A Highly Parallel Algorithm for the Reduction of a Nonsymmetric Matrix to Block Upper-Hessenberg Form

Michael W. Berry[†] Jack J. Dongarra [*†] Youngbae Kim[†]

February 6, 1994

Abstract

In this paper, we present an algorithm for the reduction to block upper-Hessenberg form which can be used to solve the nonsymmetric eigenvalue problem on message-passing multicomputers. On such multicomputers, a nonsymmetric matrix can be distributed across processing nodes configured into a network of two-dimensional mesh processor array using a block-scattered decomposition. Based on the matrix partitioning and mapping, the algorithm employs both Householder reflectors and Givens rotations within each reduction step.

We analyze the arithmetic and communication complexities and describe the implementation details of the algorithm on message-passing multicomputers. We discuss two different implementations—synchronous and asynchronous—and present performance results on the Intel iPSC/860 and DELTA. We conclude with an evaluation of the algorithm's communication cost, and suggest areas for further improvement.

1 Introduction

We present an algorithm for reducing an $n \times n$ nonsymmetric matrix to block upper-Hessenberg form in preparation for solving the nonsymmetric eigenvalue problem using orthogonal transformations on message-passing multicomputers. On sequential machines, similarity transformations are typically used to reduce an $n \times n$ nonsymmetric matrix A to upper-Hessenberg form so that the QR algorithm can be applied to the $n \times n$ upper-Hessenberg matrix H and thereby compute the eigenvalues of A. Two types of similarity transformations are available for the reduction. The first type uses orthogonal or unitary factorizations based on reflectors or rotations, and the second type uses nonorthogonal or nonunitary factorization based on Gaussian elimination (i.e. elementary reflectors). Although the first type costs twice as much as the second type, orthogonal or unitary transformations are more commonly used because they guarantee stability.

The reduction of an $n \times n$ real matrix to upper-Hessenberg form using orthogonal similarity transformations is formalized as

$$H = Q^T A Q = Q_{n-2}^T \dots Q_2^T Q_1^T A Q_1 Q_2 \dots Q_{n-2},$$

[*]Mathematical Sciences Section, Oak Ridge National Laboratory, Oak Ridge, TN 37831-8083

[†]Department of Computer Science, University of Tennessee, Knoxville, TN 37996. This work is funded in part by the Applied Mathematical Sciences subprogram of the Office of Energy Research, U.S. Department of Energy, under Contract DE-AC05-84OR21400, by the Defense Advanced Research Projects Agency under contract DAAL03-91-C-0047, administered by the Army Research Office, and by the National Science Foundation Science and Technology Center Cooperative Agreement No. CCR-8809615.

where A is an $n \times n$ real nonsymmetric matrix, Q is an orthogonal matrix and $Q^T Q = I$, and H is an upper-Hessenberg matrix. We note that Q is a product of the $n-2$ orthogonal matrices $Q_1, Q_2, \ldots, Q_{n-2}$ computed through the $n-2$ steps of an orthogonal factorization.

For a block version of the reduction to block upper-Hessenberg form on sequential machines, the $n \times n$ matrix A can be partitioned into $N \times N$ blocks with equal block size of b

$$A = \left(\begin{array}{c|c} A_{11} & A_{12} \\ \hline A_{21} & A_{22} \end{array} \right),$$

where $\boldsymbol{A}_{11} \in \boldsymbol{R}^{b \times b}$, $\boldsymbol{A}_{12} \in \boldsymbol{R}^{b \times (n-b)}$, $\boldsymbol{A}_{21} \in \boldsymbol{R}^{(n-b) \times b}$, $\boldsymbol{A}_{22} \in \boldsymbol{R}^{(n-b) \times (n-b)}$, and $n = bN$.

Suppose that we have computed the QR factorization $A_{21} = \tilde{Q}_1 R_1$ and that $\tilde{Q}_1 = I + W_1 Y_1^T$, an orthogonal matrix of the WY form, where W and Y are $(n-b) \times n$ matrices [9]. The block representation of the reduction algorithm is then given by

$$Q_1^T A Q_1 = \left(\begin{array}{c|c} A_{11} & A_{12}\tilde{Q}_1 \\ \hline R_1 & \tilde{Q}_1^T A_{22} \tilde{Q}_1, \end{array} \right), \ where \ \ Q_1 = \left(\begin{array}{c|c} I_b & 0 \\ \hline 0 & \tilde{Q}_1 \end{array} \right),$$

and I_b is an $b \times b$ identity matrix.

Finally, after $N-1$ block reduction steps, we obtain a block upper-Hessenberg matrix, H of the following form

$$H = Q^T A Q = \begin{pmatrix} H_{11} & H_{12} & \ldots & \ldots & H_{1N} \\ H_{21} & H_{22} & \ldots & \ldots & H_{2N} \\ 0 & H_{32} & \ddots & \ldots & H_{3N} \\ \vdots & \vdots & \ddots & \ddots & \vdots \\ 0 & 0 & \ldots & H_{N,N-1} & H_{NN} \end{pmatrix},$$

where each H_{ij} is a $b \times b$ dense matrix and $Q = Q_1 \ldots Q_{N-1}$ with each Q_k in WY form. Note that $H_{i+1,i}$, the subdiagonal blocks of H are upper triangular so that the matrix H has lower bandwidth b. In computing the nonsymmetric eigensystem associated with A, the original matrix A must also be postmultiplied by the orthogonal matrix, Q_k.

Two schemes are available for orthogonal factorizations: Householder reflectors and Givens rotations. Householder reflectors involve half the number of arithmetic operations as do Givens rotations [9]. On traditional sequential machines, therefore, where the cost of a floating-point operation dominates the cost of a memory reference, Householder-based algorithms are generally preferred. Several sequential algorithms have been developed for the reduction of a general dense matrix to upper-Hessenberg form. For example, one can use `DGEHRD` from `LAPACK` [5]. Since block methods on high-performance computers improve processing efficiency by grouping memory references, `DGEHRD` is implemented using block Householder reflectors.

On a parallel machine where memory access times may dominate flop times, Givens algorithms may be preferable. In [4], for example, it was shown that on the Denelcor HEP (a shared-memory multiprocessor), Givens algorithms were twice as fast as Householder algorithms (see also [2]).

Dongarra and Ostrouchov have also developed a parallel algorithm for reducing a matrix to upper-Hessenberg form on distributed-memory multiprocessors [3]; the implementation in `LAPACK` is discussed by Dongarra and van de Geijn in [7]. In 1989, Pothen and Raghavan [10] showed that a hybrid algorithm could take advantage of both low-cost arithmetic operations of Householder algorithm and low-cost communication costs of Givens algorithms.

In this paper, we design and implement a parallel algorithm for message-passing multicomputers, in which processors are easily reconfigured into a $p \times q$ mesh-connected processor array. Our approach achieves processing efficiency by partitioning a general dense matrix into block submatrices and distributing the blocks among processors in block-scattered fashion to exploit the nature of row-oriented and column-oriented computations. It then applies QR factorizations based on Householder reflectors and Givens rotations for all off-diagonal blocks at each reduction step.

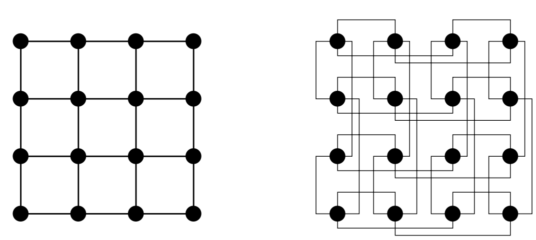
The remainder of this paper is organized as follows. In Section 2, we discuss our parallel algorithm for the reduction to block upper-Hessenberg form. We present a communication model for message-passing multicomputers, and discuss matrix partitioning and mapping of the data using such a model. In Section 3, two different implementations of our parallel algorithm on some message-passing multicomputers are presented. Section 4 discusses the arithmetic and communication complexities of the algorithm. Section 5 presents the performance results of experiments with two implementations on the Intel iPSC/860 hypercube and the Intel Touchstone DELTA mesh multicomputers. Finally, Sections 6 and 7 discuss the communications complexities and suggest areas for enhancing the new parallel algorithm.

2 The Parallel Algorithm

In this section, we describe a general communication topology and the details of our parallel algorithm. We also show how the original matrix is partitioned and mapped onto processors of message-passing multicomputers. The specific details of the algorithm are discussed in Section 2.3.

