ITER periodically, and either ITER or SOLVER calls PRINT, which executes and, then, stops the procedure. The main program is written by the user, and contains the necessary dimension of the real and integer vectors, calculation of these dimensions is described in the simulation program in Appendix A. The main program calls the BUFFER subroutine. The BUFFER subroutine reads the data and creates pointers to tell where each vector in the SOLVER subroutine will be located in the original real vector and integer vector. Then, it calls the SOLVER subroutine. The SOLVER subroutine
is the main subroutine that controls solving the linear system. It controls the timing. It, also, calls the ITER subroutine to iterate, and the SEARCH subroutine when it wants to know which row is next to be iterated. The SOLVER subroutine, also, checks for no convergence, and checks to see whether the maximum number of iterations has been exceeded. If this is true, then, it will call the PRINT subroutine. SEARCH compares row clocks to find which row needs to iterated, and also tells whether row clocks are finishing an iteration simultaneously. ITER subroutine iterates, and checks for convergence by a radius of convergence. If convergence is achieved, the PRINT subroutine is called. The PRINT subroutine prints the solution and the time, then it stops the process. This completes the hierarchy of the simulation program.

The simulation program BUFFER subroutine reads the data. The data is input in three files, one for real numbers, one for integers, and another for convergence and timing. Full details are given in Appendix A, but one thing to note is that the matrix A is stored only by nonzero elements. There is a column pointer vector and a row information vector to sort this data. This is because less memory is used since the matrix is sparse.

The sparse matrix problem that was first tried, example problem 1, is shown in Fig. 4.6. The system is a linear, unsymmetric, unstructured, sparse one. It is an 8x8 problem that has eight equations and eight unknowns. The matrix is 50% sparse. The solution vector, x, is known and is exact. This is a theoretical example, which is used to show the convergence on a general sparse system. In
Fig. 4.6 Example Problem 1: Unsymmetric, Unstructured, Sparse System

Table 4.1 shows how fast this simulation program's internal clock will converge on a certain radius of the solution, which is the same timing as if the parallel machine tried the problem. The table shows the maximum number of iterations. This corresponds to the fastest row's number of iterations. The minimum number of iterations is of more importance because this is the slower row, which usually takes more time to converge. The other rows have more iterations, because they have converged and are waiting for the other rows to converge. So, this number is then used to find the multiples of the dimension it took to converge. This gives the coefficient of the order of time (N), so that this method can be compared easily with other methods.
Table 4.1 Timing Simulation of the Parallel Asynchronous Jacobi Method on Example Problem 1

<table>
<thead>
<tr>
<th>Radius of Convergence</th>
<th>Min./Max. Number of Iterations</th>
<th>Multiples of Dimension N</th>
<th>Execution Time of the Longest Row Clock</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.0 E 00</td>
<td>6 - 7</td>
<td></td>
<td>14 - 7 - 7 - 7 - 7 - 98</td>
</tr>
<tr>
<td>1.0 E-01</td>
<td>7 - N</td>
<td>24 - 8 - 8 - 8 - 120</td>
<td></td>
</tr>
<tr>
<td>1.0 E-02</td>
<td>13 - 15</td>
<td>30 - 15 - 15 - 210</td>
<td></td>
</tr>
<tr>
<td>1.0 E-03</td>
<td>17 - 20</td>
<td>20N - 20 - 20 - 280</td>
<td></td>
</tr>
<tr>
<td>1.0 E-04</td>
<td>22 - 25</td>
<td>88 - 22 - 22 - 352</td>
<td></td>
</tr>
<tr>
<td>1.0 E-05</td>
<td>25 - 30</td>
<td>3N - 27 - 27 - 405</td>
<td></td>
</tr>
<tr>
<td>1.0 E-06</td>
<td>27 - 31</td>
<td>29 - 29 - 435</td>
<td></td>
</tr>
<tr>
<td>1.0 E-07</td>
<td>33 - 37</td>
<td>4N - 33 - 33 - 528</td>
<td></td>
</tr>
<tr>
<td>1.0 E-08</td>
<td>37 - 43</td>
<td>43 - 43 - 43 - 602</td>
<td></td>
</tr>
<tr>
<td>1.0 E-09</td>
<td>40 - 46</td>
<td>5N - 43 - 43 - 645</td>
<td></td>
</tr>
<tr>
<td>1.0 E-10</td>
<td>46 - 52</td>
<td></td>
<td>184 - 46 - 46 - 736</td>
</tr>
</tbody>
</table>
The best way to compare time is by a more exact means than a coefficient. At the right-hand side of the table is the time in the units of execution time for additions, multiplications, and divisions for the longest row to converge. Then, there is a total time that corresponds to the equivalent number of executions of an addition operation. This number is calculated by using a 1-3-9 system as described in section 2.5. With the total time, an accurate comparison can be made between methods.

The simulation program converges rather fast. It takes approximately N iterations to gain an accuracy of two decimal places. The method would converge even faster for a sparser matrix.

So, the example problem has been solved and an estimate of the rate of convergence for this particular system has been given. This was done by the simulation program rather than by mathematics, which would have to be redone for every new system.

4.3 Architecture

Now that the algorithm and a simulator program have been discussed, how could this algorithm be realized? Thus, the next step is a study of the architecture of the Parallel Asynchronous Jacobi method. In the past, the only architecture that has been developed to simulate this algorithm has been an array of microprocessors designed by Loyce Adams. This thesis proposes a new architecture to be realized in VLSI. This type of architecture is faster and smaller. Faster because communication links are closer, and smaller because area is wasted using a microprocessor to just multiply and add. Thus, a VLSI design would be best for this algorithm.
The design is dependent on the manner in which the data is input. There will be a multiplier unit for every off-diagonal, nonzero element in the matrix. To configure this architecture, a complex Central Processing Unit (CPU) is required. This unit will restructure the buses for any type of matrix that is input. To ease its job, the data is fed in, as in the simulator program, by nonzero elements of matrix with column pointers. This helps the CPU because it does not need to sense which elements are nonzero before it configures the buses. Thus, the CPU just needs to add or subtract the row information to decide how many multipliers are needed for each row. To determine all the connections between the multipliers and the solution vector memory, the column pointers only need to be looked at. Thus, a CPU is needed for configuring and loading.

The restructuring that the CPU does will be considered to be complete, so that the architecture can be viewed as already being structured. This will make it easier to view the structure, since the diagrams will show the hardware spread out.

The architecture for row processors is given in Fig. 4.7. Each row will have a multiplier for every nonzero element in matrix A. The multipliers will have a one number memory, which will be loaded with a nonzero element. The solution vector is multiplied by the corresponding nonzero elements of matrix A. These products must be summed together. Several ways can be used to do this, as will be shown in Chapter 5, but binary tree adders were used. A divider with the diagonal element, \( a_{11} \), in memory is also needed. This output goes to a buffer, which creates a greater fanout than possible from
Fig. 4.7 Row i Processor for the Parallel Asynchronous Jacobi Architecture
the divider. This buffer continually updates the solution vector, which is stored separately for each row. This is necessary for asynchronousity, because the multipliers of different rows would have problems accessing the same element of vector \( x \) simultaneously. Thus, the architecture of one row is given, and this same architecture tailored to each row's number of nonzero elements, is repeated for all \( N \) rows.

The row processors take different lengths of time due to different numbers of nonzero elements. With independent row processors, all rows can be run independently of each other. There are two different ways of running this architecture. First is the method used in the simulation program that updates the solution vector and then starts the calculation of the next iterates. This involves very little pipelining. The adders run independently of the multipliers in this method. The second method pipelines the guesses of the solution vector every instruction cycle. This method was developed after designing the architecture. This makes the architecture more sequential, but adds to asynchronousity, which increases the speed. The pipelining does delay calculation of a particular iterate because the divider is the slowest device and the adders could have performed eight more operations while waiting for the divider. The throughput, though, is increased greatly. So, there are two types of synchronization that could be used.

So far, a CPU is needed and hardware for the row processors is needed. In addition to this, hardware is required for convergence checking. Convergence can be checked for in several ways, though
only one simple method will be discussed here. This scheme involves having the row buffers drive one more line, which is connected to a shift register. The shift register must contain at least the last two iterates for each element of the solution vector. The number of iterates stored depends on the method of synchronization. Only two iterates need to be stored for the first method described for synchronization. A comparator is connected to compare the newest iterate with the oldest iterate, and, if it is within a specified margin, the comparator pulls the line low. A NAND gate or a series of NAND gates are connected so that a high is outputted to signal convergence. When the CPU gets this signal, the process is halted and the newest iterates are output. Also, with this system, there must be a counter to tell the CPU how many iterations have occurred. The CPU or a comparator can be used to see if this is higher than or equal to the maximum allowed. If so, then the CPU halts the process and signals the user that there is no convergence. So, the architecture for checking convergence consists of only a couple shift registers, comparators, NAND gates, and a counter.

