REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*

2. REPORT DATE
11 JUL 97

3. REPORT TYPE AND DATES COVERED

4. TITLE AND SUBTITLE
PARALLEL QR DECOMPOSITION FOR ELECTROMAGNETIC SCATTERING PROBLEMS

5. FUNDING NUMBERS

6. AUTHOR(S)
JEFF BOLENG

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
COLORADO SCHOOL OF MINES

8. PERFORMING ORGANIZATION REPORT NUMBER
97-080

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
DEPARTMENT OF THE AIR FORCE
AFIT/CI
2950 P STREET
WRIGHT-PATTERSON AFB OH 45433-7765

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION AVAILABILITY STATEMENT

DISTRIBUTION STATEMENT A
Approved for public release
Distribution Unlimited

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

14. SUBJECT TERMS

15. NUMBER OF PAGES
102

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT

18. SECURITY CLASSIFICATION OF THIS PAGE

19. SECURITY CLASSIFICATION OF ABSTRACT

20. LIMITATION OF ABSTRACT

Parallel QR Decomposition for Electromagnetic Scattering Problems

by

Jeff Boleng

A thesis submitted to the Faculty and the Board of Trustees of the Colorado School of Mines in partial fulfillment of the requirements for the degree of Master of Science (Mathematical and Computer Sciences).

Golden, Colorado
Date 20 Jun 97

Signed:
Jeff Boleng

Approved: Manavendra Misra
Dr. Manavendra Misra
Assistant Professor of Mathematical and Computer Sciences
Thesis Advisor

Golden, Colorado
Date 23 June 1997

Steve Pruess for GF
Dr. Graeme Fairweather
Professor and Head
Department of Mathematical and Computer Sciences

ABSTRACT

This report introduces a new parallel QR decomposition algorithm. Test results are presented for several problem sizes, numbers of processors, and data from the electromagnetic scattering problem domain. The development of the algorithm marks a departure from past parallel QR algorithms. The load balancing method used considers total computational work as opposed to just balancing Givens rotations. This results in expected efficiencies which approach optimal as problem size grows relative to number of processors. The hybrid nature of the algorithm, which maximizes computation between communication and synchronization, indicates potential for good performance on distributed memory machines and networks of workstations. Implementation results on shared memory and distributed shared memory architectures show promise and track expected performance well up to 12 processors.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

I would like to extend sincere thanks to everyone that I have had the pleasure to work with over the last eighteen months in the pursuit this goal. Special credit is due to the Colorado School of Mines Multi-disciplinary University Research Initiative group of Dr. John DeSanto, Dr. Willy Hereman, Dr. Manavendra Misra, and Grant Erdmann. My involvement in their work has been interesting and rewarding. Thanks to some of the best professors anywhere. Especially my advisor Dr. Manavendra Misra, Dr. Steve Pruess, and of course, Dr. Ruth Maurer.

I would also like to acknowledge the generous support of the U.S. Air Force Academy and the Air Force Institute of Technology. In addition, the National Center for Supercomputing Applications (NCSA) has been particularly helpful and tolerant with access to their computing resources.

I am especially grateful to Maj. Rob Kaufman and Col. Samuel Grier. Thanks for the chance.

I will always be indebted to Jack Webb, who helped me build the strongest foundations imaginable, and to Warren Parker who always reminded me why they were important and taught me how to keep it all in perspective. Finally, the greatest thanks always goes to Gina for her friendship and support.

Chapter 1

INTRODUCTION

The problem of solving dense systems of linear equations on parallel computers has been widely studied. Initial results treat the problem with theoretical rigor but do not usually implement the algorithms because real machines which meet the requirements and assumptions of the derivation rarely or never exist. Practitioners have taken these studies, added their own analysis and creativity, and adapted them to produce realistic algorithms which can be implemented on parallel computers of the day. This process creates two general classes of algorithms:

1. those that are theoretically optimal with respect to some parameter, and
2. those that might not be theoretically optimal but are tuned to run on a specific architecture.

This second class of algorithms must be extended and adapted as machine architectures evolve.

One final option provided to practitioners is the smaller class of algorithms which is beginning to gain popularity. There are algorithm designers which design, rigorously analyze, and develop methods for widely available production machines. This class of algorithms is characterized by realistic assumptions which result from considering existing architectures. The goal is production quality numerical techniques which are possible and cost effective to develop. Oftentimes, optimal results are achieved and measured on the target machines. This work aims to develop algorithms in this last class.

1.1 Outline

This report will begin with an overview of the algorithms currently available for the QR factorization of a dense matrix. These algorithms are summarized in Chapter 2, and predicted and actual performance results presented where they are available. Chapter 3 then introduces and develops a new algorithm for parallel QR factorization. The application problem that motivated the development of this algorithm involves computationally modelling electromagnetic scattering from a rough surface. The electromagnetic scattering problem is described in Chapter 4. Numerical results from general matrices and the application of the new algorithm to the electromagnetic scattering problem are presented in Chapter 5.

1.2 Notation and Conventions

We write the QR factorization of A as

$$A = QR, \tag{1.1}$$

where A is $m \times n$, Q is an $m \times m$ orthogonal matrix, R is $m \times n$ upper trapezoidal, and $m \geq n$. The number of processors will be denoted by p. A is assumed to be full rank; $rank(A) = rank(R) = n$. Additionally, all vectors $\vec{v}$ are considered to be column vectors, and therefore $\vec{v}\vec{v}^T$ represents an outer product and results in an $m \times m$ matrix when $\vec{v} \in \Re^m$, while $\vec{v}^T\vec{v}$ is a scalar.

Throughout this paper, the sequential complexity of QR factorization will be based on the results from (Golub & Van Loan, 1996) (see Table 1.1). The complexities listed in Table 1.1, which count the number of floating point operations performed, assume $m = n$ for Gaussian Elimination, and $m \geq n$ for all other cases. Additional

Method	Complexity
Gaussian Elimination	$\geq (2n^3/3)$
Normal Equations with Cholesky Factorization	$\geq (mn^2 + n^3/3)$
Householder Reflections	$\geq (2n^2(m - n/3))$
Givens Rotations	$\geq (3n^2(m - n/3))$

Table 1.1. Complexity comparison of common factorization methods.

notational conventions will be introduced when needed.

1.3 Least Squares Problem

All computational results in this paper assume A has full rank and $m \geq n$. Therefore, when $m = n$ the system of equations

$$A\vec{x} = \vec{b}$$

with known A and $\vec{b}$ has exactly one solution. When $m > n$, the solution is taken to be the vector $\vec{x}$ that minimizes [1]

$$\| A\vec{x} - \vec{b} \|^2 .$$

Given $A = QR$ from 1.1 the solution becomes

$$\min_{\vec{x}} \| A\vec{x} - \vec{b} \|^2 = \min_{\vec{x}} \| QR\vec{x} - \vec{b} \|^2 = \min_{\vec{x}} \| R\vec{x} - Q^T\vec{b} \|^2 .$$

[1] All vector and matrix norms will be considered the 2-norm unless specifically subscripted.

Letting $\vec{d} = Q^T\vec{b}$ we have

$$\min_{\vec{x}} \left\| \begin{bmatrix} R_1 \\ 0 \end{bmatrix} \vec{x} - \begin{bmatrix} d_1 \\ \dots \\ d_n \\ d_{n+1} \\ \dots \\ d_m \end{bmatrix} \right\|^2 = \min_{\vec{x}} \left\| \begin{bmatrix} R_1\vec{x} & - & \begin{bmatrix} d_1 \\ \dots \\ d_n \end{bmatrix} \\ 0 & - & \begin{bmatrix} d_{n+1} \\ \dots \\ d_m \end{bmatrix} \end{bmatrix} \right\|^2 =$$

$$\min_{\vec{x}} \left\| R_1\vec{x} - \begin{bmatrix} d_1 \\ \dots \\ d_n \end{bmatrix} \right\|^2 + \sum_{i=n+1}^{m} d_i^2,$$

where the residual is

$$\sqrt{\sum_{i=n+1}^{m} d_i^2}.$$

The above equations lead us to a general QR decomposition algorithm:

1. Calculate Q and R,

2. $\vec{d} = Q^T\vec{b}$,

3. use back substitution to solve $R_1\vec{x} = \begin{bmatrix} d_1 \\ \dots \\ d_n \end{bmatrix}$,

4. residual $= \sqrt{\sum_{i=n+1}^{m} d_i^2}$.

The most computationally demanding step in the algorithm is calculating Q and R (step 1), therefore this research will focus on the parallelization of this process.

1.4 Computing Q and R

There are two primary ways to compute Q and R – Householder reflections and Givens rotations. Each of these methods has advantages and disadvantages, especially when considered in the context of a parallel QR algorithm. The following chapters survey existing parallel QR techniques (Chapter 2) and the development of a new parallel QR algorithm (Chapter 3), and use both Householder reflections and Givens rotations.

1.4.1 Householder Reflections

An $n \times n$ matrix H of the form

$$H = I - \beta \vec{v}\vec{v}^T \tag{1.2}$$

where $\beta = \frac{2}{\vec{v}^T\vec{v}}$ is called a Householder reflection (or matrix, or transformation). The vector $\vec{v}$ is called a Householder vector. It is easy to verify that Householder matrices are symmetric and orthogonal (Golub & Van Loan, 1996). Householder reflections are rank-1 modifications of the identity matrix and can be used to zero selected components of a vector.

Consider $\vec{x} \in \Re^n$ and $H\vec{x}$ such that

$$H\vec{x} = \pm\alpha\vec{e_1} \tag{1.3}$$

where $\vec{e_1}$ is the first column of the $n \times n$ identity matrix. By substituting we get

$$H\vec{x} = (I - \beta\vec{v}\vec{v}^T)\vec{x} = \vec{x} - \beta\vec{v}\vec{v}^T\vec{x} = \alpha\vec{e_1},$$

and solving for $\vec{v}$ yields

$$\vec{v} = \frac{\vec{x} - \alpha\vec{e_1}}{\beta\vec{v}^T\vec{x}}.$$

This suggests trying

$$\vec{v} = \gamma(\vec{x} - \alpha\vec{e_1}) = \vec{x} - \alpha\vec{e_1}$$

for some constant γ. The value of γ has no effect on H, so we will take it to be 1.[2] Therefore, given a non-zero vector $\vec{x}$, let

1. $\alpha = \| \vec{x} \|_2$,

2. $\vec{v} = \vec{x} - \alpha\vec{e_1}$,

3. $\beta = \frac{1}{\alpha(x_1+\alpha)}$, and

4. $H = I - \beta\vec{v}\vec{v}^T$, then

H is a Householder reflection, and $H\vec{x} = \alpha\vec{e_1}$.

Householder reflections can be used to find Q and R as follows:

1. Form H_1 as in Equation 1.2 to zero elements 2 through m in $\vec{x}$ where $\vec{x}$ is column 1 of matrix A ($\vec{x} = A\vec{e}_1$).

2. Form $H_i, 2 \leq i \leq n$ similarly to zero elements $(i+1)...m$ in column i of A.

3. Form $Q^T = H_n H_{n-1} ... H_1$.

4. $R = Q^T A$.

[2]

$$\text{Letting } \gamma = \frac{1}{\beta\vec{v}^T\vec{x}}, \text{ then } \beta\vec{v}^T\vec{x} = \frac{\vec{x}^T\vec{x} + \alpha\vec{e}_1^T\vec{x}}{\alpha\,(x_1+\alpha)} = \frac{\alpha^2 + \alpha x_1}{\alpha^2 + \alpha x_1} = 1.$$

1.4.2 Complexity of One Householder Reflection

A quick presentation of the computational complexity of one Householder rotation is necessary for later use during algorithm development. The complexity of one Givens rotations is similarly presented in Section 1.4.5. Following the steps presented previously at the end of Section 1.4.1, the computational complexity is (for simplicity, additions, subtractions, multiplications, divisions, and square roots are all assumed to be unit time operations.)[3]

$$\begin{array}{rcll} \alpha & = & \| \vec{x} \|_2 & \sim & 2n \\ \vec{v} & = & \vec{x} + \alpha \vec{e_1} & \sim & 1 \\ \beta & = & \frac{1}{\alpha(x_1+\alpha)} & \sim & 3 \\ H & = & I - \beta \vec{v}\vec{v}^T & \sim & n. \end{array}$$

Notice that the complexity given for forming the Householder matrix is not that of forming an outer product ($\vec{v}\vec{v}^T$, normally $O(n^2)$). This is because, in practice, explicitly forming H is not necessary. The formation of $\alpha, \vec{v}$, and β is sufficient, so formation of the Householder matrix requires $\sim (3n+4)$ flops. Application of the Householder matrix can exploit the high degree of structure in H and can be done with a cost of ($A \in \Re^{m \times n}$ and $H \in \Re^{m \times m}$)

$$HA = (I - \beta \vec{v}\vec{v}^T)A \sim 4mn. \tag{1.4}$$

1.4.3 Householder Summary

Householder reflections work well for introducing large numbers of zeros in one matrix operation. Normally, all the elements below the diagonal of an entire column of

[3] The $\sim$ symbol will be used throughout this report to denote the approximate operation count needed to compute the value shown.

the matrix A are eliminated by one Householder reflection. This leads to the primary disadvantage of Householder matrices when used in parallel processing. One reflection affects multiple rows, and therefore Householder reflections can not be carried out in parallel in a straightforward way. Householder reflections are not disjoint when applied to an entire matrix, which is the case in traditional QR decomposition. There is a way to apply multiple Householder reflections to the same matrix in parallel by applying smaller reflections to blocks of the main matrix. This is the key idea behind the hybrid algorithms included in Chapters 2 and 3 and will be explored in more detail later. The alternative is QR decomposition via Givens rotations.

1.4.4 Givens Rotations

Givens rotations can selectively annihilate individual matrix elements, as opposed to Householder reflections which eliminate whole columns or rows as described above. One rotation only affects two rows of the matrix; the row containing the element being zeroed, and the row being used to zero the element. We will use the notation introduced in (Cosnard & Trystram, 1995) where $G(i, j, k)$ is used to denote the Givens rotation that zeroes element $A(i, k)$ by rotating rows i and j through angle θ in the (i, j) plane. Givens matrices are rank-two corrections to the identity matrix, and can be easily shown to be orthogonal. An example best illustrates the structure of a Givens rotation (ref. Fig. 1.1).

When performing Givens rotations computationally, it is not necessary to explicitly compute the rotation angle (θ). Instead, for $G(i, j, k)$, it is enough to compute c and s (which denote $\sin\theta$ and $\cos\theta$) as follows (Cosnard & Trystram, 1995):

$$c = \frac{a_{jk}}{\sqrt{a_{jk}^2 + a_{ik}^2}} \qquad (1.5)$$

$$G(i,j,k) = \begin{array}{r|ccccccc|}
 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
j \rightarrow & 0 & c & 0 & 0 & 0 & s & 0 \\
 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\
 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\
 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\
i \rightarrow & 0 & -s & 0 & 0 & 0 & c & 0 \\
 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\
\multicolumn{1}{r}{} & \multicolumn{1}{c}{} & \multicolumn{1}{c}{\uparrow} & & & & \multicolumn{1}{c}{\uparrow} & \multicolumn{1}{c}{} \\
\multicolumn{1}{r}{} & \multicolumn{1}{c}{} & \multicolumn{1}{c}{k} & & & & \multicolumn{1}{c}{i} & \multicolumn{1}{c}{}
\end{array}$$

$$\begin{aligned} c &= \cos\theta \\ s &= \sin\theta \end{aligned}$$

FIG. 1.1. Example 7 × 7 Givens rotation where $i = 6, j = 2$, and $k = 2$.

$$s = \frac{a_{ik}}{\sqrt{a_{jk}^2 + a_{ik}^2}} \tag{1.6}$$

Using Givens rotations to find Q and R is similar to the procedure described in Section 1.4.1, however, the number of Givens matrices needed is

$$r = \frac{n(n-1)}{2},$$

as opposed to the number of Householder matrices needed, n when $m > n$, or $n - 1$ when $m = n$. The procedure to produce the QR decomposition of A using Givens rotations is to find choices of i, j, and k such that

$$G_r \ldots G_1 A = R$$

then

$$Q = G_1^T \ldots G_r^T.$$

Assume $m > n$, then a common ordering for combining Givens rotations is formed by zeroing the elements from bottom to top and left to right, or from rows m to 2 in column 1, to column n, rows m to $m-n+1$. Mathematically this looks like the following:

$$\begin{aligned}
C_1 &= \underbrace{G(2,1,1)G(3,1,1)...G(m-1,1,1)G(m,1,1)}_{\text{zeros} \quad \text{column} \quad 1} \\
C_2 &= \underbrace{G(3,2,2)G(4,2,2)...G(m-1,2,2)G(m,2,2)}_{\text{zeros} \quad \text{column} \quad 2} \\
C_{n-1} &= \underbrace{G(m-n,n-1,n-1)G(m-n+1,n-1,n-1)...G(m,n-1,n-1)}_{\text{zeros} \quad \text{column} \quad n-1} \\
C_n &= \underbrace{G(m-n+1,n,n)G(m-n+2,n,n)...G(m-1,n,n)G(m,n,n)}_{\text{zeros} \quad \text{column} \quad n}.
\end{aligned}$$

Combining the above matrices results in

$$C_n C_{n-1} ... C_2 C_1 A = R.$$

1.4.5 Complexity of One Givens Rotation

Given two vectors $\vec{u}, \vec{v} \in \Re^n$, their Givens rotation can be depicted as (Cosnard & Trystram, 1995)

$$\begin{bmatrix} c & s \\ -s & c \end{bmatrix} \begin{bmatrix} u_1 & u_2 & ... & u_n \\ v_1 & v_2 & ... & v_n \end{bmatrix} = \begin{bmatrix} u'_1 & u'_2 & ... & u'_n \\ 0 & v'_2 & ... & v'_n \end{bmatrix}$$

where c and s are calculated as in equations 1.5 and 1.6 respectively, and $\vec{u}'$ and $\vec{v}'$ are calculated by

$$\begin{aligned} u_i' &= cu_i + sv_i \qquad & 1 \leq i \leq n \\ v_i' &= -su_i + cv_i \qquad & 1 \leq i \leq n. \end{aligned}$$

The number of individual operations to zero element $\vec{v}_1$ by combining $\vec{u}$ and $\vec{v}$ using a Givens rotation is shown in the following table:

	$+,-$	$*,/$	$\sqrt{\ }$
computation of c and s	1	4	1
computation of $\vec{u}$'	n	$2n$	
computation of $\vec{v}$'	n	$2n$	

Assuming that additions, subtractions, multiplications and divisions are unit time operations, the complexity of the computation is given by:

$$G \begin{bmatrix} \vec{u}^T \\ \vec{v}^T \end{bmatrix} \sim 6n + 6. \tag{1.7}$$

1.4.6 Reflections vs. Rotations

A few final notes must be made concerning Householder reflections and Givens rotations before proceeding to an overview of existing parallel QR algorithms. First, although many more Givens rotations are required to perform the decomposition than Householder reflections, each Givens rotation is much simpler and less computationally demanding than each Householder reflection. Given these contrasting observations, examination of Table 1.1 reveals that performing the entire decomposition via Givens rotations costs only about one and one half times as much as performing the entire decomposition via Householder reflections on a sequential machine.

Second, Givens rotations are ideally suited for parallel execution as long as the rows being used in the parallel rotations are disjoint. This idea is further expanded in Chapter 2 when specific algorithms are described. Third, a Givens rotation, $G(i, j, k)$, is possible to eliminate any arbitrary element $A(i, k)$, using any two rows i and j of the matrix. However, if the goal is to apply a series of Givens rotations in order to form a triangular matrix, then all the elements of the two rows i and j left of column k must be zero, or the rotation will introduce non-zero elements (or fill-ins) into previously zeroed positions. Doing this would destroy previous work with subsequent rotations. These two ideas are crucial limitations on the parallel Givens based QR algorithms discussed next.

Finally, it is possible to perform Householder reflections on sub-matrices. This idea, due to Pothen and Raghavan (1989), makes it possible to perform a great deal of the QR decomposition in perfect parallelism using the most efficient annihilation method known, Householder reflections. Givens rotations are then applied (in parallel) to "clean up the ragged edges." This idea is key in the development of the new parallel QR algorithm introduced here.

Chapter 2

PREVIOUS QR DECOMPOSITION ALGORITHMS

This chapter provides a summary of existing parallel QR factorization algorithms. Current literature is largely polarized. Two primary assumptions are made concerning the target architecture which leads to two large classes of algorithms. The first and oldest class of algorithms is a direct descendant of early theoretical work and is targeted for efficient execution on large scale vector computers (Sameh & Kuck, 1978). The primary assumption made in this analysis is that an operation on a vector takes the same amount of computation time regardless of the vector length. This assumption is completely valid on vector processors as long as vector length never exceeds that of the target machine's vector registers. These algorithms are covered in Section 2.1. One extension to this class of algorithms takes place when vector lengths become longer, thereby invalidating the original assumption. These algorithms will be covered in Section 2.2.

The second class of algorithms is newer and still growing. These algorithms are targeted at distributed memory architectures including massively parallel machines and networks of workstations. The number of algorithms available in this class is fewer, and the demands placed on the algorithm designer are higher. Section 2.3 provides an overview of these efforts.

2.1 Early Algorithms

The starting point for any discussion of parallel factorization techniques must provide some acknowledgement to the very early algorithms in the field. The first five algorithms discussed include the still widely referenced work by A.H. Sameh and D.J. Kuck, some discussion of parallel LU factorization, and early attempts to produce variations on the original Givens algorithms discussed by Sameh and Kuck (1978).

2.1.1 Sameh and Kuck's Algorithm

The earliest and most widely referenced work on parallel QR factorization is that of Sameh and Kuck which introduces a stable parallel algorithm for solving dense systems of linear equations, and two algorithms for solving a tridiagonal system (Sameh & Kuck, 1978). All the approaches in the paper by Sameh and Kuck make use of Givens transformations (see Section 1.4.4, (Cosnard & Trystram, 1995), and (Golub & Van Loan, 1996)). As mentioned earlier, these algorithms assume that the time to perform a single Givens transformation is the same regardless of vector length. This assumption simplifies the problem to one of balancing Givens rotations across processors.

The first algorithm introduced is widely known as Sameh and Kuck's algorithm. Excellent descriptions and analysis are given in (Cosnard & Trystram, 1995) and (Sameh & Kuck, 1978). For our purposes, a figure will illustrate the general algorithm most effectively (see Fig. 2.1). This figure shows the order in which the elements of A are zeroed by the algorithm. An integer number placed in the matrix position indicates the step at which that element is zeroed using a Givens rotation.

As can be seen in the diagram, this algorithm begins with the annihilation of element $A(m,1)$ using rows $m-1$ and m (written as $G(m,m-1,1)$) and proceeds up and to the right. In the first two steps, only one element is annihilated, but at every other step the number of simultaneous rotations increases by one. The maximum number of rotations is $\min[n, \lfloor\frac{m}{2}\rfloor]$ and is reached by step $\left(2 \times \min[n, \lfloor\frac{m}{3}\rfloor] - 1\right)$

*						
12	*					
11	13	*				
10	12	14	*			
9	11	13	15	*		
8	10	12	14	16	*	
7	9	11	13	15	17	*
6	8	10	12	14	16	18
5	7	9	11	13	15	17
4	6	8	10	12	14	16
3	5	7	9	11	13	15
2	4	6	8	10	12	14
1	3	5	7	9	11	13

FIG. 2.1. Sameh and Kuck algorithm, $m = 13$, $n = 7$.

(Cosnard & Trystram, 1995).

This method requires $n(n-1)/2$ rotations as in the sequential case. It is shown in Sameh and Kuck (1978) that $5(2n-3)$ steps (rotations) are required to produce the QR factorization using the optimal number of processors $p = n(3n-2)/2$. Data mapping is not presented in the paper because uniform memory access is assumed and therefore no communication costs are included. For very large matrices, this produces an $O(n^2)$ speedup and efficiency of $O(1)$. Modifications to this basic algorithm are available and described in (Cosnard & Trystram, 1995). These modifications do not provide performance which differs substantially from that originally reported in (Sameh & Kuck, 1978).

