ELSEVIER

Parallel Computing 27 (2001) 1321–1345

PARALLEL COMPUTING

www.elsevier.com/locate/parco

Load balanced parallel QR decomposition on shared memory multiprocessors [☆]

Jeff Boleng [*], Manavendra Misra

Department of Math and Computer Sciences, Colorado School of Mines, Golden, CO 80401, USA

Received 5 August 1998; received in revised form 25 September 1998; accepted 20 May 1999

Abstract

This paper introduces a new parallel QR decomposition algorithm with two key advantages over previous techniques. It is specifically designed to achieve high parallel efficiency on shared memory parallel computers with a modest number of processors, and the novel load balancing method described here 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 seeks to maximize computation between communication and synchronization. Implementation results on shared memory multiprocessors track expected performance well up to 12 processors, and initial performance results on a distributed shared memory machine are presented. © 2001 Elsevier Science B.V. All rights reserved.

Keywords: Parallel solutions of linear systems; Shared memory multiprocessors; Load balancing; QR decomposition

1. Introduction

The problem of solving dense systems of linear equations on parallel computers has been studied extensively. Research efforts tend to treat the problem with theoretical rigor but often do not actually implement the algorithms on machines because real machines which meet the requirements and assumptions of the theoretical derivation rarely or never exist. Practitioners have taken these studies, added their

[☆] Partial support for this work was provided by the AFOSR under the MURI program, grant #F49620-96-1-0039, and the NCSA under grants TRA960029N and ASC970018N.

[*] Corresponding author.

E-mail addresses: boleng@rmi.net (J. Boleng), manav@kbkids.com (M. Misra).

own analysis and creativity, and adapted them to produce realistic algorithms which can be implemented on parallel computers of the day. However, these implementations rarely achieve theoretically promised parallel efficiency or speedup. This process creates two general classes of algorithms: (i) those that are theoretically optimal with respect to some parameter, and (ii) those that might not be theoretically optimal but are tuned to run on a specific architecture.

The problem with the second class of algorithms is that they often require extensive adaptation as machine architectures evolve. A third option is the smaller class of algorithms which is beginning to gain popularity. These algorithms are designed for, and rigorously tested on widely available production machines [2]. This class of algorithms is characterized by realistic assumptions which result from considering existing architectures during algorithm design. The goal is to create production quality numerical techniques which are cost-effective to develop. Often, excellent results are achieved on the target machines. This work aims to develop algorithms in this last class.

This paper begins with a brief overview of the algorithms currently available for the QR factorization of a dense matrix in Section 2. Section 3 introduces a new algorithm for parallel QR factorization. Performance results from applying the algorithm to general matrices are presented in Section 4. Section 5 concludes the paper.

1.1. Notation and conventions

The QR factorization of a general matrix A is represented by

$$A = QR, \tag{1}$$

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

Throughout this paper, the sequential complexity of QR factorization will be based on the results from [10] (see Table 1). [1] The time requirements listed in Table 1, which count the number of floating point operations performed, [2] assume $m = n$ for Gaussian elimination, and $m \geqslant n$ for all other cases.

All computational results in this paper assume A has full rank and $m \geqslant n$. 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 [3]

[1] The fast Givens technique is not included here due to possible overflow problems. The complexity of fast Givens is better than the traditional Givens method ($\sim (2n^2(m - n/3))$ vs. $(3n^2(m - n/3))$).

[2] For simplicity, additions, subtractions, multiplications, divisions, and square roots are all assumed to be unit time operations. The $\sim$ symbol will be used throughout this paper to denote the approximate operation count needed to compute the value shown.

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

Table 1
Number of computational steps for common factorization methods

Method	Number of steps
Gaussian elimination	$\sim(2n^3/3)$
Normal equations with Cholesky factorization	$\sim(mn^2 + n^3/3)$
QR using Householder reflections	$\sim(2n^2(m - n/3))$
QR using Givens rotations	$\sim(3n^2(m - n/3))$

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

A general linear least squares solution to a system of equations $A\vec{x} = \vec{b}$ utilizing QR decomposition has the following steps:
1. Compute Q and R as defined in Eq. (1),
2. Compute vector $\vec{d}$ defined as $\vec{d} = Q^{\mathrm{T}}\vec{b}$,
3. Use back substitution to solve for $\vec{x}$ in $R_1\vec{x} = \vec{d}$,
4. Compute the residual $= \sqrt{\sum_{i=n+1}^{m} d_i^2}$.

The most computationally demanding step in the algorithm involves the computation of Q and R (step 1), therefore this research focuses on the parallelization of this process.