The total amount of hardware required for the Parallel Asynchronous Jacobi method is shown in Table 4.2. At first glance, this seems like a great amount of hardware, but for a very large and very sparse matrix, this is very small. Notice the dependency on the dimension of the matrix is at most on the order of $N$. The multiplier number is dependent on the number of nonzeros. For Gauss elimination on an unstructured sparse matrix, there must be $N^2 + N$ multipliers and adders. In Table 4.3 a comparison is given of the number of pro-
Table 4.2  Required Hardware for the Parallel Asynchronous Jacobi Method (N=number of rows and Z=number of nonzeros)

<table>
<thead>
<tr>
<th>Hardware Item</th>
<th>Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>Z</td>
</tr>
<tr>
<td>Multipliers</td>
<td>log2(Z)</td>
</tr>
<tr>
<td>with one cell of memory</td>
<td></td>
</tr>
<tr>
<td>Adders</td>
<td>N</td>
</tr>
<tr>
<td>Dividers</td>
<td>N</td>
</tr>
<tr>
<td>Memory vectors containing N+1 cells</td>
<td>N+1</td>
</tr>
<tr>
<td>Row buffers</td>
<td>&gt;2N</td>
</tr>
<tr>
<td>Shift Registers</td>
<td></td>
</tr>
<tr>
<td>Comparators</td>
<td>1</td>
</tr>
<tr>
<td>Counter</td>
<td>Several</td>
</tr>
<tr>
<td>NAND gates</td>
<td></td>
</tr>
</tbody>
</table>

Several CPU Multipliers with one cell of memory Adders Dividers Memory vectors containing N+1 cells Row buffers Shift Registers Comparators Counter NAND gates

processors required for the Parallel Asynchronous Jacobi method and L-U decomposition with back substitution[14].

Table 4.3  Comparison of L-U Decomposition and the Parallel Asynchronous Jacobi Architecture's Hardware

<table>
<thead>
<tr>
<th>Architecture</th>
<th>Number of Processors</th>
<th>Worst Case</th>
</tr>
</thead>
<tbody>
<tr>
<td>L-U Decomposition and Back Substitution[14]</td>
<td>p*q + p</td>
<td>N^2 + N</td>
</tr>
<tr>
<td>Parallel Asynchronous Jacobi</td>
<td>Z - multipliers</td>
<td>N^2 - multipliers</td>
</tr>
</tbody>
</table>

In this table, Z again stands for the number of nonzero elements in the matrix. This comparison shows that if the matrix is not banded, p and q are equal to N, then L-U decomposition requires N^2 + N processors. Most methods don't work well with unstructured sparse matrices, since they must treat them as though they were dense. Other methods that must do this are Choleski's method[10], Gauss elimination (see Chapter 2), and the Q-R method (see Chapter 3). A possible way that has been proposed to alter these methods is to reorder the matrix into a banded matrix, but this requires additional hardware and computational time.
If the matrix is treated as dense, the algorithms require the most hardware and the most time to complete; in other words, it's their worst case. The Parallel Asynchronous Jacobi architecture requires extra time for the CPU to reorganize its buses, but, once this is accomplished, it can handle any diagonally dominant matrix, and it can solve an unstructured sparse matrix very quickly. This method does require more processing elements for a narrowly banded matrix, but it still solves it fast. One architecture that does require a similar number of processors for the unstructured case is C. Pottle's L-U decomposition[11]. This method stores only the nonzero elements, but it performs L-U decomposition, which continually creates more nonzero elements. It seems difficult to guess where another nonzero element will be created, and it also requires more processors that must wait for these elements to be created. So, for the unstructured sparse case, this architecture requires less hardware than the well-known methods.

4.4 Similar Methods

Other methods that require similar hardware are the Jacobi Over Relaxation method and the Parallel Conjugate Gradient method. These are both iterative methods for solving sparse linear equations. The Jacobi method was discussed in detail because it is simpler than these methods. All these methods use a guessing technique of finding the solution. There are other similar methods, but these were chosen for their variety and/or because they have a strong advocacy of their advantages.
4.4.1 Parallel Conjugate Gradient Method

The Parallel Conjugate Gradient method was chosen for both the previously discussed reasons. It is a method that differs because of its stability. This method is very close to being a direct method because it is supposed to converge within N iterations[2]. Whether it converges within N iterations or not will be determined in section 4.5. Loyce Adams is a strong advocate of this method. She has proposed an architecture of microprocessor arrays for this method[2]. In particular, this method was chosen for comparison reasons because Loyce Adams advocates it over the Parallel Jacobi method.

As stated earlier, this is a stable method. It is this way because it normalizes the vectors in the system. The algorithm of this method is given below:

For the system \( Ax = b \),

1) Choose \( x(0) \)
2) \( r(0) = b - Ax(0) \)
3) \( p(0) = r(0) \)
4) \( k = 0 \)
5) For \( k = 0,1,\ldots \), \( k_{\text{max}} \)
   1) \( \alpha = ((\|r(k)\|_2^2)/(p(k)^T A p(k))) \)
   2) \( x(k+1) = x(k) + \alpha p(k) \)
   3) If \( \|x(k+1) - x(k)\|_\infty < \xi \) then stop
   4) \( r(k+1) = r(k) - \alpha A p(k) \)
This algorithm is rather complicated, and requires more hardware than the Jacobi method. Most of the extra calculations are to form coefficients which will normalize the operations. This makes the process very stable, and, hopefully, makes it converge faster. The name of the method and the above algorithm suggest that a type of gradient is calculated so that the next guess will be a better approximation of the solution than by Jacobi's method. Thus, the Parallel Conjugate Gradient method requires more operations for each iteration, which uses more execution time and requires more hardware, but, since this method guesses better at a solution, is it better than the Parallel Asynchronous Jacobi method? The answer to this question will be discussed in section 4.5.

4.4.2 JOR Method

This method was chosen to give a variety of techniques used to accelerate iterative methods. This method involves choosing a coefficient to multiply the iterated value by to have a closer approximation of the solution. The algorithm is given below:

\[ x_i = \sum_{j \neq i}^N a_{ij} x_j + (1-\omega) x_i^{old} + \omega c \quad c \text{ is a constant} \]

The coefficient \( \omega \) is chosen between 0 and 2, and it can be shown that, when \( \omega \) equals one, this method is the same as the Jacobi method. The problem associated with this method is guessing a value for \( \omega \). The best value for \( \omega \) is dependent on the smallest eigenvalue. This is
usually not known. So, only on special systems, where the matrix remains the same for several problems, it is good to use this method. Therefore, this algorithm will not be shown or compared in parallel with the other methods. This method is good to study to see how some methods can be accelerated, which may eventually be of use on special systems.

4.5 Performance

So, how good are these methods? For comparison, the Parallel Asynchronous Jacobi method, the Parallel Gauss elimination, and Parallel Conjugate Gradient method have been simulated with FORTRAN programs listed in Appendices A, B, and C, respectively. One direct method was used to show how these methods compare to indirect methods. Also, the comparison between Jacobi's method and the Conjugate Gradient method is of use to show whether the extra hardware to gain stability will actually decrease the number of iterations significantly enough to be even faster than the Jacobi method.

The first example is Example Problem 1, which is discussed in section 4.2. It is an unsymmetric, unstructured, sparse system. Table 4.4 shows a comparison of the three methods on this problem. Initial guesses for the Parallel Asynchronous Jacobi method and the Parallel Conjugate Gradient method were chosen to be zero for each element. The radius of convergence was set equal to 1.00E-05. For this particular example, Parallel Gauss elimination takes about half as much time as the Parallel Asynchronous Jacobi method. Thus, the Parallel Asynchronous Jacobi method must be used on a much less than 50% sparse
system to take advantage of its speed. The Jacobi method, though, does require less hardware for this problem; 32 multipliers to 72 multipliers for Parallel Gauss elimination. The Parallel Conjugate Gradient method was unable to converge on a solution, probably because it was an unsymmetric matrix. So, for this example problem, it is very close to decide which method is best; though, automatically, the Parallel Conjugate Gradient method can be ruled out.