2.1.2 Gaussian Elimination

Lord *et al.* (1983) focus on the parallelization of Gaussian Elimination as well as a parallel algorithm using Givens transformations. Similar algorithms are developed and analyzed for LU factorization on a distributed memory multiprocessor in Geist and Romine (1987). Lord *et al.* develop their LU decomposition algorithm using a parallel task dependency graph and achieve an efficiency of $\frac{2}{3}$ for an $n \times n$ matrix with $p = \lceil n/2 \rceil$. The mapping of the tasks to processors of a parallel machine achieves a natural load balancing. Actual performance measures for this algorithm are presented which track the predicted performance very closely.

The primary drawback of parallel LU factorization/Gaussian elimination algorithms is their numerical instability without pivoting. Introduction of pivoting greatly increases the communication requirements and interrupts the task assignment of the processors by forcing a synchronization for row interchange.

The next sections (2.1.3 and 2.1.4) describe the two Givens variants presented by Lord *et al.* (1983). In contrast to the method discussed in Section 2.1.1 where $p = O(n^2)$, the Givens methods described here focus on the cases where $p \leq \lceil (n-1)/2 \rceil$. Two algorithms are developed, implemented, and tested on the Denelcor HEP machine. Again, predicted and actual results correlate very well.

2.1.3 ZIGZAG

The first variant of the Givens method presented by Lord *et al.* (1983) is called ZIGZAG and is recommended for cases where $p = \lceil (n-1)/2 \rceil$. Figure 2.2 gives the general idea of the algorithm. Again, an integer in a matrix position denotes the time step at which that element is annihilated. The numbers in this figure are subscripted with processor identifiers to help depict the zigzag nature of the algorithm. In all the rotations the row containing the zeroed element is combined with the row immediately above it. Notice that each processor is assigned to perform all the rotations for two diagonals as designated in the figure.

	*								
P1	1_{P1}	*							
↘	2_{P1}	3_{P1}	*						
P2	1_{P2}	4_{P1}	5_{P1}	*					
↘	2_{P2}	3_{P2}	6_{P1}	7_{P1}	*				
P3	1_{P3}	4_{P2}	5_{P2}	8_{P1}	9_{P1}	*			
↘	2_{P3}	3_{P3}	6_{P2}	7_{P2}	10_{P1}	11_{P1}	*		
P4	1_{P4}	4_{P3}	5_{P3}	8_{P2}	9_{P2}	12_{P1}	13_{P1}	*	
↘	2_{P4}	3_{P4}	6_{P3}	7_{P3}	10_{P2}	11_{P2}	14_{P1}	15_{P1}	*

FIG. 2.2. ZIGZAG algorithm, $n = 9$, $p = 4$.

One observation is immediately apparent. The algorithm does a poor job of load balancing the required computation. The maximum number of rotations is $2n - 3$ and is performed by processor P1. The last processor ($\lceil (n-1)/2 \rceil = 4$) only performs 3 rotations.

This method results in a parallel speedup of approximately $\frac{2n}{9}$ with an efficiency of only $\frac{4}{9}$ for sufficiently large n. Further discussion and details are available in (Lord *et al.*, 1983).

2.1.4 COLSWEEP

The second variant of the Givens method presented by Lord *et al.* (1983) is called COLSWEEP and is recommend for cases where $p \leq \lceil (n-1)/2 \rceil$. Figure 2.3 gives the general idea of the algorithm. In this algorithm, columns are assigned in a round robin fashion to processors as designated in the figure. The scheduling of rotations is critical to avoid access conflicts between processors. As an example, processor i must be finished modifying its pivot row before processor $i - 1$ can annihilate that row's leftmost element in the next step. Each processor uses the diagonal element as a pivot.

*								
1	*							
2	3	*						
3	4	5	*					
4	5	6	7	*				
5	6	7	8	9	*			
6	7	8	9	10	11	*		
7	8	9	10	11	12	13	*	
8	9	10	11	12	13	14	15	*
P1	P2	P3	P1	P2	P3	P1	P2	

FIG. 2.3. COLSWEEP algorithm, $n = 9$, $p = 3$.

This method results in a parallel speedup and efficiency numbers similar to those reported for ZIGZAG above. In fact, for the case of $p = \lceil (n-1)/2 \rceil$, the time complexity is exactly the same as that for ZIGZAG. After closer examination, this algorithm has dramatic similarities to Sameh and Kuck's discussed in 2.1.1. Further discussion and details are available in (Lord *et al.*, 1983).

2.1.5 Greedy Algorithm

This algorithm was independently introduced in (Cosnard & Robert, 1983) and (Modi & Clarke, 1984). At each step, all disjoint rotations are performed simultaneously. An example of the complexity of this algorithm requires $2n+O(n)$ rotations for the case $m = o(n^2)$ with $p = \lfloor m/2 \rfloor$ $(m \geq n)$ processors (Cosnard & Robert, 1986). "However, no exact formula is known (for the general complexity of the greedy algorithm), even in the case of a square matrix $(m = n)$" (Cosnard & Daoudi, 1994). For consistency, Figure 2.4 demonstrates the main idea behind the algorithm (Cosnard & Trystram, 1995).

The basic concept of maximizing the number of possible rotations at each time

*					
4	*				
3	6	*			
3	5	8	*		
2	5	7	10	*	
2	4	7	9	12	*
2	4	6	8	11	14
1	3	6	8	10	13
1	3	5	7	9	12
1	3	5	7	9	11
1	2	4	6	8	10
1	2	4	6	8	10
1	2	3	5	7	9

FIG. 2.4. Greedy algorithm.

step is an elegant one, but actually determining a rotation schedule to achieve this is the main difficulty of successful algorithm implementation. Small cases can be easily assigned, but as matrix size grows, a considerable amount of effort is required at each step to assign the rotations which are disjoint and can therefore take place in parallel.

2.2 Block Algorithms

This group of algorithms has been extensively studied and developed in (Dongarra *et al.*, 1991) and (Gallivan *et al.*, 1990). Additional discussions can be found in (Anderson *et al.*, 1995), (Cosnard & Trystram, 1995), (Dongarra, 1993), (Dongarra & Walker, 1996), (Dongarra *et al.*, 1986), and (Golub & Van Loan, 1996) to name a few. The recently released LAPACK software makes extensive use of block algorithms. The demand for block algorithm development was spawned by the limits imposed by real machines on vector length and the need to carefully schedule memory accesses to avoid swapping.

The primary idea is to split the matrix to be factored into blocks which can then be assigned to processors. The calculation of these blocks can then be postponed or scheduled to increase performance. The key to achieving speedup is the proper choice of blocksize, which has proven to be heavily machine dependent and somewhat of an "art form". A simple example taken from (Dongarra *et al.*, 1991) should suffice to illustrate the key concept.

Consider the decomposition of matrix A into its LU factorization with the following matrix partitioning (blocking) [1]:

$$\begin{pmatrix} A_{11} & A_{12} & A_{13} \\ A_{21} & A_{22} & A_{23} \\ A_{31} & A_{32} & A_{33} \end{pmatrix} = \begin{pmatrix} L_{11} & 0 & 0 \\ L_{21} & L_{22} & 0 \\ L_{31} & L_{32} & L_{33} \end{pmatrix} \begin{pmatrix} U_{11} & U_{12} & U_{13} \\ 0 & U_{22} & U_{23} \\ 0 & 0 & U_{33} \end{pmatrix}$$

Multiplying L and U together and equating terms with A, we have

$$\begin{aligned} A_{11} &= L_{11}U_{11}, & A_{12} &= L_{11}U_{12}, & A_{13} &= L_{11}U_{13}, \\ A_{21} &= L_{21}U_{11}, & A_{22} &= L_{21}U_{12} + L_{22}U_{22}, & A_{23} &= L_{21}U_{13} + L_{22}U_{23}, \\ A_{31} &= L_{31}U_{11}, & A_{32} &= L_{31}U_{12} + L_{32}U_{22}, & A_{33} &= L_{31}U_{13} + L_{32}U_{23} + L_{33}U_{33}. \end{aligned}$$

The computation of the elements of L and U can then be scheduled according to the methods in Section 2.1.2 allowing the algorithmic variants to be used in cases where $p \ll m, n$ or vector length is less than m, n.

Further discussion and details of a blocked Householder algorithm with performance measures can be found in (Dongarra *et al.*, 1991). It should be noted that there is little benefit from the development of a block Givens algorithm because the individual rotations are commutative provided they are disjoint.

[1] In this example, A_{ij} denotes a block of matrix A, not an individual element.

2.3 Recent Work

The most recent work regarding parallel QR factorization has been done in (Cosnard & Daoudi, 1994) and (Pothen & Raghavan, 1989). The first paper extends the ideas introduced in (Modi & Clarke, 1984) and is covered in Section 2.3.1, and the other is concerned with QR factorization using a distributed memory programming model (covered in Sections 2.3.2, 2.3.3, and 2.3.4).

2.3.1 Fibonacci

This represents an entire class of algorithms which was first introduced by Modi and Clarke (1984). A good description and full analysis can be found in (Cosnard & Daoudi, 1994) and (Cosnard & Trystram, 1995). In their paper, Cosnard and Daoudi present several alternate formulations of Fibonacci algorithms. The basic concept is represented by Figure 2.5.

The time-step determination and processor assignment of Givens rotations proceeds in two stages. In stage one, the initial column is filled from the top down; $u_1 = 1$ zero is placed in position $(2, 1)$, then $u_2 = 2$ copies of -1 are placed below that, then $u_3 = 3$ copies of $-2, \ldots, u_k = u_{k-1} + 1$ copies of $-(k-1)$ below that until the column is completely filled. The second column is filled by adding two to each element in column one and shifting it one position downward. This proceeds until the n^{th} column is filled. In stage two, the elimination assignment is obtained by adding $u + 1$ to each element in the previous table with $-u$ being the element in position $(m, 1)$. Figure 2.5 shows this assignment for a Fibonacci algorithm of order 1.

Similar algorithms of order q can be constructed by replacing the relation $u_k = u_{k-1} + 1$ with $u_k = u_{k-1} + u_{k-2} + \ldots + u_{k-q+1}$ (with $u_0 = u_{u-1} = \ldots u_{-q+1} = 0$) and adding $q + 1$ instead of 2 to the elements in column j, in order to obtain column $j + 1$

$$
\begin{array}{rrrrrr}
* & & & & & \\
0 & * & & & & \\
-1 & 2 & * & & & \\
-1 & 1 & 4 & * & & \\
-2 & 1 & 3 & 6 & * & \\
-2 & 0 & 3 & 5 & 8 & * \\
-2 & 0 & 2 & 5 & 7 & 10 \\
-3 & 0 & 2 & 4 & 7 & 9 \\
-3 & -1 & 2 & 4 & 6 & 9 \\
-3 & -1 & 1 & 4 & 6 & 8 \\
-3 & -1 & 1 & 3 & 6 & 8 \\
-4 & -1 & 1 & 3 & 5 & 8 \\
-4 & -2 & 1 & 3 & 5 & 7
\end{array}
\quad \Longrightarrow \quad
\begin{array}{rrrrrr}
* & & & & & \\
5 & * & & & & \\
4 & 7 & * & & & \\
4 & 6 & 9 & * & & \\
3 & 6 & 8 & 11 & * & \\
3 & 5 & 8 & 10 & 13 & * \\
3 & 5 & 7 & 10 & 12 & 15 \\
2 & 5 & 7 & 9 & 12 & 14 \\
2 & 4 & 7 & 9 & 11 & 14 \\
2 & 4 & 6 & 9 & 11 & 13 \\
2 & 4 & 6 & 8 & 11 & 13 \\
1 & 4 & 6 & 8 & 10 & 13 \\
1 & 3 & 6 & 8 & 10 & 12
\end{array}
$$

FIG. 2.5. Example time-step assignment for Fibonacci algorithm of order 1.

(Cosnard & Trystram, 1995).

Cosnard and Daoudi present the following results for the complexity and performance of their multi-stage Fibonacci algorithm in (Cosnard & Daoudi, 1994):

$$T_{opt}(p) = \frac{n^2}{2p} + p + O(n) \qquad \text{for} \qquad 1 \le p \le \frac{n}{2+\sqrt{2}} + o(n)$$

and that the minimum number of processors in order to compute the Givens factorization in asymptotically optimal time $(2n+o(n))$ is equal to $p_{opt} = n/(2+\sqrt{2})+o(n)$.

2.3.2 Distributed Givens

Pothen and Raghavan (1989) implement and discuss two of the three parallel Givens techniques originally introduced in (Pothen *et al.*, 1987). These algorithms are labeled *sgs* and *ggs* for standard Givens sequence and greedy Givens sequence respectively. The *sgs* algorithm is previously described in Section 2.1.1, and the *ggs* algorithm is previously described in Section 2.1.5. The important extension made by Pothen and Raghavan (1989) is the inclusion of a predicted and measured communication cost for the two algorithms, in addition to calculating its computational complexity. The third algorithm discussed in (Pothen *et al.*, 1987) is the "recursive fine partition algorithm". It has not been implemented because examination shows it to be best only when a large number of processors are available, which has not been the case for the authors.

The complexities for the *sgs* and *ggs* algorithms are derived by Pothen and Raghavan and are included in Table 2.1.

Algorithm	A/C	Complexity
sgs	A	$\frac{2n^2}{p}(m - n/3)\tau + n(3m + n/2)\tau$
sgs	C	$\frac{2n^2}{p}(m - n/3)\alpha + n(3m + n/2)\alpha + \frac{2n}{p}(2m - n)\beta + 2(m + n)\beta$
ggs	A	$\frac{2n^2}{p}(m - n/3)\tau + n^2 \log(p)\tau$
ggs	C	$n^2 \log(p)\alpha + 2n \log(p)\beta$

Table 2.1. Arithmetic (A) and communication (C) complexities for the standard and greedy parallel Givens sequences. Matrix size is $m \times n$, with p processors, $\tau =$ time needed for one flop, and message transfer time is $M\alpha + \beta$ for M bytes at α bytes per time unit with a latency of β.

2.3.3 Distributed Householder

Pothen and Raghavan also discuss two distributed Householder algorithms. These algorithms are labelled *p_house* and *b_house* for pipelined and broadcast Householder respectively. Each algorithm performs the QR factorization with the same mathematical elimination scheme. The difference lies in the communication of the results to the other processors between stages.

This algorithm proceeds by mapping the columns of the matrix A onto a ring of p processors. Each processor holds an $m \times \lceil n/p \rceil$ submatrix. At stage j, processor one computes the Householder vector required to zero out the sub-diagonal elements in column j. This processor applies this reflection to the columns assigned to it, and passes the reflection information to the next processor in the ring to apply to its columns. Execution proceeds in this fashion until the final processor on the ring receives and applies the final Householder reflection. This sort of nearest neighbor communication is a key characteristic in what is called "systolic" algorithms.

The only modification for the broadcast version of the algorithm is that the communication of the Householder vector is broadcast to all other processors and the other processors then apply the reflection simultaneously. The broadcast communication operation costs more, but the computation sequence doesn't suffer from the loading and unloading of the pipeline.

The complexities for *p_house* and *b_house* are included in Table 2.2.

Algorithm	A or C	Complexity
p_house	A	$\frac{n^2}{p}(m - n/3)\tau + 3n(m - n/2)\tau$
p_house	C	$n(m - n/2)\alpha + mp\alpha + (n + p)\beta$
b_house	A	$\frac{n^2}{p}(m - n/3)\tau + 3n(m - n/2)\tau$
b_house	C	$n(m - n/2)\log(p)\alpha + n\log(p)\beta$

Table 2.2. Arithmetic (A) and communication (C) complexities for the pipelined and broadcast Householder algorithms. Matrix size is $m \times n$, with p processors, τ = time needed for one flop, and message transfer time is $M\alpha + \beta$ for M bytes at α bytes per time unit with a latency of β.

2.3.4 Distributed Hybrid

The final algorithm discussed by Pothen and Raghavan (1989) is a hybrid based on the two classes of algorithms described in Sections 2.3.2 and 2.3.3. An observation which Pothen and Raghavan bring to light is that when the matrix A is highly overdetermined, communication costs in a parallel Givens algorithm could be substantially lower than those in a Householder algorithm because the Givens technique communicates rows where the Householder technique communicates columns. This motivated the authors to develop an algorithm with the lower local computation costs of a Householder algorithm and the lower communication costs of a Givens algorithm.

The matrix is partitioned by rows numbered from 0 to $m-1$, and a ring of p processors numbered from 0 to $p-1$ is employed. The first n rows are mapped onto the processors such that row 0 is on processor 0, row 1 on processor 1, etc. The rest of the $m-n$ rows can be equally distributed among the processors in any manner.

The matrix is then transformed by columns from left to right. Each column is transformed in two phases:

1. the internal reflection phase (IP), and
2. a recursive elimination phase (RP).

During the internal reflection phase, each processor applies a local Householder reflection to zero all but one element of the current column. The recursive elimination phase proceeds when processors communicate with each other to annihilate the remaining subdiagonal elements by means of Givens rotations. The complexities for the hybrid algorithm are presented in Table 2.3.

Based on the results listed above, predicted and actual performance figures are included in (Pothen & Raghavan, 1989). Run time results were measured on the Intel

Algorithm	A or C	Complexity
hybrid	A	$\frac{n^2}{p}(m - n/3)\tau + n^2 \log(p)\tau + \frac{3}{2}n^2\tau + \frac{mn}{p}\tau$
hybrid	C	$n^2 \log(p)\alpha + 2n \log(p)\beta$

Table 2.3. Arithmetic (A) and communication (C) complexities for the hybrid algorithm. Matrix size is $m \times n$, with p processors, τ = time needed for one flop, and message transfer time is $M\alpha + \beta$ for M bytes at α bytes per time unit with a latency of β.

Paragon iPSC-286 hypercube with 16 processors. The experimental results obtained correlate closely to those predicted, therefore validating the derived complexities listed in Tables 2.1, 2.2, and 2.3.

Chapter 3

A LOAD BALANCED HYBRID PARALLEL QR ALGORITHM

This chapter will introduce a new parallel QR decomposition algorithm. The approach taken for algorithm design and development has three key goals:

1. design with modern-day, widely available parallel architectures in mind,
2. maximize the parallel work between communications, and
3. load balance the amount of parallel work evenly among all processors.

The algorithm developed here achieves all three goals. In many respects it is similar to and draws on work done previously, however, the first goal creates different considerations with regard to maximizing parallelism and balancing load. Where the majority of previous algorithms balance Givens rotations across processors, the algorithm presented here load balances at a much finer level. It balances multiplications and additions evenly across processors by considering vector length when assigning Givens rotations to processors. As a result of the first goal, the first few sections will discuss the motivation behind the choice of the targeted architectures. Next, the details and an example of the new parallel QR algorithm are presented. Finally, a complexity analysis of the algorithm is performed and predicted performance is presented. Chapter 5 will present a comparison of the predicted and actual performance on a variety of matrix sizes.

3.1 Trends in Parallel Architectures

The market turmoil of the late 80's and early 90's that hit parallel computer manufacturers has been widely discussed in the literature. Just a few of the articles covering this topic include (Bell, 1994), (Cownie, 1994), (Cybenko, 1996), (Quinn, 1994), (Siegel *et al.*, 1996), (Wallach, 1994), and (Wladawsky-Berger, 1994). One can draw several conclusions from the lessons still being learned from the "dark ages" of parallel computing. In the following sections, specific conclusions will be discussed and speculation on emerging trends in parallel systems, hardware and software, will be offered.

3.1.1 Parallel Programming Is Hard

This may seem obvious, but the real implications of this statement are still being discovered and sinking in. When designing and implementing parallel algorithms and applications, it is not enough to identify and exploit concurrency. This is arguably the easiest part of the problem. The considerably harder part is picking the appropriate parallel programming model, designing with respect to that model, mapping the model to an architecture, and identifying a real world machine with a similar architecture. Then, possibly the most difficult step, is optimizing the algorithm in light of all the constraints imposed by the hardware implementation and still achieving a cost effective speedup in light of the cost of the machine, cost of the software development, and speedup achieved.

It is exactly the unexpected finer points of parallel implementation that are beginning to dominate discussions in parallel computing. It has been said that "everybody who wanted a CM-5 (and could pay for it) already had one (Quinn, 1994)." While this is less true for parallel machines in general, after the initial wave of new machines in the late 80's, organizations stopped buying parallel machines. Why? One debatable answer is that once they had them, they didn't know what to do with them. Achieving any speedup at all was proving much more difficult than expected because of hardware design realities. Memory throughput and inter-processor communication began to become the bottlenecks. It became apparent that parallelism wasn't for everyone (Pancake, 1996).

In light of early applications, more thought went into correctly matching a problem with an appropriate solution method. Excellent discussions of this include (Foster, 1995), (Morton & Tyler, 1996), and (Pancake, 1996). A spectrum of parallel programming models also began to form, with implementation difficulty increasing as the model moved from sequential, to shared memory, to message passing.

3.1.2 Parallel Computer Manufacturers Did Not Survive

Not one of the companies whose primary business was producing parallel computers survived. Most failed, and the lucky few were acquired by more traditional computer chip and hardware manufacturers. The parallel machines commercially available today fall into two main categories, and they are produced by:

- companies whose primary business is traditional sequential computing and that have been successful enough to support the research and development required to produce state of the art parallel machines, namely Intel and IBM, or

- companies that have a successful sequential architecture and for a small investment have developed parallel architectures using small numbers of existing mass market CPU's, namely SGI, Sun, HP, and Intel.

Regardless of technical merit, the market has won the first battle of this war. Currently, the most cost effective machines to program and to buy are shared memory machines with modest numbers of processors (2 to 64). The shared memory programming model is also the easiest to work with when developing parallel applications. There is no reason not to expect the parallel computing industry to follow in the footsteps of traditional software development where we will see the cost of computational solutions dominated by parallel software development.

3.1.3 Distributed Shared Memory

The largest hurdle for shared memory architectures is due to physical restrictions which limit their scalability. This is beginning to be overcome with the introduction of Distributed Shared Memory machines (Protiĉ *et al.*, 1996) and also with shared memory programming models on networks of workstations (Clarke, 1997) and (Cordsen *et al.*, 1997). With the above trends in mind, and with distributed shared memory machines now available, it seems most appropriate to target new algorithm development in this direction. Further discussions of designing, implementing, and tuning shared memory algorithms can be found in (Adve & Gharachorloo, 1996), (Charny, 1996), (Islam & Campbell, 1992), (Sun & Zhu, 1995), and (Sun & Zhu, 1996).

3.2 Goals and Approach

The algorithms being proposed here differ from existing techniques in three key ways which directly support the previously stated goals:

- The assumption that each Givens rotation can be computed in the same amount of time regardless of vector length will not be made. While this assumption was true for vector machines and smaller problem sizes, the trends in parallel architectures are moving away from vector machines.[1] Therefore, work will be balanced across processors by considering vector length when assigning the vectors to processors.

- The amount of computation done between communication steps will be maximized. Instead of performing one Householder reflection, or enough Givens rotations to zero one column, each processor will be allowed to proceed with computation and zero as many elements as possible before a communication step is required.

- Algorithm development specifically targets shared memory parallel machines, and the shared memory programming model. These machines are the most widely available and cost effective machines. In addition, there is some movement toward a shared memory programming model even on the most cost effective parallel computer, networks of workstations ((Clarke, 1997) and (Cordsen *et al.*, 1997)).

These algorithms depart from common practice with the first item, but with the second they will build on the ideas introduced by Pothen and Raghavan in (Pothen & Raghavan, 1989) by considering communication as an algorithmic cost in a QR factorization routine. One version will also use the idea from (Pothen & Raghavan, 1989) of a hybrid algorithm, i.e. using Householder reflections for computation local to one processor because of the lower complexity, and using Givens rotations for inter-processor annihilation because of their independence.

[1] In general, U.S. manufacturers have moved away from production of vector machines. There is active research, development, and manufacturing of new parallel vector machines elsewhere by such companies as NEC and Fujitsu.

These algorithms will be most suitable for non-vector parallel computers with fewer processors than equations in the system. This is expected to be the case for most current and future environments. The size of the systems of equations in modelling physical systems and solving problems has grown dramatically, and the trends in parallel computing and physical limitations of hardware are working together to keep this true. Also, by maximizing computation between communication steps, these algorithms may become well suited to the highly distributed processing found in networks of workstations.

3.3 Algorithm Overview

The new QR decomposition algorithm proceeds in two stages, with the potential of the two stages being repeated multiple times in the process of one decomposition. For simplicity, these stages will be named in a way similar to the naming of the stages in the hybrid algorithm introduced by Pothen and Raghavan in (Pothen & Raghavan, 1989). A description of the two stages and their names follows:

1. The internal reflections stage (IR): the rows of the matrix are divided evenly among the processors with each processor getting a block of size $(m/p \times n)$. During this stage, each processor performs $(m/p) - 1$ Householder reflections which results in a matrix with p upper trapezoidal submatrices as shown in Figure 3.1.