2.1 Communication Topology

We assume that processing nodes of message-passing multicomputers can easily be reconfigured into a network of a two-dimensional mesh, or a processor grid. Assume the processor grid has p processors in each row and q processors in each column, and that neighboring processors in the grid may or may not be physically connected. For example, in hypercube architectures, a subcube can be reconfigured into a processor grid, but it has different connections between the neighboring processors. Figure 1(a) shows a typical processor grid as a mesh-connected processor array, and Figure 1(b) shows a 2^4 subcube reconfigured into a 4×4 mesh-connected processor array. Note



(a) A mesh-connected processor grid (b) A cube reconfigured into a 4 x 4 mesh

Figure 1: 4×4 mesh-connected processor arrays

that a "•" represents a processing node and that each physical connection is represented by a solid line.

In our algorithm, each intermediate transformation matrix computed to reduce a block must be broadcast to all processors on the corresponding row and column to update the remaining blocks of the original matrix at each reduction step. This broadcasting strategy totally depends upon the network of message-passing multicomputers, and therefore has a great effect on the performance of our algorithm.

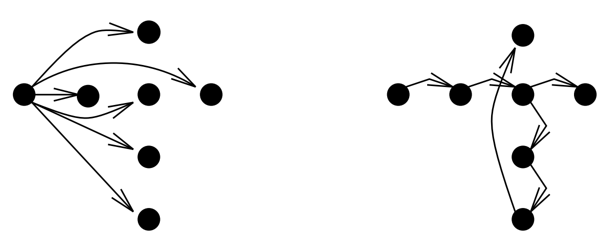
Two general message-passing methods can be used to update the rest of the matrix A: one-to-one broadcasting in pipelined fashion and one-to-many broadcasting. Figure 2 shows those broadcasting methods. For convenience, we use $P_{i,j}$ to denote the processor in the (i, j) position of the processor grid, and $P_{i,*}$ and $P_{*,j}$ to denote the processors assigned to the ith row and jth column of the grid. In one-to-one broadcasting, the processor being reduced first sends a message to the next row processor, $P_{i,j+1}$ and then sends it to the processor $P_{i,j+2}$, etc. As soon as the diagonal processor, $P_{i,i}$, receives a message, it sends the message to both the next row and column processors. In one-to-many broadcasting, the processor, $P_{i,j}$ being reduced broadcasts a message to the corresponding row and column processors, $P_{i,*}$ and $P_{*,j}$ simultaneously. Further details of such message-passing methods will be discussed in the next section.

2.2 Matrix Partitioning and Mapping

Since message-passing multicomputers typically have no globally-shared memory, the data must be distributed among the processing nodes in some way. Typically, data is distributed by rows if the computation is row-oriented, and by columns if it is column-oriented. For the reduction algorithms using similarity transformations, the matrix A must be updated by both premultiplying and postmultiplying an intermediate orthogonal matrix at each reduction step. Thus, the computations of the algorithm are both row-oriented and column-oriented, and hence parallelism of our algorithm stems from the computations along both row and column with processors configured into a processor grid.

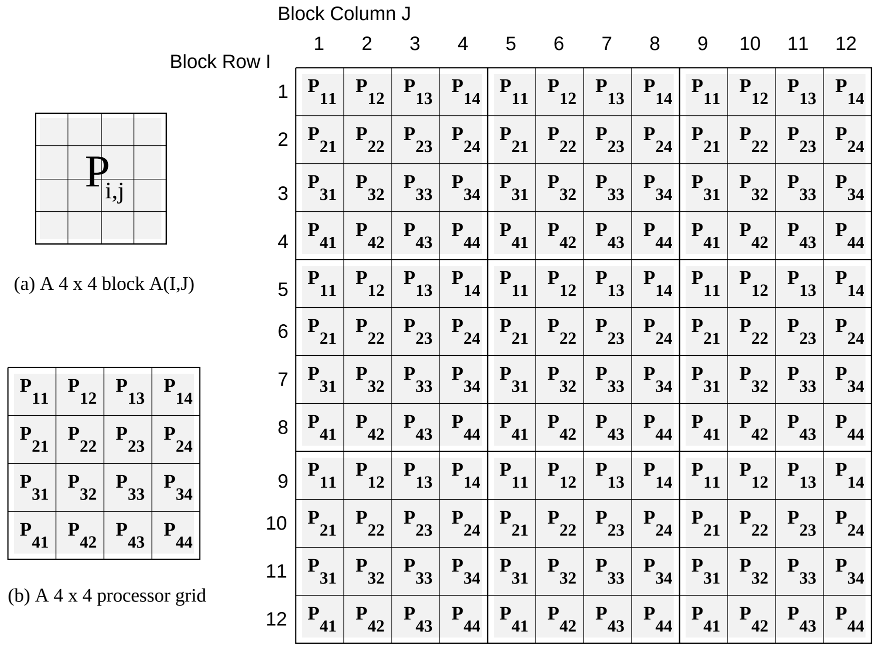
In our algorithm, an $n \times n$ matrix is distributed into processors in block-scattered fashion, i.e. via $N \times N$ blocks of equal size b, where $N = n/b$. Figure 3 illustrates the partitioning of a 48×48 matrix into 12×12 block submatrices with equal block size of 4 so that the partitioned block submatrices are mapped onto a 4×4 processor grid.

In summary, the matrix mapping strategy is essentially a wrapping of the rows of the $n \times n$ matrix A around the p processor rows, namely, $P_{1,*}$, $P_{2,*}$, ..., $P_{p,*}$ or a wrapping of the columns of A around the q processor columns, namely, $P_{*,1}$, $P_{*,2}$, ..., $P_{*,q}$. Hence, if the nodes of a $p \times q$



(a) One-to-many broadcasting (b) One-to-one broadcasting using a unidirectional ring

Figure 2: The communication patterns for broadcasting on a 4×4 processor grid



(a) A 4 x 4 block A(I,J)

(b) A 4 x 4 processor grid

(c) A 12 x 12 block matrix mapped onto a 4 x 4 processor grid.

Figure 3: Matrix Partitioning and Mapping

processor grid are indexed by (i, j), where $1 \leq i \leq p$, and $1 \leq j \leq q$, then the blocks can be wrapped onto this grid by assigning $A_{I,J}$ to node $(I \bmod p, J \bmod q)$.

2.3 The Algorithm

Based on the matrix partitioning, mapping, and communication model described in the preceding sections, the algorithm applies both Householder reflectors and Givens rotations to reduce all off-diagonal blocks at each step. The algorithm consists of two different phases: a *QR factorization phase* and a *Givens phase*. While the QR factorization phase is based on Householder reflectors to reduce each block in the current block column to upper-triangular form, the Givens phase uses Givens rotations to annihilate all elements of each triangularized block except for the main subdiagonal block.

Suppose that the jth block column is being reduced and i is the row index of a block submatrix in block column j. The processor that owns the block submatrix A_{ij} has a row index of $i \bmod p$ and a column index of $j \bmod q$. In the QR factorization phase, Householder reflectors are used to compute Q_{ij} such that $A_{ij} = Q_{ij}R_{ij}$. The off-diagonal blocks in the current block column are reduced to upper triangular form. This phase can be carried out locally by each processor without any global communication. The rest of the original matrix A is then updated by pre- and postmultiplication by Q_{ij}. Once each processor computes an orthogonal matrix Q_{ij} for each off-diagonal block using Householder reflectors, it broadcasts Q_{ij} to all processors on its processor row and on its corresponding processor column i. Figure 4 illustrates the subsequent matrix triangularization. Figure 5 depicts the message flows in this phase.