Depending on the importance of speed compared to size, both Gauss elimination (twice as fast) and the Parallel Asynchronous Jacobi method (twice as small) are fairly good methods.

Example Problem 2 is shown in Fig. 4.8.

\[
\begin{pmatrix}
1.00 & 0.99 \\
0.99 & 0.98
\end{pmatrix}
\begin{pmatrix}
1.00 \\
1.00
\end{pmatrix}
= 
\begin{pmatrix}
1.99 \\
1.97
\end{pmatrix}
\]

Fig. 4.8 Example Problem 2

This problem was taken from A Survey of Numerical Mathematics, Volume II, [17]. It is a simple problem, but it can cause errors
because the elements are very close in values. The results of the three simulator programs are listed in Table 4.5.

Table 4.5 Results of Example Problem 2
(Radius of Convergence is 1.00E-05)

<table>
<thead>
<tr>
<th>Method</th>
<th>Additions</th>
<th>Multiplications</th>
<th>Divisions</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parallel Gauss Elimination</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>39</td>
</tr>
<tr>
<td>Parallel Asynchronous Jacobi</td>
<td>4</td>
<td>2</td>
<td>2</td>
<td>28</td>
</tr>
<tr>
<td>Parallel Conjugate Gradient</td>
<td>14</td>
<td>14</td>
<td>5</td>
<td>101</td>
</tr>
</tbody>
</table>

The Jacobi method is the fastest and it requires about the same hardware as Gauss elimination. The Conjugate Gradient method did converge, though it did take longer than the other methods. The Jacobi method was more accurate than the other two methods, being exact to 16 digits, while the other two methods were exact to 13 digits. For this particular example, the Parallel Asynchronous Jacobi method is the best.

Another example was run on these three programs to see how well they can solve an ill-conditioned example. Example problem 3, [17], is given in Fig. 4.9.

\[
\begin{bmatrix}
0.000003 & 0.215512 & 0.173257 \\
0.213472 & 0.375623 & 0.663257 \\
0.332147 & 0.476625 & 0.625675 \\
\end{bmatrix} \begin{bmatrix}
x \\
\end{bmatrix} = \begin{bmatrix}
-0.991289 \\
0.532039 \\
0.674121 \\
\end{bmatrix} \begin{bmatrix}
0.235262 \\
0.127653 \\
0.285321 \\
\end{bmatrix}
\]

Fig. 4.9 Example Problem 3
The element \( a_{11} \) is very small compared to the other elements. This causes the matrix to be ill-conditioned, because it is a diagonal element. The results are listed in Table 4.6.

Table 4.6 Results of Example Problem 3
(Radius of Convergence is 1.00E-05)

<table>
<thead>
<tr>
<th>Method</th>
<th>Additions</th>
<th>Multiplications</th>
<th>Divisions</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parallel Gauss Elimination</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>65</td>
</tr>
<tr>
<td>Parallel Asynchronous Jacobi</td>
<td></td>
<td></td>
<td></td>
<td>ARITHMETIC OVERFLOW</td>
</tr>
<tr>
<td>Parallel Conjugate Gradient</td>
<td></td>
<td></td>
<td></td>
<td>DOES NOT CONVERGE</td>
</tr>
</tbody>
</table>

Only Gauss elimination was able to solve the system of equations. The Jacobi method had an arithmetic overflow because of the way the algorithm handled the small pivot element. The Conjugate Gradient method failed to converge, but it was able to normalize the calculations so it wouldn't overflow. So, Parallel Gauss elimination seems to work the best on a small, ill-conditioned problem.

For the final comparison of the three methods a passive circuit was solved. In Fig. 4.10 is the passive circuit; this circuit is the combination of several Cauer 1 and Cauer 2 realizations given in Analog and Digital Filters: Design and Realization[16]. The systems of equations is sparse, symmetrical and banded. The matrix representation is given in Fig. 4.11. This system contains imaginary numbers. Two methods can be used to solve complex numbered systems: 1) double the number of variables and double the number of equations, by creating separate equations for the real part and imaginary part of
Fig. 4.10  Passive Circuit Used in Example Problem 4[16]
\[
\begin{array}{cccccccccccc}
(5,100) & (-4,0) & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & (10,0) \\
(-4,0) & (6.4,160) & (-2.4,0) & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & (-2.4,0) & (3.2,80) & (-0.8,0) & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & (-0.8,0) & (.9143, 22.86) & (-.1143, 0) & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & (-.1143, 0) & (8.75, 269.5) & (0, -266.7) & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & (0, -266.7) & (4.57, 446.3) & (0, -179.6) & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & (0, -179.6) & (49.10, 1746.3) & (-3, 0) & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & (-3, 0) & (3,220) & (0,-70) & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & (0,-70) & (1.36, 73.601) & (0, -3.601) & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & (0, -3.601) & 3.601 & 0 \\
\end{array}
\]

Fig. 4.11 Matrix Representation of Example Problem 4
each variable, or 2) change the simulation program to use complex representation of numbers. The second method was chosen. The results are given in Table 4.7.

Table 4.7 Results of Example Problem 4
(Radius of Convergence is 1.00E-05)

<table>
<thead>
<tr>
<th>Method</th>
<th>Additions</th>
<th>Multiplications</th>
<th>Divisions</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parallel Gauss Elimination</td>
<td>19</td>
<td>19</td>
<td>19</td>
<td>247</td>
</tr>
<tr>
<td>Parallel Asynchronous Jacobi</td>
<td>40</td>
<td>20</td>
<td>20</td>
<td>280</td>
</tr>
<tr>
<td>Parallel Conjugate Gradient</td>
<td>34</td>
<td>20</td>
<td>7</td>
<td>217</td>
</tr>
</tbody>
</table>

A comparison of the three methods yields the Parallel Conjugate Gradient method as being the fastest, and Parallel Gauss elimination is second fastest. The execution times of this example should not be compared to other examples because timing is for execution of these operations on complex numbers, which requires more time.

So, now after trying simulations on large, small, ill-conditioned, sparse, dense, symmetric, and unsymmetric systems the Parallel Asynchronous Jacobi method is competitive with the well-known methods on the basis of execution time. Also, as stated earlier, this method requires less hardware than these other methods on unstructured, sparse matrices; therefore, the Parallel Asynchronous Jacobi method is an excellent method to use on unstructured, sparse matrices.
V. MULTIPLICATION

Since there is a need to solve large systems of linear equations very rapidly, all parts of the architecture and algorithm should be investigated for the possibility of increasing speed. The focus of this thesis has been on the efficiency, amount of hardware versus speed, of the algorithms and macroscopic parts of the architecture; but, now, the focus is on microscopic parts of the architecture, in particular, the multiplier unit. This unit must be very fast because it is used very often, some 20-100 times, to solve a linear system. Also, the multiplier must be small because there may be need to have over 100 of them on one chip. So, this eliminates the possibility of using look-up tables, which require a chip or two of memory. Thus, a multiplication algorithm must be used, which, most likely, is to be highly parallel. Three algorithms are to be discussed which were chosen by the author because of their high speed and their size. These three methods are binary tree multiplication, array multiplication with carry-save adders, and array multiplication with 3-D carry save adders.

These three multipliers are similar in the way they create product terms. They use an array type multiplication. An array is created of shifted multiplicand terms, as is shown in Fig. 5.1. Since this is binary arithmetic, each product term is either zero or a shifted multiplicand. This array of terms can be created in one step, if it is done in parallel. This saves time over other techniques that simply shift and add. The three methods differ in how they sum the product terms.
Fig. 5.1 Array Multiplication

5.1 Binary Tree Multiplier

The binary tree multiplier sums the product terms by using a binary tree of parallel adders, as is depicted in Fig. 5.2.

Fig. 5.2 Binary Tree of Parallel Adders

The parallel adders are of the type discussed by Jean Vuillemin and Leonidas Guibas[15], which require \(O(N\log_2 N)\) area and \(O(\log_2 N)\) time, where \(N\) is the number of bits. If \(M\) is the number of bits in both the
multiplicand and the multiplier, then, in this use of the parallel
adders, equation 5.1 is true.

\[ N = 2^M - 1 \quad (5.1) \]

The number of stages of adders is equal to \( \log_2 M \); this is shown in
Table 5.1.