2. The balanced rotations stage (BR): the jagged edges of the matrix are cleaned up using Givens rotations. This stage consists of $(m/p) - 1$ smaller **steps**. During each step, the following sequence takes place:

- The rows of block 1 (matrix rows 1 to m/p which were annihilated by processor 1 via Householder reflections in the IR stage) are assigned in a balanced manner as pivot rows to the processors.
- The step proceeds by the processors zeroing (via Givens rotations) all the elements in the remaining blocks that are of the same length as their assigned pivot row(s) from block 1. This step annihilates the first diagonal in blocks 2 to p.
- The above two steps are repeated, annihilating the diagonals of decreasing length in blocks 2 to p until the matrix appears as shown in Figure 3.2.
- The balanced assignment of rows to processors will be described fully in Section 3.5.

The first m/p rows of the matrix are now in the desired format. The next step is to perform both the IR and BR stages on the submatrix depicted in Figure 3.2 by Y entries (of size 25×9 in this case). This process is repeated in a recursive nature on the remaining submatrices until one of two terminating conditions is reached. Terminating condition one occurs when, during the IR stage processor one reaches the n^{th} column of the currently active submatrix. When this is the case, the BR stage will fully annihilate all the elements in the rows greater than n. The second terminating condition occurs when processor one performs all its Householder reflections in the IR stage and the BR stage completes, leaving a submatrix with only one column to be eliminated. In this case, the final column can be eliminated with either a processor pairing strategy as described in (Pothen & Raghavan, 1989), or sequentially, since in the end, the final Givens rotation must use element $A(n, n)$ to eliminate $A(n+1, n)$.

A quick note should be made concerning the size of the submatrix used during the recursive call of the IR and BR stages. Examining the example in Figure 3.2

$$\begin{array}{r|cccccccccccccccc|}
1 & X & X & X & X & X & X & X & X & X & X & X & X & X & X & X & X \\
2 & 0 & X & X & X & X & X & X & X & X & X & X & X & X & X & X & X \\
3 & 0 & 0 & X & X & X & X & X & X & X & X & X & X & X & X & X & X \\
4 & 0 & 0 & 0 & X & X & X & X & X & X & X & X & X & X & X & X & X \\
5 & 0 & 0 & 0 & 0 & X & X & X & X & X & X & X & X & X & X & X & X \\
6 & 0 & 0 & 0 & 0 & 0 & X & X & X & X & X & X & X & X & X & X & X \\
7 & 0 & 0 & 0 & 0 & 0 & 0 & X & X & X & X & X & X & X & X & X & X \\
8 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & X & X & X & X & X & X & X & X & X \\
9 & X & X & X & X & X & X & X & X & X & X & X & X & X & X & X & X \\
10 & 0 & X & X & X & X & X & X & X & X & X & X & X & X & X & X & X \\
11 & 0 & 0 & X & X & X & X & X & X & X & X & X & X & X & X & X & X \\
12 & 0 & 0 & 0 & X & X & X & X & X & X & X & X & X & X & X & X & X \\
13 & 0 & 0 & 0 & 0 & X & X & X & X & X & X & X & X & X & X & X & X \\
14 & 0 & 0 & 0 & 0 & 0 & X & X & X & X & X & X & X & X & X & X & X \\
15 & 0 & 0 & 0 & 0 & 0 & 0 & X & X & X & X & X & X & X & X & X & X \\
16 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & X & X & X & X & X & X & X & X & X \\
17 & X & X & X & X & X & X & X & X & X & X & X & X & X & X & X & X \\
18 & 0 & X & X & X & X & X & X & X & X & X & X & X & X & X & X & X \\
19 & 0 & 0 & X & X & X & X & X & X & X & X & X & X & X & X & X & X \\
20 & 0 & 0 & 0 & X & X & X & X & X & X & X & X & X & X & X & X & X \\
21 & 0 & 0 & 0 & 0 & X & X & X & X & X & X & X & X & X & X & X & X \\
22 & 0 & 0 & 0 & 0 & 0 & X & X & X & X & X & X & X & X & X & X & X \\
23 & 0 & 0 & 0 & 0 & 0 & 0 & X & X & X & X & X & X & X & X & X & X \\
24 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & X & X & X & X & X & X & X & X & X \\
25 & X & X & X & X & X & X & X & X & X & X & X & X & X & X & X & X \\
26 & 0 & X & X & X & X & X & X & X & X & X & X & X & X & X & X & X \\
27 & 0 & 0 & X & X & X & X & X & X & X & X & X & X & X & X & X & X \\
28 & 0 & 0 & 0 & X & X & X & X & X & X & X & X & X & X & X & X & X \\
29 & 0 & 0 & 0 & 0 & X & X & X & X & X & X & X & X & X & X & X & X \\
30 & 0 & 0 & 0 & 0 & 0 & X & X & X & X & X & X & X & X & X & X & X \\
31 & 0 & 0 & 0 & 0 & 0 & 0 & X & X & X & X & X & X & X & X & X & X \\
32 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & X & X & X & X & X & X & X & X & X \\
\multicolumn{1}{r}{} & \multicolumn{1}{c}{1} & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 & \multicolumn{1}{c}{16}
\end{array}$$

FIG. 3.1. Matrix A after completion of the IR stage. Example using $m = 32, n = 16$, and $p = 4$. The X symbol denotes an element with information while a zero denotes an annihilated entry.

shows that row 8 is in the desired format, but it is included in the submatrix for the 2^{nd} recursive call. During the first iteration of the IR and BR stages, column 8 could have been annihilated using Givens rotations in the BR stage. Instead, the algorithm stops at column 7 in order for more efficient Householder reflections to be used in the annihilation of column 8 during the IR stage in the 2^{nd} recursive call.

This new hybrid algorithm is different from that introduced in (Pothen & Raghavan, 1989) and described in Chapter 2 in two primary ways. First, in Pothen and Raghavan's algorithm, the IR phase always consists of exactly p Householder reflections performed in parallel followed by their recursive elimination phase using processor pairing to eliminate $p - 1$ single elements via Givens rotations. At this point a parallel synchronization and a communication is required which is normally very costly. The algorithm described above maximizes the work done in the IR stage by letting each processor perform the maximum possible number of Householder reflections $(\frac{m}{p} - 1)$ before a parallel synchronization is required to proceed with the BR stage. Second, the algorithm described above balances the Givens rotation load across processors and performs $\left((p-1)\lfloor\frac{m}{p} - 1\rfloor\right)$ simultaneous Givens rotations between synchronizations.

$$\begin{array}{r|cccccccccccccccc|}
1 & X & X & X & X & X & X & X & X & X & X & X & X & X & X & X & X \\
2 & 0 & X & X & X & X & X & X & X & X & X & X & X & X & X & X & X \\
3 & 0 & 0 & X & X & X & X & X & X & X & X & X & X & X & X & X & X \\
4 & 0 & 0 & 0 & X & X & X & X & X & X & X & X & X & X & X & X & X \\
5 & 0 & 0 & 0 & 0 & X & X & X & X & X & X & X & X & X & X & X & X \\
6 & 0 & 0 & 0 & 0 & 0 & X & X & X & X & X & X & X & X & X & X & X \\
7 & 0 & 0 & 0 & 0 & 0 & 0 & X & X & X & X & X & X & X & X & X & X \\
8 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & Y & Y & Y & Y & Y & Y & Y & Y & Y \\
9 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & Y & Y & Y & Y & Y & Y & Y & Y & Y \\
10 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & Y & Y & Y & Y & Y & Y & Y & Y & Y \\
11 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & Y & Y & Y & Y & Y & Y & Y & Y & Y \\
12 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & Y & Y & Y & Y & Y & Y & Y & Y & Y \\
13 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & Y & Y & Y & Y & Y & Y & Y & Y & Y \\
14 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & Y & Y & Y & Y & Y & Y & Y & Y & Y \\
15 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & Y & Y & Y & Y & Y & Y & Y & Y & Y \\
16 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & Y & Y & Y & Y & Y & Y & Y & Y & Y \\
17 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & Y & Y & Y & Y & Y & Y & Y & Y & Y \\
18 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & Y & Y & Y & Y & Y & Y & Y & Y & Y \\
19 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & Y & Y & Y & Y & Y & Y & Y & Y & Y \\
20 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & Y & Y & Y & Y & Y & Y & Y & Y & Y \\
21 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & Y & Y & Y & Y & Y & Y & Y & Y & Y \\
22 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & Y & Y & Y & Y & Y & Y & Y & Y & Y \\
23 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & Y & Y & Y & Y & Y & Y & Y & Y & Y \\
24 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & Y & Y & Y & Y & Y & Y & Y & Y & Y \\
25 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & Y & Y & Y & Y & Y & Y & Y & Y & Y \\
26 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & Y & Y & Y & Y & Y & Y & Y & Y & Y \\
27 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & Y & Y & Y & Y & Y & Y & Y & Y & Y \\
28 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & Y & Y & Y & Y & Y & Y & Y & Y & Y \\
29 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & Y & Y & Y & Y & Y & Y & Y & Y & Y \\
30 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & Y & Y & Y & Y & Y & Y & Y & Y & Y \\
31 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & Y & Y & Y & Y & Y & Y & Y & Y & Y \\
32 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & Y & Y & Y & Y & Y & Y & Y & Y & Y \\
\multicolumn{1}{r}{} & \multicolumn{1}{c}{1} & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 & \multicolumn{1}{c}{16}
\end{array}$$

FIG. 3.2. Matrix A after completion of the BR stage. Example using $m = 32, n = 16$, and $p = 4$. The X symbol denotes an element with information while a zero denotes an annihilated entry. The Y symbol denotes the elements of the submatrix to be worked on during the second application of the IR and BR stages.

$$\begin{array}{r|cccccccccccccccc|}
1 & X & X & X & X & X & X & X & X & X & X & X & X & X & X & X & X \\
2 & 0 & X & X & X & X & X & X & X & X & X & X & X & X & X & X & X \\
3 & 0 & 0 & X & X & X & X & X & X & X & X & X & X & X & X & X & X \\
4 & 0 & 0 & 0 & X & X & X & X & X & X & X & X & X & X & X & X & X \\
5 & 0 & 0 & 0 & 0 & X & X & X & X & X & X & X & X & X & X & X & X \\
6 & 0 & 0 & 0 & 0 & 0 & X & X & X & X & X & X & X & X & X & X & X \\
7 & 0 & 0 & 0 & 0 & 0 & 0 & X & X & X & X & X & X & X & X & X & X \\
8 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & X & X & X & X & X & X & X & X & X \\
9 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & X & X & X & X & X & X & X & X \\
10 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & X & X & X & X & X & X & X \\
11 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & X & X & X & X & X & X \\
12 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & X & X & X & X & X \\
13 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & X & X & X & X \\
14 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & X & X & X \\
15 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & X & X \\
16 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & X \\
17 & X & X & X & X & X & X & X & X & X & X & X & X & X & X & X & X \\
18 & 0 & X & X & X & X & X & X & X & X & X & X & X & X & X & X & X \\
19 & 0 & 0 & X & X & X & X & X & X & X & X & X & X & X & X & X & X \\
20 & 0 & 0 & 0 & X & X & X & X & X & X & X & X & X & X & X & X & X \\
21 & 0 & 0 & 0 & 0 & X & X & X & X & X & X & X & X & X & X & X & X \\
22 & 0 & 0 & 0 & 0 & 0 & X & X & X & X & X & X & X & X & X & X & X \\
23 & 0 & 0 & 0 & 0 & 0 & 0 & X & X & X & X & X & X & X & X & X & X \\
24 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & X & X & X & X & X & X & X & X & X \\
25 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & X & X & X & X & X & X & X & X \\
26 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & X & X & X & X & X & X & X \\
27 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & X & X & X & X & X & X \\
28 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & X & X & X & X & X \\
29 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & X & X & X & X \\
30 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & X & X & X \\
31 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & X & X \\
32 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & X \\
\multicolumn{1}{r}{} & \multicolumn{1}{c}{1} & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 & \multicolumn{1}{c}{16}
\end{array}$$

FIG. 3.3. Case 1: After IR stage. $m = 32, n = 16, p = 2$, and $\frac{m}{p} = 16 = n$.

3.4 A Note on Matrix "Shape"

When applying the parallel QR decomposition algorithm presented here, one of three conditions can occur with respect to matrix shape. Each case yields slightly different results.

3.4.1 Case 1: $\frac{\mathbf{m}}{\mathbf{p}} = \mathbf{n}$

When this condition occurs, all the blocks of the matrix are square, including the first. The IR stage of the algorithm will annihilate all the elements in the first block with $(m/p) - 1$ Householder reflections. The resulting matrix that is passed to the BR stage will have all the entries in rows $n + 1$ to m eliminated by Givens rotations and the matrix will be in the desired form after only one iteration of the algorithm. Figure 3.3 depicts the structure of the matrix after the IR stage in this case.

$$\begin{array}{r cccccccccccccccc}
1 & X & X & X & X & X & X & X & X & X & X & X & X & X & X & X & X \\
2 & 0 & X & X & X & X & X & X & X & X & X & X & X & X & X & X & X \\
3 & 0 & 0 & X & X & X & X & X & X & X & X & X & X & X & X & X & X \\
4 & 0 & 0 & 0 & X & X & X & X & X & X & X & X & X & X & X & X & X \\
5 & 0 & 0 & 0 & 0 & X & X & X & X & X & X & X & X & X & X & X & X \\
6 & 0 & 0 & 0 & 0 & 0 & X & X & X & X & X & X & X & X & X & X & X \\
7 & 0 & 0 & 0 & 0 & 0 & 0 & X & X & X & X & X & X & X & X & X & X \\
8 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & X & X & X & X & X & X & X & X & X \\
9 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & X & X & X & X & X & X & X & X \\
10 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & X & X & X & X & X & X & X \\
11 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & X & X & X & X & X & X \\
12 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & X & X & X & X & X \\
13 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & X & X & X & X \\
14 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & X & X & X \\
15 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & X & X \\
16 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & X \\
17 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
18 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
19 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
20 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
21 & X & X & X & X & X & X & X & X & X & X & X & X & X & X & X & X \\
22 & 0 & X & X & X & X & X & X & X & X & X & X & X & X & X & X & X \\
23 & 0 & 0 & X & X & X & X & X & X & X & X & X & X & X & X & X & X \\
24 & 0 & 0 & 0 & X & X & X & X & X & X & X & X & X & X & X & X & X \\
25 & 0 & 0 & 0 & 0 & X & X & X & X & X & X & X & X & X & X & X & X \\
26 & 0 & 0 & 0 & 0 & 0 & X & X & X & X & X & X & X & X & X & X & X \\
27 & 0 & 0 & 0 & 0 & 0 & 0 & X & X & X & X & X & X & X & X & X & X \\
28 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & X & X & X & X & X & X & X & X & X \\
29 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & X & X & X & X & X & X & X & X \\
30 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & X & X & X & X & X & X & X \\
31 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & X & X & X & X & X & X \\
32 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & X & X & X & X & X \\
33 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & X & X & X & X \\
34 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & X & X & X \\
35 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & X & X \\
36 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & X \\
37 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
38 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
39 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
40 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 & 16
\end{array}$$

FIG. 3.4. Case 2: After IR stage. $m = 40, n = 16, p = 2$, and $\frac{m}{p} = 20 > n$.

3.4.2 Case 2: $\frac{\mathbf{m}}{\mathbf{p}} > \mathbf{n}$

When this occurs, the IR stage of the algorithm will annihilate all the elements of the matrix in the upper block with m/p Householder reflections, and the resulting matrix will have bands of zeroes present. This is the case where the highest parallel efficiencies are possible because the greatest amount of work is done in the IR stage. All that is left for the BR stage is to eliminate the remaining, relatively few elements via Givens rotations. Figure 3.4 depicts the structure of the matrix after the IR stage in this case.

3.4.3 Case 3: $\frac{\mathbf{m}}{\mathbf{p}} < \mathbf{n}$

When this final case occurs, the structure after the IR stage will be similar to that depicted in Figure 3.1. Once the BR stage completes, the whole algorithm must be recursively applied to the $(m - \frac{m}{p} + 1) \times (n - \frac{m}{p} + 1)$ submatrix depicted with Y symbols in Figure 3.2. The algorithm will terminate when the entering condition is that of either Case 1 or Case 2 above.

3.5 Algorithm Load Balancing Details

This section describes the algorithmic details of the load balancing performed in the Balanced Rotations (BR) stage. When the structure of the matrix is examined after completion of the IR stage (see Fig. 3.1), there are $(p-1)$ blocks that have m/p rows and n columns, and it can be determined that

$$(p-1)\left[\frac{\frac{m}{p}\left(\frac{m}{p}-1\right)}{2}\right]$$

Givens rotations are needed to "clean up" the ragged edges of the matrix in the BR stage. The aforementioned goal is to balance work (multiplications and additions, not Givens rotations) across processors. On non-vector parallel machines, it takes different amounts of time to perform Givens rotations on vectors of different length.

Inspiration for the load balancing method used in this algorithm was gained while examining the number of Givens rotations used to perform sequential QR decomposition. Consider an $n \times n$ matrix. In order to transform the matrix to upper triangular, column one needs $n-1$ Givens rotations, column two needs $n-2$ Givens rotations, etc. up to column $n-1$, which needs only one Givens rotation. This simplifies to

$$\frac{n(n-1)}{2}$$

Givens rotations, which is simply the sum of the integers from 1 to $n-1$. The general form to sum integers is

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$$

which leads to the simple realization that the sum of 1 and n is equal to the sum of 2 and $n-1$, which is equal to the sum of 3 and $n-2$, etc. This is precisely the idea behind balancing the work in the BR stage of the algorithm.

Applying the idea above, a way to balance the BR stage load across p processors is shown in Figure 3.5. This figure shows two blocks of a matrix as an example with $m/p = 12, n = 16$, and $p = 3$. The top block in the figure is the pivot block (rows 1 to m/p in a full matrix example), and the next block is the block whose elements will be zeroed. Notice that the processor assignment is done in a cyclic way. This results in 2 cycles that contain 6 pivot rows each. In the general case, cycle length is $2p$. This example illustrates one step of the BR stage of the algorithm, and the work in this case is split among the processors as follows:

Processor	Rotations	Lengths	Total Work
1	4	16,11,10,5	276 flops
2	4	15,12,9,6	276 flops
3	4	14,13,8,7	276 flops

The work is calculated using the complexity of one Givens rotation from Equation 1.7 in Section 1.4.5 and applying it to the vector length. This results in a perfectly load balanced series of Givens rotations. Recall that the matrix in question contains

$$\begin{array}{r|cccccccccccccccc|}
1 & X & X & X & X & X & X & X & X & X & X & X & X & X & X & X & X \\
2 & 0 & X & X & X & X & X & X & X & X & X & X & X & X & X & X & X \\
3 & 0 & 0 & X & X & X & X & X & X & X & X & X & X & X & X & X & X \\
4 & 0 & 0 & 0 & X & X & X & X & X & X & X & X & X & X & X & X & X \\
5 & 0 & 0 & 0 & 0 & X & X & X & X & X & X & X & X & X & X & X & X \\
6 & 0 & 0 & 0 & 0 & 0 & X & X & X & X & X & X & X & X & X & X & X \\
7 & 0 & 0 & 0 & 0 & 0 & 0 & X & X & X & X & X & X & X & X & X & X \\
8 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & X & X & X & X & X & X & X & X & X \\
9 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & X & X & X & X & X & X & X & X \\
10 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & X & X & X & X & X & X & X \\
11 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & X & X & X & X & X & X \\
12 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & X & X & X & X & X \\
13 & X & X & X & X & X & X & X & X & X & X & X & X & X & X & X & X \\
14 & 0 & X & X & X & X & X & X & X & X & X & X & X & X & X & X & X \\
15 & 0 & 0 & X & X & X & X & X & X & X & X & X & X & X & X & X & X \\
16 & 0 & 0 & 0 & X & X & X & X & X & X & X & X & X & X & X & X & X \\
17 & 0 & 0 & 0 & 0 & X & X & X & X & X & X & X & X & X & X & X & X \\
18 & 0 & 0 & 0 & 0 & 0 & X & X & X & X & X & X & X & X & X & X & X \\
19 & 0 & 0 & 0 & 0 & 0 & 0 & X & X & X & X & X & X & X & X & X & X \\
20 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & X & X & X & X & X & X & X & X & X \\
21 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & X & X & X & X & X & X & X & X \\
22 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & X & X & X & X & X & X & X \\
23 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & X & X & X & X & X & X \\
24 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & X & X & X & X & X \\
\multicolumn{1}{r}{} & \multicolumn{1}{c}{1} & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 & \multicolumn{1}{c}{16}
\end{array}$$

Pivot Row	Zero Row	Processor Assigned	Length	flops $\sim (6n+6)$
1	13	1	16	102
2	14	2	15	96
3	15	3	14	90
4	16	3	13	84
5	17	2	12	78
6	18	1	11	72
7	19	1	10	66
8	20	2	9	60
9	21	3	8	54
10	22	3	7	48
11	23	2	6	42
12	24	1	5	36

FIG. 3.5. Load balanced Givens assignment, $m/p = 12, n = 16$, and $p = 3$.

many blocks like that in Figure 3.5, and that it looks like the matrix from the previous example in Figure 3.1. The BR stage of the algorithm actually assigns all the rows in blocks $2 \ldots p$ to be eliminated by pivoting on rows $1 \ldots m/p$ using this scheme.

After one application of the load balancing scheme described above, the matrix is not yet in the desired form depicted in Figure 3.2. It appears as shown in Figure 3.6. Only the first diagonal of each block has been eliminated. The same load balancing scheme is applied to the matrix repeatedly until it appears in the form depicted in Figure 3.2. Figure 3.7 shows the row-wise assignment for each step and the work done during the full Balanced Rotations (BR) stage of the algorithm for an example matrix. Table 3.1 summarizes the work done by each processor during each step for this example.

$$
\begin{array}{r|cccccccccccccccc|}
1 & X & X & X & X & X & X & X & X & X & X & X & X & X & X & X & X \\
2 & 0 & X & X & X & X & X & X & X & X & X & X & X & X & X & X & X \\
3 & 0 & 0 & X & X & X & X & X & X & X & X & X & X & X & X & X & X \\
4 & 0 & 0 & 0 & X & X & X & X & X & X & X & X & X & X & X & X & X \\
5 & 0 & 0 & 0 & 0 & X & X & X & X & X & X & X & X & X & X & X & X \\
6 & 0 & 0 & 0 & 0 & 0 & X & X & X & X & X & X & X & X & X & X & X \\
7 & 0 & 0 & 0 & 0 & 0 & 0 & X & X & X & X & X & X & X & X & X & X \\
8 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & X & X & X & X & X & X & X & X & X \\
9 & 0 & X & X & X & X & X & X & X & X & X & X & X & X & X & X & X \\
10 & 0 & 0 & X & X & X & X & X & X & X & X & X & X & X & X & X & X \\
11 & 0 & 0 & 0 & X & X & X & X & X & X & X & X & X & X & X & X & X \\
12 & 0 & 0 & 0 & 0 & X & X & X & X & X & X & X & X & X & X & X & X \\
13 & 0 & 0 & 0 & 0 & 0 & X & X & X & X & X & X & X & X & X & X & X \\
14 & 0 & 0 & 0 & 0 & 0 & 0 & X & X & X & X & X & X & X & X & X & X \\
15 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & X & X & X & X & X & X & X & X & X \\
16 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & X & X & X & X & X & X & X & X & X \\
17 & 0 & X & X & X & X & X & X & X & X & X & X & X & X & X & X & X \\
18 & 0 & 0 & X & X & X & X & X & X & X & X & X & X & X & X & X & X \\
19 & 0 & 0 & 0 & X & X & X & X & X & X & X & X & X & X & X & X & X \\
20 & 0 & 0 & 0 & 0 & X & X & X & X & X & X & X & X & X & X & X & X \\
21 & 0 & 0 & 0 & 0 & 0 & X & X & X & X & X & X & X & X & X & X & X \\
22 & 0 & 0 & 0 & 0 & 0 & 0 & X & X & X & X & X & X & X & X & X & X \\
23 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & X & X & X & X & X & X & X & X & X \\
24 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & X & X & X & X & X & X & X & X & X \\
25 & 0 & X & X & X & X & X & X & X & X & X & X & X & X & X & X & X \\
26 & 0 & 0 & X & X & X & X & X & X & X & X & X & X & X & X & X & X \\
27 & 0 & 0 & 0 & X & X & X & X & X & X & X & X & X & X & X & X & X \\
28 & 0 & 0 & 0 & 0 & X & X & X & X & X & X & X & X & X & X & X & X \\
29 & 0 & 0 & 0 & 0 & 0 & X & X & X & X & X & X & X & X & X & X & X \\
30 & 0 & 0 & 0 & 0 & 0 & 0 & X & X & X & X & X & X & X & X & X & X \\
31 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & X & X & X & X & X & X & X & X & X \\
32 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & X & X & X & X & X & X & X & X & X \\
\hline
\multicolumn{1}{r}{} & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 & \multicolumn{1}{c}{16}
\end{array}
$$

FIG. 3.6. Matrix A after step 1 of the BR stage. Example using $m = 32, n = 16$, and $p = 4$. The X symbol denotes an element with information while a zero denotes an annihilated entry.