Throughout the Givens phase, all off-diagonal blocks except for the main subdiagonal block are annihilated. This phase applies a sequence of Givens rotations between two blocks for eliminating all elements of one block with the other block as a pivot block. Figure 6 illustrates the process

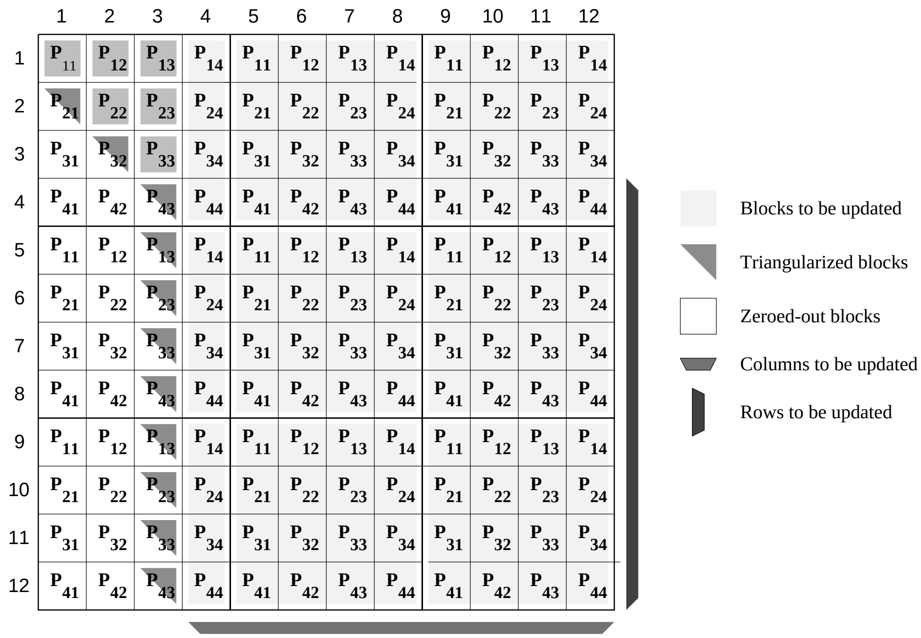
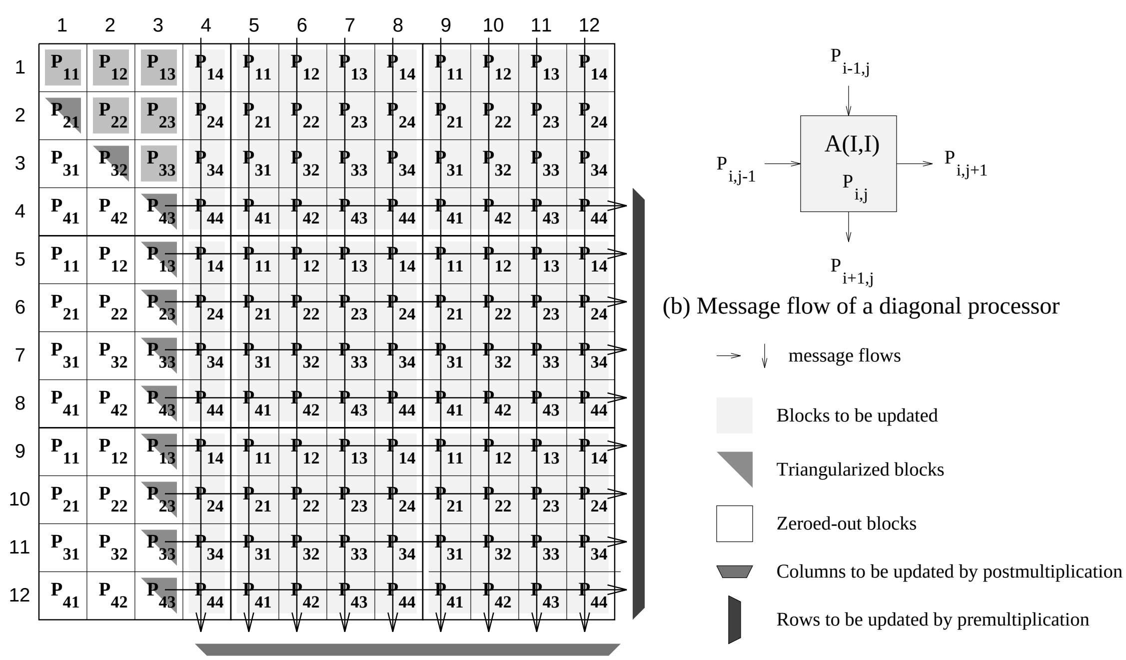


Figure 4: The matrix triangularized in the QR factorization phase



(a) Message Flows after triangularized in the QR factorization phase.

Figure 5: Message Flows after triangularized by Householder algorithms

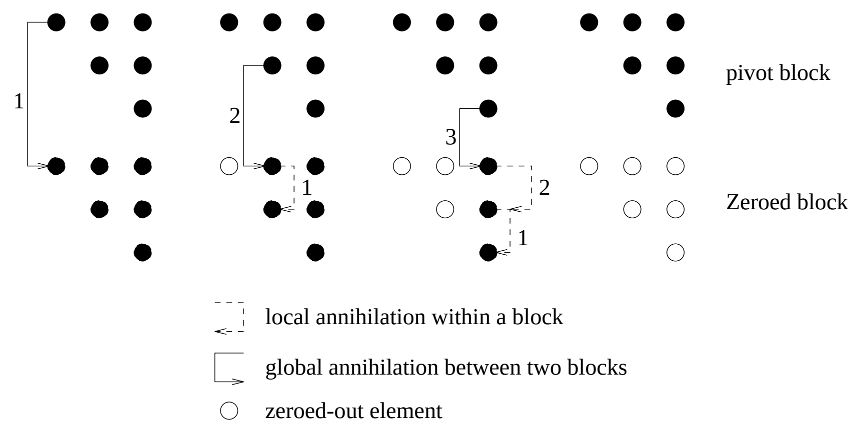


Figure 6: A sequence of Givens rotations between two blocks

of computing a sequence of Givens rotations between two 3×3 blocks. The reduction of the jth current block column to block upper-Hessenberg form is composed of two steps: *local annihilation* and *global annihilation*.

In Step 1 of the Givens phase (local annihilation), each processor in the current column that owns all off-diagonal blocks annihilates all elements of its local off-diagonal blocks with its lowest-numbered local block as a pivot block. Once a sequence of Givens rotations for each of two blocks is computed, it must be broadcast to all processors on its row and corresponding column for updating the rest of matrix. Hence, after this step, each processor has left only one local off-diagonal block with lowest-numbered index (see Figure 7).

In Step 2 of the Givens phase (global annihilation), the lowest-numbered block of each processor on the current column is paired with that of other processor, and then one of two global blocks is annihilated by computing a sequence of Givens rotations with other block as a pivot block. This annihilation step requires all pairs of processors to participate in computing a sequence of Givens rotations between two paired blocks regardless of whether any computational work remains for a particular processor. This step requires communications between the paired processors for exchanging its whole row blocks. The step can be done recursively in $\log_2 p$ stages, where p is the number of processor rows (see Figure 8).

For convenience, we denote a "pivoting processor" as a processor that owns a pivot block and a "zeroing processor" as a processor that owns a block to be annihilated. The pivoting processor has a lower-numbered row block index than that of the zeroing processor; hence, only the pivoting processors at the previous stage participate in the next stage. At the beginning of each of $\log p$ stages, every processor is paired with the nearest neighboring processor that owns a nonzero block. Then, every processor exchanges its current local pivot block to the current paired processor. Both processors should compute the same sequence of Givens rotations to zero out one block. While one processor zeros out the received block with a pivot block as its own block, the other processor zeros out its own block with a pivot block as the received block. Between the two stages, both the pivoting and zeroing processors must broadcast a Givens sequence computed at the previous stage to the processors on its row and corresponding column of the processor grid. After $\log_2 p$ stages, only one triangular block (i.e., the main subdiagonal block) remains. Thus, the current block column is reduced to block upper-Hessenberg form. This process completes a reduction step of the algorithm and is repeated on the remaining unreduced block columns. A complete pseudocode of the algorithm is given in Figure 9.