Table 5.1 Comparison of the Number of Addition Stages Required to the
Number of Product Terms in a Binary Tree Multiplier

<table>
<thead>
<tr>
<th>No. of Addition Stages</th>
<th>Max. Number of Product Terms</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>8</td>
</tr>
<tr>
<td>4</td>
<td>16</td>
</tr>
<tr>
<td>5</td>
<td>32</td>
</tr>
<tr>
<td>6</td>
<td>64</td>
</tr>
<tr>
<td>7</td>
<td>128</td>
</tr>
<tr>
<td>8</td>
<td>256</td>
</tr>
<tr>
<td>9</td>
<td>512</td>
</tr>
<tr>
<td>10</td>
<td>1024</td>
</tr>
</tbody>
</table>

So, the total time necessary for the multiplication of two \( M \) bit num-
bers is \( O(\log_2 M^*(\log_2 (2^M - 1))) \), which is less than \( O(\log_2^2 M + \log_2^2 N) \).
The total multiplier unit is depicted in Fig. 5.3. This multiplier
unit can be made to work even faster if it doesn't have a
nonchangeable design and, instead, uses a little CPU to organize its
buses. If it has this capability, the multiplier term can be checked
for nonzero elements and be able to produce only nonzero product
terms. Then, the buses of the adders would be changed so that there
would be \( \log_2 (k) \), which is less than or equal to \( \log_2 (M) \), stages,
where \( k \) is the number of nonzeros. On the average, the number of stages
would be \( \log_2(M) - 1 \). Thus, the total time to multiply could be reduced. But, the complexity of reordering the buses and the complexity of designing the CPU suggest that this speed-up be sacrificed for simplicity. Thus, the binary tree multiplier would be best implemented as a processing unit as depicted in Fig. 5.3.

5.2 CSA Tree Multiplier

The multiplier that uses array multiplication with carry-save adders follows a scheme very similar to the binary tree multiplier. It produces an array of numbers it must add. The difference is that
carry-save adders are used to add these product terms. A carry-save adder, generally, takes in three inputs and produces two outputs. The three inputs are the product terms that are to be summed, and the two outputs are a moduli 2 addition and a carry term. Fig. 5.4 gives a scheme for the multiplication of two M-bit numbers.

Fig. 5.4 Addition of Product Terms by the Use of Carry-Save Adders

This reduces the inputs by one third in each stage of carry-save adders. This method does require one adder similar to the adders in the binary tree multiplication scheme; these adders are called carry-lookahead adders. They take in two numbers and produce one sum output. Thus, the addition of all the product terms takes many stages; Table 5.2 gives a few examples. Thus, a 16-bit multiplication requires at least 7 stages. Other configurations have been developed that require less hardware, but require more addition steps[8]; these methods use feedback. The CSA method with feedback works fairly well, and it is used in the IBM 360 mainframes. The array multiplication by carry-save adders accomplishes its reduction in product terms by the 3 to 2 carry-save adders.
Table 5.2  Comparison of the Number of Addition Stages Required to the Number of Product Terms in a CSA Multiplier

<table>
<thead>
<tr>
<th>No. of Addition Stages</th>
<th>Max. Number of Product Terms</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>6</td>
</tr>
<tr>
<td>5</td>
<td>9</td>
</tr>
<tr>
<td>6</td>
<td>13</td>
</tr>
<tr>
<td>7</td>
<td>19</td>
</tr>
<tr>
<td>8</td>
<td>28</td>
</tr>
<tr>
<td>9</td>
<td>42</td>
</tr>
<tr>
<td>10</td>
<td>63</td>
</tr>
</tbody>
</table>

5.3 3-D CSA Tree

A variation of this scheme for multiplication is array multiplication with 3-D carry-save adders. The difference in the described adders is depicted in Fig. 5.5.

![Fig. 5.5 Three Adders: a) Binary Tree Adder (Parallel Adder), b) CSA, and c) 3-D CSA](image)

The parallel adder used in binary tree multiplication is depicted in Fig. 5.5a. This adder is simple; it sums two numbers and outputs the sum. The CSA in Fig. 5.5b inputs three numbers and outputs the moduli-2 sum and the carry. The 3-D CSA in Fig. 5.5c inputs seven numbers and outputs three numbers: the moduli-2 sum, the primary
carry, and the secondary carry. It was named the 3-D CSA by the author of this thesis because of its three outputs which provide three "dimensions" of N-bit information about the actual sum. This adder was developed in the hope that its reduction in the number of adders and the number of addition stages required to add N partial products to form a multiplier would outweigh its cost in internal hardware.

The 3-D carry-save adder takes seven N-bit numbers, adds them, and produces three N-bit outputs that must be added. A scheme of how the outputs are produced is given below in Fig. 5.6.

\[
\begin{array}{ccc}
1 & 0 & 1 \\
C_{2i} & C_{1i} & S_i \\
\end{array}
\]

Fig. 5.6 Example of How Carry Bits and Sum Bits Are Determined in a 3-D Carry-Save Adder

This shows the seven numbers are just added together, and \( C_{2i} \), secondary carry term bit \( i \), gets a one, the primary carry bit \( i \), \( C_{1i} \), gets a zero, and the sum bit \( i \), \( S_i \), gets a one, for a seven term sum of bit \( i \) equal to five.

To design a multiplier that uses these 3-D carry-save adders, one regular carry-save adder and one carry-lookahead adder is required, as is shown in Fig. 5.7. This shows how the product terms are added, once they are produced by a shifted multiplicand array. The system in Fig. 5.7 adds 15 product terms together. It uses only four stages, as compared to seven for regular carry-save adders and four for binary tree adders. Table 5.3 gives a comparison between the number of sta-
Fig 5.7 15 Term Adder Used In a Array Multiplier with 3-D Carry-Save Adders

Table 5.3 Comparison of the Number of Addition Stages Required to the Number of Product Terms in a 3-D CSA Multiplier

<table>
<thead>
<tr>
<th>No. of Addition Stages</th>
<th>Max. Number of Product Terms</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>..................................7</td>
</tr>
<tr>
<td>4</td>
<td>..................................15</td>
</tr>
<tr>
<td>5</td>
<td>..................................35</td>
</tr>
<tr>
<td>6</td>
<td>..................................79</td>
</tr>
<tr>
<td>7</td>
<td>..................................183</td>
</tr>
<tr>
<td>8</td>
<td>..................................427</td>
</tr>
<tr>
<td>9</td>
<td>..................................995</td>
</tr>
<tr>
<td>10</td>
<td>..................................2319</td>
</tr>
</tbody>
</table>
ges in a 3-D carry-save adder tree and the maximum number of product
terms that can be added. After six stages, the maximum number of pro-
duct terms becomes idealistic, since the errors due to the word size
would be too great for a reasonable amount of accuracy. Though, as
the number of stages increases, it can be seen that the number of pro-
duct terms increases faster than either the binary tree adder or the
regular CSA. This is its big advantage, less stages are needed.

The only drawback of this method is the question of how the 3-D
CSA can be designed to have a speed and a size similar to the other
two adders. This is a definite problem which the author hasn't been
able to answer. There are a few approaches, but most are either too
big or too slow. One approach is to build the 3-D CSA with seven-bit-
slice Wallace trees. But, as shown in [8] on page 166, these Wallace
trees are made of four regular CSA in a three stage tree. This makes
the delay of the 3-D CSA at least equal to three times the time for a
regular CSA to add, and it would be more compact to design the whole
addition by CSA than to use this type of architecture. Another method
would be a PLA realization. This architecture requires 127 product
terms, since, by using two software reduction schemes for VLSI design,
PRESTO and BLAM by University of California at Berkeley, a reduction
of a possible 128 inputs was only by one. PRESTO reduces truth tables
to a regular PLA, and BLAM unsuccessfully tried to fold the PLA to try
to reduce the area. Another possibility is to design the 3-D CSA by a
logic design. The sum bit could be realized with six exclusive-OR
gates, and the secondary carry bit could be realized with 35 NAND
gates, wire-ANDing, and an inverter. But, how do you realize the pri-
mary carry? It can be done with a PLA, but it requires 48 product terms after reduction. This is large, but not as large as the other PLA. If an architecture can be developed that is smaller and faster than these architectures, this method, based on 3-D CSA, probably is the best.

5.4 Performance

Which design of a multiplier should be used in a VLSI chip that is dedicated to the solving of large-scale linear systems? All three methods seem good; from the two methods developed by the author, which are the binary tree multiplier and the 3-D CSA tree multiplier, as well as to the regular CSA tree used in the IBM 360. The binary tree multiplier is the simplest, though the adders are the second simplest. The CSA multiplier has the simplest elements, but requires the most stages. The 3-D CSA requires the least stages, but may be too big or too slow, depending on the architecture. A comparison is given in Table 5.4: the number of stages versus the maximum number of terms that can be added.