	Step							Work						
Row	1	2	3	4	5	6	7	1	2	3	4	5	6	7
1	P_1													
2	P_2	P_1												
3	P_3	P_2	P_1											
4	P_4	P_3	P_2	P_1										
5	P_4	P_4	P_3	P_2	P_1									
6	P_3	P_4	P_4	P_3	P_2	P_1								
7	P_2	P_3	P_4	P_4	P_3	P_2	P_1							
8														
9	P_1	P_1	P_1	P_1	P_1	P_1	P_1	102	96	90	84	78	72	66
10	P_2	P_2	P_2	P_2	P_2	P_2		96	90	84	78	72	66	
11	P_3	P_3	P_3	P_3	P_3			90	84	78	72	66		
12	P_4	P_4	P_4	P_4				84	78	72	66			
13	P_4	P_4	P_4					78	72	66				
14	P_3	P_3						72	66					
15	P_2							66						
16														
17	P_1	P_1	P_1	P_1	P_1	P_1	P_1	102	96	90	84	78	72	66
18	P_2	P_2	P_2	P_2	P_2	P_2		96	90	84	78	72	66	
19	P_3	P_3	P_3	P_3	P_3			90	84	78	72	66		
20	P_4	P_4	P_4	P_4				84	78	72	66			
21	P_4	P_4	P_4					78	72	66				
22	P_3	P_3						72	66					
23	P_2							66						
24														
25	P_1	P_1	P_1	P_1	P_1	P_1	P_1	102	96	90	84	78	72	66
26	P_2	P_2	P_2	P_2	P_2	P_2		96	90	84	78	72	66	
27	P_3	P_3	P_3	P_3	P_3			90	84	78	72	66		
28	P_4	P_4	P_4	P_4				84	78	72	66			
29	P_4	P_4	P_4					78	72	66				
30	P_3	P_3						72	66					
31	P_2							66						
32														

FIG. 3.7. Example of work done during the BR stage, $m = 32, n = 16$, and $p = 4$. Entries in the Step column indicate which processor is assigned to do the annihilation of that element during that step. Integers in the Work column contain the number of operations required to perform the rotation at that stage. The rows of block 1 are used as the pivots (rows 1 to 7), and the remaining rows are assigned as shown for annihilation.

	Steps							
Processor	1	2	3	4	5	6	7	Total
1	102	96	90	84	78	72	66	588
2	162	90	84	78	72	66		552
3	162	150	78	72	66			528
4	162	150	138	66				516
Total Work								2184
Max/Step	162	150	138	84	78	72	66	
Parallel Work								750

Table 3.1. Summary of work *for one block* during BR stage. Parallel work is 750 flops, total work is 2184 flops, for a parallel speedup of 2.9 and an expected parallel efficiency of 72.8%.

Even in this simple example, the work is well balanced among the processors. We'll see in the next section that the parallel efficiency (measure of load balancing) becomes better as problem size increases and the number of processors available remains relatively small (with respect to problem size). One quick note is appropriate concerning the current implementation of the algorithm. The code used for results and analysis in Chapter 5 forces a synchronization between steps during the algorithm's BR stage. The interleaving of the Givens rotations to solve the problem is theoretically possible, but no solution has been possible given the implementation language, architecture, and programming paradigm. This results in a slightly less efficient load balance, which is easily seen in the simple example of Table 3.1. The actual imbalance with larger problem sizes is nearly insignificant.

3.6 Algorithmic Complexity

The following sections generalize the ideas introduced in Sections 3.3 and 3.5. The complexity equations are presented only for one iteration of the two algorithmic stages (IR and BR). Depending on the problem size, ratio of rows to columns, and number of processors, multiple recursive calls to the algorithm might be necessary. In that case, the complexities presented still hold, but the input problem size changes accordingly to reflect the size of the sub-matrix being factored. This case was discussed previously in Section 3.4.

3.6.1 Complexity of Internal Reflections

The Internal Reflections (IR) stage of the algorithm is perfectly load balanced. During this stage, the matrix is partitioned into p blocks, each of size $m/p \times n$. Using these dimensions, and the complexity of sequential Householder factorization from Table 1.1, the IR stage has a parallel complexity of

$$O\left(2n^2\left(\frac{m}{p}-\frac{n}{3}\right)\right).$$

It is easy to see that linear speedup and perfect parallel efficiency will result from the IR stage of the algorithm.

3.6.2 Complexity of Balanced Rotations

Analysis of the Balanced Rotations (BR) stage of the algorithm is not nearly as straightforward. There are two sources of load imbalance in this stage. As seen previously in Section 3.5, the BR stage must perform

$$(p-1)\left[\frac{\frac{m}{p}\left(\frac{m}{p}-1\right)}{2}\right]$$

Givens rotations at the cost of $6i+6$ flops per rotation where i is the length of the vectors in the rotation. The total work done in one block, during one step, of the BR stage of the algorithm is [2]

$$\sum_{i=n-\frac{m}{p}+2}^{n-j} (6i+6)$$

where j is the column index of the left most non-zero element in each of the 2 to p non-pivot blocks. There are $(m/p)-1$ steps and $p-1$ blocks, so the total work done during the BR stage of the algorithm is

$$(p-1)\left(\sum_{j=0}^{\frac{m}{p}-2}\left(\sum_{i=n-\frac{m}{p}+2}^{n-j}(6i+6)\right)\right)$$

which has one possible closed form of [3]

$$(p-1)\left(\frac{-2m^3}{p^3}+\frac{9m^2}{p^2}+\frac{3m^2n}{p^2}-\frac{7m}{p}-\frac{3mn}{p}\right).$$

Using the work assignment scheme described in Section 3.5, recall that work is assigned in cycles to the processors; the load and balance is therefore cyclic in nature. The blocksize (number of rows per block) at the beginning of the BR stage is

$$\text{blocksize} = \lfloor m/p - 1 \rfloor.$$

One row is subtracted from the m/p rows assigned to each processor during the IR stage because the last row of every block is already the desired vector length (for example, consider rows 8, 16, 24, and 32 in Figure 3.1).

At each step, a large number of the rows can be combined via Givens rotations with the work perfectly balanced across the processors. The perfectly balanced row

[2] The starting point of the summation is the length of the shortest row which has an element needing elimination at that step, and the ending point is the length of the longest row which has an element needing elimination at that step.

[3] Simplification of the above summation was done by the software package Mathematica.

assignment is done cyclicly as in Figure 3.5 with each cycle containing $2p$ rows. Each of the p blocks will have

$$\text{cycles} = \lfloor \frac{\text{blocksize}}{2p} \rfloor$$

cycles. After assignment of the cycles, there are

$$\text{leftovers} = \text{blocksize} - 2p(\text{cycles})$$

remaining rows which create the first source of the load imbalance. The maximum number of leftover rows at any one step is $2p - 1$. Considering this, there are three possible scenarios regarding the efficiency of the BR stage at each step:

1. blocksize is evenly divisible by p, and the leftmost non-zero column of blocks 2 to p is between 1 and (blocksize $- p$): the work is balanced perfectly (see Figure 3.5),

2. blocksize is not evenly divisible by p, and the leftmost non-zero column of blocks 2 to p is between 1 and (blocksize $- p$): the maximum work is done by processor $(p + 1) -$ (leftovers $\quad mod \quad p$) (see Figure 3.7, steps 1 to 3), and

3. the algorithm is working on the last p steps of the BR stage: the maximum work is always done by processor 1. This is the second source of the load imbalance. In this case, the number of idle processors increases by one each step until completion of the BR stage. A simple example of this "tailing off" can be seen in Table 3.1 during the last 4 steps.

The algorithm's BR stage must perform $\lfloor m/p - 1 \rfloor$ steps, and the parallel work is the sum of the maximum work done at each step. Dividing this by the total work in Equation 3.6.2 and normalizing with the number of processors gives a general form for expected parallel efficiency. Table 3.2 includes several calculated parallel efficiencies for differing problems sizes and number of processors. The figure illustrates the high level of load balancing achievable as m and n grow relative to p. The efficiencies in this figure were calculated using a simple program written in MATLAB which simulates the work distribution of the algorithm during each stage of the algorithm and each step of the BR stage, by following the process illustrated above.

3.6.3 Total Algorithmic Complexity

Sections 3.6.1 and 3.6.2 describe the complexity of one application of the IR and BR stages respectively. As Section 3.4 points out, when $m/p < n$ the IR and BR stages of the algorithm must be repeatedly applied to the submatrix left from the previous application. Therefore, the total algorithmic complexity is determined by

1. the complexity of the IR stage,

2. the complexity of the BR stage, and

3. the number of applications of the algorithm needed based on m, n, and p.

Table 3.3 presents several examples of total expected efficiency for various problem sizes and numbers of processors. Again, the values were calculated by simulating the work of the algorithm with a program written in MATLAB.

Two details that merit attention in Tables 3.2 and 3.3 are the effect of using more processors, and the effect of changing problem size. As the number of processing elements available increases, the efficiency for the same problem size decreases. This is because increasing the number of processors also increases the number of blocks and decreases the size of the blocks. This requires more recursive applications of the two stages of the algorithm to fully factor the input matrix. In order to make up for

p=2						
$m \times n$	50	100	200	400	800	1600
50	96.9406	96.4834	96.3063	96.2273	96.1899	96.1717
500	99.7215	99.9260	99.9808	99.7396	99.6518	99.6229
1000	99.7215	99.9260	99.9808	99.9951	99.8706	99.8261
4000	99.7215	99.9260	99.9808	99.9951	99.9988	99.9997
8000	99.7215	99.9260	99.9808	99.9951	99.9988	99.9997
p=4						
$m \times n$	50	100	200	400	800	1600
50	79.5820	79.0469	78.8024	78.6853	78.6279	78.5996
500	99.0500	99.7442	98.4391	97.9478	97.7856	97.7174
1000	99.0500	99.7442	99.9351	99.2210	98.9620	98.8770
4000	99.0500	99.7442	99.9351	99.9835	99.9958	99.8067
8000	99.0500	99.7442	99.9351	99.9835	99.9958	99.9990
p=8						
$m \times n$	50	100	200	400	800	1600
50	36.9966	37.2492	37.3748	37.4374	37.4687	37.4844
500	97.3569	92.5137	90.6489	90.0301	89.7699	89.6499
1000	97.3569	99.2677	96.3404	95.2470	94.8888	94.7389
4000	97.3569	99.2677	99.8072	99.9528	99.0962	98.7915
8000	97.3569	99.2677	99.8072	99.9528	99.9881	99.5495
p=16						
$m \times n$	50	100	200	400	800	1600
50	9.3445	9.3596	9.3672	9.3711	9.3730	9.3740
500	71.5498	68.1066	66.9637	66.4832	66.2616	66.1550
1000	90.2660	84.7403	81.6514	80.6489	80.2308	80.0386
4000	90.2660	97.9179	99.4401	96.1273	94.9384	94.5535
8000	90.2660	97.9179	99.4401	99.8535	98.0722	97.4400
p=32						
$m \times n$	50	100	200	400	800	1600
50	n/a	n/a	n/a	n/a	n/a	n/a
500	22.3132	22.9039	23.1773	23.3090	23.3736	23.4057
1000	42.1385	45.7517	47.1867	47.8328	48.1401	48.2900
4000	75.6023	90.9803	84.8545	81.5329	80.4851	80.0523
8000	75.6023	90.9803	98.2840	92.1524	89.9813	89.2891

Table 3.2. BR stage expected efficiency examples. m is listed on the leftmost column and n is listed across the first row of each block.

the overhead incurred by repeatedly calling the IR and BR stages of the algorithm on submatrices of decreasing size, problem size must increase to keep the blocksize large relative to the number of processors.

As problem size increases, but the number of processors available stays constant, two trends are noticeable. A simple way to symbolically represent total efficiency is

$$\text{Total Efficiency} = \left(\%\text{Work}_{\text{IR}} \times \text{IR}_{\text{efficiency}}\right) + \left(\%\text{Work}_{\text{BR}} \times \text{BR}_{\text{efficiency}}\right)$$

where

- $\%\text{Work}_{\text{IR}}$ is the percentage of total work done in the IR stage of the algorithm,
- $\text{IR}_{\text{efficiency}}$ is the efficiency of the IR stage of the algorithm (always 1),
- $\%\text{Work}_{\text{BR}}$ is the percentage of total work done in the BR stage of the algorithm, and
- $\text{BR}_{\text{efficiency}}$ is the efficiency of the BR stage of the algorithm.

As problem size increases and there are more rows than columns (overdetermined case), expected efficiency increases. The algorithm is well suited to factoring matrices of this type because increasing the ratio of rows to columns moves a larger percentage of the total work to the IR stage, which is perfectly parallel. As problem size increases and there are more columns than rows (underdetermined case), expected efficiency decreases. This occurs because the percentage of the total work done in the perfectly parallel IR stage is decreasing and the effect of the load imbalances in the BR stage becomes more pronounced.

p=2 $m \times n$	50	100	200	400	800	1600
50	97.4672	96.6914	96.4248	96.3113	96.2587	96.2333
500	99.8141	99.9507	99.9872	99.8226	99.6801	99.6340
1000	99.8141	99.9507	99.9872	99.9967	99.9123	99.8401
4000	99.8141	99.9507	99.9872	99.9967	99.9992	99.9998
8000	99.8141	99.9507	99.9872	99.9967	99.9992	99.9998
p=4 $m \times n$	50	100	200	400	800	1600
50	80.5229	78.5425	77.8119	77.4925	77.3426	77.2699
500	99.3647	99.8293	99.0141	98.3679	97.6745	97.4262
1000	99.3647	99.8293	99.9568	99.5130	99.1831	98.8310
4000	99.3647	99.8293	99.9568	99.9890	99.9972	99.8800
8000	99.3647	99.8293	99.9568	99.9890	99.9972	99.9993
p=8 $m \times n$	50	100	200	400	800	1600
50	36.8041	36.0283	35.7123	35.5685	35.4998	35.4662
500	98.2223	95.0547	93.3916	91.3354	88.6419	87.6856
1000	98.2223	99.5106	97.6595	96.7140	95.6264	94.1740
4000	98.2223	99.5106	99.8714	99.9686	99.4359	99.1788
8000	98.2223	99.5106	99.8714	99.9686	99.9921	99.7200
p=16 $m \times n$	50	100	200	400	800	1600
50	12.5749	12.5749	12.5749	12.5749	12.5749	12.5749
500	77.6950	74.9954	71.8967	67.0174	60.4840	58.4636
1000	93.2931	89.4679	86.9453	84.8680	81.6500	76.7413
4000	93.2931	98.6022	99.6260	97.5302	96.6694	96.0076
8000	93.2931	98.6022	99.6260	99.9023	98.7884	98.3370
p=32 $m \times n$	50	100	200	400	800	1600
50	n/a	n/a	n/a	n/a	n/a	n/a
500	28.7825	29.0312	27.6353	23.8049	20.9518	20.1300
1000	48.8887	53.1959	53.7644	51.5051	45.0112	38.2298
4000	82.2950	93.8005	89.5886	87.2417	85.5268	83.9052
8000	82.2950	93.8005	98.8494	94.8454	93.4224	92.4743

Table 3.3. Total expected efficiency examples. m is listed on the leftmost column and n is listed across the first row of each block.

Chapter 4

ELECTROMAGNETIC SCATTERING PROBLEM

This Chapter is included to describe the background and motivation which led to the need for a parallel QR decomposition algorithm. It will also provide an introduction to the problem domain which provides a portion of the test data included in Chapter 5.

4.1 Introduction to Electromagnetic Scattering

When electromagnetic energy (such as radar waves) comes into contact with a metallic object, some of the energy is reflected off the surface. If the surface is completely reflective and perfectly smooth, all of the energy will reflect off the surface at the same angle as the angle of incidence (See Figure 4.1).

If the surface is not smooth, the reflected energy will scatter in different directions called scattered orders. If the surface is still perfectly reflective, but rough, the total energy contained in all the scattered orders is equal to the energy in the incident radiating wave (See Figure 4.2).

The number and magnitude of the scattered orders is governed by the characteristics of the surface and the incident wave. The scattering can be found through the solution of an integral equation. This can be reduced to a problem of solving a system of linear equations. However, the creation and solution of the system requires significant computational resources and time. We are therefore interested in exploiting parallel computing to speed up the solution and thereby solve larger problems with greater resolution (Cybenko, 1996).

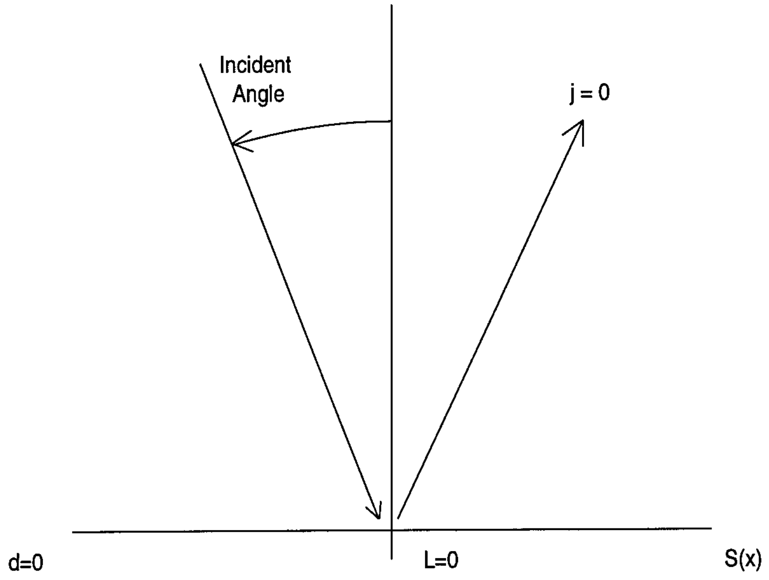


FIG. 4.1. Simple reflecting surface case. Only one reflected component with angle of reflection equal to angle of incidence.

There are widespread applications for a system which can accurately model radar scattering. For example, it could be applied in the following areas:

- aircraft recognition and identification,
- obstacle detection and recognition,
- machine vision,
- ballistic missile defense systems (from "Star Wars" to the Patriot missile system used to shield the allies during Desert Shield/Storm), and
- geological exploration for oil, gas, coal, and other minerals.

Current computational techniques only exist to solve very simple or very small problems of this type. When any real world system is modeled at high resolution,

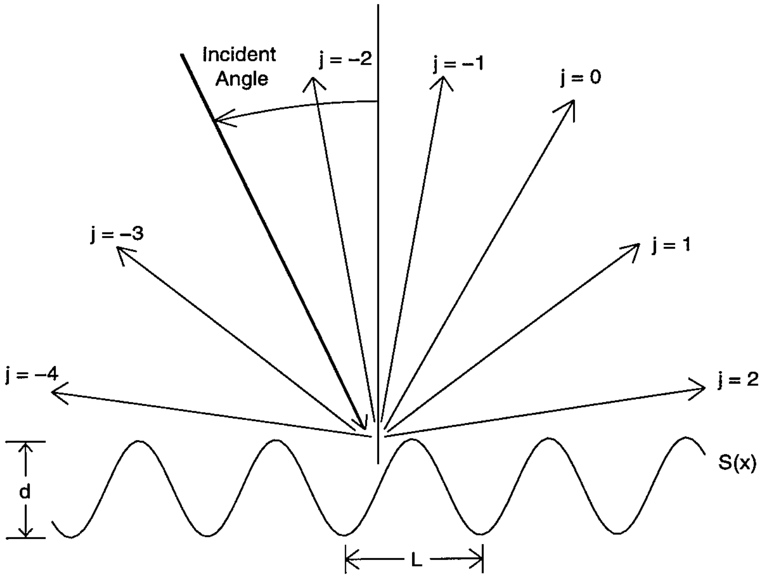


FIG. 4.2. Rough surface scattering, perfectly reflecting case. Reflected energy scatters in multiple components.

Surface Equation	$S(x) = cos(x)$
Surface Period	$L = 1$
Surface Height	$d = 1$
Scattered Components	$-4 <= j <= 2$

the amount of data necessary and the size of problem become too large to solve in a reasonable amount of time on traditional sequential computers. Our goal is to implement a solution to this problem using parallel computing. This will allow the resolution and the size of the problem to increase, thereby moving one step closer to accurate modelling of real world phenomena.

4.2 Electromagnetic Scattering Details

This section will present a brief overview of the mathematical formalism required to model electromagnetic scattering. Correct modeling of the problem is essential to obtain accurate computational results; however, the derivations of the theoretical model are beyond the scope of this document and are therefore not included here. For more information please refer to (Boleng *et al.*, 1996) and (DeSanto, 1985).

A number of simplifying assumptions concerning the model must be made so that the solution becomes feasible. Many of these assumptions will be relaxed in the future to move the mathematical and computational models closer to measurable physical results. Other assumptions are necessary to abstract unimportant details in order to focus on the essence of the problem.

The following simplifying assumptions are being made in an effort to achieve increased mathematical simplicity for initial implementations.

1. The incident field is an infinite plane wave.

2. The surface is a one dimensional infinite periodic surface defined by $S(x)$.

3. The surface is perfectly reflective.

Figures 4.1 and 4.2 show a schematic representation of the problem. The following description is only a qualitative introduction to the problem. Please refer to (DeSanto, 1985) for a more detailed discussion of the mathematical formulation.

$S(x)$ represents the profile of the surface. The surface is considered to be periodic in the initial version of this application. The period of the surface is denoted by L, and d is its depth. θ^{ic} represents the incident angle of the incoming plane wave and λ is its wavelength. Each j corresponds to a propagating order and a specific scattered angle, measured from the vertical z-direction. $j = 0$ is called the specular order and has a scattered angle equal to the incident angle.

When an incident plane wave strikes a rough surface, the angles of the scattered orders can be determined from

$$\sin\theta_j = \sin\theta^{ic} + j\frac{\lambda}{L}, \tag{4.1}$$

which is called the Bragg equation. It is common to use $\alpha_0 = \sin\theta^{ic}$ and $\alpha_j = \sin\theta_j$, and therefore the Bragg equation above can also be written as $\alpha_j = \alpha_0 + j\lambda/L$. Additionally,

$$\beta_j = \cos\theta_j = \begin{cases} +\sqrt{1-\alpha_j^2}, & |\alpha_j| \le 1 \\ +i\sqrt{\alpha_j^2 - 1}, & |\alpha_j| > 1. \end{cases}$$

In this implementation this will be the equation which governs the size and complexity of the problem. By choosing a fixed value for the problem size one can determine the wavelength range required for λ. Using the number of scattered orders ($\mathbf{n}$) to determine problem size yields an easily discretized problem and results in a system of linear equations of the form (see details in (Boleng *et al.*, 1996)):

$$\mathbf{K}\vec{\mathbf{N}} = \vec{\mathbf{F}}^{+}, \tag{4.2}$$

in which $\vec{\mathbf{N}}$ is the unknown, and

$$F_j^{+} = -2\beta_0 D\delta_{j0}, \tag{4.3}$$

where δ_{j0} is the Kronecker delta, and D is the amplitude of the incident plane wave.