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. We begin with a brief description of these two techniques.

1.1.1. Householder reflections

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

$$H = I - \beta\vec{v}\vec{v}^{\mathrm{T}}, \tag{3}$$

where $\beta = 2/\vec{v}^{\mathrm{T}}\vec{v}$ is called a Householder matrix (or reflection). The vector $\vec{v}$ is called a Householder vector. Pre-multiplication of the coefficient matrix A with H is used to zero out appropriate elements of A. It is easy to verify that Householder matrices are symmetric and orthogonal [10].

The definition of the Householder matrix involves the computation of an outer product $\vec{v}\vec{v}^{\mathrm{T}}$, an $\mathrm{O}(n^2)$ complexity operation. However, the practical time requirement of using H to zero out elements in A is lower than that of computing a full outer product. This is because the explicit computation of the full matrix H is not necessary in practice. Pre-multiplication of the Householder matrix to A can exploit the high degree of structure in H and can be done in $\sim 4mn$ steps (where $A \in \mathbb{R}^{m\times n}$ and $H \in \mathbb{R}^{m\times m}$).

Householder reflections work well for introducing large number of zeros using just one matrix multiplication (computing HA). Normally, all the elements below the diagonal of an entire column of the matrix A are eliminated by one Householder reflection. However, this leads to a difficulty when Householder transforms are implemented on parallel machines. One reflection affects multiple rows, and therefore, it is difficult to exploit fine-grained parallelism in the operation. The alternative is to exploit coarse-grained parallelism by applying multiple (smaller) Householder reflections to parts of the same matrix in parallel. Thus, smaller reflections are applied to blocks of the matrix A. This is the key idea behind the hybrid algorithms described in Sections 2 and 3 and will be presented in greater detail later. An alternative to Householder reflections is QR decomposition via Givens rotations. A brief description of this technique is presented below.

1.1.2. 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 (pivot row). We will use the notation introduced in [7] 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 easily shown to be orthogonal. Using Givens rotations to find Q and R is similar to the procedure used to compute Householder transforms, however, the number of Givens matrices needed is $r = n(n-1)/2$, as opposed to the number of Householder matrices needed which is n when $m > n$, and $(n-1)$ when $m = n$.

Given two vectors $\vec{u}, \vec{v} \in \mathbb{R}^n$, the Givens rotation to zero out the first element of $\vec{v}$ can be depicted as [7]

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

where c and s are calculated as [14]

$$c = \frac{u_1}{\sqrt{u_1^2 + v_1^2}}, \quad s = \frac{v_1}{\sqrt{u_1^2 + v_1^2}} \tag{4}$$

and $\vec{u}'$ and $\vec{v}'$ are calculated by

$$\begin{aligned} u'_i &= cu_i + sv_i, \quad 1 \leqslant i \leqslant n, \\ v'_i &= -su_i + cv_i, \quad 1 \leqslant i \leqslant n. \end{aligned}$$

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 or 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 computation requires $\sim 6n + 6$ steps.

1.1.3. Reflections vs. rotations

A few final notes must be made concerning Householder reflections and Givens rotations before proceeding further. 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 reveals that performing the entire decomposition via Givens rotations costs only about one and a 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. Third, a Givens rotation, $G(i,j,k)$ can be used 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.

Finally, it is possible to perform Householder reflections on submatrices. This idea, due to Pothen and Raghavan [13], makes it possible to perform a great deal of the QR decomposition in parallel 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 one key to the development of the new parallel QR algorithm introduced here.

2. Related work in parallel QR algorithms

This section provides a brief overview of existing parallel QR factorization algorithms and summarizes a few of the most recently developed methods. 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 [14]. 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 include Sameh and Kuck's [14] algorithm, parallel Gaussian elimination algorithms by Lord et al. [11], Greedy algorithms introduced independently in [5,12], and Block algorithms which are widely discussed in [8,9,2] to name just a few. Block-cyclic partitioning of matrices onto processors has significant advantages in distributed memory environments [2], but in a shared memory environment, a finer partitioning of the matrix can be used without extra communication penalties. The ScaLAPACK group has done significant work in developing and implementing parallel numerical algorithms, and details on their ongoing efforts can be found at http://www.netlib.org/scalapack.

Additional work in the area of parallel QR factorization has been described in [6,13]. The first extends the ideas introduced in [12] by using Fibonacci sequences to maximize the number of simultaneous Givens rotations in a Greedy algorithm. The work by Pothen and Raghavan is concerned with implementing QR factorization using a distributed memory programming model.