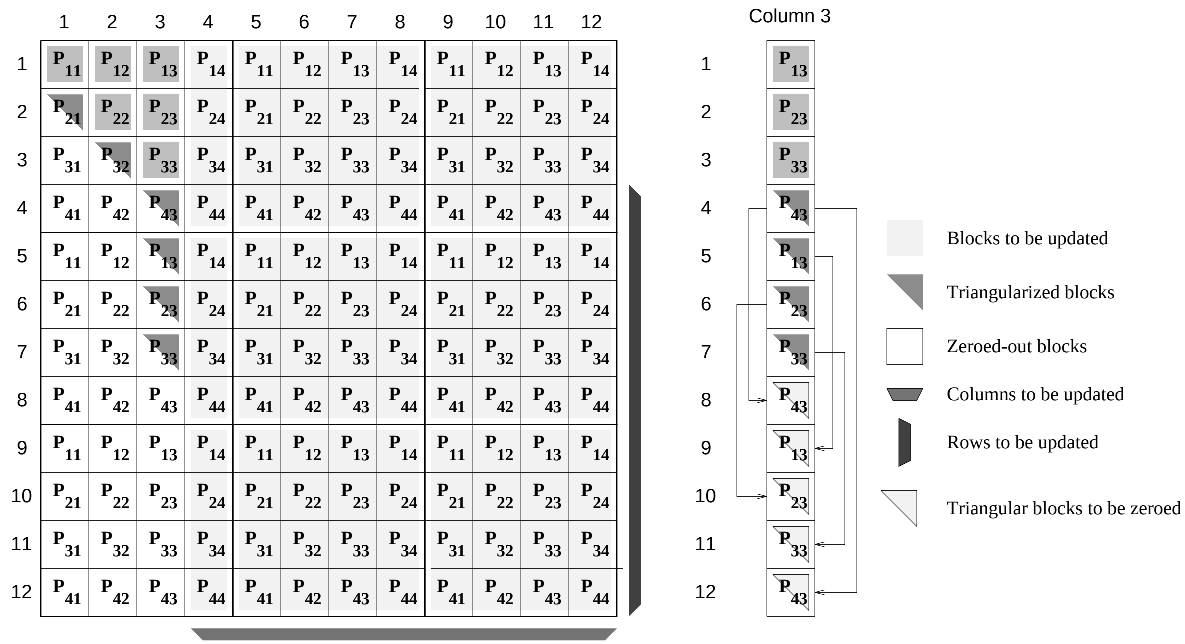


Figure 7: The matrix after the step I of Givens phase

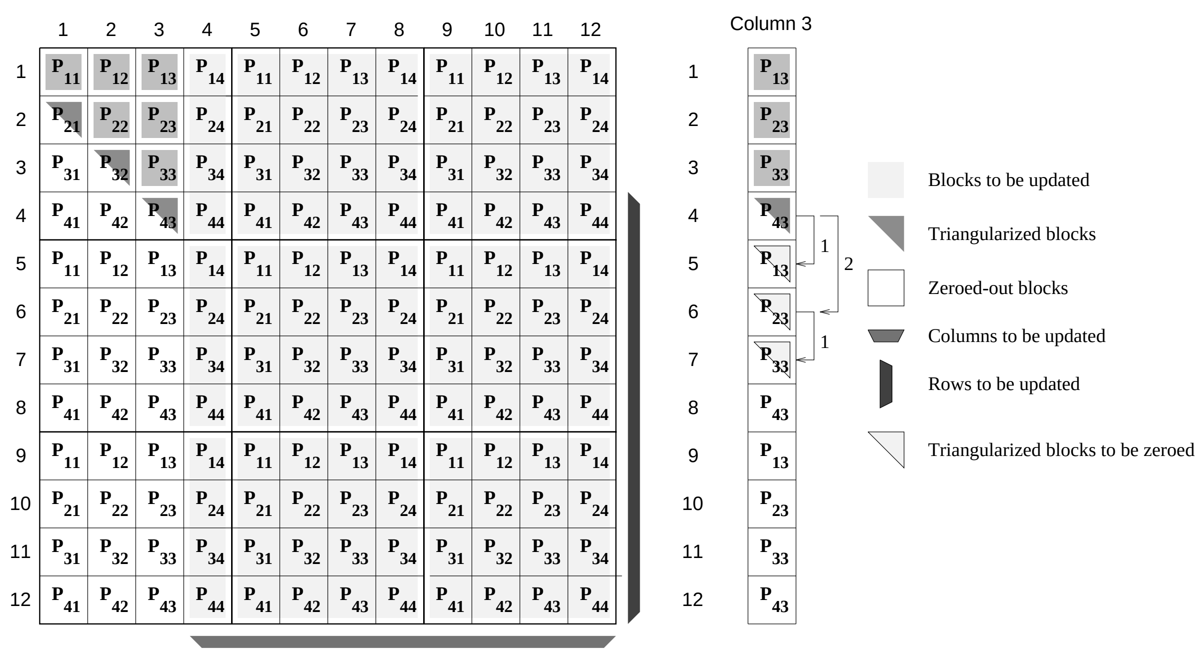


Figure 8: The matrix after the step II of Givens phase

Algorithm:

```
for each block col j = 1 to N do
  QRfact(j);  QR factorization phase
  Givens(j);  Givens phase
endfor
```

procedure: `QRfact(j)`

```
  for each local row block i = j+1 to N-1 do
    dgeqr(Q_ij);   Compute Q_ij
    bcast(Q_ij);   to processors in row i and col i
    rowupdate(Q_ij);   premultiplication, Q^T_{i,j} A_{i,*}
    if i is my col then
      colupdate(Q_ij);   postmultiplication, A_{*,i} Q_{i,j}
    endif
  endfor
```

procedure: `Givens(j)`

1. local annihilation

```
  for each local row block i = j+2 to N-1 do
    dzeroblk(G_ij);   Compute G_ij
    bcast(G_ij);   to processors in row i and col i
    rowupdate(G_ij);   premultiplication, G^T_{i,j} A_{i,*}
    if i is my col then
      colupdate(G_ij);   postmultiplication, A_{*,i} G_{i,j}
    endif
  endfor
```

2. global annihilation

```
  for k = 1 to log_2 p do
    if my top row block not zeroed out then
      exchange(all blocks in the top row)
      dzeroblk(G_ij);   Compute G_ij
      bcast(G_ij);   to processors in row i and col i
      rowupdate(G_ij);   premultiplication, G^T_{i,j} A_{i,*}
      if i is my col then
        if necessary then
          exchange (all blocks in the corresponding col)
        endif
        colupdate(G_ij);   postmultiplication, A_{*,i} G_{i,j}
      endif
    endif
  endfor
```

Figure 9: A Pseudocode of the Algorithm

3 Time Complexity Analysis

In this section, we analyze the time complexity of both the arithmetic operations and the communications of our parallel algorithm. We denote t_A as the arithmetic time complexity and t_C as the communication time complexity. For simplicity, we ignore the extra communication cost required for pairs of processors to exchange the whole block column for a column update in Step 2 of the Givens phase in cases where a processor grid is not square.

First, we compute the computation and communication costs of each phase on the jth block column at each reduction step. Then, we sum over all block columns to get the complexity of the phase for the entire algorithm. Suppose that the dimension of the block matrix is N with block size of b and that the processor grid is $p \times q$. Thus, a processor has $\lceil N/p \rceil$ row blocks and $\lceil N/q \rceil$ column blocks. We assume that p divides N and q divides N.

3.1 Arithmetic Complexity

In the QR factorization phase, a processor on the jth block column computes $(N-j)/p$ QR factorizations of the $b \times b$ off-diagonal block submatrices. Row processors that must premultiply A by these transformations have $(N-j)/p \times (N-j)/q$ blocks to update. Column processors that must postmultiply A have $N/p \times (N-j)/q$ blocks to update. For each block, the arithmetic cost for the orthogonal factorization is $\frac{4}{3}b^3$ floating-point operations (flops). The premultiplication of each block row requires $2b^3(N-j)/q$ flops, and the postmultiplication of each block column requires $2b^3N/p$ flops. While the arithmetic complexity for the premultiplication of each block row in the QR factorization phase is then multiplied by the number of block rows, $(N-j)/p$, the arithmetic complexity for the postmultiplication of each block column is multiplied by the number of block columns, $(N-j)/q$. The arithmetic complexity summed over all the block columns is then given by

$$t_A(QR) = \sum_{j=1}^{N-1} \left(\frac{N-j}{p}\right)\left(\frac{4b^3}{3} + 2b^3\frac{N-j}{q} + 2b^3\frac{N}{q}\right).$$

In the Givens phase, the arithmetic cost results from the two different steps, local and global annihilation. The arithmetic cost of local annihilation on the jth block column is

$$t_A(I) = \left(\frac{N-j}{p}\right)\left(\frac{b(b+1)}{2}\right)\left(6 + 6b\frac{N-j}{q} + 6b\frac{N}{p}\right).$$