In this formulation of the problem, a substantial computational requirement comes from the evaluation of the individual elements of the **K** and **M** matrices. Each matrix element is computed by evaluating an integral of the form:

$$K_{jj'} = \frac{1}{2\pi}\int_{-\pi}^{+\pi} e^{-i(j-j')y} e^{ik_0\left(\beta_j-\beta_{j'}\right)S\left(\frac{L}{2\pi}y\right)}\, dy \tag{4.4}$$

where $k_0 = 2\pi/\lambda$ is the wave number. The **M** matrix is filled similarly:

$$M_{jj'} = \frac{1}{2\pi}\int_{-\pi}^{+\pi} e^{-i(j-j')y} e^{-ik_0\left(\beta_j+\beta_{j'}\right)S\left(\frac{L}{2\pi}y\right)}\, dy. \tag{4.5}$$

Note that the matrix **K** has diagonal entries equal to one. After inverting **K** and solving for $\vec{\mathbf{N}}$, we can evaluate the following equation for the vector $\vec{\mathbf{A}}$:

$$\sum_{j'} M_{jj'} N_{j'} = 2\beta_j A_j, \tag{4.6}$$

where A_j is the amplitude of the jth scattered order whose angles of reflection can be found from the initial Bragg equation (4.1).

Finally, we know that the sum of the scattered energy is equal to the energy of the incident wave:

$$\sum_j |A_j|^2 \text{ Re } (\beta_j) = \beta_0 D^2.$$

We set $D = 1$ for all of the trials, and therefore the following condition should hold:

$$\sum_j |A_j|^2 \frac{\text{Re } (\beta_j)}{\beta_o} = 1. \tag{4.7}$$

This is the energy check condition used to verify the quality of the results.

4.3 Electromagnetic Scattering Algorithm

This section develops one possible computational solution for the electromagnetic scattering model outlined in Section 4.2. If needed, exact equation derivations can be found in (Boleng *et al.*, 1996) and (DeSanto, 1985). For simplicity, problem size is always n, the number of scattered orders.

The basic computational procedure for the Spectral-Spectral formalism (DeSanto, 1985) is:

1. Fix a value for λ/L. This determines the number of scattering orders, using (4.1). A range for λ/L can be calculated if a fixed problem size (n value) is desired.
2. Compute the matrix $\mathbf{K}$ (4.4).
3. Compute $\vec{\mathbf{F}}^{+}$ (4.3).
4. Solve for $\vec{\mathbf{N}}$ (4.2).
5. Compute the matrix $\mathbf{M}$ (4.5).
6. Solve for $\vec{\mathbf{A}}$ (4.6).
7. Perform the energy check to determine if energy is conserved (4.7).

Sequential implementation and performance evaluation of the model included all of the computational steps outlined above. Test results from the sequential implementation for a problem size of n =256 executed on a single R10000 processor of a Silicon Graphics Power Challenge indicated that 99.56% of total computation time is spent filling the $\mathbf{K}$ and $\mathbf{M}$ matrices. As a result, this is where the primary parallelization effort was directed. Fortunately both these processes (steps 2 and 5 above) were perfectly parallel. Section 4.4 includes the initial performance results obtained by only parallelizing the matrix fill routines (ref. equations 4.4 and 4.5).

After a detailed complexity analysis it became apparent that these speedup results could only hold for problem sizes with matrices smaller than about 500 × 500.

The matrix fill routines grow as $O(cn^2)$ where c is a (potentially large) constant. This constant ($\approx$500 in these cases) term dominates the linear system solution ($O(n^3)$ in general) until matrix size grows past it ($n > c$). Once this condition is reached efficient parallel solutions to the electromagnetic scattering problem must contain a parallel linear system solver. This has led to the development of the parallel QR decomposition algorithm presented here, and the results of its application to this problem domain are included in Chapter 5. Furthermore, as the mathematical model becomes more robust and realistic, and as fewer initial assumptions are made, the matrices to be solved in the electromagnetic scattering problem become denser and larger which increases the need for an efficient parallel linear system solver.

4.4 Parallelization Results

This section discusses the initial parallel implementation details of the electromagnetic scattering model. It includes a discussion of the computational resources, supporting code libraries, sequential implementation, and parallel implementation.

4.4.1 Computing Resources and Support

The computing resources used for implementation include a variety of machines and tools. The primary development was done on a Sun Sparc 20 using C. Parallel implementation was done using Power C extensions. Parallel test cases were executed on a Silicon Graphics Power Challenge with eight R10000 processors and two gigabytes of shared main memory and a SGI/Cray Origin 2000 with 32 R10000 processors and eight gigabytes of distributed shared main memory. The machines are located at the National Center for Supercomputing Applications (NCSA).

The matrix algebra and basic complex arithmetic routines used here are from the Meschach Library (Stewart & Leyk, 1994). This is a numerical library of C routines for performing calculations on matrices and vectors. Several alternative integration routines were written specifically for this project. The library routines included a direct linear solver for general complex matrices using Householder QR decomposition. An iterative solver for general complex matrices is not included in the library.

Parallelization of the sequential source code was very straightforward. As discussed earlier, the primary focus of the initial parallel version was the partitioning of the matrix fill routines. This portion of the problem can be termed perfectly parallel. Each matrix entry can be computed independently with no communication costs other than saving the results of the computation. Our initial approach was to divide the $\mathbf{K}$ and $\mathbf{M}$ matrices among the processors by mapping n/p rows (matrices are square $n \times n$) to each processor. This naive parallelization proved effective on the Power Challenge array due to the smaller number of processors and limited problem sizes possible.

When the application was ported to the Origin 2000 machine, larger problem sizes revealed limitations in the integration routines' sampling methods. In order to improve the performance and the accuracy of these routines we also implemented a load balancing scheme based on the maximum slope of the surface. Careful examination of equations (4.4) and (4.5) reveals that surface sampling for the integration is dependent only on the maximum surface slope and the matrix indices (jj'). Before distributing the matrix calculation tasks to the processors, an estimate is made for each matrix element of how many integration intervals will be required for an accurate evaluation. These values are used to load balance the resulting computation. The execution results for all versions of the code are included and compared below in Section 4.4.2.

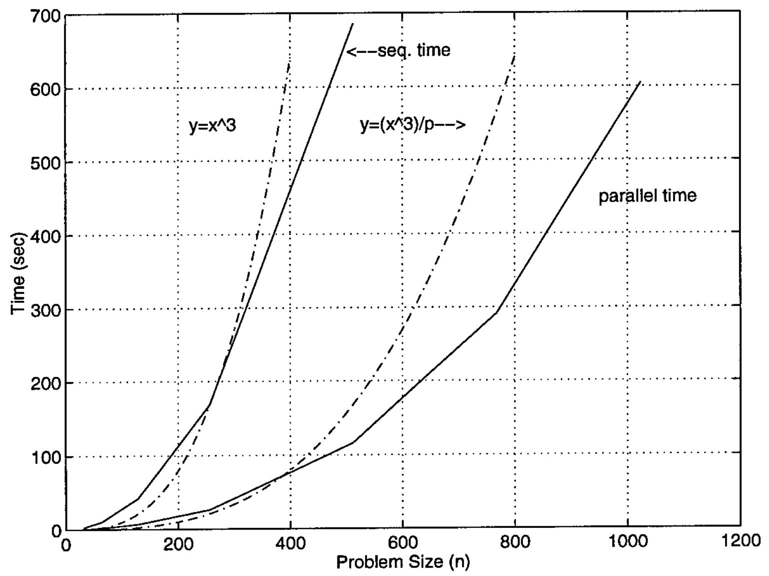


FIG. 4.3. Parallel and sequential execution times on the SGI Power Challenge. Parallel runs were done using 8 processors.

4.4.2 Results

This section summarizes the results and performance improvements due to parallel implementation. The sequential times referenced in the figures are the measured performance using only one processor on the SGI Power Challenge or SGI/Cray Origin 2000, as appropriate. Figure 4.3 compares the sequential and parallel execution times on the Power Challenge. A simple cubic function is included for comparison as well as the same function divided by $p = 8$ (the number of processing elements).

Figure 4.4 presents the speedup using

$$Speedup = \frac{T_{sequential}}{T_{parallel}}$$

and the efficiency (Foster, 1995) of the non-load balanced code on the Power Chal-

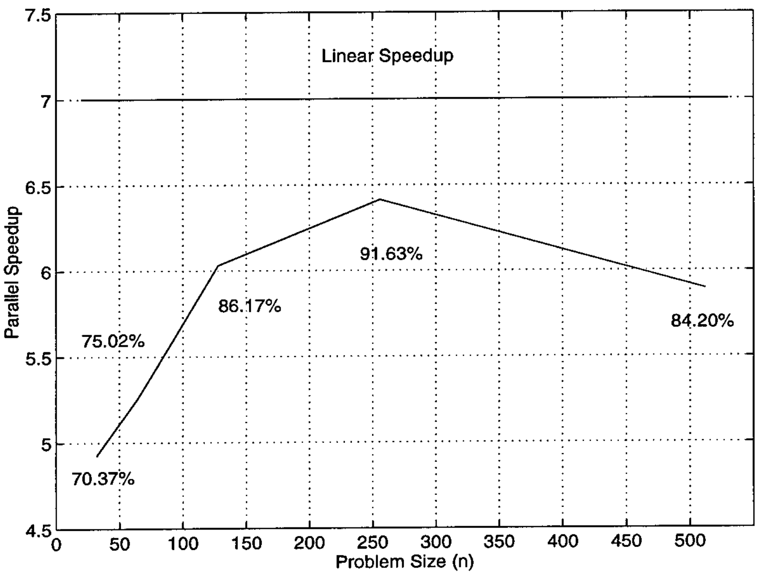


FIG. 4.4. Parallel speedup and efficiency (percentages listed with each data point), Power Challenge Array.

lenge. In these case there were eight processing elements available. The recommendation of SGI is to create one less thread of execution than the number of processors available; this is presumably for system management overhead.

Closer inspection of Figure 4.4 reveals a possible explanation for the graph's shape. The smaller test cases (n = 32, 64, and 128) result in worse performance than the best results at n =256. This is because the overhead required to create and manage the parallel regions by the operating system is a substantial fraction of the overall time. This is a fixed cost, so as the problem size increases, it has a diminishing effect. The efficiency declines after $n = 256$ because the linear system's solution time now contributes a larger percentage of the overall time. As discussed earlier, the complexity of a linear system solution is O(n^3) in general. Our matrix fill time grows as cn^2 where c is a (potentially large) constant. After n =256, this constant term

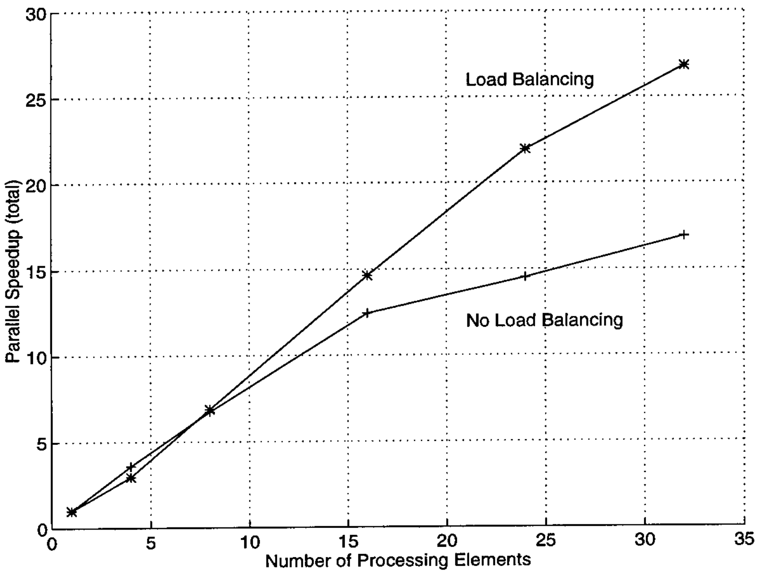


FIG. 4.5. Parallel speedup, Origin 2000.

(≈500 in these cases) begins to play a diminishing role when compared to the $O(n^3)$ complexity of the linear system solution.

The parallel scattering application was recently ported to the SGI/Cray Origin 2000 distributed shared memory machine. Movement to a parallel machine with more processors and memory allowed the execution of larger test cases. Larger cases pointed out a deficiency in the integration routines that, once corrected, led to processor load imbalances. As introduced earlier in Section 4.3, examination of equations (4.4) and (4.5) allows a conservative estimate to be made a priori of the amount of computational work required to fill each matrix element in **K** and **M**. This estimate is then used to balance the load across processors. Figures 4.5 and 4.6 depict the parallel speedup and efficiency results obtained for a fixed problem size of $n = 256$ and $\theta = 25°$.

Examination of Figures 4.5 and 4.6 reveals nearly linear speedup for the load

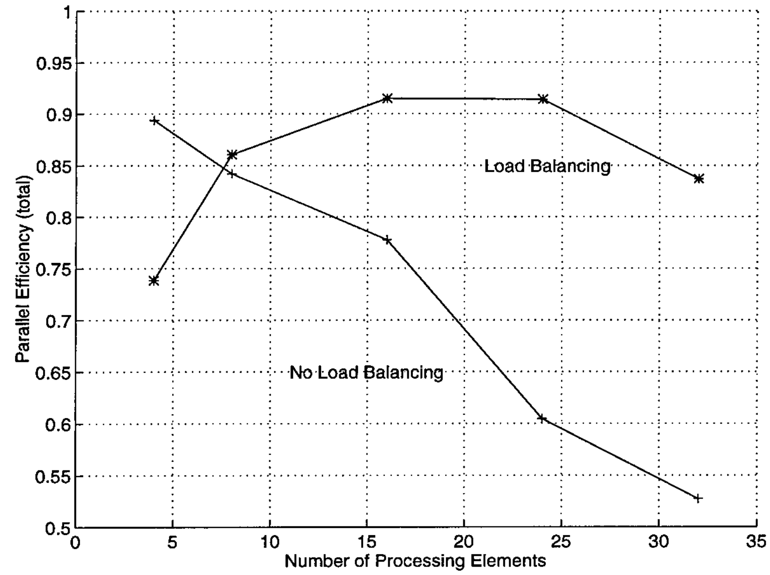


FIG. 4.6. Parallel efficiency, Origin 2000.

balanced code. The scalability of this code is also superior to that of the non-load balanced version. The slight efficiency drop over $p = 24$ in Figure 4.6 is due primarily to the relatively small problem size used for this initial testing ($n = 256$). Much larger test cases have been run (up to $n = 2048$) and efficiency results have remained over 90% for all 32 processors.

In general, the parallel solution will scale well to an arbitrary number of processors. This is to say, the portions of the code that are parallel (filling the **K** and **M** matrices) will scale well. The above figures do indicate a possible problem when scaling the entire solution. For cases greater than n =512 the matrices become more dense and the linear system's solution begins to require a noticeable amount of time. The limited data from the largest two cases tends to indicate that the total problem solution will not scale well until the QR decomposition algorithm is parallelized. These results are presented in Chapter 5.

Chapter 5

PARALLEL QR DECOMPOSITION RESULTS

This chapter presents the results of implementing the new QR decomposition algorithm in parallel for a variety of problem sizes. The test problems chosen focus on sizes where $m/p < n$ (see Section 3.4) in an effort to push the algorithm on the most demanding cases. Section 5.2 presents expected and actual performance of the algorithm using matrices with randomly generated entries, and Section 5.3 includes performance results when the algorithm is applied to a matrix from the electromagnetic scattering problem domain.

5.1 Computing Resources

The computing resources used for these tests are largely the same as those listed in Section 4.4.1. However, there are some changes which must be mentioned before results are presented. Reconfiguration of machines at NCSA has made available Power Challenge machines with up to 16 processors as opposed to only 8, as was the case for earlier testing. In addition to this, the status of the Origin 2000 machine was moved from installation/evaluation to production. This change made a substantial difference in the user load on the machine, as well as its availability. When comparing results, it will become apparent that more data is available for the Power Challenge implementations. When the Origin 2000 at NCSA was moved to production use, the migration of a large number of users from the Power Challenge array to the Origin 2000 made system response time extremely slow, and queue wait times increased dramatically. The large number of users porting code also affected the system stability which resulted in periods of outage. All these problems combined to make testing and obtaining performance results on the Origin 2000 difficult. Several cases are included nonetheless.

5.1.1 Power Challenge Architecture Overview

The SGI Power Challenge (see Fig. 5.1) is a shared-memory multiprocessor architecture based on the MIPS superscalar RISC R8000 and R10000 chips. These chips are 64-bit processors with 64-bit integer and floating-point operations, registers, and virtual addresses. The R8000 and R10000 use the MIPS IV instruction set.

The cache system consists of a 16 kilobyte (KB) direct-mapped on-chip instruction cache, a 16 KB direct-mapped on-chip integer data cache, and a 4 megabyte (MB) four-way set associative external cache. The external cache serves as the primary cache for floating-point data and the secondary cache for instructions and integer data. The length of cache lines are:

- on-chip instruction cache: 32 bytes (4 double words)
- on-chip integer data cache: 32 bytes (4 double words)
- external cache: 128 bytes (16 double words)

The processors communicate via a fast shared-bus interconnect. The bus has a bandwidth of 1.2 gigabyte (GB) per second with a 256-bit wide data bus and a separate 40-bit wide address bus that can access up to one terabyte (TB) of physical memory. The bus provides high-bandwidth, low-latency, cache-coherent communication between processors, memory, and I/O.

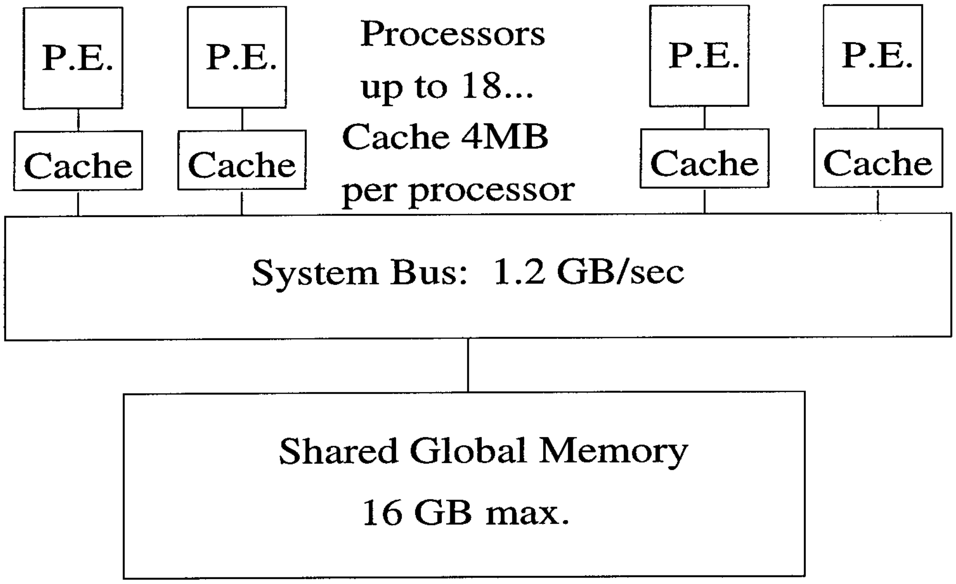


FIG. 5.1. SGI Power Challenge architecture overview.

5.1.2 Origin 2000 Architecture Overview

The SGI/Cray Origin 2000 is a follow-on to the Challenge-class symmetric multiprocessing (SMP) system. It uses Silicon Graphics' distributed shared-memory multiprocessing architecture, called S2MP. As illustrated in Figure 5.2, the Origin 2000 has a number of processing nodes linked together by an interconnection fabric. Each processing node contains either one or two processors, a portion of shared memory, a directory for cache coherence, and two interfaces: one that connects to I/O devices and another that links system nodes through the interconnection fabric (see Fig. 5.3).

The Origin 2000 uses the MIPS R10000, a high-performance 64-bit superscalar processor which supports dynamic scheduling. An important attribute of the R10000 is its capacity for heavy overlapping of memory transactions – up to twelve per processor in the Origin 2000. Each Node board added to the Origin 2000 provides additional

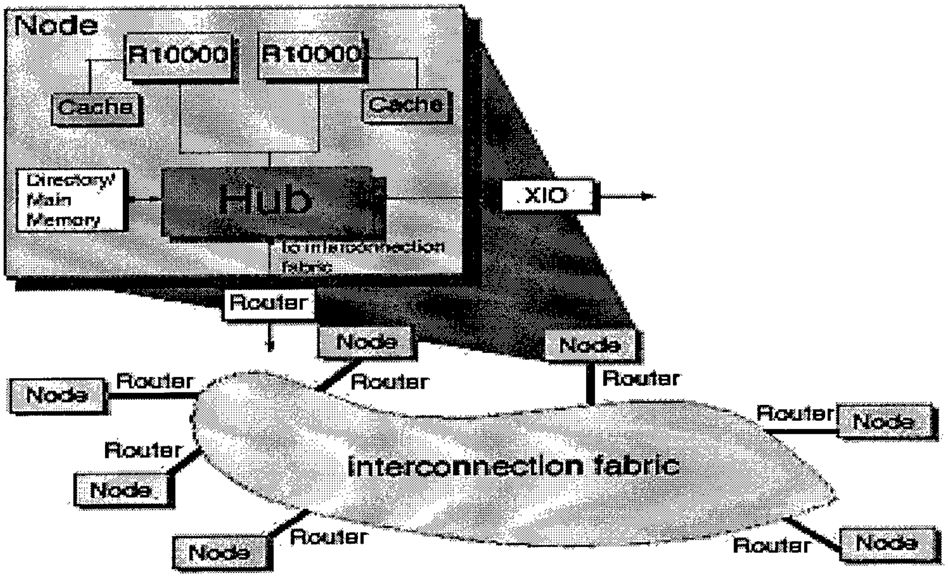


FIG. 5.2. SGI/Cray Origin 2000 architecture overview. Figure courtesy of NCSA on-line SGI documentation.

independently accessed memory, and each node is capable of supporting up to 4 GB of memory. Up to 64 nodes can be configured in a system, which implies a maximum memory capacity of 256 GB.

The Origin 2000 nodes are connected by an interconnection fabric. The interconnection fabric is a set of switches, called routers, that are linked by cables in various configurations, or topologies. The interconnection fabric differs from a standard bus in the following important ways:

- The interconnection fabric is a mesh of multiple point-to-point links connected by the routing switches. These links and switches allow multiple transactions to occur simultaneously.

- The links permit extremely fast switching. Each bidirectional link sustains as

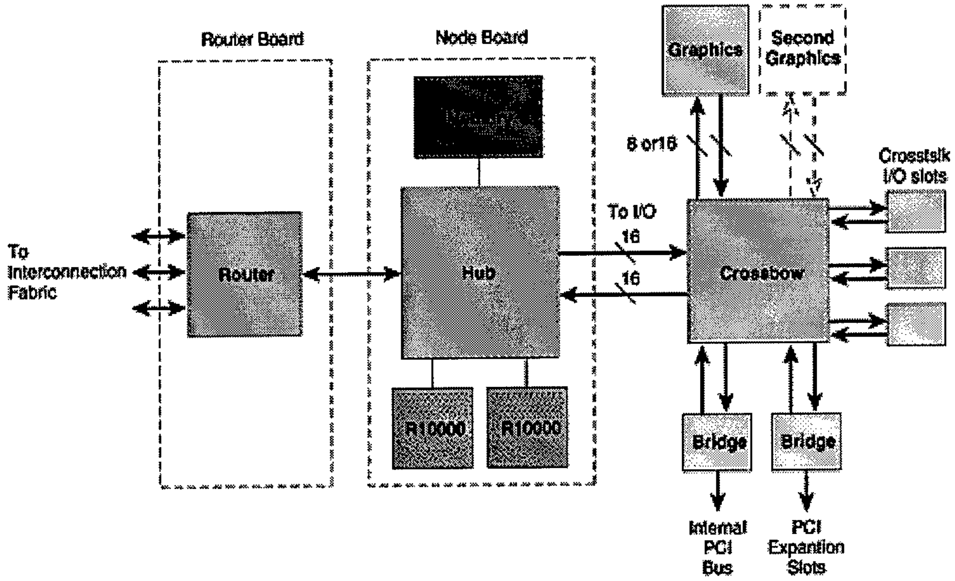


FIG. 5.3. SGI/Cray Origin 2000 node layout. Figure courtesy of NCSA on-line SGI documentation.

much bandwidth as the entire Challenge bus.

- The interconnection fabric does not require arbitration nor is it as limited by contention, while a bus must be contested for through arbitration.

- More routers and links are added as nodes are added, increasing the interconnection fabric's bandwidth. A shared bus has a fixed bandwidth that is not scalable.

- The topology of the CrayLink Interconnect is such that the bisection bandwidth grows linearly with the number of nodes in the system.