Special attention must be given to the final algorithm discussed by Pothen and Raghavan [6] because this is the starting point for the new algorithm introduced here. Pothen and Raghavan's most effective distributed algorithm is a hybrid based on the distributed Givens and Householder algorithms introduced earlier in their work. 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 (which have fewer elements since $m > n$) 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 into two phases:

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

During the IP, each processor applies a local Householder reflection to zero all but one element of the current column. The RP 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.

Based on the results listed above, predicted and actual performance figures are included in [13]. Run time results were measured on the Intel Paragon iPSC-286 hypercube with 16 processors. The experimental results obtained correlate closely to those predicted, therefore supporting the derived complexities.

Table 2
Arithmetic and communication complexities for the hybrid algorithm of Pothen and Raghavan from [13][a]

	Number of steps
Arithmetic	$\frac{n^2}{p}(m - n/3)\tau + n^2 \log(p)\tau + \frac{3}{2}n^2\tau + \frac{mn}{p}\tau$
Communication	$n^2 \log(p)\alpha + 2n \log(p)\beta$

[a] 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.

3. A load balanced hybrid QR algorithm

This section introduces 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 (in particular, we focus on shared memory machines),
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 forces 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. Since we are designing with shared memory machines in mind, the exploitation of finer grained parallelism does not incur extra communication costs. We present the details and an example of the new parallel QR algorithm. Also, a complexity analysis of the algorithm is performed, and predicted and actual performance is presented.

The algorithm presented here differs from other 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. [4] 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 among the most widely available and cost effective to buy and program. In addition, emphasis on and concern for the implementation costs of parallel software are growing and the shared memory programming model addresses these concerns effectively. There is also some movement toward a shared memory programming model even on the most cost effective kind of parallel computers, networks of workstations [3,4].

[4] In general, US manufacturers have moved away from production of vector machines and are focusing on parallel architectures using commodity processors and networks of workstations. There is active research, development, and manufacturing of new parallel vector machines elsewhere by companies such as NEC and Fujitsu.

The algorithm departs from common practice with the first item, but with the second it builds on the ideas introduced by Pothen and Raghavan [13] by considering communication as an algorithmic cost in a QR factorization routine. It uses Householder reflections for computation local to one processor because of the lower complexity, and Givens rotations for inter-processor annihilation because of their independence. The algorithm will be the most suitable for non-vector shared memory parallel computers with fewer processors than equations in the system.

3.1. 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 fully decomposing a matrix. 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 [13]. 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 result in a matrix with p upper trapezoidal submatrices as shown in Fig. 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 across the processors.
   - The step proceeds with 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 Fig. 2.
   - The balanced assignment of rows to PEs will be described fully in Section 3.3.

After one application of the two stages, 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 Fig. 2 by Y entries (of size 25×9 in this case). This process is repeated in a recursive manner 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 nth 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 [13], or sequentially, since in the end, the final Givens rotation must use element $A(n,n)$ to eliminate $A(n+1,n)$.

$$
\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. 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.

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 Fig. 2 shows that row 8 is in the desired format, but it is included in the submatrix for the second 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 so that more efficient Householder reflections can be used in the annihilation of column 8 during the IR stage in the second recursive call.

This new hybrid algorithm is different from that introduced by Pothen and Raghavan [13] 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

$$\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}{} & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 & \multicolumn{1}{c}{16}
\end{array}$$

Fig. 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 recursively during the second application of the IR and BR stages.

described above maximizes the work done in the IR stage by letting each processor perform the maximum possible number of Householder reflections ($(m/p) - 1$) before a parallel synchronization is required to proceed with the BR stage. This substantially reduces the number of processor synchronizations required. Second, the algorithm described above balances the Givens rotation load across processors and performs ($(p-1)\lfloor (m/p) - 1 \rfloor$) simultaneous Givens rotations between synchronizations.

3.2. 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.

Case 1 ($m/p = n$). When this condition occurs, all the partitioned blocks of the matrix which are distributed to the processors are square, including the first. The IR stage of the algorithm will annihilate all the desired elements in the first block with $(m/p) - 1$ Householder reflections, and each partition will be fully decomposed to upper triangular form. 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.

Case 2 ($m/p > n$). 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 in all partitioned blocks for rows greater than m/p. 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. Again, the matrix will be in the desired form after only one iteration of the algorithm.