Table 5.4 Comparison of Binary Tree, CSA, and 3-D CSA Stages to the Maximum Number of Terms that can be Summed

<table>
<thead>
<tr>
<th>Number of Stages</th>
<th>Maximum Number of Terms that can be Summed for:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Binary Tree</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>8</td>
</tr>
<tr>
<td>4</td>
<td>16</td>
</tr>
<tr>
<td>5</td>
<td>32</td>
</tr>
<tr>
<td>6</td>
<td>64</td>
</tr>
<tr>
<td>7</td>
<td>128</td>
</tr>
<tr>
<td>8</td>
<td>256</td>
</tr>
<tr>
<td>9</td>
<td>512</td>
</tr>
<tr>
<td>10</td>
<td>1024</td>
</tr>
</tbody>
</table>
Another important factor is the area. Table 5.5 gives an estimate for area of two architectures for 3-D CSA and one for a regular CSA for comparison.

Table 5.5 Area Required for Different Adders

<table>
<thead>
<tr>
<th>Number of Product Terms for PLA</th>
<th>Type</th>
<th>Area per Bit</th>
<th>Additional Hardware</th>
</tr>
</thead>
<tbody>
<tr>
<td>128</td>
<td>3-D CSA</td>
<td>$2159 \lambda^2$</td>
<td>None</td>
</tr>
<tr>
<td>48</td>
<td>3-D CSA</td>
<td>$960 \lambda^2$</td>
<td>35 NANDs, INV, and 6 exclusive-ORs</td>
</tr>
<tr>
<td>8</td>
<td>CSA</td>
<td>$56 \lambda^2$</td>
<td>None</td>
</tr>
</tbody>
</table>

This shows why, currently, the 3-D CSA needs a more compact logic design, if it is going to compete with the CSA. So, two methods seem very good and can be produced, the binary tree multiplier and CSA multiplier. But, the third multiplier, that uses 3-D CSA, needs a small, fast architecture before it is at this stage. Thus, the best method, without further data to compare adder area and adder time, is probably the binary tree array multiplier, or the CSA tree array multiplier, because of their simplicity. Further research must be done in this area, though, since both area and time are of great importance, especially when 100-1000 multiplier units are used and 10-100 multiplication steps are performed in one solution of a large-scale linear system of equations.
VI. CONCLUSION

This thesis has shown many methods for solving large-scale linear systems, and has also shown several techniques of multiplying to reduce size and time necessary. But, how should this information be used? There are several decisions that must be made to choose an appropriate method to solve one's particular system. This final chapter will address this decision making.

6.1 Decision of System Type

The first decision is concerned with the type of system that is to be solved. The user of these architectures must classify his system. For instance, a person who always is going to be solving linear, passive, electronic circuits will always have a diagonally dominant sometimes sparse system, if he uses standard techniques for labeling and solving his circuit. If he chooses his equations randomly, he cannot guarantee diagonal dominance. So, the user must define his task and the method he is going to use to produce his system of equations. Once he has decided this, he can limit the number of types of systems he needs to be able to solve. For instance, if he does have only equations that form very much more than 50% sparse, unsymmetric, unstructured systems, he should use the Parallel Asynchronous Jacobi method. If he has dense systems, he should consult Chapter 2, which shows several direct methods. So, the user must define and limit the types of systems he will be solving, and then consult the appropriate chapter of the thesis so he can narrow his choice to one architecture.
6.2 Decision of Speed vs. Hardware

Another decision that influences which architecture is best is the choice of comparative value of hardware versus speed. In Chapters 2, 3, and 4, comparisons have been given of several methods' efficiency. Efficiency was defined as the inverse of the product of hardware and execution time. This gave a rough comparison of the methods discussed. To get a good enough comparison needed to choose an architecture, one needs to change this efficiency factor to show the need for speed, as compared to need for a small size, which implies small cost. If the user has unlimited funds and hardware, he should be concerned only with time. If, instead, he has unlimited time to solve his system, then he should find the smallest, cheapest architecture. Neither of these situations usually occurs, and the user must balance his need for speed and his limited funds. After he has determined his limitations, the tables in Chapters 2, 3, and 4 should be very valuable in choosing an architecture that fits his time and budget.

6.3 Decision of How Much Adaptability vs. Speed

The final decision that must be addressed is how much adaptability is needed versus the need for speed. This is similar to the first decision, but it is a little more involved with the calculations of speed for every type system that must be solved. The question of which system types need to be solved may yield an answer of more than one system. Can sacrifices in execution time be tolerated for a certain system type, so as to save money in not building an additional architecture to solve this system type separately? For instance,
Gauss elimination is best on dense and banded matrices, but is slower and requires more hardware for unstructured and very sparse matrices. A user needs both dense matrices solved and unstructured and very sparse matrices to be solved. The question he must ask himself is: Can I accept a system that may have a long execution time for the sparse matrix, but is very good on a dense matrix? If he can, Gauss elimination may be the architecture he should use. If he can't, then one answer is to have two architectures, Gauss elimination and the Jacobi method. So, the user must balance his speed and hardware requirements with the adaptability of the architecture to several system types.

After the user has made these decisions, this thesis will be a good reference manual from which to choose an appropriate architecture for his problem. In the future, the hope is that all these architectures will be completely designed, fabricated, and readily available to interface with the engineer's or scientist's host computer so that his particular problem can be solved efficiently.
References


APPENDIX A: PARALLEL ASYNCHRONOUS JACOBI SIMULATION PROGRAM
PARALLEL ASYNCHRONOUS JACOBI METHOD
for solving large-scale linear systems

This program simulates the Parallel Asynchronous
Jacobi method for solving a system of linear
equations.

THE LINEAR SYSTEM IS OF THE FORM A\times x = b.
A is a given matrix; b is a given vector, and x is to
be solved. The system is assumed to be sparse; so,
only the nonzero elements of A are stored in memory.

INPUT:  I/O device number 2, which corresponds to
        file FOR002.DAT on the VAX 11/750, is used
to input real values. This starts with the
        the nonzero elements of matrix A, row by row,
        followed by vector b, and then an initial
        guess at vector x. Elements can be placed
        in any format with a comma or a space between
        them as long as they follow this order.
        I/O device number 3, which corresponds to
        file FOR003.DAT on the VAX 11/750, is used
to input integer values. First the column
        pointer vector is stored; this vector shows
        which column the corresponding nonzero element
        in matrix A is located in. Next, the vector
        that contains the ending address of the rows
        nonzero elements that is stored in file 2.
        Another I/O device can be altered optionally;
        this device, 4 or FOR004.DAT, contains a key
        number to tell whether to limit the iterations
        by radius of convergence (input a 1), or by
        number of iterations (input a 2). This file also
        contains the radius of convergence or the
        number of iterations.

OUTPUT: I/O device number 7, which corresponds to
        file FOR007.DAT on the VAX 11/750, is used
to output the radius of convergence or
        the limit on the number of iterations; the
        solution vector and the number of times each
        row processor added, multiplied, and divided.

MAIN PROGRAM

IMPORTANT CONSIDERATIONS IN WRITING THE MAIN PROGRAM:
The main program needs two vectors dimensioned: one for real numbers, RVEC, and one for integers, IVEC. The dimensions of these vectors are calculated as follows:

\[ \text{RDIMEN} = \text{number of nonzero elements of matrix } A \text{ of } A(N \times N) + 5 \times \text{dimension } N \]

\[ \text{IDIMEN} = 8 \times \text{dimension } N + \text{number of nonzeros} \]

RDIMEN stands for the dimension of RVEC, the real vector. IDIMEN stands for the dimension of IVEC, the integer vector. These two dimensions are used to dimension two vectors, IVEC and RVEC, and are then used to call the subroutine BUFFER. The subroutine has four parameters:

- RDIMEN: The dimension of the real vector
- IDIMEN: The dimension of the integer vector
- RVEC: The real vector
- IVEC: The integer vector

REAL*8 RVEC(72)
INTEGER IVEC(96)
CALL BUFFER(72, 96, RVEC, IVEC)
STOP
END

SUBROUTINE BUFFER(RDIMEN, IDIMEN, RVEC, IVEC)

BUFFER subroutine

BUFFER controls the dimensioning of the arrays for real numbers and integer numbers. Also, pointers are calculated to determine where the vectors and arrays will be located within the real vector and the integer vector. Then, the SOLVER subroutine is called to handle the creation of the necessary vectors.