The interconnection fabric provides a minimum of two separate paths to every pair of the Origin 2000 nodes. This redundancy allows the system to bypass failing routers or broken interconnection fabric links. Each fabric link is additionally protected by

Interface	Sustained Bandwidth [Peak BW in brackets]
Memory	780 MB per second [780]
Node Card	1.25 GB per second [1.56 GB]
Crossbow	2.5 GB per second [3.12 GB]
Module (deskside)	5.0 GB per second [6.24 GB]
Rack	80 GB per second [100 GB]

Table 5.1. Origin 2000 Peak and Sustained Bandwidths.

a CRC code and a link-level protocol, which retry any corrupted transmissions and provide fault tolerance for transient errors.

There are three types of bandwidths:

- Peak bandwidth, which is a theoretical number derived by multiplying the clock rate at the interface by the data width of the interface.

- Sustained bandwidth, which is derived by subtracting the packet header and any other immediate overhead from the peak bandwidth. This best-case figure, sometimes called Peak Payload bandwidth, does not take into account contention and other variable effects.

- Bisection bandwidth, which is derived by dividing the interconnection fabric in half, and measuring the data rate across this divide. This figure is useful for measuring data rates when the data is not optimally placed.

Table 5.1 gives a comparison between peak and sustained data bandwidths of the Origin 2000.

5.1.3 Shared vs. Dedicated Mode

One final note concerns an observed detail about the timings presented below. During testing, it was noticed that execution times were considerably lower and observed efficiencies where higher when the test case was submitted to run in dedicated versus shared mode. This presented some difficulties because the NCSA machines are only available for dedicated runs on Sundays, but it was still possible to obtain dedicated execution results for some problem sizes.

5.2 Measured Algorithm Performance

This section will present the performance of the parallel QR decomposition algorithm when applied to matrices of randomly generated elements. Section 5.2.1 includes a discussion of the sequential performance of the parallel algorithm and compares it with the sequential performance of standard decomposition techniques. Sections 5.2.2 and 5.2.3 present and discuss the measured results on the SGI Power Challenge and SGI/Cray Origin 2000 respectively.

5.2.1 Performance vs. Standard Methods

When measuring the relative performance of a parallel algorithm a decision must be made concerning the standard of comparison for the single processor or sequential timing. Parallel speedup is calculated as

$$\text{Speedup} = \frac{T_{\text{sequential}}}{T_{\text{parallel}}} .$$

Several alternatives have been suggested concerning how $T_{\text{sequential}}$ is measured. The most widespread measurement technique is simply to run the parallel code on

Problem Size	Parallel QR Algorithm		Householder Algorithm
1024 × 1024	$T_{1/1}$	= 67.95	$T_{\text{best}} = 67.64$
1024 × 1024	$T_{1/16}$	= 44.91	
1024 × 1024	$T_{1/32}$	= 46.96	
8000 × 1600	$T_{1/1}$	= 1859.4	$T_{\text{best}} = 1868.97$
8000 × 1600	$T_{1/16}$	= 1716.47	
8000 × 1600	$T_{1/32}$	= 1365.96	

Table 5.2. Comparison of sequential performance for the parallel algorithm and the best known sequential algorithm. Execution times reported in seconds and measured on one R10000 processor of the SGI Power Challenge.

one processor of the machine being used for testing. This technique has been criticized because it is often the case that the parallel algorithm, when run sequentially, is much less efficient than the best sequential algorithm. An alternative technique uses the time of the best known sequential algorithm for $T_{\text{sequential}}$. This results in a more realistic speedup measure, but the resulting efficiency is not a good measure of how "busy" the algorithm keeps the processors. For the results below, both measurements will be presented when possible. The best known sequential algorithm for QR decomposition uses Householder reflections, and this is reported as the time T_{best}. The corresponding speedup and efficiency measures are reported as $\text{Speedup}_{\text{best}}$ and $\text{Efficiency}_{\text{best}}$. The performance of the parallel algorithm executed on one processor is reported as $T_{1/b}$, where b is the number of blocks used in the sequential run. Speedup and efficiency of the parallel algorithm are reported in a similar manner as $\text{Speedup}_{1/b}$ and $\text{Efficiency}_{1/b}$.

When gathering performance results, it was noticed that in most cases, the timing for the parallel algorithm executing on one processor, $T_{1/b}$, was better than the timing for the best known sequential algorithm, T_{best}. This was especially true as the problem size grew and different values of b where used. Table 5.2 contains a comparison of some of the results. The difference in performance for these algorithms is presumably due to the memory hierarchy of the machines being used. When the algorithms eliminate elements via Householder reflections the code instructs the compiler to maintain the Householder vector in local cache to increase performance. Each Householder vector is m elements long. As the matrix size grows, particularly the number of rows, the traditional sequential algorithm cannot maintain the entire Householder vector in cache, so performance suffers. The parallel QR algorithm acts in a block manner and annihilates blocks of the matrix with Householder vectors that are long enough to be maintained in the cache during their entire application. For the purpose of calculating parallel speedup and efficiency in Sections 5.2.2 and 5.2.3, both T_{best} and $T_{1/1}$ are used.

5.2.2 Performance on the Power Challenge

This section summarizes the performance results of the parallel QR decomposition algorithm implemented on the SGI Power Challenge. Each of the following sets of tables and graphs is presented for different problem sizes. Examination of Figures 5.4 and 5.5 shows good parallel efficiency on the Power Challenge for up to four processors, and speedup increases on up to 12 processors. Efficiency peaks at four processors (77.9% and 80.8%, respectively), but speedup is still increasing. The efficiency begins to drop off quickly for the eight and 12 processor cases. Peak performance is reached at 12 processors with a speedup of 5.65, but a parallel efficiency of only 47.1%.

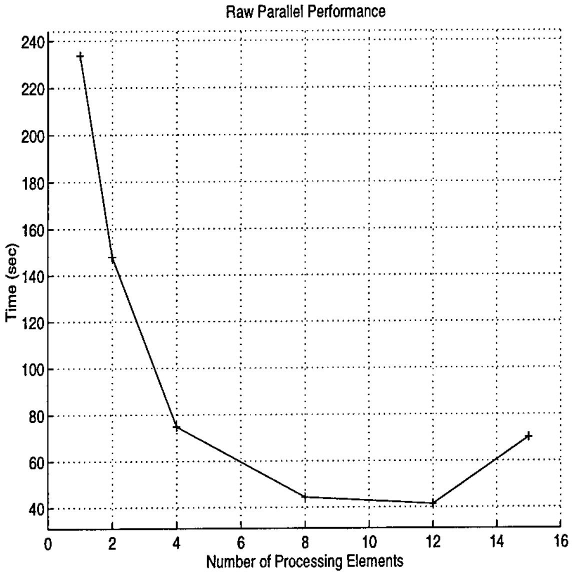
Two comments can be made about the shapes of the graphs in Figures 5.4 and 5.5. First, actual performance differs substantially from the expected performance. This occurs primarily because the prediction model derived in Chapter 3 and used here does not include any system management overhead. In addition to the relatively small load imbalances inherent in the algorithm, which the model includes, are the much higher costs of creating the parallel threads of execution, memory contention/delays, synchronization costs, etc. As the number of processors increases for a fixed problem size, less and less work is being done between thread creation, synchronizations, and destruction. Any speedup achieved by concurrent operation is soon lost in the cost of the system overhead. By the 16 processor case, problem size was just too small to benefit from further parallelization. Larger problem sizes yield higher efficiencies for greater numbers of processors.

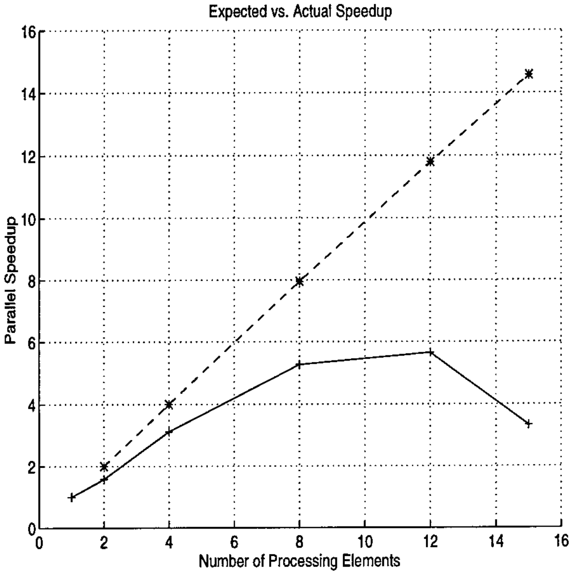
The second notable observation is apparent in Figure 5.5 at $p = 2$ where the sharp trough appears. This could be a result of anomalous data attributable to heavy system load at run time (these data points were gathered while running in shared mode), or it is more likely caused again by system management overhead. At the case of only two processors, there is very little potential speedup to absorb the fixed cost incurred from creating the parallel threads. As more processors are used the variable cost of adding more threads of execution is still incurred, but the fixed parallelization startup costs can be spread across greater potential speedup.

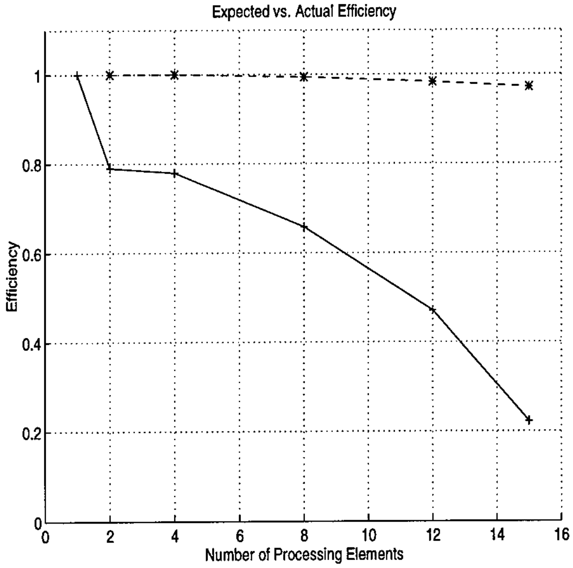
An important comparison can be made from Figures 5.6 and 5.7. These results present performance for the same matrix size and processor numbers in shared vs. dedicated mode. In shared mode, the program is time shared with other parallel jobs, while in dedicated mode, the program is executed from start to finish on one Power Challenge using as many processors as requested. The raw performance run times are significantly better for the parallel cases when executed in dedicated mode. This observation is apparent later on the Origin 2000 machine as well. This performance difference is likely attributable to the action of context switching between applications in shared mode which creates extra system overhead that does not exist in dedicated mode.

It is encouraging to note that with the increased problem sizes in Figures 5.6 and 5.7, the actual results began to follow the predicted performance more closely. Speedup was achieved for this problem size all the way up to and including the maximum available number of processors on the NCSA machines. This is in contrast to the previous cases where speedup peaked and then began to decrease before the processor limit. Efficiency does steadily decline as in the previous cases, but the decrease is not as soon or as sharp.

Finally, examine Figure 5.8 which plots the parallel efficiency and speedup for many processors as the problem size changes. This comparison demonstrates a characteristic of the algorithm predicted in Chapter 3. As problem size increases relative to the number of processors, the efficiency of the algorithm increases. This occurs because the two sources of possible load imbalance (the leftover rows and last p steps during the BR stage) become a smaller and smaller fraction of the total work done. It is expected that this trend would continue for larger problem sizes.



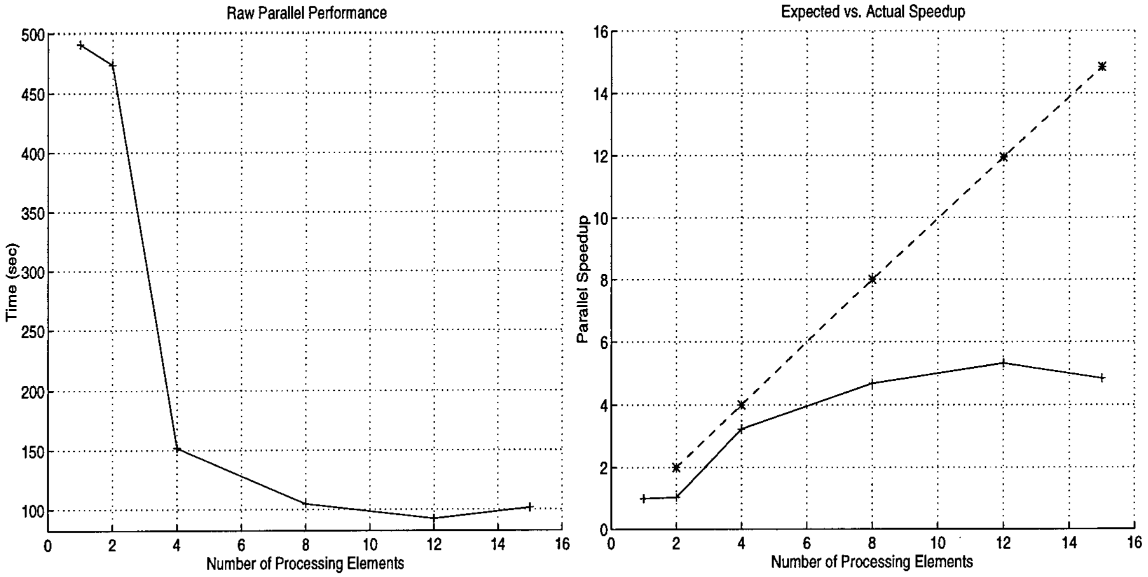


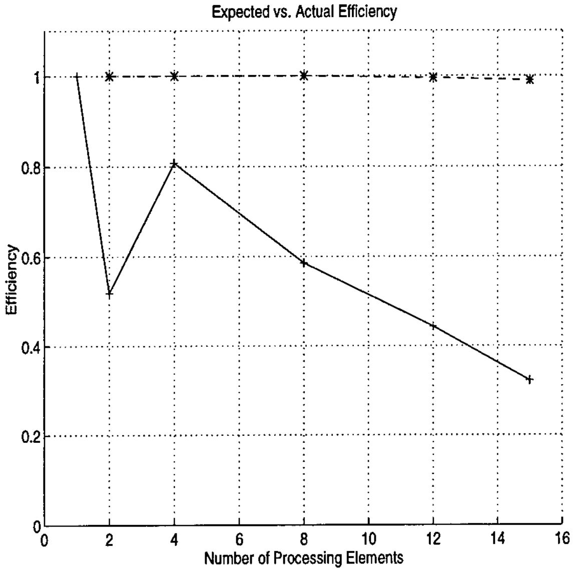


4000 × 800	p=1	p=2	p=4
Householder	233.7		
Parallel QR	231.3	147.9	44.38
$\text{Speedup}_{\text{expected}}$		1.99	3.99
$\text{Efficiency}_{\text{expected}}$		100%	99.9%
$\text{Speedup}_{\text{best}}$		1.58	3.12
$\text{Efficiency}_{\text{best}}$		79.2%	77.9%
$\text{Speedup}_{1/1}$		1.56	3.09
$\text{Efficiency}_{1/1}$		78.2%	77.2%

	p=8	p=12	p=15
Parallel QR	104.79	92.24	101.49
$\text{Speedup}_{\text{expected}}$	7.95	11.79	14.57
$\text{Efficiency}_{\text{expected}}$	99.4%	98.3%	97.2%
$\text{Speedup}_{\text{best}}$	5.27	5.65	3.33
$\text{Efficiency}_{\text{best}}$	65.8%	47.1%	22.2%
$\text{Speedup}_{1/1}$	5.21	5.59	3.30
$\text{Efficiency}_{1/1}$	65.1%	46.6%	21.9%

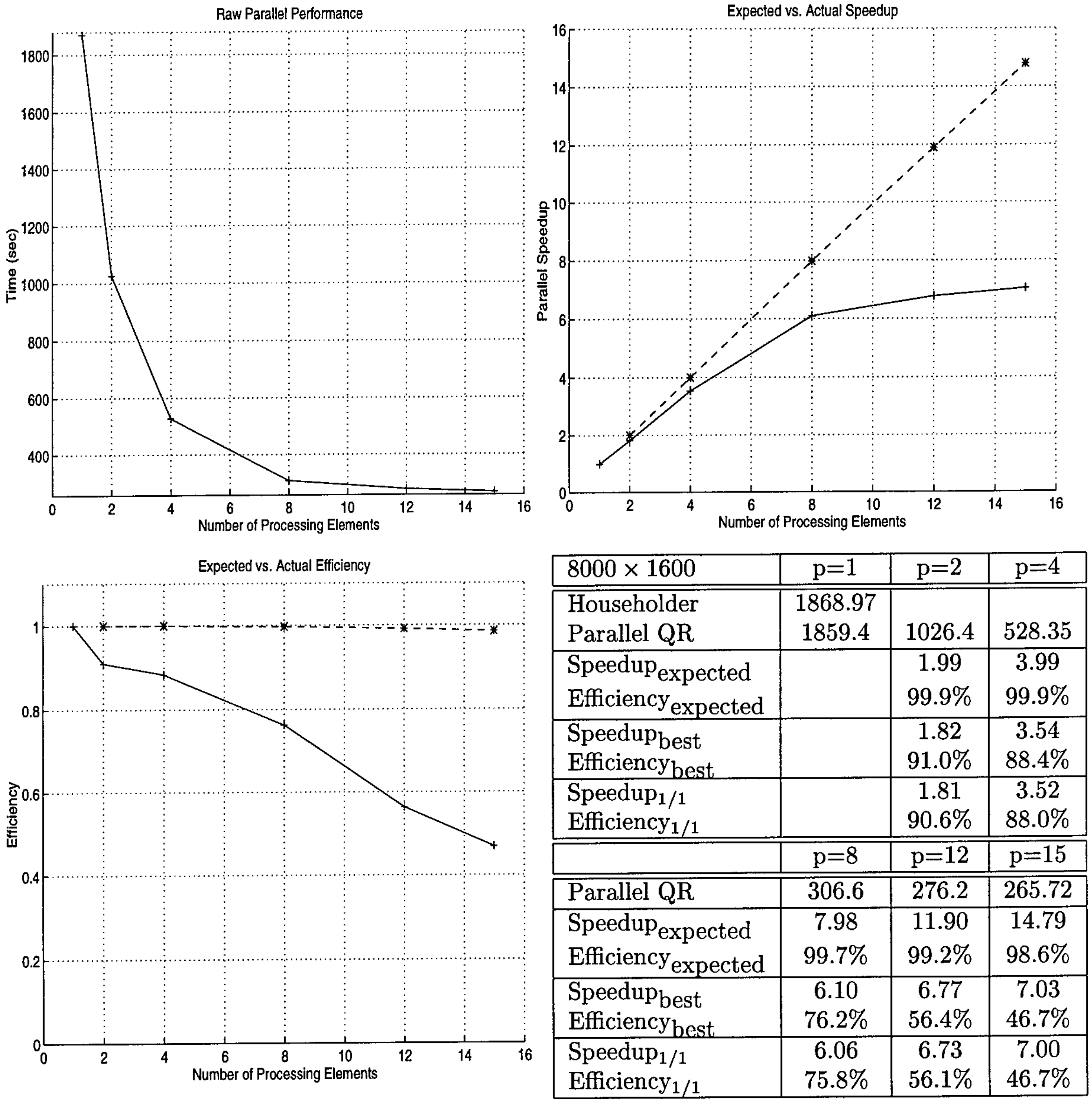
FIG. 5.4. Parallel performance on the SGI Power Challenge (R10000) in shared mode. Matrix size is 4000 × 800. Expected results shown as a dashed line with '$*$' symbol. Actual results shown as a solid line with '+' symbol.





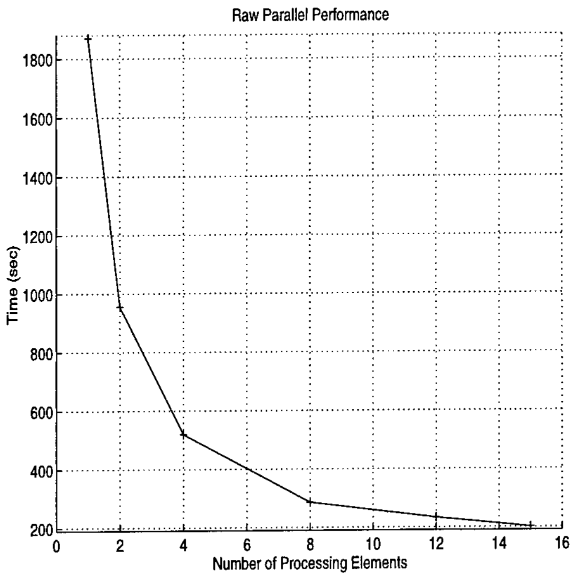
8000 × 800	p=1	p=2	p=4
Householder Parallel QR	490.9 479.4	 473.7	 151.94
$\text{Speedup}_{\text{expected}}$ $\text{Efficiency}_{\text{expected}}$		2.00 99.9%	3.99 99.9%
$\text{Speedup}_{\text{best}}$ $\text{Efficiency}_{\text{best}}$		1.04 51.8%	3.23 80.8%
$\text{Speedup}_{1/1}$ $\text{Efficiency}_{1/1}$		1.01 50.6%	3.16 78.9%
	p=8	p=12	p=15
Parallel QR	104.79	92.24	101.49
$\text{Speedup}_{\text{expected}}$ $\text{Efficiency}_{\text{expected}}$	7.99 99.9%	11.95 99.6%	14.85 98.9%
$\text{Speedup}_{\text{best}}$ $\text{Efficiency}_{\text{best}}$	4.68 58.6%	5.32 44.4%	4.84 32.3%
$\text{Speedup}_{1/1}$ $\text{Efficiency}_{1/1}$	4.58 57.2%	5.20 43.3%	4.72 31.5%

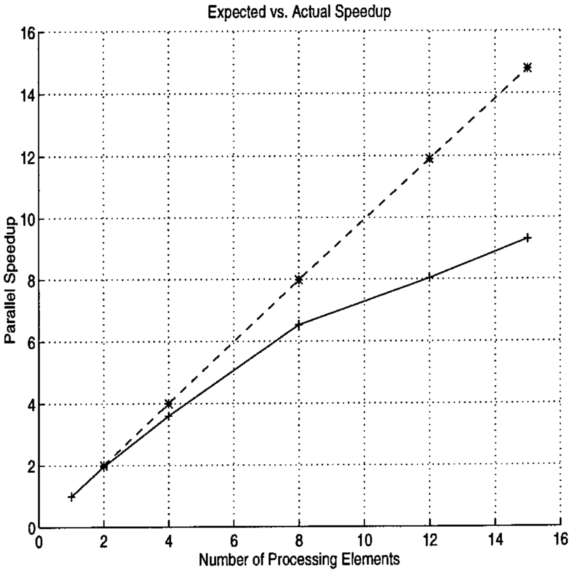
FIG. 5.5. Parallel performance on the SGI Power Challenge (R10000) in shared mode. Matrix size is 8000 × 800. Expected results shown as a dashed line with '$*$' symbol. Actual results shown as a solid line with '+' symbol.

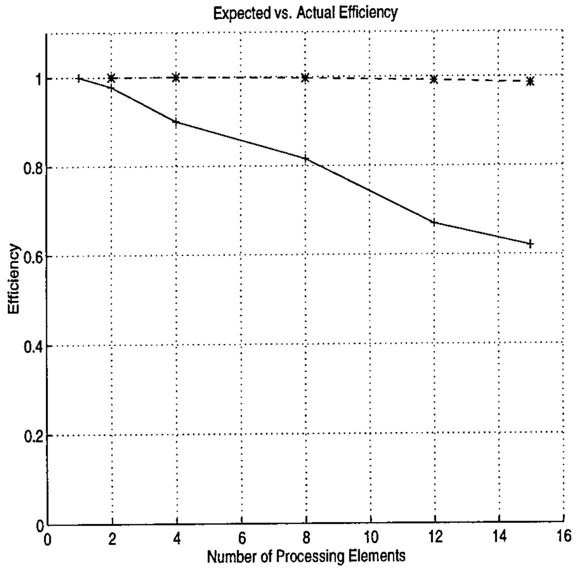


8000 × 1600	p=1	p=2	p=4	p=8	p=12	p=15
Householder	1868.97					
Parallel QR	1859.4	1026.4	528.35	306.6	276.2	265.72
Speedup$_{expected}$		1.99	3.99	7.98	11.90	14.79
Efficiency$_{expected}$		99.9%	99.9%	99.7%	99.2%	98.6%
Speedup$_{best}$		1.82	3.54	6.10	6.77	7.03
Efficiency$_{best}$		91.0%	88.4%	76.2%	56.4%	46.7%
Speedup$_{1/1}$		1.81	3.52	6.06	6.73	7.00
Efficiency$_{1/1}$		90.6%	88.0%	75.8%	56.1%	46.7%

FIG. 5.6. Parallel performance on the SGI Power Challenge (R10000) in shared mode. Matrix size is 8000 × 1600. Expected results shown as a dashed line with '$*$' symbol. Actual results shown as a solid line with '+' symbol.