Case 3 ($m/p < n$). When this final case occurs, the structure after the IR stage will be similar to that depicted in Fig. 1. Once the BR stage completes, the whole algorithm must be recursively applied to the

$$\left(m - \frac{m}{p} + 1\right) \times \left(n - \frac{m}{p} + 1\right)$$

submatrix depicted with Y symbols in Fig. 2. The algorithm will terminate when the entering condition (based on submatrix "shape") is that of either Case 1 or Case 2 above.

3.3. 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. 1), there are $(p-1)$ blocks that have m/p rows and n columns, and it can be determined that

$$(p-1)\left[\frac{\frac{m}{p}\left(\frac{m}{p}-1\right)}{2}\right] \tag{5}$$

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, because on non-vector parallel machines it takes different amounts of time to perform Givens rotations on vectors of different lengths.

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 $n(n-1)/2$ Givens rotations, which is simply the sum of the integers from 1 to $n-1$. The general form to sum integers is $\sum_{i=1}^{n} i = n(n+1)/2$ 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 above idea, a way to balance the BR stage load across p processors is shown in Fig. 3. 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. Note that the processor assignment is done in a cyclic way. This results in two "cycles" that contain six 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 perfectly 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 amount of work is calculated using the complexity of one Givens rotation presented in Section 1.1.2 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 many blocks like those in Fig. 3, and that it looks like the matrix from the previous example in Fig. 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 Fig. 2. It appears as shown in Fig. 4. 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 Fig. 2. Fig. 5 shows the row-wise assignment for each step and the work done during the full BR stage of the algorithm for an example matrix. Table 3 summarizes the work done by each processor during each step for this example.

Even in this simple example, the work is well balanced among the processors. We will see in the following 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 Section 4 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

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

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

slightly less efficient load balance, which is easily seen in the simple example of Table 3. The actual imbalance with larger problem sizes is nearly insignificant.

3.4. Algorithmic complexity

The following is a brief discussion on the complexity of the algorithm. For a complete analysis and description see Boleng [1]. The complexity equations are presented for one iteration of the two algorithmic stages (IR and BR).

$$\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 \\
\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. 4. 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.

3.4.1. Complexity of internal reflections

The internal reflections (IR) stage of the algorithm are 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, the IR stage has a parallel complexity of

$$\mathrm{O}\left(2n^2\left(\frac{m}{p} - \frac{n}{3}\right)\right). \tag{6}$$

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

		S	t	e	p				W	o	r	k		
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. 5. 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 Givens rotation at that stage. The rows of block 1 are used as the pivots (rows 1–7), and the remaining rows are assigned as shown for annihilation.

3.4.2. Complexity of balanced rotations

Analysis of the 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 3.3, the BR stage must perform

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

Table 3
Summary of work *for one block* during BR stage[a]

Processor	Steps							Total
	1	2	3	4	5	6	7	
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

[a] 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%.

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 [5]

$$\sum_{i=n-(m/p)+2}^{n-j} (6i + 6), \tag{7}$$

where j is the column index of the leftmost 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

$$(p-1)\left(\sum_{j=0}^{(m/p)-2}\left(\sum_{i=n-(m/p)+2}^{n-j}(6i+6)\right)\right), \tag{8}$$

which has one possible closed form of [6]

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

When we use the work assignment scheme described in Section 3.3, 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

$$\text{blocksize} = \lfloor m/p - 1 \rfloor. \tag{10}$$

[5] 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.

[6] Simplification of the above summation was done using the software package Mathematica.

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 Fig. 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 assignment is done cyclicly as in Fig. 3 with each cycle containing $2p$ rows. Each of the p blocks will have

$$\text{cycles} = \left\lfloor \frac{\text{blocksize}}{2p} \right\rfloor \tag{11}$$

cycles. After assignment of the cycles, there are

$$\text{leftovers} = \text{blocksize} - 2p(\text{cycles}) \tag{12}$$

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 $(\text{blocksize} - p)$: the work is balanced perfectly (see Fig. 3),
2. blocksize is not evenly divisible by p, and the leftmost non-zero column of blocks 2 to p is between 1 and $(\text{blocksize} - p)$: the maximum work is done by processor $(p + 1) - (\text{leftovers} \mod p)$ (see Fig. 5, steps 1–3), 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 during the last four 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 Eq. (9) and normalizing with the number of processors gives a general form for expected parallel efficiency. Boleng [1] includes several examples which illustrate the high level of load balancing achievable as m and n grow relative to p. It also presents several examples of the total expected efficiency for various problem sizes and number of processors. As long as m and n remain large relative to the number of processors p, very high efficiencies are expected (>90%).

4. Results

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 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 noted 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 were used. Table 4 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, both T_{best} and $T_{1/1}$ are used.