INTEGER RDIMEN, IDIMEN
REAL*8 REAVEC(RDIMEN)
INTEGER INTVEC(IDIMEN)
NNN = (IDIMEN - RDIMEN) / 3
MM=(IDIMEN-8*NNN)
READ(2,*,END=3) REAVEC
READ(3,*,END=5) INTVEC
CONTINUE

Pointers are calculated to determine where each array begins in REAVEC or INTVEC. IAD_ is an
address pointer for REAVEC. INTAD_ is an
address pointer for INTVEC.

IAD1=1
IAD2=1+MM
IAD3=NNN+IAD2
IAD4=NNN+IAD3
IAD5=NNN+IAD4
IAD6=NNN+IAD5
INTAD1=1
INTAD2=1+MM
INTAD3=INTAD2+NNN
INTAD4=INTAD3+NNN
INTAD5=INTAD4+NNN
INTAD6=INTAD5+NNN
INTAD7=INTAD6+NNN
INTAD8=INTAD7+NNN
INTAD9=INTAD8+NNN
CALL SOLVER(N,MM,REAVEC(IAD1),REAVEC(IAD2),
*REAVEC(IAD3),REAVEC(IAD4),REAVEC(IAD5),INTVEC(INT
*AD1),INTVEC(INTAD2),INTVEC(INTAD3),INTVEC(INTAD4)
*INTVEC(INTAD5),INTVEC(INTAD6),INTVEC(INTAD7),INT
*VEC(INTAD8),INTVEC(INTAD9))
RETURN
END

SUBROUTINE SOLVER(N,NN,A_NONZ,B,XC,XT,XTT,
*INDCOL,HROWINFO,FLAG,NONZERO,ITEMP,LOG2,TIMELY,
*NCURTIM,HAXCLK)

SOLVER subroutine

SOLVER creates the vectors and arrays needed to solve the linear system. It also calculates the timings,
calls the other subroutines (ITER, SEARCH, and PRINT),
and checks to see if there has been no convergence.

INPUTS: N ........ dimension of matrix A (NxN)
NN ........ number of nonzero elements
in matrix A
A_NONZ ... vector containing nonzero elements
of matrix A, row by row.
B ........ elements of vector b
XC ........ initial guess at solution vector, x.
INDCOL ... column indeces, contains column each
element of A_NONZ is in matrix A.
MROWINFO . contains ending address of row's
nonzero elements stored in vector
A_NONZ.

INTERNAL
SYMBOLS: XT ........ first temporary location for solution
vector, #: stores iterate information
prior to updating the current solution
vector, XC,
XTT ........ secondary temporary location for
solution vector. Stores most recent
iterate,
FLAG ....... vector that contains convergence
information for each row, element
equals one to signify that row is
converging within the specified
radius.
NONZERO .. vector contains the number of
nonzero elements in each row of
matrix A.
ITEMF .... temporary location to store the
row number of row processors that
This information is used to update
all these elements of the solution
vector at one time.
LOG2 ...... vector contains the log base two of
each element of NONZERO
TIMDLY ... vector containing the time delay of
each row's processor to calculate
an iterate. Helps in simulating
the process as being asynchronous
and in parallel.
It is equal to the equivalent number
of addition steps to calculate a
a row's iterate.
NCURTIM .. equals the current time of each
row's processor clock. Units are
in equivalent time to perform an
addition.
MAXCLK ... vector that limits the time each
row clock can have. Is used to
set a limit on the number of
iterations. Its default limiter is
1000 iterations.
LIMIT .... equals one if the limitation is to
be by means of radius of convergence;
equals two if the limitation is to be by the number of iterations.

ACC ...... when its initially read it can have two meanings depending on LIMIT. Its the radius of convergence or the limiting number of iterations. In other subroutines its always a radius of convergence.

NO-IT .... is the limiting number of iterations which is 1000 unless otherwise specified.

REAL*8 A_NONZ(NN),B(N),XC(N),XT(N),XTI(N)
INTEGER MROWINFO(N),FLAG(N),NONZERO(N),ITEMP(N)
INTEGER LOG2(N),TIMELY(N),NCURTIM(N)
INTEGER INDCOL(NN),MAXCLK(N)
REAL*8 ACC
READ(4,*) LIMIT,ACC,MULT,DIV

Calculate the number of nonzero elements in each row to use to calculate timings of row processors. Do this by using information from where the row information is stored.

NONZERO(1)=MROWINFO(1)
DO 10 I=2,N
    NONZERO(I)=MROWINFO(I)-MROWINFO((I-1))
10
One is added to the nonzeros to account for the addition of bi.

DO 20 I=1,N
    NONZERO(I)=NONZERO(I)+1
20
A binary tree adder is used in addition. So to calculate the timings, the log base 2 of the number of elements is used.

DO 30 I=1,N
    RNONZERO=FLOATJ(NONZERO(I))
    TLOG2=ALOG(RNONZERO)/ALOG(2.0)
    ILOG2=JINT(TLOG2)
    RLOG2=FLOATJ(ILOG2)
    IF (RLOG2.EQ.TLOG2) GOTO 25
    LOG2(I)=ILOG2+1
    GOTO 30
30
    LOG2(I)=ILOG2
25
    LOG2(I)=ILOG2+1
    CONTINUE
Calculate new clock time by adding row's time
delay to its current time, which is in NCURTIM.
Then iterate.

\[
\text{NOLD} = \text{NCURTIM(NTERATE)}
\]
\[
\text{NCURTIM(NTERATE)} = \text{NOLD} + \text{TIMELY(NTERATE)}
\]

CALL ITER(N, ACC, NTERATE, NFLAGM, A_NONZ, B, XC, XTT, *
  INDCOL, FLAG, MROWINFO, NN, NCURTIM, NO_IT, TIMELY)

IF(NCURTIM(NTERATE), LT, MAXCLK(NTERATE)) GOTO 50

IF(LIMIT, NE, 2) GOTO 100

CALL PRINT(N, NN, A_NONZ, INDCOL, B, XC, NCURTIM, *
  ACC, LIMIT, NO_IT, TIMELY)

STOP

100 WRITE (6,*) 'COUNT IS OVER 1000*N'
WRITE (6,*) 'NO CONVERGENCE'
RETURN
END

SUBROUTINE ITER(M, ACCURY, II, NFLAGSM, A_NONZ, BF, XCF, *
  XTTF, INDCOL, NFLAG, ROWINFO, MM, CLOCK, ITR, TIM)

ITERate subroutine

ITER gives the next iteration and checks for convergence.

INPUTS:
- M .......... dimension of matrix A (N x N)
- MM .......... number of nonzeros in A
- II .......... row to be iterated
- ACCUR ....... radius of convergence
- A_NONZ ...... nonzeros of matrix A, row by row,
- BF .......... vector b
- XCF ...... current solution vector used for
  the calculation of the next iterate
- INDCOL .. column index, corresponding to the
  column each element of A_NONZ is in
  matrix A.
- ROWINFO .. contains ending address of row's
  nonzero elements stored in vector A_NONZ
- NFLAG ...... vector that contains 1 for each row
  that is within the radius of convergence
- CLOCK ...... current time of each row processor.
  It's used in the print subroutine
  that is called.
- ITR ...... contains the limiting number of
  iterations. It's used in the print
subroutine.

OUTPUTS: 
XTTF .... contains most recent iterates of the solution vector.
NFLAG .... convergence information on each row
NFLAGSM ... sum of elements of NFLAG equals M when there is convergence.

REAL*8 A_NZ(MM),BF(M),XCF(M),XTTF(M),XA,ACCUR
REAL*8 SUM
INTEGER IND_COL(MM),ROWINFO(M),NFLAG(M)
INTEGER CLOCK(M),TIM(M)
SUM = 0.0

Compute starting and ending address of rows nonzero elements.

IF (II,NE,1) GOTO 1000
  ISTART_ADDR=1
  IEND_ADDR=ROWINFO(1)
  GOTO 1010
1000
  ISTART_ADDR=ROWINFO((II-1)+1)
  IEND_ADDR=ROWINFO(II)

Compute summation of Aij*Xj, and store the location of the diagonal element which is used to divide Bj minus the summation by.