8000 × 1600	p=1	p=2	p=4
Householder	1868.97		
Parallel QR	1859.4	955.34	519.35
$\text{Speedup}_{\text{expected}}$		1.99	3.99
$\text{Efficiency}_{\text{expected}}$		99.9%	99.9%
$\text{Speedup}_{\text{best}}$		1.96	3.60
$\text{Efficiency}_{\text{best}}$		97.8%	90.0%
$\text{Speedup}_{1/1}$		1.95	3.58
$\text{Efficiency}_{1/1}$		97.3%	89.5%

	p=8	p=12	p=15
Parallel QR	286.36	232.62	200.92
$\text{Speedup}_{\text{expected}}$	7.98	11.90	14.79
$\text{Efficiency}_{\text{expected}}$	99.7%	99.2%	98.6%
$\text{Speedup}_{\text{best}}$	6.53	8.03	9.30
$\text{Efficiency}_{\text{best}}$	81.6%	67.0%	62.0%
$\text{Speedup}_{1/1}$	6.49	7.99	9.25
$\text{Efficiency}_{1/1}$	81.2%	66.6%	61.7%

FIG. 5.7. Parallel performance on the SGI Power Challenge (R10000) in **dedicated** mode. Matrix size is 8000 × 1600. Expected results shown as a dashed line with '*' symbol. Actual results shown as a solid line with '+' symbol.

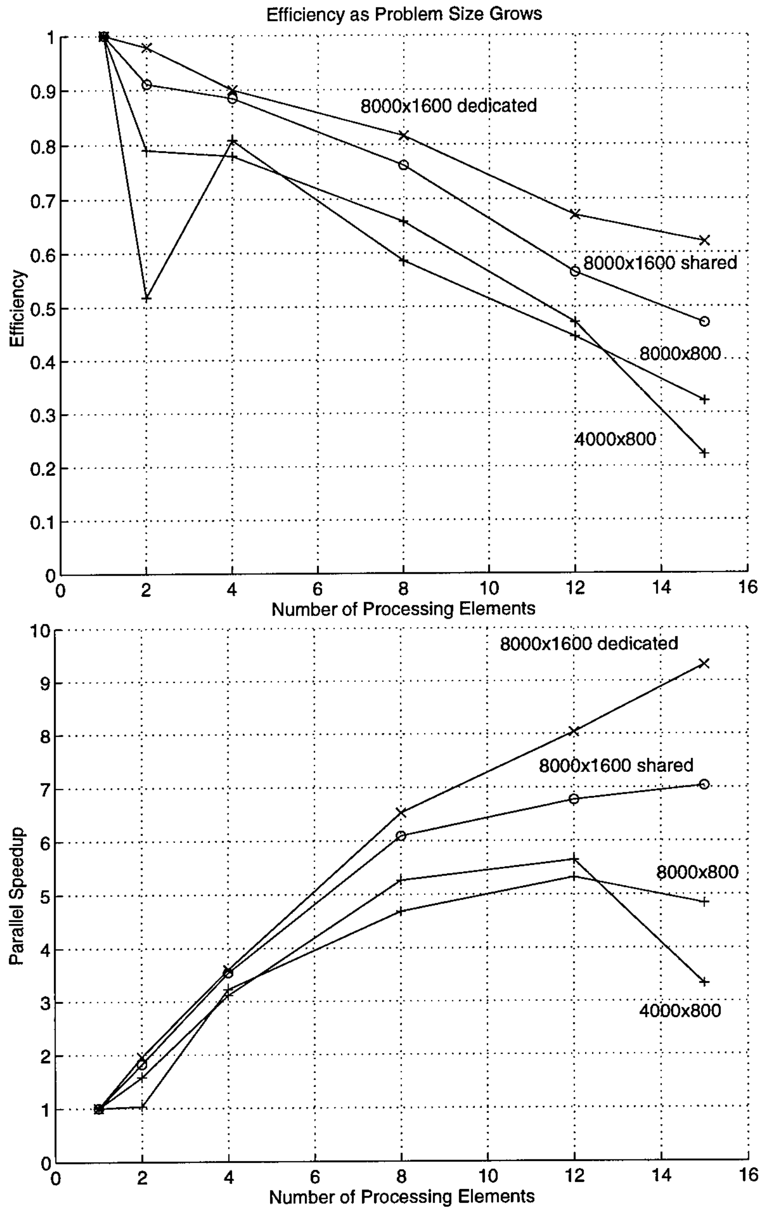


FIG. 5.8. Parallel performance comparison on the SGI Power Challenge (R10000) as problem size grows. Matrix sizes are shown next to their representative line.

5.2.3 SGI/Cray Origin2000 Performance

This section summarizes the performance results of the parallel QR decomposition algorithm implemented on the SGI/Cray Origin 2000. Each of the following sets of tables and graphs is presented for different problem sizes. The matrices used for testing on the Origin 2000 are the same as those used previously on the Power Challenge. The performance achieved on the Origin 2000 differs substantially from that on the Power Challenge. Raw performance, or execution times, are faster, but parallel performance is not nearly as good on large problem sizes. Only for the smaller matrix cases (4000×800 and 8000×800) did the Origin 2000 significantly outperform the Power Challenge.

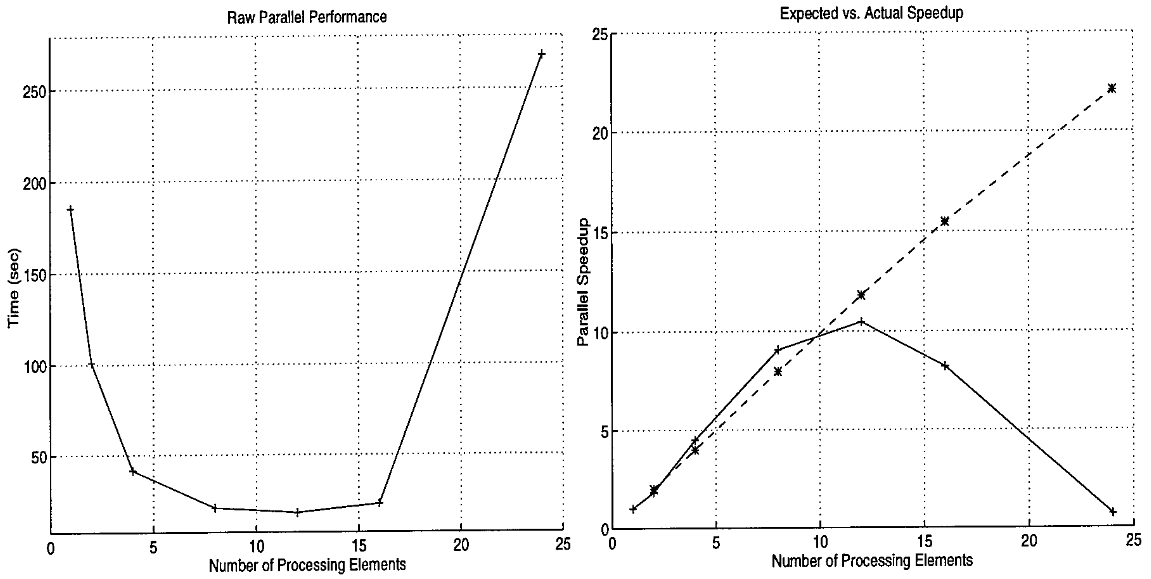
Figure 5.9 shows promising performance up to 12 processors. In fact, super-linear speedup is achieved, which is certainly attributable to the blocking of the matrix and Householder vector length as described earlier in Section 5.2.1. When the parallel algorithm is used, the Householder vectors are shorter and more easily maintained in cache memory. This blocking effect alone is enough to help performance dramatically without parallelization. The parallel execution times take advantage of these access differences in the memory hierarchy and combine concurrent operations to achieve super-linear performance.

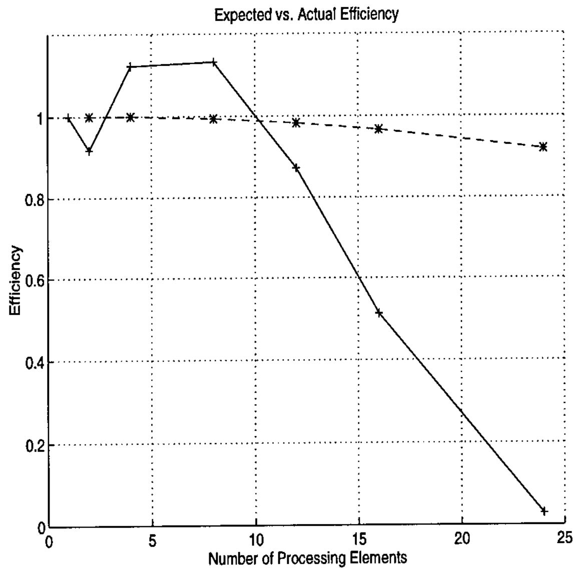
Using more than 12 processors on this matrix size results in over-parallelization and decreased performance for the same reasons as those described above on the Power Challenge. This is shown by the dramatic decrease in efficiency for 16 and 24 processors. In the case of the smallest matrix (see Figure 5.9) speedup decreased more sharply than in any other case. Figure 5.10 represents the best results achieved. Speedup remains close to predicted results, and 85-90% efficiency results. Speedup continues past the 12 processor case, but after 16 processors, the problem size is again too small to benefit from further parallelization.

Figures 5.11 and 5.12 represent the comparison of using shared vs. dedicated mode on the Origin 2000 to factor the largest problem size. Run times are consistently better when problems are executed in dedicated mode. Parallel performance at this large problem size is disappointing on the Origin 2000, especially when compared to the increasing speedup obtained on the Power Challenge for the same problem. In both cases (dedicated and shared), the use of more than 16 processors does not improve performance. In fact, the move from 16 to 24 processors created the biggest observed increase in run time during parallel execution.

Recall that the Origin 2000 is a Distributed Shared Memory (DSM) architecture. The physical design of the machine consists of a variety of nodes linked by a high speed cross-bar switch. There are only two processors per node that physically share memory. The logical address space is mapped by the operating system across the $p/2$ distributed nodes. The current algorithm implementation makes no attempt to schedule memory accesses to reduce communication. A few alternative scheduling techniques were tried for the BR stage, but memory mapping and movement as well as processor assignment is operating system dependent. No method was found that consistently addressed all the mysteries of the operating system's thread and memory management.

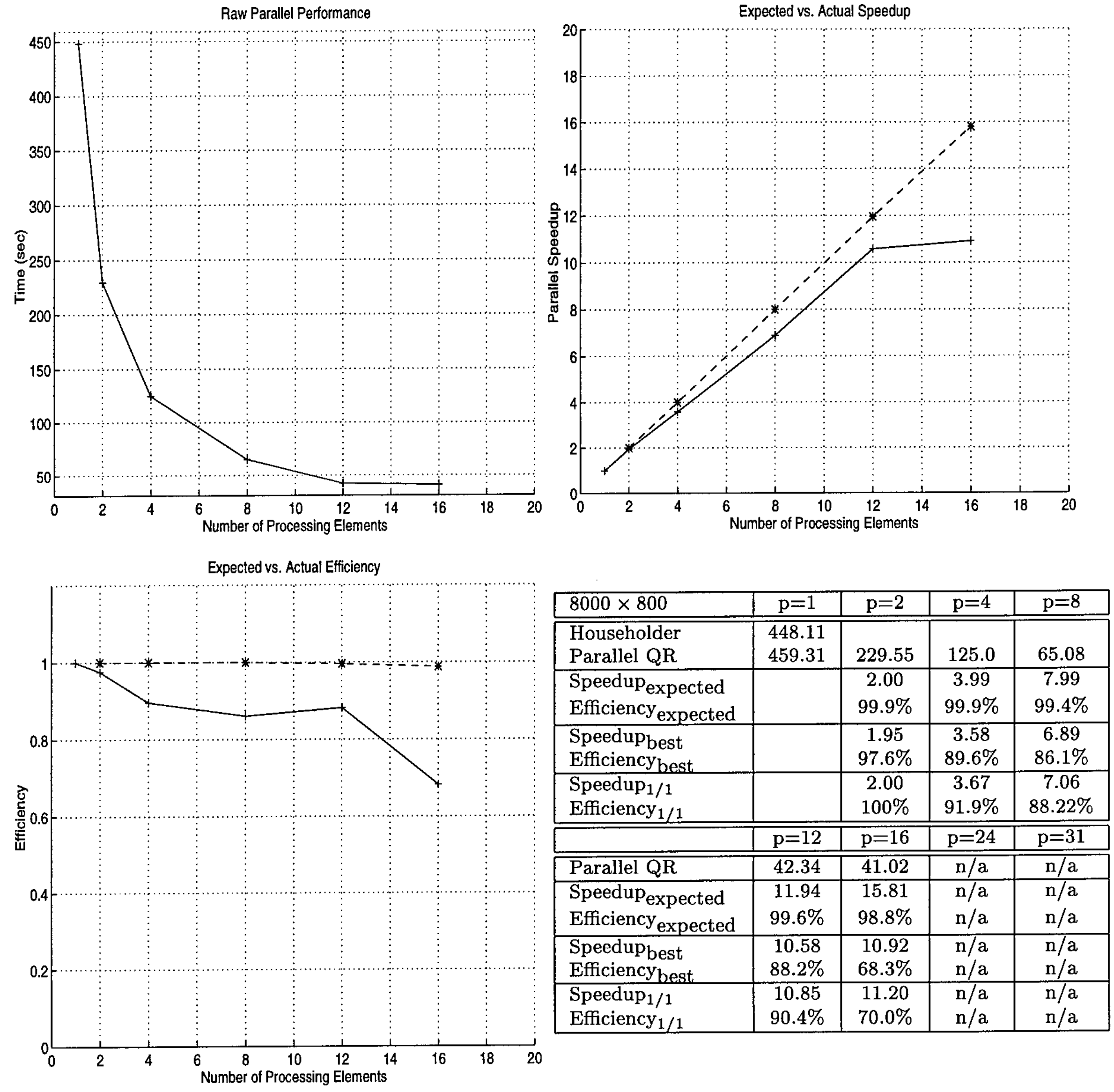
The final Origin 2000 figure compares all the problem sizes together (see Fig. 5.13). The results and conclusions here are not nearly as obvious or encouraging as those seen in the Power Challenge results. In only one case did the Origin 2000 maintain a speedup with more than 12 processors, and when the efficiency figure is included, the outlook becomes even worse. This distributed shared memory machine is not only running into the same system management overhead and problem size limitations as the Power Challenge, but it has the added cost of a more complex memory management and transfer system. More careful scheduling and memory accesses during the BR stage must be done on this machine.





4000 × 800	p=1	p=2	p=4	p=8
Householder Parallel QR	185.49 194.23	 101.1	 41.29	 20.47
Speedup$_{\text{expected}}$ Efficiency$_{\text{expected}}$		1.99 99.9%	3.99 99.9%	7.95 99.4%
Speedup$_{\text{best}}$ Efficiency$_{\text{best}}$		1.83 91.7%	4.49 112.3%	9.06 113.3%
Speedup$_{1/1}$ Efficiency$_{1/1}$		1.92 96.1%	4.70 117.6%	9.49 118.6%
	p=12	p=16	p=24	p=31
Parallel QR	17.72	22.59	268.21	n/a
Speedup$_{\text{expected}}$ Efficiency$_{\text{expected}}$	11.80 98.3%	15.47 96.7%	22.06 91.9%	n/a n/a
Speedup$_{\text{best}}$ Efficiency$_{\text{best}}$	10.47 87.2%	8.21 51.3%	0.69 2.88%	n/a n/a
Speedup$_{1/1}$ Efficiency$_{1/1}$	10.96 91.3%	8.60 53.7%	0.72 3.02%	n/a n/a

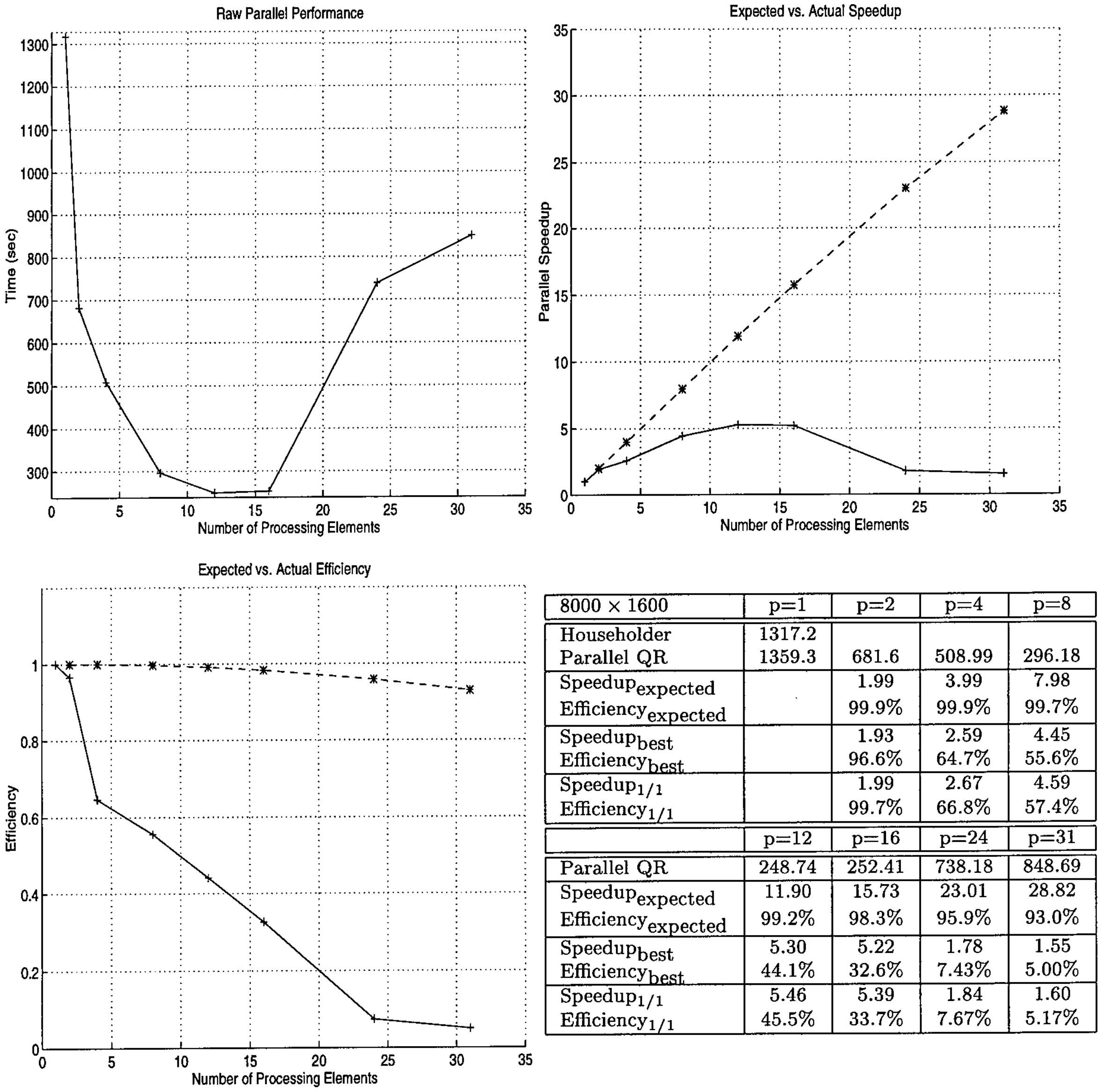
FIG. 5.9. Parallel performance on the SGI/Cray Origin 2000 in shared mode. Matrix size is 4000 × 800. Expected results shown as a dashed line with '$*$' symbol. Actual results shown as a solid line with '$+$' symbol.



8000 × 800	p=1	p=2	p=4	p=8
Householder	448.11			
Parallel QR	459.31	229.55	125.0	65.08
Speedup$_{\text{expected}}$		2.00	3.99	7.99
Efficiency$_{\text{expected}}$		99.9%	99.9%	99.4%
Speedup$_{\text{best}}$		1.95	3.58	6.89
Efficiency$_{\text{best}}$		97.6%	89.6%	86.1%
Speedup$_{1/1}$		2.00	3.67	7.06
Efficiency$_{1/1}$		100%	91.9%	88.22%

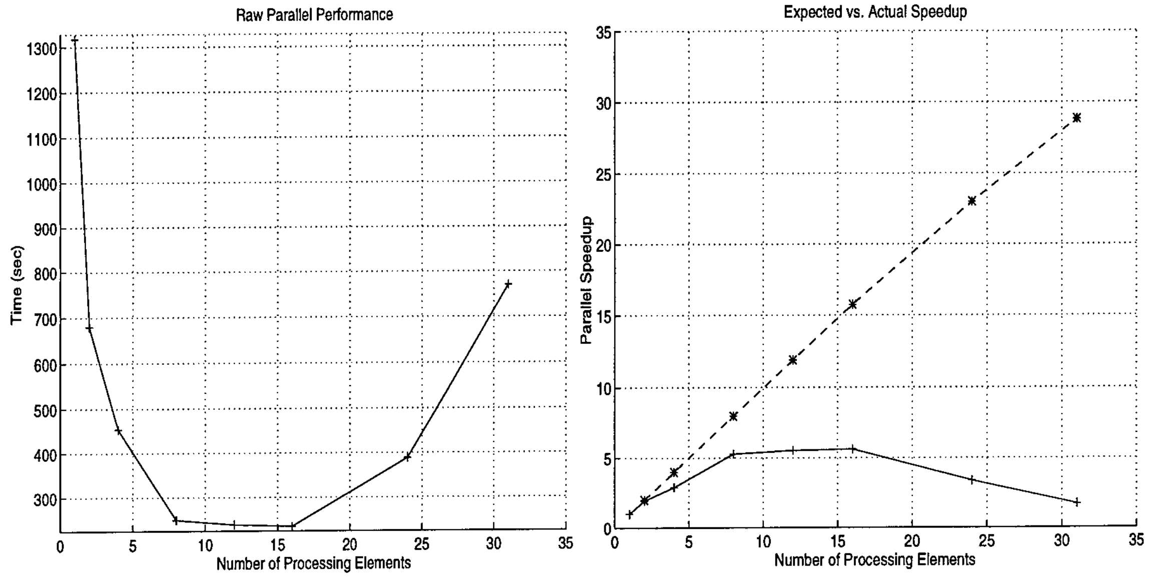
	p=12	p=16	p=24	p=31
Parallel QR	42.34	41.02	n/a	n/a
Speedup$_{\text{expected}}$	11.94	15.81	n/a	n/a
Efficiency$_{\text{expected}}$	99.6%	98.8%	n/a	n/a
Speedup$_{\text{best}}$	10.58	10.92	n/a	n/a
Efficiency$_{\text{best}}$	88.2%	68.3%	n/a	n/a
Speedup$_{1/1}$	10.85	11.20	n/a	n/a
Efficiency$_{1/1}$	90.4%	70.0%	n/a	n/a

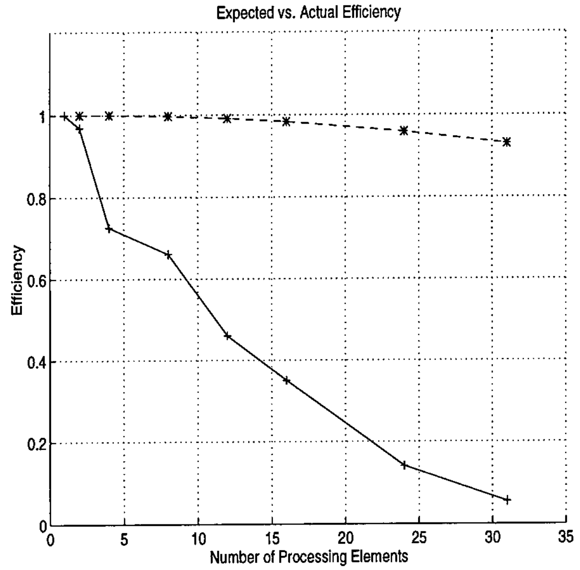
FIG. 5.10. Parallel performance on the SGI/Cray Origin 2000 in shared mode. Matrix size is 8000 × 800. Expected results shown as a dashed line with '$*$' symbol. Actual results shown as a solid line with '+' symbol.