Table 4
Comparison of sequential performance for the parallel algorithm and the best known sequential algorithm[a]

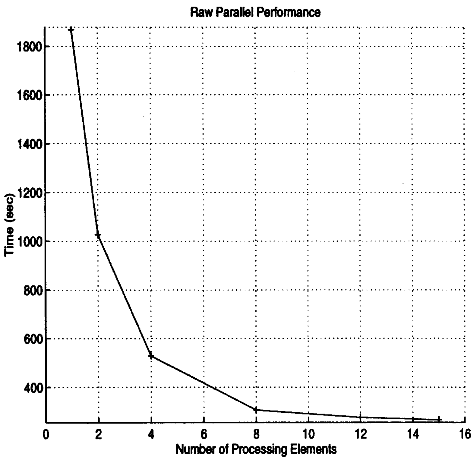
Problem size	Parallel QR algorithm	Householder algorithm
1024×1024	$T_{1/1} = 67.95$	$T_{\text{best}} = 67.64$
1024×1024	$T_{1/16} = 44.91$	
1024×1024	$T_{1/32} = 46.96$	
8000×1600	$T_{1/1} = 1859.4$	$T_{\text{best}} = 1868.97$
8000×1600	$T_{1/16} = 1716.47$	
8000×1600	$T_{1/32} = 1365.96$	

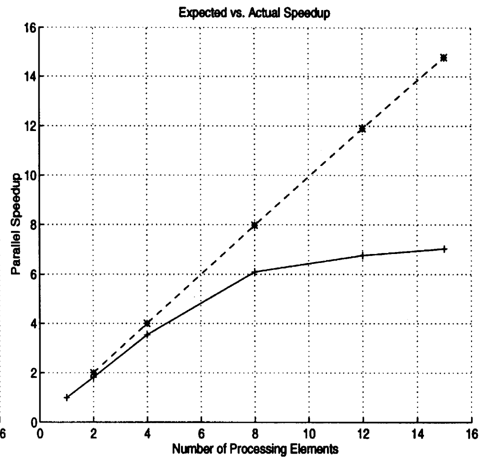
[a] Execution times reported in seconds and measured on one R10000 processor of the SGI Power Challenge.

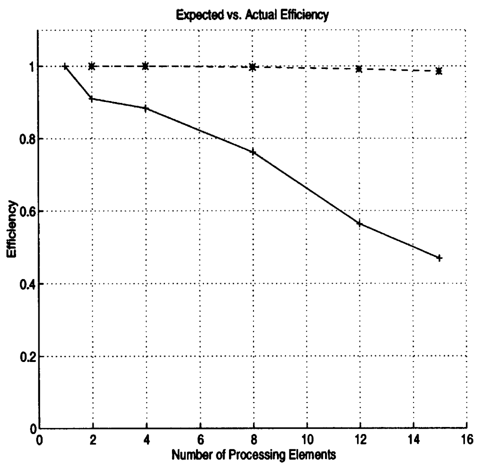
4.1. Performance on the Power Challenge and the origin 2000

SGI Power Challenge. The algorithm was implemented on the shared memory architecture of the SGI Power Challenge for different problem sizes and different number of processors. Since the algorithm was designed with the shared memory model in mind, we expected good performance on the Power Challenge architecture. We also tested the difference in performance between the machine being run in dedicated versus shared mode. Detailed results can be found in [1].

Here we present the results on a representative problem size on a machine shared with other users (Fig. 6), and when run in dedicated mode (Fig. 7). Finally, we present a summary of the performance results across a number of problem sizes