While the arithmetic cost for global annihilation in $log_2 p$ stages is

$$t_A(II) = (\log_2 p)\left(\frac{b(b+1)}{2}\right)\left(6 + 6b\frac{N-j}{q} + 6b\frac{N}{p}\right).$$

Hence, the total arithmetic cost of the Givens phase is

$$t_A(Givens) = \sum_{j=1}^{N-2} (t_A(I) + t_A(II)) = \sum_{j=1}^{N-2} \left(\frac{N-j}{p} + \log_2 p\right)\left(\frac{b(b+1)}{2}\right)\left(6 + 6b\frac{N-j}{q} + 6b\frac{N}{p}\right).$$

3.2 Communication Complexity

The communication complexity of the QR factorization phase on the jth block column is dominated by the cost of broadcasting a transformation of each block matrix, A_{ij} to all processors on the row and column. Suppose that α is the communication startup cost and β is the transmission time per floating-point number. In the store-and-forward mechanism, the message transfer time between two adjacent processors can be represented as $\alpha+m\beta$, where m is the number of floating-point numbers in the messages. If a message is delivered h hops away, the message transfer time can be roughly estimated as $h(\alpha+m\beta)$. In the wormhole routing mechanism used in the Intel DELTA, the message transfer time is almost independent of the distance (number of hops) between processors [1]. In this case, if network contention is not considered, the message transfer time can be represented as $\alpha+m\beta$ regardless of the distance that a message has to traverse. In our analysis, for convenience, we assume that the message transfer time is $\alpha+m\beta$ for m floating-point numbers. The communication cost for the QR factorization phase is therefore given by

$$t_C(QR) = \sum_{j=1}^{N-1} \left(\frac{N-j}{p}\right) \left(\alpha + \beta\frac{b(b+1)}{2}\right).$$

Similarly, the communication costs for two different steps of the Givens phase on the jth block column are given by

$$t_C(I) = \left(\frac{N-j}{p}\right) (\alpha + b(b+1)\beta)\,,$$

and

$$t_C(II) = (\log_2 p)\,(\alpha + b(b+1)\beta) + (2\log_2 p)\left(\left(\alpha + b^2\frac{N-j}{q}\beta\right) + \left(\alpha + b^2\frac{N}{p}\beta\right)\right).$$

Hence, the total communication cost of the Givens phase is given by

$$t_C(Givens) = \sum_{j=1}^{N-2} (t_C(I) + t_C(II))$$

$$= \sum_{j=1}^{N-2}\left[\left(\frac{N-j}{p} + \log_2 p\right)(\alpha + b(b+1)\beta) + (2\log_2 p)\left(\left(\alpha + b^2\frac{N-j}{q}\beta\right) + \left(\alpha + b^2\frac{N}{p}\beta\right)\right)\right].$$

	Arithmetic Cost (t_A)	Communication Cost (t_C)
QR	$\frac{5}{3pq}N^3b^3 + (\frac{2}{3p} + \frac{2}{pq})N^2b^3 + O(n)$	$\frac{1}{2p}(N^2+4N)\alpha + \frac{1}{4p}(N^2b^2 + N^2b + O(n))\beta$
Givens I	$\frac{1}{pq}(N^3b^3 + N^3b^2) + \frac{3}{2p^2}(N^3b^3 + N^3b^2) + O(n^2)$	$\frac{1}{2p}N^2\alpha + \frac{1}{2p}\left(N^2b^2 + N^2b + O(n)\right)\beta$
Givens II	$\left(\frac{3}{2q} + \frac{3}{p}\right)(N^2b^3 + N^2b^2)\log_2 p + O(n\log_2 p)$	$5N\alpha\log_2 p + (\frac{1}{q} + \frac{2}{p})(N^2b^2 + O(n))\beta\log_2 p$

Table 1: Complexity Summary of the Algorithm

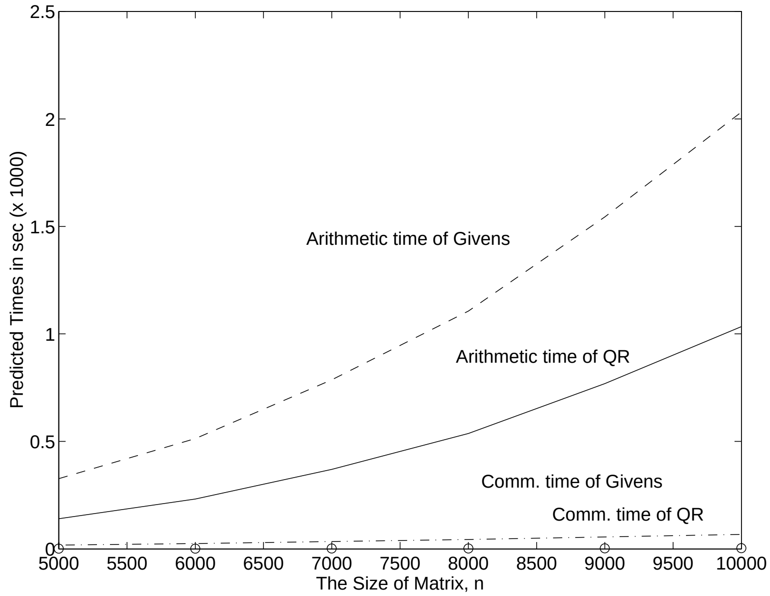


Figure 10: Predicted Timing Results of the Algorithm based on the Complexity Analysis.(For the Intel DELTA, block size, $b = 25$, processor grid $= 16 \times 16$, and $N = n/b$)

3.3 Complexity Summary

The complexity of the algorithm is summarized in Table 1. While the arithmetic cost of our algorithm remains unchanged once a processor grid is given, the communication cost dominates the total cost of the algorithm. Furthermore, the communication cost depends on how a message is broadcast to the row and column. For a rectangular processor grid, for example, the communication cost of the Givens phase is increased by the extra communications required for a column update in both Step 1 and Step 2. Based on our complexity analysis, Figure 10 shows the predicted timing results of the algorithm for the block size 25 on a 16×16 processor grid of the Intel DELTA as the matrix size is varied between $5,000$ and $10,000$.

4 Implementations

The target architectures for our algorithm are message-passing multicomputers such as the Intel iPSC/860, the Intel DELTA, the TMC CM-5, the Cray T3D, and the Ncube nCUBE. A message-passing multicomputer is a distributed-memory multiprocessor in which each memory module is physically associated with each processor. A point-to-point interprocessor communication network provides a mechanism for communication between processors.

Several factors affect the performance of the algorithm on such multicomputers. First, with block-scattered decomposition, the block size raises the issue of granularity in communication as well as in computation of the algorithm. Specifically, as the block size increases, the number of computations and the size of a message to transfer also increase, while the number of communications to broadcast a message decreases. Second, an important aspect in the mapping technique is load balancing. Our algorithm maintains load balancing statically by wrapped mapping on both the row and column. Third, and most important, is the waiting time when messages are broadcast along the row and column simultaneously. This waiting time is affected mostly by the broadcasting mechanisms that the machine's system software supplies.

In this section, we describe two different implementations of the algorithm on the Intel iPSC/860 and the DELTA: blocking and nonblocking. Blocking refers to the case when a processor receiving/sending a message from/to another processor must wait until the receiving/sending process is complete; nonblocking refers to the case when receiving/sending a message does not block the processor. These blocking and nonblocking mechanisms are often referred to as "synchronous" and "asynchronous" message passing, respectively. We have used the `BLAS` [8] and `LAPACK` routines for doing all basic block computations. No global combine operations among processors are required in the current implementations.

4.1 Communication routines on Intel machines

On Intel machines, a program running on one processor can send a message to and receive a message from another processor by calling two different sets of communication primitives provided by the operating system. The set for blocking calls comprises `csend()` and `crecv()`. The set for nonblocking calls comprises `isend()` and `irecv()`; the `msgdone()` call is used to check whether an asynchronous operation has completed.

4.2 Synchronous Implementation

For our synchronous implementation, we use the BLACS (Basic Linear Algebra Communication Subprograms) communication library [6]. The BLACS are a linear algebra communication library written using communication primitives of message-passing multicomputers. The library provides portable, efficient, and modular high-level routines for manipulating and communicating data structures that are distributed among the memories of message-passing multicomputers. Also, it embeds several different communication topologies and supports various broadcasting schemes for processing nodes of multicomputers logically configured as a rectangular mesh, or grid. [1]

Our synchronous implementation contains three different versions: `STREE`, `IRING`, and `DRING`. Each of which depends on how a message is broadcast in a linear array of row processors or column processors. `IRING`, `DRING`, and `STREE`, were implemented using broadcasting schemes that the BLACS supports: *increasing ring*, *decreasing ring*, and *minimum spanning tree*, respectively. Those broadcasting schemes require unidirectional ring topologies or linear arrays.