1010 DO 1020 J=ISTART_ADDR,IEND_ADDR
  IPTR=IND_COL(J)
  IF (II.EQ.IPTR) GOTO 1015
  SUM=SUM+A_NZ(J)*XCF(IPTR)
  GOTO 1020
1015 ISAVE=J
1020 CONTINUE
  XA=(BF(II)-SUM)/A_NZ(ISAVE)

Check for convergence within the specified radius, ACCUR. The flag vector is then updated with this information, and the flag sum is also updated. If one was added to the flag sum then it is checked to see if it equals the number of equations. If it does, then there is convergence, so to print and stop; otherwise, continue.

IF(DABS(XA-XCF(II))).GT.ACCUR GOTO 1030
  IF(NFLAG(II).EQ.1) GOTO 1040
    NFLAGSM=NFLAGSM+1
  NFLAG(II)=1
  IF(NFLAGSM.NE.M) GOTO 1040
CALL PRINT(M,MM,A_NZ,IND_COL,BF,XCF, 
CLOCK,ACCRU,1,ITR,TIM) 
STOP 

1030 IF(NFLAG(II),EQ.1) NFLAGSM=NFLAGSM-1 
NFLAG(II)=0 
1040 XTTF(II)=XA 
RETURN 
END 

SUBROUTINE SEARCH(M,ITERATE,NCURTME,NEW) 

SEARCH subroutine 

SEARCH subroutine finds the next row to iterate and 
tells if the time on its clock is different than the 
last row's clock that was picked to iterate. 

INPUTS: M ........ dimension of vectors 
NCURTME .. current time of each row's processor 
clock 
NEW ........ last lowest clock time to be 
compared to see if it is the same 
as the one found this time the 
subroutine was called. 

OUTPUTS: ITERATE .. row that has the lowest clock 
NEW ........ equals one if the lowest clock found 
is different than before; otherwise, 
it equals zero. 

DIMENSION NCURTME(M) 

Search for lowest clock and store time in 
ILOWEST and store row number in ITERATE. 

ITERATE=1 
ILOWEST=NCURTME(1) 
DO 2000 IM=2,M 
  IF(ILOWEST.LE.NCURTME(IM)) GOTO 2000 
  ITERATE=IM 
  ILOWEST=NCURTME(IM) 
2000 CONTINUE 

See if it is the same time as the last time that 
was found, the last time search was called. Store
the answer in NEW.

IF(ILOWEST.EQ.NEW) GOTO 2010
NEW=1
GOTO 2020
2010 NEW=0
GOTO 2020
RETURN
END

SUBROUTINE PRINT(K,KK,AK,INDCOLK,BK,XK,TIMEK,
* ACCK,LIM,NO_ITER,DELAY)

PRINT subroutine

PRINT writes the solution vector, the elapsed time of each equation's clock, and the maximum number of iterations or the radius of convergence.

INPUTS: K ........ dimension of matrix A (KxK)
KK ........ number of nonzeros in A
AK ........ contains nonzeros of A
INDCOLK .... column index of AK
BK ........ vector b
XK ........ solution vector
TIMEK .... elapsed time of each equation
ACCK ...... radius of convergence
LIM ....... tells what was the limiter:
          one for the radius of convergence
          two for the number of iterations
NO_ITER .. limiting number of iterations

OUTPUTS: Output goes to I/O device number 7 or file FOR007.DAT. Output consists of solution vector, elapsed time of each equation's clock, and the maximum number of iterations or the radius of convergence.

REAL*8 AK(KK),BK(K),XK(K),ACCK
INTEGER INDCOLK(KK),TIMEK(K),DELAY(K)
INTEGER ADDK,MULTK,IDIVK
DO 3000 I=1,K
WRITE(7,3040) I,XK(I)
3000 CONTINUE
DO 3010 I=1,K
C Calculate time in additions, multiplications, and divisions.

MULTK = TIMEK(I) / DELAY(I)
IDIUK = MULTK
ADDK = TIMEK(I) - 3 * MULTK - 9 * IDIUK

WRITE(7, 3050) ADDK, MULTK, IDIUK, TIMEK(I)
IF (LIM.EQ.2) GOTO 3020
WRITE(7, 3060) ACCK
GOTO 3030

3020 WRITE(7, 3070) NO_ITER
3030 CONTINUE
RETURN

3040 FORMAT(T5, 'X', T7, I2, T9, '= ', T11, G22.15)
3050 FORMAT(T5, 'NUMBER OF ADDITIONS', T25, I5, T31, 'NUMBER
# OF DIVISIONS', T50, I5, T60, 'NUMBER OF MULTIPLICATIONS',
# T85, I5, T95, FOR ROW PROCESSOR', T112, I5, T120, 'TOTAL
*L TIME', T130, I4)
3060 FORMAT(T5, 'RADIUS OF CONVERGENCE IS', T32, G22.15
*)
3070 FORMAT(T5, 'NO. OF MAXIMUM ITERATIONS IS', T35, I3
*)
END
APPENDIX B: PARALLEL GAUSS ELIMINATION

SIMULATION PROGRAM
GAUSS ELIMINATION WITH BACK SUBSTITUTION
WITHOUT PIVOTING

SOLVES THE LINEAR SYSTEM OF \( A\mathbf{x} = b \)
\( A \) is a given matrix; \( b \) is a given vector, and \( \mathbf{x} \) is to be solved. Both the square matrix \( A(N \times N) \) and the vector \( b(N) \) are stored together in augmented matrix \( A(N \times N+1) \); \( \mathbf{x} \) will take the position of vector \( b \) in column \( N+1 \) of \( A \).

To be able to compare this method to other parallel methods, a table is given of the execution time. This table consists of the number of additions, multiplications, and divisions that take up the actual execution time if the Gaussian elimination were actually done on a parallel systolic array.

**INPUT:** I/O device number 10 is used for input; on the VAX 11/750 this corresponds to file FOR010.DAT. First matrix \( A \) is input column by column leaving a space between elements or a comma. Then vector \( b \) is input in the same manner.

**OUTPUT:** I/O device number 17 receives the output which is file FOR017.DAT on the VAX 11/750. This file contains the timings and the solution.

**MAIN PROGRAM**

**IMPORTANT CONSIDERATIONS IN WRITING THE MAIN PROGRAM:**

The main program must be written with a double precision real vector having a dimension equal to \( N \times N+1 \), where \( N \) is the dimension of the square matrix \( A(N \times N) \). Then this dimension followed by the name of the vector must be used in a call statement calling subroutine SOLVER.

```plaintext
REAL*8 RVEC(420)
CALL SOLVER(420,RVEC)
STOP
```
SUBROUTINE SOLVER(MM, A_SOL)

SOLVER subroutine

SOLVER handles the calling of the routines that solve the linear system of equations and prints the results. Also dimension of the array is calculated and data read in from I/O device 10.

REAL*8 A_SOL(MM)
INTEGER MULT_S/0/, DIV_S/0/, ADD_S/0/
READ (010,*) A_SOL
N_SOL=JNINT((SORT(1.0+FLOAT((4*MM)))-1.0)/2.0)
CALL GAUSS(N_SOL, A_SOL, MULT_S, DIV_S, ADD_S)
CALL PRINT(N_SOL, A_SOL, MULT_S, DIV_S, ADD_S)
RETURN
END

SUBROUTINE GAUSS(N, A, MULT, DIV, ADD)

GAUSS elimination and back substitution subroutine

GAUSS solves the linear system Ax=b by Gauss elimination and back substitution.

Time is computed as if the equations were solved in parallel. Parallelism occurs in calculating all the elements in the pivot row and in adding the pivot elements to all the rows.

KK stands for the pivot row, I the row being operated on, and J the column being operated on. MULT is the number of multiplication steps that take up execution time, the other multiplications occur in parallel and need not be added in since this is just to see what an approximate of the execution time is. DIV is the number of division steps that take up execution time and ADD is the number of addition steps that take up execution time.
REAL*8 A(N,(N+1))
INTEGER MULT,DIV,ADD

Gauss elimination

DO 20 KK=1,N-1
  DO 10 I=KK+1,N
    DO 10 J=KK+1,N+1
      A(I,J)=A(I,J)-(A(I,KK)/A(KK,KK))*A(KK,J)
      MULT=MULT+1
      DIV=DIV+1
    10 ADD=ADD+1
  20
 BACK substitution

DO 40 K=1,N
  KK=N-K+1
  A(KK,(N+1))=A(KK,(N+1))/A(KK,KK)
  DO 30 I=1,KK-1
    A(I,(N+1))=A(I,(N+1))-A(I,KK)*A(KK,(N+1))
    MULT=MULT+1
    DIV=DIV+1
  40 ADD=ADD+1
RETURN
END

SUBROUTINE PRINT(N_PRINT,A_PRINT,MULT_P,DIV_P,
                ADD_P)

PRINT subroutine

PRINT outputs data to I/O device 17, which is FOR017.DAT
This file contains the time and the solution.