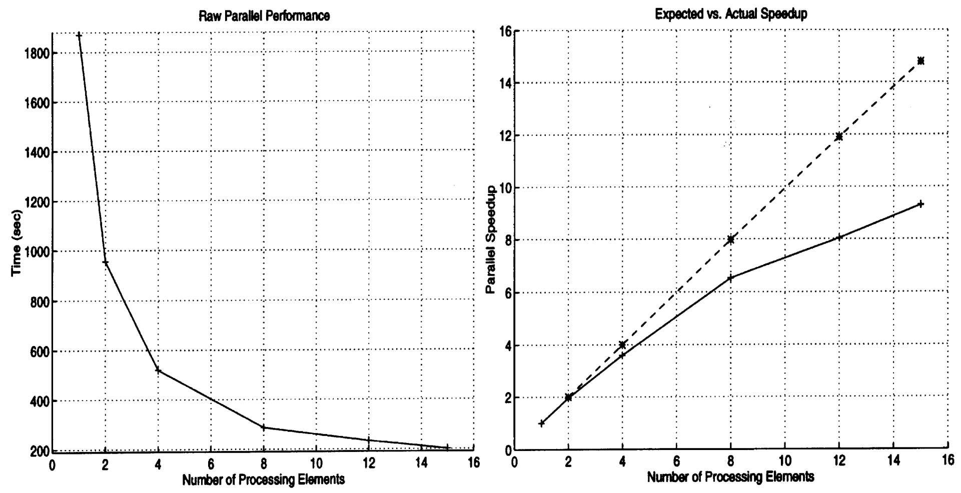


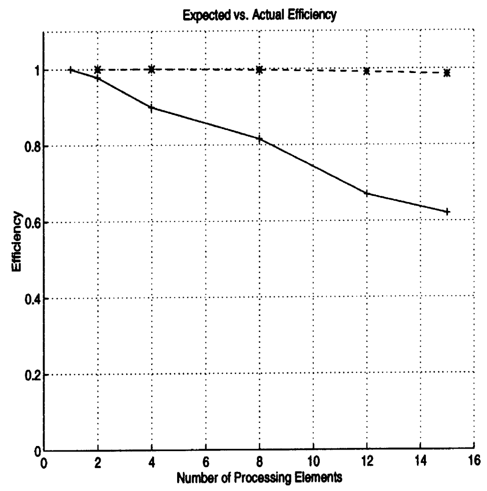


8000 × 1600	p=1	p=2	p=4
Householder Parallel QR	1868.97 1859.4	1026.4	528.35
$\text{Speedup}_{\text{expected}}$ $\text{Efficiency}_{\text{expected}}$		1.99 99.9%	3.99 99.9%
$\text{Speedup}_{\text{best}}$ $\text{Efficiency}_{\text{best}}$		1.82 91.0%	3.54 88.4%
$\text{Speedup}_{1/1}$ $\text{Efficiency}_{1/1}$		1.81 90.6%	3.52 88.0%

	p=8	p=12	p=15
Parallel QR	306.6	276.2	265.72
$\text{Speedup}_{\text{expected}}$ $\text{Efficiency}_{\text{expected}}$	7.98 99.7%	11.90 99.2%	14.79 98.6%
$\text{Speedup}_{\text{best}}$ $\text{Efficiency}_{\text{best}}$	6.10 76.2%	6.77 56.4%	7.03 46.7%
$\text{Speedup}_{1/1}$ $\text{Efficiency}_{1/1}$	6.06 75.8%	6.73 56.1%	7.00 46.7%

Fig. 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.



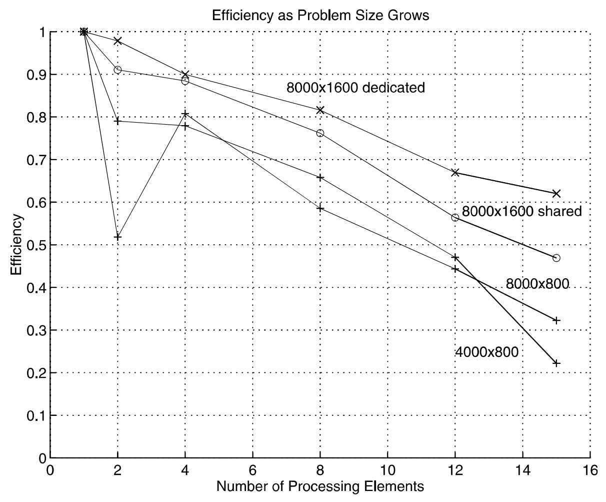


8000 × 1600	p=1	p=2	p=4
Householder	1868.97		
Parallel QR	1859.4	955.34	519.35
Speedup$_{\text{expected}}$		1.99	3.99
Efficiency$_{\text{expected}}$		99.9%	99.9%
Speedup$_{\text{best}}$		1.96	3.60
Efficiency$_{\text{best}}$		97.8%	90.0%
Speedup$_{1/1}$		1.95	3.58
Efficiency$_{1/1}$		97.3%	89.5%
	p=8	p=12	p=15
Parallel QR	286.36	232.62	200.92
Speedup$_{\text{expected}}$	7.98	11.90	14.79
Efficiency$_{\text{expected}}$	99.7%	99.2%	98.6%
Speedup$_{\text{best}}$	6.53	8.03	9.30
Efficiency$_{\text{best}}$	81.6%	67.0%	62.0%
Speedup$_{1/1}$	6.49	7.99	9.25
Efficiency$_{1/1}$	81.2%	66.6%	61.7%

Fig. 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. 8). The results obtained show good parallel efficiency on the Power Challenge up to four processors, and speedup increases up to 12 processors. However, efficiency begins to drop off quickly for the eight and 12 processor cases as more processors have to spend idle time.