Seidel [11] has shown that in ring broadcasts such as increasing ring and decreasing ring, while the original sending processor is required to spend only the amount of time to send a message to the next processor, the last processor must waste p times as much as that of receiving and sending a message.

Minimum spanning tree broadcasting requires that p be an integer power of 2. This scheme takes $log_2 p$ times as much as that of receiving and sending a message. Thus, the minimum spanning tree broadcasting consumes less waiting time than ring broadcasting. In addition, we use force-typed messages for paired processors to exchange a whole block row with each other in the Givens Step 2.

4.3 Asynchronous Implementation

Synchronous implementations could block further computations that are independent of incoming messages. The goal of an asynchronous implementation is to reduce the time wasted in waiting for incoming messages.

[1] Note that the BLACS currently do not support nonblocking routines.

In the asynchronous implementation, a message is broadcast to all processors on the row and column simultaneously, without being send to intermediate nodes. This scheme has two principal disadvantages: it can cause network contentions, and the sending processor takes much more time than receiving processors. For this implementation, we must use the Intel nonblocking low-level communication primitives directly. Note that in order to maintain the ordering of computations for row updating and column updating, some data structures are used to manage queues for incoming messages received by each processor. Also, no force-typed messages are used.

5 Experimental Results

Our implementations were run and timed on both the Intel iPSC/860 and the DELTA. All test runs were performed in 64-bit arithmetic for random matrices with values between 0.0 and 1.0.

The asynchronous implementation performed better than the synchronous implementations on a rectangular processor grid, say 16×4. Figure 11 shows this performance improvement in cases where the number of processor rows is greater than that of processor columns. The explanation rests with the fact that for a rectangular processor grid $p \times q$, where $p > q$, more parallelism is achieved along row computations, and the waiting time of broadcasting messages along rows is less in the asynchronous implementation than in the synchronous implementations.

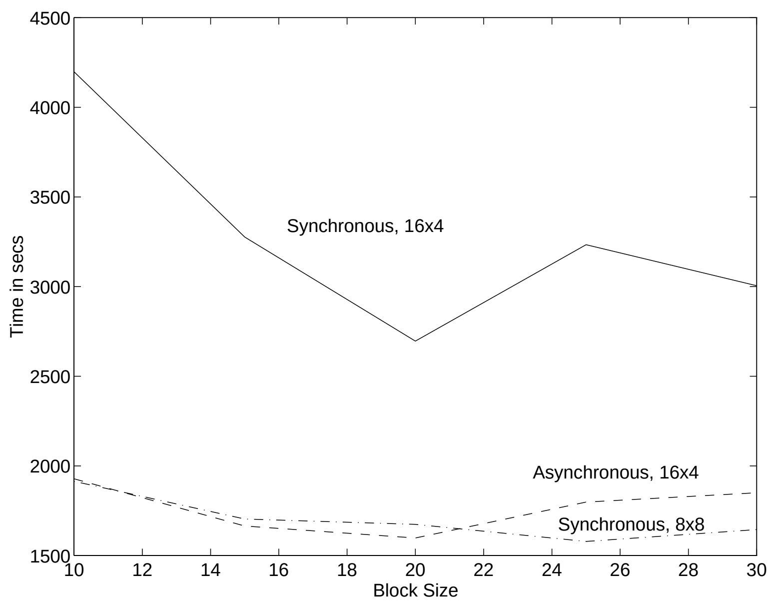


Figure 11: Performances on the iPSC/860 for matrices of order $5,000$ on different processor grids of 64 processing nodes.

The synchronous implementations perform better on a square processor grid than on a rectangular processor grid, due to significant amount of waiting time or extra communication spent in the Givens phase. Hence, all test runs were performed on the square processor grid of maximum size that the machines provide. On the iPSC/860, the processor grid 8×8 was used for a matrix of order $6,000$. On the DELTA, the processor grid was 16×16 for a matrix of order $10,000$. The asynchronous implementation was also run with identical matrices on the both machines for comparison purposes. All timings were performed for different block sizes to find the best block size.

Figures 12 and 13 show the performance results among the current implementations.

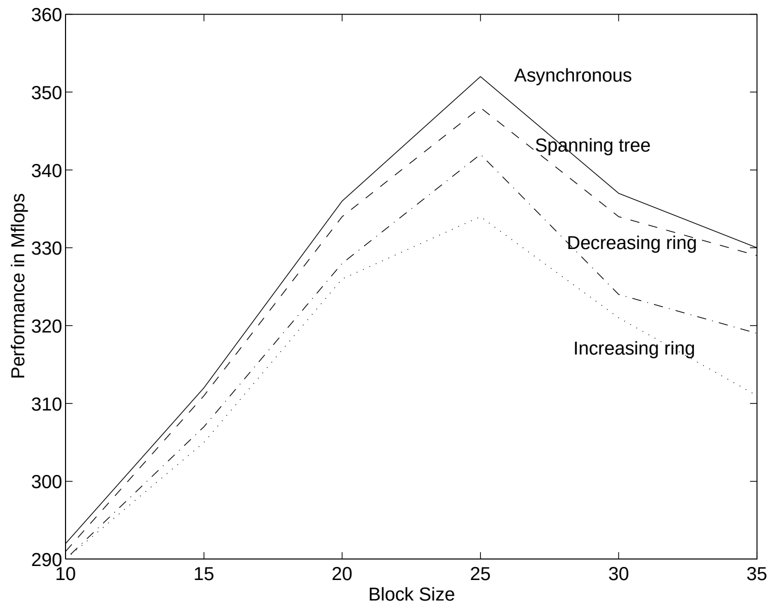


Figure 12: Performances for `STREE`, `DRING`, `IRING`, and asynchronous codes on the iPSC/860 for matrices of order $6,000$ on an 8×8 processor grid.

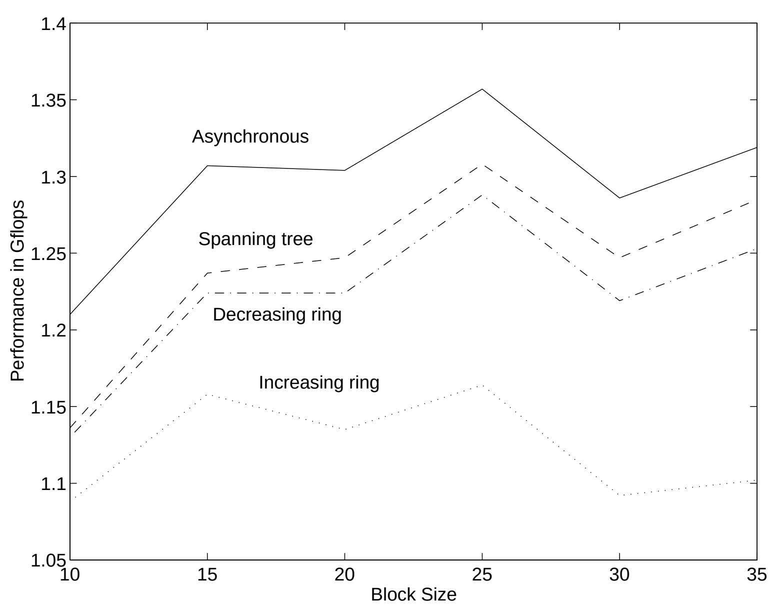


Figure 13: Performance `STREE`, `DRING`, `IRING`, and asynchronous codes on the Intel DELTA for matrices of order $10,000$ on a 16×16 processor grid.

For the purpose of comparing detailed timing and performance results of computation and communication parts of the algorithm, the `STREE` code of the synchronous implementation was run for a matrix of order $6,000$ with the same processor grid 8×8 on the both machines. Detailed timing results are shown in Figures 14 and 15, and their numeric data are provided in Tables 2 and 3 of the Appendix. In these tables, "Row" stands for row updating of blocks, "Col" for column updating, "QR" for computing QR factorization of blocks, "Givens" for computing Givens rotations of blocks, and "Bcast" for broadcasting messages to their corresponding row and column processors.