REAL*8 A_PRINT(N_PRINT,(N_PRINT+1))
INTEGER MULT_P,DIV_P,ADD_P
WRITE(017,50)
WRITE(017,51)
WRITE(017,52) ADD_P,MULT_P,DIV_P
DO 45 I=1,N_PRINT
  WRITE(017,60) I,A_PRINT(I,(N_PRINT+1))
45 RETURN
FORMAT(T5,'THE NUMBER STEPS TO SOLVE THIS SYSTE
#M IS')
51 FORMAT(T7,'ADD',T23,'MULT',T35,'DIV')
52 FORMAT(T6,I4,T23,I4,T34,I4)
60 FORMAT(T10,'X',T11,I2,T14,'=','T16,G22.15)
$
APPENDIX C: PARALLEL CONJUGATE GRADIENT SIMULATION PROGRAM
PARALLEL CONJUGATE GRADIENT METHOD

SOLVES THE LINEAR SYSTEM OF \( A \cdot x = b \)

Using the parallel conjugate gradient as developed by Joyce Adams, this linear system is solved where \( A \) is a given \( M \times N \) matrix, \( b \) is a given vector, and \( x \) is to be solved for. The number of additions, multiplications, and divisions that affect execution time are stored to be able to compare execution time to other methods.

INPUTS: I/O device number 29, which corresponds to file FOR029.DAT on the VAX 11/750, is used to input matrix \( A \) column by column, vector \( b \), and an initial guess at the solution vector, \( x \). I/O device number 21 is used to specify the radius of convergence.

OUTPUT: I/O device number 39, which corresponds to file FOR039.DAT on the VAX 11/750, is used to output the solution followed by the radius of convergence and then the timings which is done by the equivalent number of divisions, multiplications, and additions to solve the linear system by using the conjugate gradient method executed in parallel.

Main Program

IMPORTANT CONSIDERATIONS IN WRITING THE MAIN PROGRAM:

Main Program must have a double precision real vector that has a dimension equal to \( 6 \times N + N \times N \), where \( N \) is the dimension of square matrix \( A \) (\( N \times N \)). Then subroutine buffer is called with this dimension and the name of the real vector.

```plaintext
REAL*8 RVEC(520)
CALL BUFFER(520,RVEC)
STOP
END
```

SUBROUTINE BUFFER(DIMEN,REVEC)
**BUFFER subroutine**

BUFFER calculates pointers and calls the SOLVER subroutine.

```fortran
INTEGER DIMEN
REAL*8 REVEC(DIMEN)
N_BUF=JNINT((SQRT(25.0+FLOAT((4*DIMEN)))-5.0)/
#2.0)
I1=1
I2=1+N_BUF*N_BUF
I3=I2+N_BUF
I4=I3+N_BUF
I5=I4+N_BUF
I6=I5+N_BUF
CALL SOLVER(N_BUF,REVEC(I1),REVEC(I2),REVEC(I3)
+REVEC(I4),REVEC(I5),REVEC(I6))
RETURN
END
```

**SUBROUTINE SOLVER(N,A,R,P,XC,XT,T2)**

SOLVER subroutine

SOLVER solves the linear system by the conjugate gradient method as though it were done in parallel and calculates timing as though it were done in parallel.

```fortran
REAL*8 A(N,N),R(N),P(N),XC(N),XT(N),T2(N)
REAL*8 T1,T3,T4,ACCUR,ALPHA,BETA
INTEGER ADD/0/,MULT/0/,DIV/0/,LOG2N,NFLAG
READ(029,*) A,R,XC
READ(21,*) ACCUR
C
Calculate initial R vector which equals
R=b-Ax
C
DO 300 I=1,N
  T2(I)=0.0
  DO 300 K=1,N
  300 T2(I)=A(I,K)*XC(K)+T2(I)
DO 310 I=1,N
```
R(I)=R(I)-T2(I)

Initialize maximum iteration counter
COUNT=0

Initialize P vector
DO 2 I=1,N
   P(I)=R(I)

Calculate how many additions take place in a summation which is executed in a binary tree of adders.

MAXK=0
DO 5 I=1,N
   K=0
   DO 6 J=1,N
      IF(A(I,J).EQ.0.0) GOTO 6
      K=K+1
   CONTINUE
   IF(K.LE.MAXK) GOTO 5
   MAXK=K
CONTINUE
RN=FLOAT(MAXK)
TLOG2N=ALOG(RN)/ALOG(2.0)
ILOG2N=JINT(TLOG2N)
RLOG2N=FLOAT(ILOG2N)
IF(RLOG2N.EQ.TLOG2N) GOTO 3
   LOG2N=ILOG2N+1
   GOTO 4
3    LOG2N=ILOG2N
4    CONTINUE
Timing is calculated for the initialization steps
MULT=MULT+1
ADD=ADD+LOG2N+1

T1=0
DO 10 I=1,N
   T1=R(I)*R(I)+T1
10   MULT=MULT+1
    ADD=ADD+LOG2N

DO 20 I=1,N
   T2(I)=0
20   DO 20 K=1,N
CONVERGENCE IS CHECKED IN PARALLEL WITH THE CALCULATION OF THE SquARED NORM OF R.

20  T2(I) = A(I,K)*F(K) + T2(I)
    MULT = MULT + 1
    ADD = ADD + LOG2N

C C

    T3 = 0
    DO 30 I = 1, N
      T3 = T3 + P(I) * T2(I)
      MULT = MULT + 1
      ADD = ADD + LOG2N

C C

    ALPHA = T1 / T3
    DIV = DIV + 1

C C

    DO 40 I = 1, N
      XT(I) = XC(I) + ALPHA * P(I)
      R(I) = R(I) - ALPHA * T2(I)
      MULT = MULT + 1
      ADD = ADD + 1

C C

   Convergence is checked in parallel with the calculation of the squared norm of R.

C C

    NFLAG = 0
    DO 50 I = 1, N
      IF ((DABS(XT(I) - XC(I))).LT. ACCUR) GOTO 50
      NFLAG = 1
    CONTINUE
    IF (NFLAG .NE. 0) GOTO 55
    CALL PRINT(N, XT, ADD, MULT, DIV, ACCUR)
    RETURN

50  CONTINUE

55  T4 = 0
    DO 60 I = 1, N
      T4 = T4 + R(I) * R(I) + T4
      MULT = MULT + 1
      ADD = ADD + LOG2N

C C

    DO 70 I = 1, N
      XC(I) = XT(I)

C C

    BETA = T4 / T1
    DIV = DIV + 1
    T1 = T4

C C

    DO 80 I = 1, N
      P(I) = R(I) + BETA * P(I)

80  CONTINUE
MULT=MULT+1
ADD=ADD+1
COUNT=COUNT+1
IF(COUNT.LE.(4*N)) GOTO 15
WRITE(6,85)

85 FORMAT(T5,'THE NUMBER OF ITERATIONS EXCEEDED 'N
** THE DIMENSION OF THE MATRIX.')
CALL PRINT(N,XT,ADD,MULT,DIV,ACC)
RETURN
END

SUBROUTINE PRINT(N,P,XT,P,ADD,P,MULT,P,DIV,P,ACC,P)
PRINT subroutine
PRINT puts the solution vector the radius of
convergence, and the number of additions,
multiplications, and the number of divisions
in I/O device 39 or file FOR039.DAT.

REAL*8 XT_P(N_P)
INTEGER ADD_P,MULT_P,DIV_P
REAL*8 ACC_P
WRITE(39,110)
DO 100 I=1,N_P
100 WRITE(39,120) I,XT_P(I)
WRITE(39,130) ACC_P
WRITE(39,140)
WRITE(39,150)
WRITE(39,160) ADD_P,MULT_P,DIV_P
RETURN
110 FORMAT(T5,'THE SOLUTION VECTOR IS')
120 FORMAT(T5,'X',T7,I2,T9,=',T11,G22.15)
130 FORMAT(T5,'RADIUS OF CONVERGENCE IS',T32,G22.15
**)
140 FORMAT(T5,'THE NUMBER OF STEPS TO SOLVE THIS SY
**STEM IS')
150 FORMAT(T7,'ADD',T23,'MULT',T35,'DIV')
160 FORMAT(T6,I4,T23,I4,T34,I4)
END