Two comments can be made about the shapes of the graphs. First, actual performance differs substantially from the expected performance. This occurs primarily because the prediction model derived in Section 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, synchroniza-



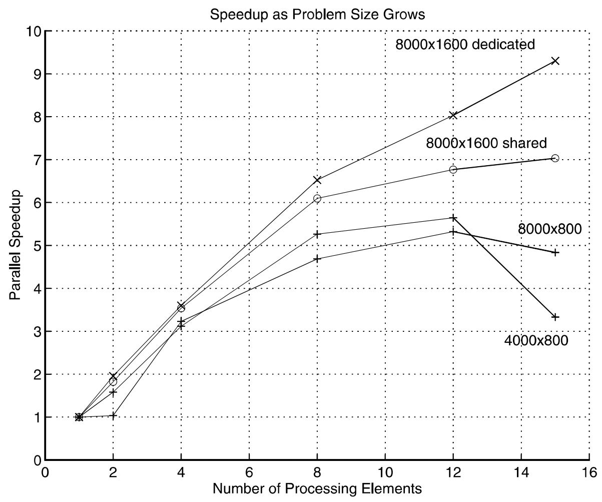


Fig. 8. Parallel performance comparison on the SGI Power Challenge (R10000) as problem size grows. Matrix sizes are shown next to their representative line.

tion 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 number of processors.

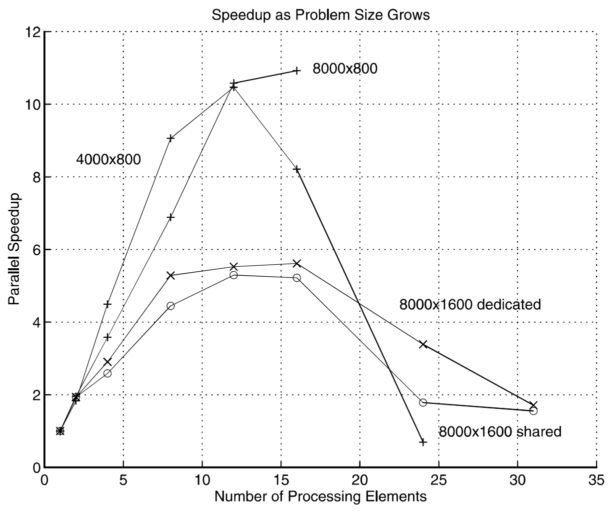
An important comparison can be made from Figs. 6 and 7. These results present the performance for the same matrix size and processor numbers in shared vs. dedicated mode. In shared mode, the program is time shared with other parallel jobs, while in dedicated mode, the program is executed from start to 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 was apparent on the Origin 2000 machine as well. This performance difference is 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 Figs. 6 and 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 smaller problems where speedup peaked and then began to decrease before the processor limit. Efficiency does steadily decline as in the smaller problems, but the decrease is not as soon or as sharp.

Finally, examine Fig. 8 which plots the parallel efficiency and speedup for many processors as the problem size changes. 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.

SGI/Cray Origin 2000. Although the algorithm was developed for shared memory machines, we decided to test its performance on a distributed shared memory system. Therefore, the code developed for the Power Challenge was ported to the Origin 2000 and was tested for various problem sizes and number of processors. Detailed results are presented in [1]. Here, we present a summary of the speedup and efficiency results in Fig. 9. 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 is to be expected since the fine grained partitioning incurs an extra communication overhead on the distributed memory system.

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. 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. The current algorithm implementation makes no attempt to schedule memory accesses to reduce communication contention. 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 and varies with each run based on system load, problem size, processors used, etc. No method was found that consistently addressed all the mysteries of the operating system's thread and memory management. More careful scheduling and memory accesses during the BR stage



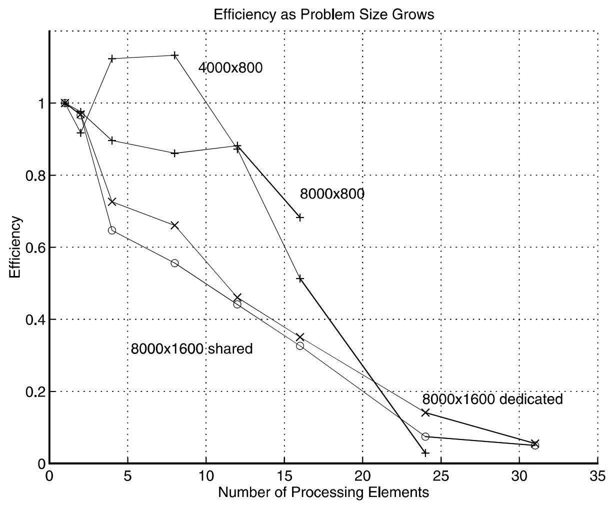


Fig. 9. Parallel performance comparison on the SGI/Cray Origin 2000 as problem size grows. Matrix sizes are shown next to their representative line.