6 Conclusions

The performance results of the current implementations of our algorithm on the Intel iPSC/860 and DELTA are quite consistent, considering the distinct characteristics of both machines. We draw several conclusions from these results:

- The `STREE` code performs better than any of the other synchronous codes on the both machines, even though the DELTA has a mesh architecture.
- The asynchronous implementation performs best among implementations. This superior performance stems from less waiting time because of nonblocking communications.
- The most important factor limiting the performance of the synchronous implementation is blocking receiving processors. Specifically, the receiving processors spend most of their time waiting for messages (transformation matrices) from row and column processors. Furthermore, once messages are received, all blocks on the corresponding row and column are updated synchronously. Thus, synchronous communications limit the performance.
- Premultiplication (row updating) is much slower than postmultiplication (column updating), even without any communications. Furthermore, the ratio of premultiplication versus postmultiplication is improved and both premultiplication and postmultiplication are faster as the block size becomes larger; however, communication becomes slower for larger block size choices.
- The timing ratio of row updating versus column updating the in Givens phase is larger than that in the QR-factorization phase. This may be due to differences in memory access patterns and possible cache misses when blocks are accessed for updating in the Givens phase.
- The best block size for our current implementations is between 20 and 30.

7 Future Work

The reduction to Hessenberg form inherently requires very large numbers of computations of $\frac{10}{3}n^3$ flops for $n \times n$ matrices. In our parallel algorithm, the computation cost still dominates the total cost of the algorithm, and the communication cost is inexpensive. Thus, there is a tradeoff between the improved performance possible by using more number of processors and the degradation resulting from more communications.

In view of the overhead of managing some data structures for handing asynchronous sends and receives, one might explore the use of `hsend()` and `hrecv()`. These Intel routines behave like interrupt handlers, invoking a routine specified as arguments when a program running on a processor receives an incoming message. To use them, however, we need to use large `common` blocks in Fortran to handle asynchronous message passing.

Another area for research is load balancing. When the block size is large, wrap-mapping in a round-robin fashion is not optimal. A mapping created by wrapping processors on the row and column alternately in reverse order would be better.

Finally, we envision this algorithm as a highly parallel algorithm for massive parallel multicomputers with more than one-thousand processing nodes, such as the Intel Paragon, or on message-passing multicomputers with smaller communication latency.

8 Acknowledgement

We acknowledge valuable comments and suggestions by Prof. Gene H. Golub of Stanford University. This research was performed in part using the Intel Touchstone DELTA System operated by the California Institute of Technology on behalf of the Concurrent Supercomputing Consortium and also conducted on the Intel iPSC/860 System located at the Oak Ridge National Laboratory.

References

[1] M. Barnett, D. G. Payne, and R. van de Geijn. Optimal broadcasting in mesh-connected architectures. Technical Report TR-91-38, University of Texas, 1991.

[2] Eleanor Chu and Alan George. QR factorization of a dense matrix on a hypercube multiprocessor. *SIAM J. Sci. Stat. Comput.*, 11(5):990–1028, 1990.

[3] J. J. Dongarra and S. Ostrouchov. LAPACK block factorization algorithms on the intel ipsc/860. Technical Report CS-90-115, University of Tennessee, 1990. LAPACK Working Note 24.

[4] J. J. Dongarra, A. H. Sameh, and D. C. Sorensen. Implementation of some concurrent algorithms for matrix factorizations. *Parallel Computing*, 3:25–34, 1986.

[5] J. J. Dongarra, D. C. Sorensen, and S. Hammarling. Block reduction of matrices to condensed forms for eigenvalue computations. *Journal of Computational and Applied Mathematics*, 27:215–227, 1989.

[6] J. J. Dongarra, R. van de Geijn, and R. C. Whaley. A users' guide to the BLACS. Technical Report CS-93-187, University of Tennessee, 1993. LAPACK Working Note 57.

[7] J. J. Dongarra and R. A. van de Geijn. Reduction to condensed form for the eigenvalue problem on distributed memory architectures. *Parallel Computing*, 18:973–982, 1992.

[8] J.J. Dongarra, J. DuCroz, and S. Hammerling. A set of level 3 basic linear algebra subprograms. *ACM Trans.on Math. Soft.*, 16(1):1–17, 1990.

[9] G. H. Golub and C. V. Van Loan. *Matrix Computations, 2nd ed.* The Johns Hopkins University Press, lBaltimore, Mayland, 1989.

[10] Alex Pothen and Padma Raghavan. Distributed orthogonal factorization: Givens and Householder algorithms. *SIAM J. Sci. Stat. Comput.*, 10(6):1113–1134, 1989.

[11] Steven R. Seidel. Broadcasting on linear arrays and meshes. Technical Report ORNL/TM-12356, Oak Ridge National Laboratory, 1990.

Appendix

		QR fact. (Ops, Secs, %)			Givens (Ops, Secs, %)		
NB	Total (secs)	Op Counts	Time	%	Op Counts	Time	%
10	3320.96	0.3559	1181.96	35.6	0.591	2138.97	64.4
15	3072.65	0.3577	923.06	30.0	0.571	2149.57	70.0
20	2861.63	0.3581	856.20	29.9	0.561	2005.42	70.1
25	2735.18	0.3582	780.80	28.5	0.555	1954.37	71.5
30	2852.59	0.3582	763.60	26.8	0.550	2088.98	73.2
35	2908.16	0.3580	795.54	27.3	0.546	2112.61	72.7
40	2947.40	0.3578	853.50	29.0	0.543	2093.89	71.0

	QR fact. Phase (secs)				Givens Phase I (secs)				Givens Phase II (secs)			
NB	QR	Bcast	Row	Col	Givens	Bcast	Row	Col	Givens	Bcast	Row	Col
10	6.17	186.34	725.66	261.90	7.16	102.95	1354.93	478.86	2.73	125.61	36.01	23.38
15	4.57	190.39	486.22	241.02	6.66	109.29	1275.16	512.88	3.88	151.83	50.64	35.69
20	3.92	210.65	401.41	239.71	6.61	70.04	1236.66	451.71	5.19	119.98	65.86	47.66
25	3.39	227.38	327.71	221.99	6.31	134.05	1001.41	479.01	6.26	197.33	72.27	56.65
30	3.19	250.30	292.06	217.80	6.21	147.12	1131.41	406.15	7.55	235.89	92.94	62.06
35	3.22	270.80	289.47	231.83	6.62	103.84	1136.93	499.16	9.03	168.61	107.48	81.68
40	3.02	371.75	259.06	219.50	6.30	144.78	1118.76	404.05	10.37	197.07	124.69	89.11

Table 2: `STREE`: Detailed Timing Results on the Intel iPSC/860 for a matrix of order $6,000$ on an 8×8 processor grid

		QR fact. (Ops, Secs, %)			Givens (Ops, Secs, %)		
NB	Total (secs)	Op Counts	Time	%	Op Counts	Time	%
10	3138.22	0.3559	1099.42	35.0	0.591	2038.77	65.0
15	2906.21	0.3577	830.71	28.6	0.571	2075.49	71.4
20	2692.83	0.3581	749.86	27.8	0.561	1942.96	72.2
25	2543.63	0.3582	663.70	26.1	0.555	1879.92	73.9
30	2639.85	0.3582	638.68	24.2	0.550	2001.17	75.8
35	2677.06	0.3580	647.57	24.2	0.546	2029.49	75.8
40	2709.29	0.3578	679.67	25.1	0.543	2029.62	74.9

	QR fact. Phase (secs)				Givens Phase I (secs)				Givens Phase II (secs)			
NB	QR	Bcast	Row	Col	Givens	Bcast	Row	Col	Givens	Bcast	Row	Col
10	4.60	109.51	722.95	260.64	3.46	54.44	1321.15	478.41	1.69	114.7	35.34	22.83
15	3.49	100.41	485.25	240.76	3.27	43.10	1264.03	512.45	2.00	161.31	51.22	34.63
20	3.06	107.59	400.00	238.74	3.27	24.65	1224.01	450.24	2.63	123.06	66.47	46.64
25	2.71	111.04	327.08	222.58	3.17	38.67	1003.83	478.64	3.19	222.56	73.16	55.46
30	2.59	127.31	290.77	217.78	3.16	37.89	1119.47	405.51	3.84	276.63	92.68	61.43
35	2.67	123.54	288.80	232.36	3.46	19.31	1139.42	498.32	4.74	175.11	107.81	81.24
40	2.56	199.62	258.24	219.10	3.30	29.96	1110.17	403.37	5.45	265.65	124.68	88.00