8000 × 1600	p=1	p=2	p=4	p=8
Householder Parallel QR	1317.2 1359.3	681.6	508.99	296.18
$\text{Speedup}_{\text{expected}}$ $\text{Efficiency}_{\text{expected}}$		1.99 99.9%	3.99 99.9%	7.98 99.7%
$\text{Speedup}_{\text{best}}$ $\text{Efficiency}_{\text{best}}$		1.93 96.6%	2.59 64.7%	4.45 55.6%
$\text{Speedup}_{1/1}$ $\text{Efficiency}_{1/1}$		1.99 99.7%	2.67 66.8%	4.59 57.4%
	p=12	p=16	p=24	p=31
Parallel QR	248.74	252.41	738.18	848.69
$\text{Speedup}_{\text{expected}}$ $\text{Efficiency}_{\text{expected}}$	11.90 99.2%	15.73 98.3%	23.01 95.9%	28.82 93.0%
$\text{Speedup}_{\text{best}}$ $\text{Efficiency}_{\text{best}}$	5.30 44.1%	5.22 32.6%	1.78 7.43%	1.55 5.00%
$\text{Speedup}_{1/1}$ $\text{Efficiency}_{1/1}$	5.46 45.5%	5.39 33.7%	1.84 7.67%	1.60 5.17%

FIG. 5.11. Parallel performance on the SGI/Cray Origin 2000 in shared mode. Matrix size is 8000 × 1600. Expected results shown as a dashed line with '$*$' symbol. Actual results shown as a solid line with '+' symbol.





8000 × 1600	p=1	p=2	p=4	p=8
Householder Parallel QR	1317.2 1359.3	679.8	453.47	249.2
$\text{Speedup}_{\text{expected}}$ $\text{Efficiency}_{\text{expected}}$		1.99 99.9%	3.99 99.9%	7.98 99.7%
$\text{Speedup}_{\text{best}}$ $\text{Efficiency}_{\text{best}}$		1.94 96.9%	2.90 72.6%	5.29 66.1%
$\text{Speedup}_{1/1}$ $\text{Efficiency}_{1/1}$		1.99 99.9%	2.99 74.9%	5.45 68.2%
	p=12	p=16	p=24	p=31
Parallel QR	238.33	234.69	388.73	769.71
$\text{Speedup}_{\text{expected}}$ $\text{Efficiency}_{\text{expected}}$	11.90 99.2%	15.73 98.3%	23.01 95.9%	28.82 93.0%
$\text{Speedup}_{\text{best}}$ $\text{Efficiency}_{\text{best}}$	5.53 46.1%	5.61 35.1%	3.39 14.1%	1.71 5.52%
$\text{Speedup}_{1/1}$ $\text{Efficiency}_{1/1}$	5.70 47.5%	5.79 36.2%	3.50 14.6%	1.77 5.70%

FIG. 5.12. Parallel performance on the SGI/Cray Origin 2000 in **dedicated** mode. Matrix size is 8000 × 1600. Expected results shown as a dashed line with '∗' symbol. Actual results shown as a solid line with '+' symbol.

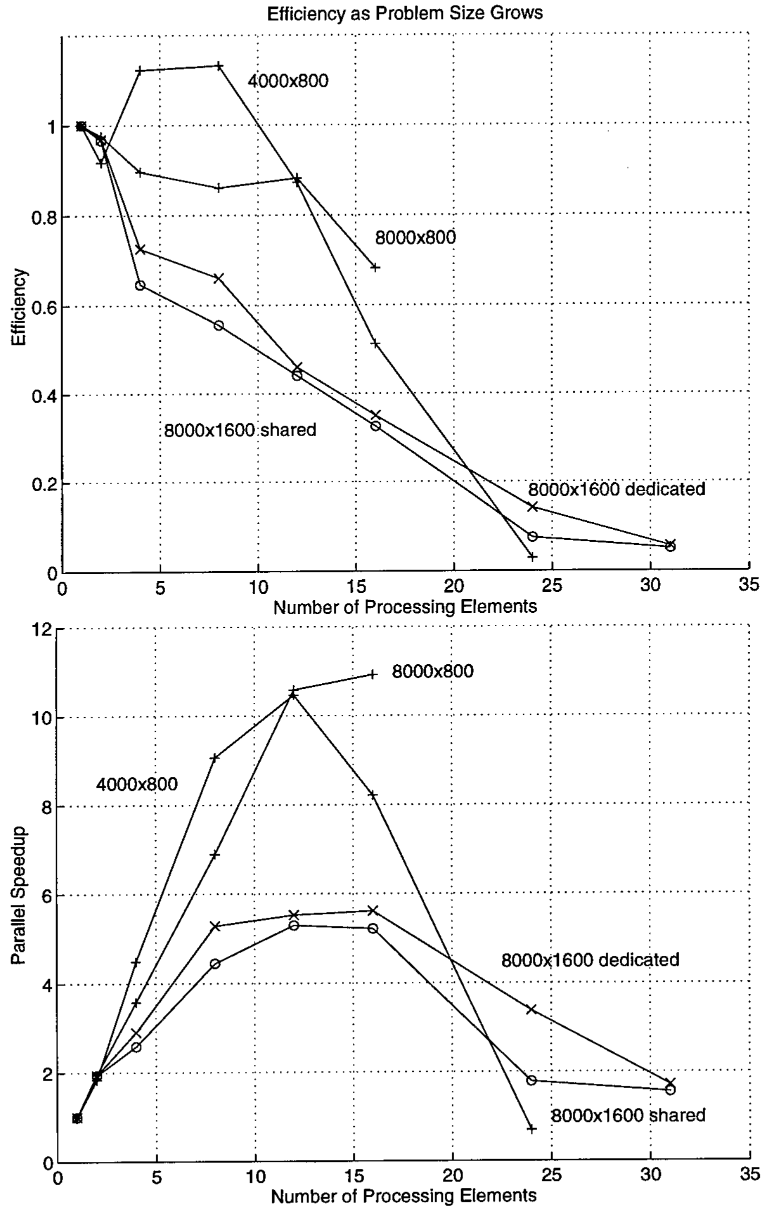


FIG. 5.13. Parallel performance comparison on the SGI/Cray Origin 2000 as problem size grows. Matrix sizes are shown next to their representative line.

5.3 Application to Electromagnetic Scattering

This section presents one set of results obtained by applying the parallel QR decomposition algorithm to a dense matrix with data from the electromagnetic scattering problem described in Chapter 4. The largest data set available is a matrix size of 1024×1024. Because the matrix size is relatively small, timing information was gathered for all processors ranging from 1 to 12 instead of using the previous method (1, 2, 4, 8, ...).

Figure 5.14 shows the performance results from the Power Challenge and Origin 2000 when the parallel QR decomposition algorithm is applied to electromagnetic scattering data. The results are promising because they show increased performance on both machines from 1 to 10 processors (9 on the Power Challenge). Speedup gains are in the same range. Given the small problem size and fast execution times, maintaining parallel speedup for such a range of processors demonstrates the effective use of synchronization and load balancing in the algorithm. If the implementation involved large amounts of processor waiting or poor load balancing, speedup would have fallen off sooner than 9 or 10 processors. The decreasing tendency of parallel efficiency is mainly due to the relatively high system overhead when compared to the short total execution time.

It is important to note the difference in the two machines when over-parallelization occurs and run times begin to increase. On the Power Challenge, when too many processors are used relative to the problem size, performance degradation occurs, but in a gradual manner. On the Origin 2000 however, once over-parallelization occurs, performance degrades in a very pronounced way. This is represented by the sharp increase in run time shown for the 11 and 12 processor cases in Figure 5.14. This is again attributable to the distibution of shared memory on the Origin 2000. When parallel overhead begins to dominate performance on the Power Challenge, memory access times remain relatively constant, so performance decreases gradually. When parallel overhead begins to dominate performance on the Origin 2000, memory access and contention during the BR stage also increase, which makes performance decrease much more quickly.

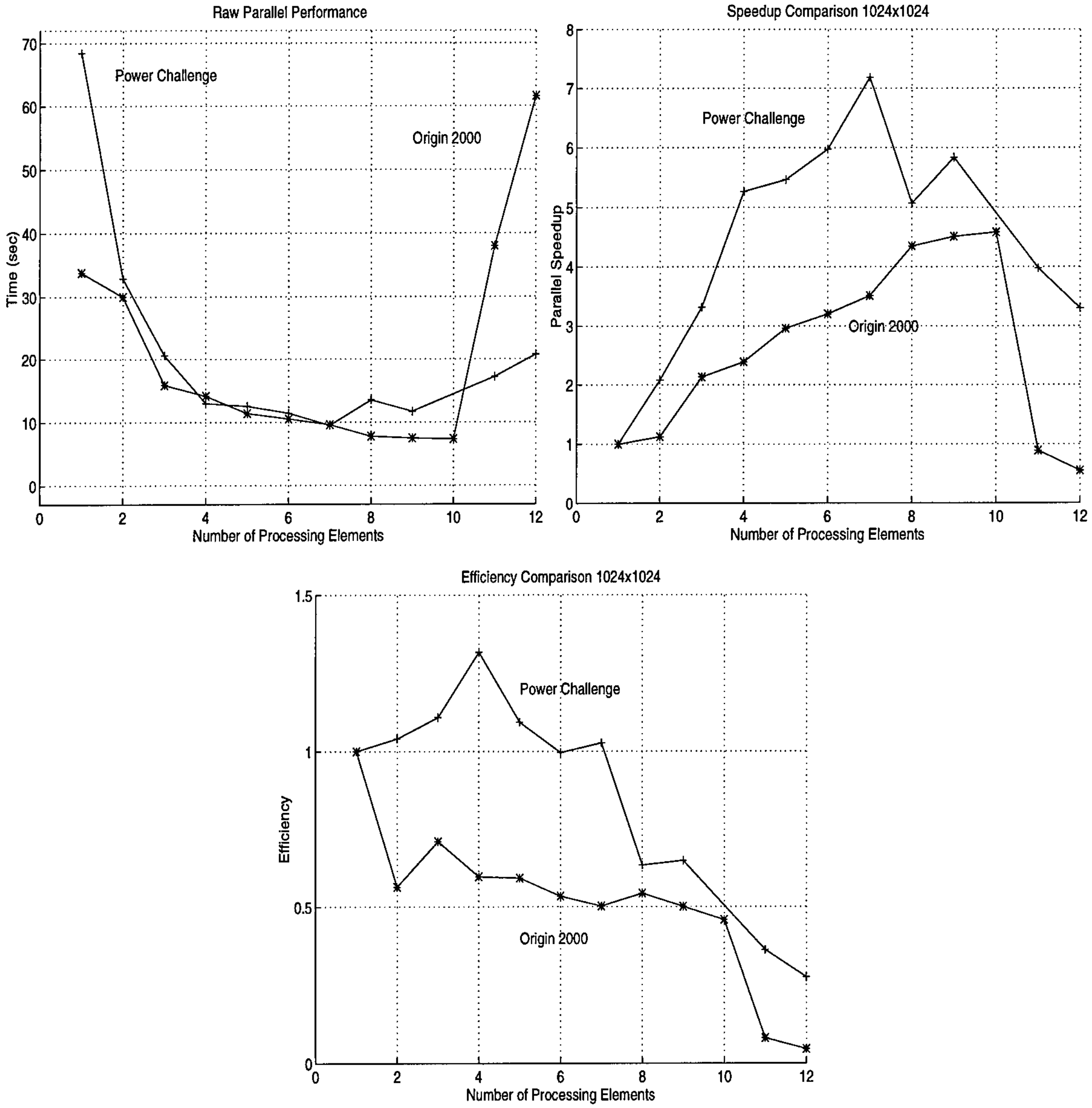


FIG. 5.14. Parallel performance comparison of the Power Challenge (R10000) and the SGI/Cray Origin 2000 on electromagnetic scattering data. Matrix size is 1024 × 1024. Power Challenge results shown with '+' symbol. Origin 2000 results shown with '$*$' symbol.

Chapter 6

CONCLUSIONS

This report introduces a new parallel QR decomposition algorithm. The algorithm is described, analyzed, and tested. The motivation for the development of a new algorithm was provided by ongoing work on electromagnetic scattering problems. A brief presentation of the algorithm's performance on data from this problem domain is also presented.

The goals surrounding the development of the algorithm were driven by pragmatism, but they did serve to extend this class of algorithms in important ways. The balancing of work across processors was done at a finer grain than earlier attempts, which is more appropriate for today's parallel architectures. This makes much higher efficiencies possible on current computers than previous algorithms. The use of a hybrid elimination technique in support of another of the goals, maximizing computation between communication, also indicates potential for good performance on distributed memory machines and networks of workstations.

The analysis of the algorithm's expected performance is very promising. The load balancing scheme derived for the balanced rotations stage of the algorithm approaches perfect efficiency for large problem sizes on modest numbers of processors. The hybrid nature of the algorithm is also beneficial because it uses the most efficient elimination technique whenever possible.

The measured performance results of the algorithm's implementation are good, but they do not track closely enough to the predicted performance. Results on 12 and fewer processors follow expected performance well, but the use of higher numbers of processors adds little speed improvement. It is anticipated that, given sufficient dedicated computing time, higher processor numbers would yield better results on larger problem sizes. As is the case with all parallel algorithms, there are a myriad of subtle implementation details that must be considered and worked on before a tuned version of the algorithm is optimized for any specific parallel machine. Considering the performance results in light of the fact that the algorithm was implemented exactly as described, with no optimization tricks, is a promising indicator.

Finally, a comment regarding the comparison of the predicted and actual performance is appropriate. In all the testing (except for one case), predicted and actual performance differed by a significant margin. One point did hold true throughout the testing. The trends predicted by the expected performance analysis were realized by the measured results. This helps to reinforce the general correctness of the prediction model, but indicates that fine tuning and inclusion of more realistic overheads are in order. Overall, the algorithm and its predicted performance were successful. They met and exceeded the goals set out, and useful methods were added to the computational linear algebra and parallel processing "tool boxes."

6.1 Contributions

The work included in this report contributed to the computer science, parallel computing, and computational linear algebra communities in the following ways:

- A new parallel QR decomposition algorithm was developed and analyzed. Expected performance promises excellent, achievable results.

- The new algorithm is specifically designed for parallel architectures with a modest number of processing elements. All but one of the previous algorithms covered in Chapter 2 are designed without limiting the number of processors.

These algorithms normally require at least $m/2$ processors to achieve optimal performance. This is unreasonable for large problem sizes, and using less processors in these algorithms results in poor performance. The new algorithm can theoretically achieve optimal performance on as few as two processors.

- Initial implementation and testing results of the new parallel algorithm are encouraging. In some cases, over 90% efficiency was achieved with no specialized tuning required.

- A novel approach was taken to predicting algorithmic performance by simulating the work and operation of a parallel algorithm on a sequential computer.

- The idea of a hybrid parallel algorithm which uses the most efficient techniques available in different sections of the algorithm to reduce overall complexity was demonstrated and extended.

- This parallel algorithm achieves better performance in the sequential case than the best known sequential algorithm as it exploits the hierarchical nature of memory through blocking. Even for sequential computing, this parallel QR algorithm is a good candidate for improved application performance.

- Several strengths and limitations of the distributed shared memory model for parallel computers were tested and reinforced, and in some cases, performance suffered as a result of the current implementations of the model.

6.2 Future Work

There is still a good deal of study and analysis that should be pursued concerning parallel QR decomposition. The frequency of new research and parallel QR algorithms has decreased in the current literature. However, the need for good solutions to this problem which would support a wide range of engineering and scientific applications has never been higher. Highly efficient algorithms for solving large systems of equations, which are optimized for current parallel architectures are not widely available. This is especially true in the case of highly distributed, low communication solutions. Future research directions specific to the algorithm presented in this report include:

- Optimize the algorithm and test larger cases on a shared memory machine.
- Since the work done for each step during the BR stage is predictable, re-order the assignment method to increase performance on a distributed shared memory machine by taking into account locality of access when assigning Givens rotations. This work could be the starting point for a distributed memory algorithm.
- Integrate the parallel decomposition routines into the existing electromagnetic scattering code described in Chapter 4.
- Implement and test the algorithm on a distributed memory machine and a network of workstations.
- Continue refinement of the predicted performance method for this type of hybrid algorithm. Accurate prediction for a wide range of problem sizes would aid greatly in the proper selection and application of appropriate algorithms.

Broader research goals, which are a natural extension of the work performed here are:

- Implement a wide range of the known parallel QR decomposition algorithms on the same hardware platform and compare performance. Not only would this provide direct performance comparisons in place of predicted values which are based on complexities, but it also would help characterize the range of problems each algorithm was bested suited to solve.

- Experiment with duplicating computation at distributed nodes to reduce communication costs. One example of this is to give every processor a duplicate copy of the pivot rows to use in order to fully annihilate all the elements in their assigned blocks with no communication. Research could then focus on recombining the now unique information in the pivot rows in an effort to maintain the integrity of the solution.

References

Adve, Sarita V., & Gharachorloo, Kourosh. 1996. Shared Memory Consistency Models: A Tutorial. *IEEE Computer*, **29**(12), 66–76.

Anderson, E., Bai, Z., Bischof, C., Demmel, J., Dongarra, J., Du Croz, J., Greenbaum, A., Hammarling, S., McKenney, A., Ostrouchov, S., & Sorensen, D. 1995. *LAPACK Users' Guide*. Second edn. Philadelphia: SIAM.

Bell, Gordon. 1994. Why there won't be apps: The problem with MPPs. *IEEE Parallel and Distributed Technology*, **2**(3), 5–6.

Boleng, J., Craig, C., DeSanto, J., Erdmann, G., Hereman, W., Khebchareon, M., Misra, M., & Sinex, A. 1996 (October). *Computational Modelling of Rough Surface Scattering, CSM-MCS-96-09.* Tech. rept. Colorado School of Mines, Golden, CO. Tech. Report for MURI project AFOSR grant F49620-96-1-0039, Report of work in progress.

Charny, Benjamin. 1996. Matrix Partitioning on a Virtual Shared Memory Parallel Machine. *IEEE Transactions on Parallel and Distributed Systems*, **7**(4), 343–355.

Clarke, Jerry A. 1997. Emulating Shared Memory to Simplify Distributed-Memory Programming. *IEEE Computational Science and Engineering*, **4**(1), 55–62.

Cordsen, Jörg, Garnatz, Thomas, Sander, Michael, Gerischer, Anne, Gubitoso, Marco Dimas, Haack, Ute, & Schröder-Preikschat, Wolfgang. 1997. Vote for Peace: Implementation and Performance of a Parallel Operating System. *IEEE Concurrency*, **5**(2), 16–27.

Cosnard, M., & Robert, Y. 1983. Complexité de la factorisation QR en parallèle. *C.R. Acad. Sci. Paris 297A*, 549–552.

Cosnard, M., & Robert, Y. 1986. Complexity of parallel QR decomposition. *Journal of the Association for Computing Machinery*, **33**(4), 712–723.

Cosnard, Michel, & Daoudi, El Mostafa. 1994. Optimal Algorithms for Parallel Givens Factorization on a Coarse Grained PRAM. *Journal of the Association for Computing Machinery*, **41**(2), 399–421.

Cosnard, Michel, & Trystram, Denis. 1995. *Parallel Algorithms and Architectures.* Boston, MA: PWS Publishing Company.

Cownie, James. 1994. Why MPPs? *IEEE Parallel and Distributed Technology*, **2**(3), 7–8.

Cybenko, George. 1996. Large-Scope Computing: A Challenge for the 21st Century. *IEEE Computational Science and Engineering*, **3**(2), 1,10.

DeSanto, J. A. 1985. Exact spectral formalism for rough-surface scattering. *Journal of the Optical Society of America*, **2**, 2202–2206.

Dongarra, Jack. 1993. Linear Algebra Libraries for High-Performance Computers: A Personal Perspective. *IEEE Parallel and Distributed Technology*, **1**(1), 17–24.

Dongarra, Jack J., & Walker, David W. 1996. *Software libraries for linear algebra computations on high performance computers.* Tech. rept. University of Tennessee and Oak Ridge National Laboratory, Knoxville, TN and Oak Ridge, TN. Submitted to SIAM Review.

Dongarra, Jack J., Duff, Iain S., Sorensen, Danny C., & van der Vorst, Henk A. 1991. *Solving Linear Systems on Vector and Shared Memory Computers.* Philadelphia: SIAM.

Dongarra, J.J., Sameh, A.H., & Sorensen, D.C. 1986. Implementation of some concurrent algorithms for matrix factorization. *Parallel Computing*, **3**, 25–34.

Foster, Ian. 1995. *Designing and Building Parallel Programs: Concepts and Tools for Parallel Software Engineering.* Reading, MA: Addison-Wesley Company, Inc.

Gallivan, K. A., Heath, Michael T., Ng, Esmond, Ortega, James M., Peyton, Barry W., Plemmons, R.J., Romine, Charles H., Sameh, A. H., & Voigt, Robert G. 1990. *Parallel Algorithms for Matrix Computations.* Philadelphia: SIAM.

Geist, G.A., & Romine, C.H. 1987. *LU factorization algorithms on distributed memory multiprocessor architectures.* Tech. Report ORNL/TM-10383. Oakridge National Laboratory, Oak Ridge, TN.

Golub, Gene H., & Van Loan, Charles F. 1996. *Matrix Computations.* Third edn. Baltimore, MD: The John Hopkins University Press.

Islam, Nayeem, & Campbell, Roy H. 1992. Design Considerations for Shared Memory Multiprocessor Message Systems. *IEEE Transactions on Parallel and Distributed Systems*, **3**(6), 702–711.

Lord, R. E., Kowalik, J. S., & Kumar, S. P. 1983. Solving Linear Algebraic Equations on an MIMD Computer. *Journal of the Association for Computing Machinery*, **30**(1), 103–117.

Modi, J.J., & Clarke, M.R.B. 1984. An alternative Givens ordering. *Numerical Mathematics*, **43**, 83–90.

Morton, Donald J., & Tyler, John M. 1996. Minimizing Development Overhead with Partial Parallelization. *IEEE Parallel and Distributed Technology*, **4**(3), 15–24.

Pancake, Cherri M. 1996. Is Parallelism for You? *IEEE Computational Science and Engineering*, **3**(2), 18–37.

Pothen, A., Jha, S., & Vemulapati, U. 1987. Orthogonal factorization on a distributed memory multiprocessor. *Pages 587–596 of:* Heath, M.T. (ed), *Hypercube Multiprocessors 1987*. Philadelphia: SIAM.

Pothen, Alex, & Raghavan, Padma. 1989. Distributed Orthogonal Factorization: Givens and Householder Algorithms. *SIAM Journal of Scientific and Statistical Computing*, **10**(6), 1113–1134.

Protič, Jelica, Tomaševič, Milo, & Milutinovič, Veljko. 1996. Distributed Shared Memory: Concepts and Systems. *IEEE Parallel and Distributed Technology*, **4**(2), 63–79.

Quinn, Michael J. 1994. Thinking about Thinking Machines. *IEEE Parallel and Distributed Technology*, **2**(3), 2.

Sameh, A. H., & Kuck, D. J. 1978. On Stable Parallel Linear System Solvers. *Journal of the Association for Computing Machinery*, **25**(1), 81–91.

Siegel, Howard J., Watson, Daniel W., & Antonio, John K. 1996. What will it take to sell a massive number of massively parallel machines? *IEEE Parallel and Distributed Technology*, **4**(3), 63–69.

Stewart, David E., & Leyk, Zbigniew. 1994. *Meschach Library Version 1.2b Description and Tutorial.* Australian National University, Canberra, Australia.

Sun, Xian-He, & Zhu, Jianping. 1995. Performance Considerations of Shared Virtual Memory Machines. *IEEE Transactions on Parallel and Distributed Systems*, **6**(11), 1185–1194.

Sun, Xian-He, & Zhu, Jianping. 1996. Performance Prediction; A case study using a scalable shared-virtual-memory machine. *IEEE Parallel and Distributed Technology*, **4**(4), 36–49.

Wallach, Steve. 1994. Teraflops into laptops. *IEEE Parallel and Distributed Technology*, **2**(3), 8–9.

Wladawsky-Berger, Irving. 1994. Parallel applications: The next frontier for computer industry breakthroughs. *IEEE Parallel and Distributed Technology*, **2**(3), 10–11.