must be done on this machine. These results also show that although the Origin 2000 allows an easy port of code from the Power Challenge, a simple port is not always sufficient because of the differences in the underlying memory organizations.

5. Conclusions

This paper introduces a new parallel QR decomposition algorithm. The balancing of work across processors is done at a finer grain than earlier attempts, which is more appropriate for shared memory parallel architectures utilizing a modest number of commodity processors. This makes much higher efficiencies possible on current shared memory multiprocessors than previous algorithms. The use of a hybrid elimination technique in support of another of the goals (maximizing computation between communication) also demonstrates the pragmatic tradeoff necessary to create higher performance than earlier demonstrated at the possible cost of scalability (especially for systems that implement a physically distributed memory). As such, the new algorithm is specifically designed for parallel architectures with a modest number of processing elements. Our algorithm can achieve practically 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.

The fine-grained load balancing technique does introduce a significant hurdle. The cyclic one-dimensional assignment of the rows to processors during the Balance Rotations stage has been done without regard to communication contention. Results on the Origin 2000 underline that this must be considered for good performance to be possible on distributed memory machines and networks of workstations. A detailed analysis of the pattern of communication in the BR stage and structuring it according to the interconnection network of the machine is the emphasis for future work.

References

[1] J. Boleng, Parallel qr decomposition for electromagnetic scattering problems, Master's thesis, Department of MCS, Colorado School of Mines, July 1997.

[2] J. Choi, J. Dongarra, S. Ostrouchov, A. Petitet, D.W. Walker, R.C. Whaley, The design and implementation of the ScaLAPACK LU, QR, and Cholesky factorization routines, Scient. Programm. 5 (1996) 173–184.

[3] J.A. Clarke, Emulating shared memory to simplify distributed-memory programming, IEEE Comput. Sci. Eng. 4 (1) (1997) 55–62.

[4] J. Cordsen, T. Garnatz, M. Sander, A. Gerischer, M.D. Gubitoso, U. Haack, W. Schröder-Preikschat, Vote for peace: Implementation and performance of a parallel operating system, IEEE Concurrency 5 (2) (1997) 16–27.

[5] M. Cosnard, Y. Robert, Complexité de la factorisation QR en parallèle, C.R. Acad. Sci. Paris 297A (1983) 549–552.

[6] M. Cosnard, E.M. Daoudi, Optimal algorithms for parallel Givens factorization on a coarse grained PRAM, J. Assoc. Comput. Mach. 41 (2) (1994) 399–421.

[7] M. Cosnard, D. Trystram, Parallel Algorithms and Architectures, PWS Publishing Company, Boston, MA, 1995.

[8] J.J. Dongarra, I.S. Duff, D.C. Sorensen, H.A. van der Vorst, Solving Linear Systems on Vector and Shared Memory Computers, SIAM, Philadelphia, 1991.

[9] K.A. Gallivan, M.T. Heath, Esmond Ng, J.M. Ortega, B.W. Peyton, R.J. Plemmons, C.H. Romine, A.H. Sameh, R.G. Voigt, Parallel Algorithms for Matrix Computations, SIAM, Philadelphia, 1990.

[10] G.H. Golub, C.F. Van Loan, Matrix Computations, third ed., The John Hopkins University Press, Baltimore, MD, 1996.

[11] R.E. Lord, J.S. Kowalik, S.P. Kumar, Solving linear algebraic equations on an MIMD computer, J. Assoc. Comput. Mach. 30 (1) (1983) 103–117.

[12] J.J. Modi, M.R.B. Clarke, An alternative Givens ordering, Numer. Math. 43 (1984) 83–90.
[13] A. Pothen, P. Raghavan, Distributed orthogonal factorization: Givens and Householder algorithms, SIAM J. Scient. Stat. Comput. 10 (6) (1989) 1113–1134.
[14] A.H. Sameh, D.J. Kuck, On stable parallel linear system solvers, J. Assoc. Comput. Mach. 25 (1) (1978) 81–91.