Table 3: `STREE`: Detailed Timing Results on the Intel DELTA for a matrix of order $6,000$ on an 8×8 processor grid

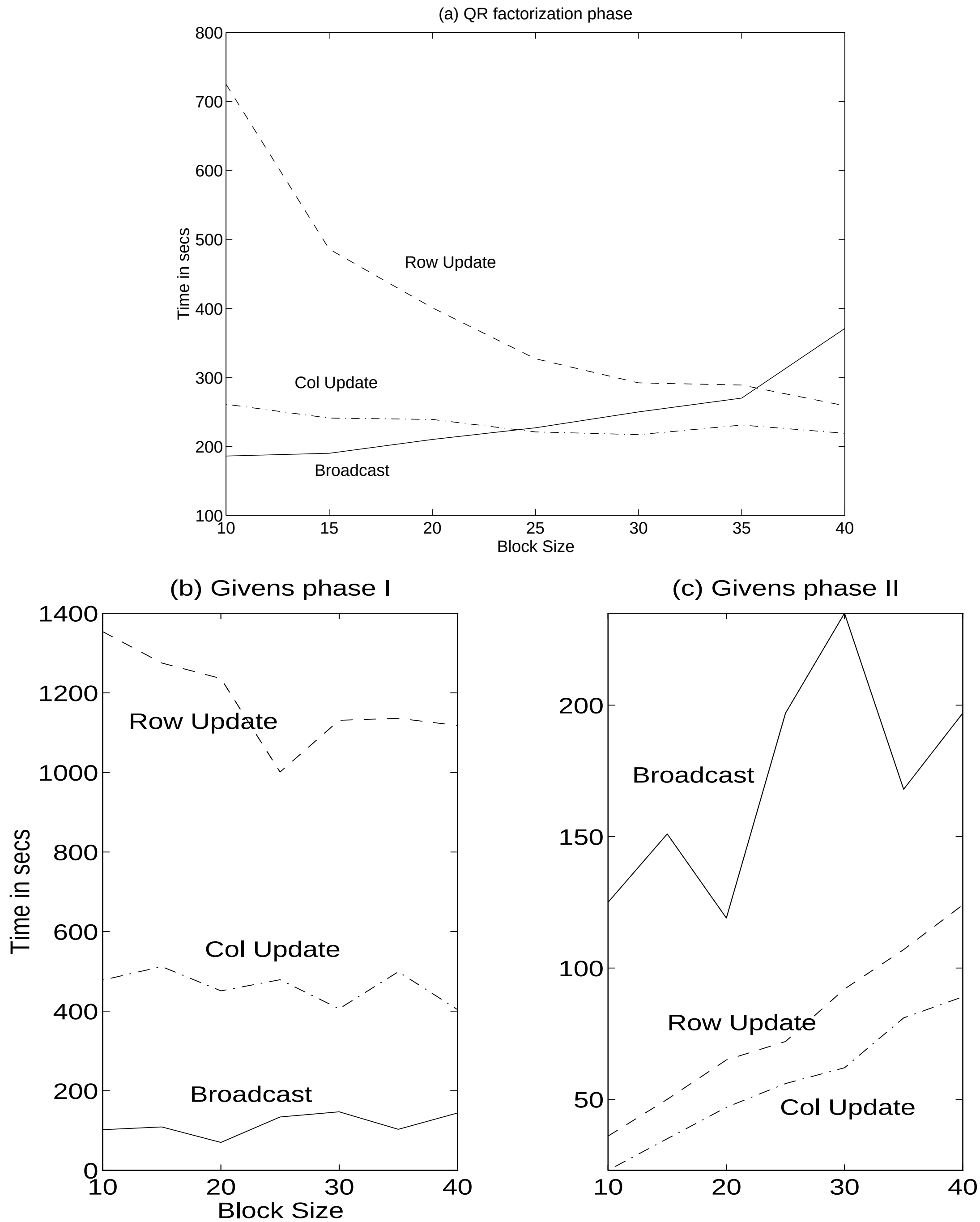


Figure 14: Detailed times on the iPSC/860 for matrices of order 6,000 on an 8×8 processor grid

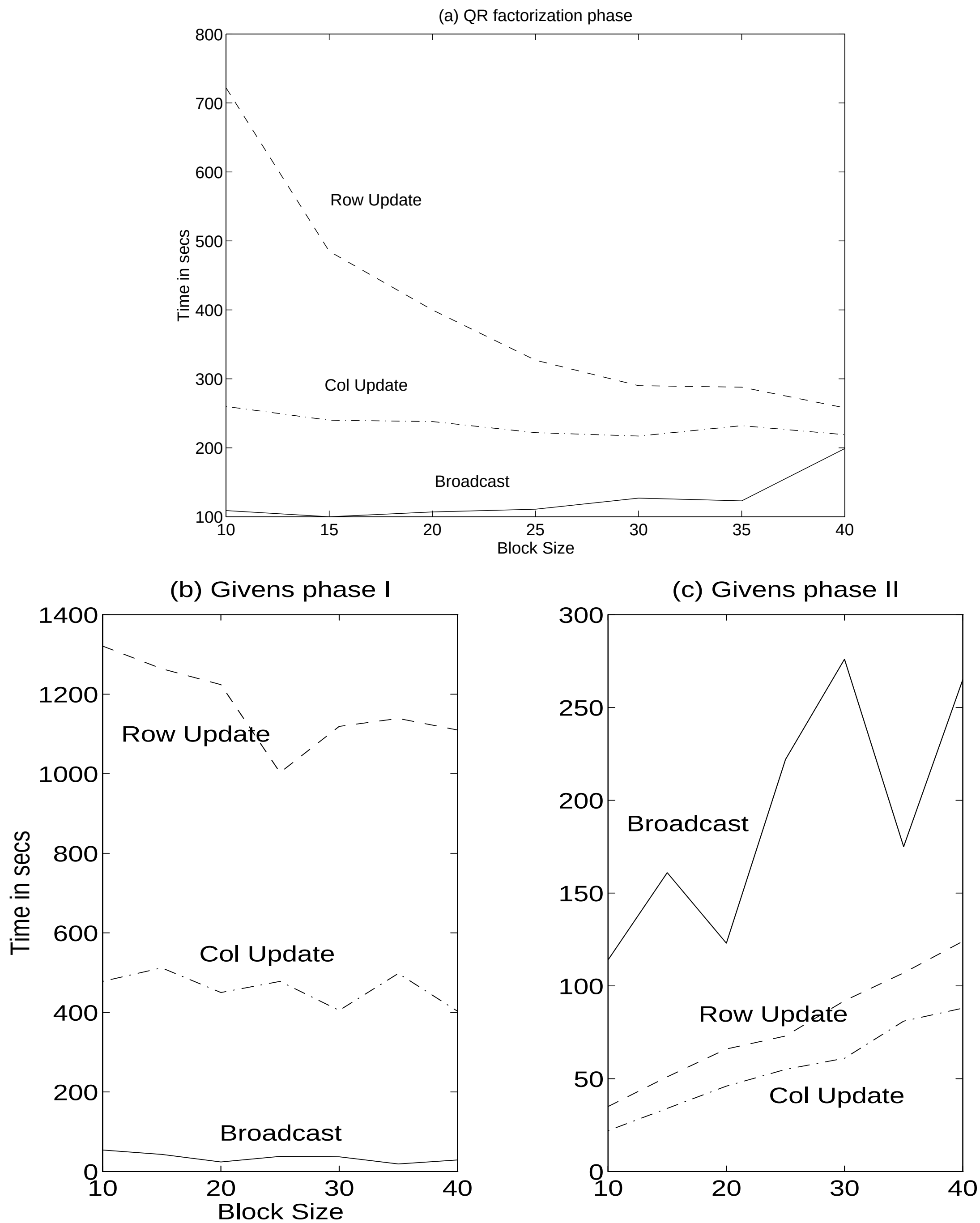


Figure 15: Detailed times on the Intel DELTA for matrices of order $6,000$ on an 8×8 processor grid