Givens and Householder Reductions for Linear Least Squares on a Cluster of Workstations

Ömer Eğecioğlu and *Ashok Srinivasan*

Department of Computer Science
University of California Santa Barbara, CA 93106

Abstract

We report on the properties of implementations of fast-Givens rotation and Householder reflector based parallel algorithms for the solution of linear least squares problems on a cluster of workstations. It is shown that the Givens rotations enable communication hiding and take greater advantage of parallelism than Householder reflectors, provided the matrices are sufficiently large.

Keywords: Linear least squares, Givens rotation, Householder reflection, parallel algorithm, workstation cluster.

1 Introduction

The linear least squares (LLS) problem is the minimization of the residual

$$\min_{x \in \mathbb{R}^n} \|Ax - b\|_2 \tag{1}$$

where $A \in \mathbb{R}^{m \times n}$ and $b \in \mathbb{R}^m$ are given, and the subscript denotes the ordinary Euclidean 2–norm. Such problems typically arise in evaluating unknown parameters $x_1, x_2, \ldots, x_n$ of a linear model to fit measured data. Each row of A represents the values of the independent variables in an experiment, and

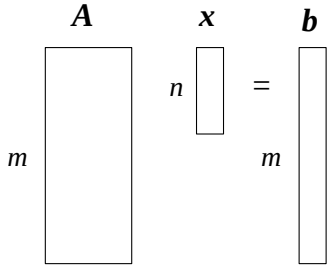

Figure 1: $Ax = b$.

the corresponding row of b represents the value of the dependent variable for that particular experiment. The measured values may be imprecise because of various sources of error, and consequently the number of experiments conducted exceeds the number of unknown parameters. Thus it is natural to assume that $m \gg n$. In this case the linear system $Ax = b$ is overdetermined. An overdetermined system usually has no exact solution, and the alternative is the minimization of the residual over a suitable norm, such as (1). A further simplifying assumption is that the columns of A are linearly independent: i.e. it has full column rank.

There are two main methods to solve (1) that use orthogonal updates: Givens (or fast-Givens) rotations, and Householder reflectors. Both use the fact that $\|Ax - b\|_2 = \|QA - Qb\|_2$, when Q is an orthogonal matrix, and through a series of premultiplications (whose product we denote by Q^T) reduce A to

$$Q^T A = \begin{bmatrix} R \\ 0 \end{bmatrix} \begin{matrix} n \\ m-n \end{matrix} \tag{2}$$

where R is $n \times n$ upper triangular. This is the QR factorization of A. Once the reduction to upper triangular form for A as in (2) is accomplished, x can be obtained by back substitution by solving $Rx = c$ where

$$Q^T b = \begin{bmatrix} c \\ d \end{bmatrix} \begin{matrix} n \\ m-n \end{matrix} \tag{3}$$

It is well known that both methods are numerically stable, and are preferred over the solution of the normal equations for LLS problems [7, 8], [14]. While the number of additions for Givens and Householder methods are the same for the QR factorization, the number of multiplications and square roots in the Householder method is fewer than the Givens method [7, 8]. Furthermore, if fast-Givens transformations are used to reduce the number of multiplications of Givens transformations, then periodic monitoring and scaling of the row multipliers are necessary. This makes the Householder method the sequential algorithm of choice for dense QR, and consequently for LLS problems. In a parallel setting, however, this advantage is offset by the higher communication requirements of the Householder reduction algorithm. This is because each annihilation requires access to not only an entire row, but also an entire column, and for $m \gg n$ there does not seem to be a straightforward way of overlapping communication with annihilation, irrespective of how the matrix elements are allocated to the processors.

In this paper, we consider a parallel implementation of the fast-Givens reduction for LLS formulated as in (1) with $m \gg n$ on a cluster of workstations, and compare our results with the Householder based ScaLAPACK QR factorization run in the same environment. In our tests, the output is $Q^T A = R$ and $Q^T b$, and back substitution step is not performed as it is identical for both algorithms. The experiments were conducted using PVM [3] on a cluster of Sun SPARCStation LX workstations connected over an Ethernet LAN, for matrices of three different sizes.

The implementation of the LLS minimization algorithm using Givens rotations takes maximum advantage of parallelism due to its ability to hide communication costs, since computations can take place while messages are in transit. In this case, however, it turns out to be more problematic to make Householder reflectors take equally significant advantage of parallelism due to the difficulty of hiding communication costs.

The outline of this paper is as follows: first we present an overview of Givens and fast-Givens transformations in Section 2, and Householder transformations in Section 3. Various aspects of these methods including applications and more detailed analyses of their properties can be found in [7], [8], [9], [12], [14], [15]. In Section 4 we present our distributed memory implementation of the Givens reduction algorithm for LLS. In Section 5 we present our test results, compare the performance of the two methods, and present our conclusions.

2 Givens transformations

2.1 Givens rotations

Consider the matrices

$$E = \begin{bmatrix} c & s \\ -s & c \end{bmatrix} \qquad A = \begin{bmatrix} a_1 \\ a_2 \end{bmatrix}$$

where a_1 and a_2 are two row vectors. Then

$$EA = \begin{bmatrix} ca_1 + sa_2 \\ -sa_1 + ca_2 \end{bmatrix} . \tag{4}$$

If the first $k-1$ components of a_1 and a_2 are zero, $k \geq 1$, then we can introduce an additional zero in the kth component of a_2 by choosing c and s as

$$c = \frac{a_{1,k}}{\sqrt{a_{1,k}^2 + a_{2,k}^2}} , \quad s = \frac{a_{2,k}}{\sqrt{a_{1,k}^2 + a_{2,k}^2}} .$$

With this choice $c^2 + s^2 = 1$, and $c = \cos\theta$ and $s = \sin\theta$ for some angle θ. The matrix E is a plane rotation matrix through angle θ and consequently orthogonal. By embedding E into the identity matrix we construct a matrix E_{ij} where $c_{ij} = \cos\theta$ and $s_{ij} = \sin\theta$ appear in the ith and jth rows and columns as shown in (5). The matrix E_{ij} defines a rotation in the (i,j) plane in $\mathbb{R}^n$ and it can be used to introduce a zero into the kth entry in the jth row of A.

$$E_{ij} = \begin{bmatrix} 1 & & & & & & & & & \\ & \ddots & & & & & & & & \\ & & 1 & & & & & & & \\ & & & c_{ij} & & & & s_{ij} & & \\ & & & & 1 & & & & & \\ & & & & & \ddots & & & & \\ & & & & & & 1 & & & \\ & & & -s_{ij} & & & & c_{ij} & & \\ & & & & & & & & 1 & \\ & & & & & & & & & \ddots \\ & & & & & & & & & & 1 \end{bmatrix} \tag{5}$$

Let $rotate(i,j)$ denote the function that makes the (i,j)th element of a matrix A zero, using rows a_{i-1} and a_i in place of a_1 and a_2 in (4). The sequence of Givens transformations in Figure 2 reduces $A \in \mathbb{R}^{m \times n}$ into upper triangular form. This straightforward algorithm takes approximately $2n^2(m - \frac{n}{3})$

```
for j = 1 to n do
   for i = m downto j + 1 do
      rotate(i, j)
   end
end
```

Figure 2: Reduction to upper triangular form (QR) by Givens rotations.

multiplications, since each rotation requires about $4(n-j)$ multiplications. The number of multiplications can be reduced by fast-Givens transformations.

2.2 Fast-Givens transformations

Fast-Givens (or fast-scaled) transformations allows for each rotation to be performed using about $2(n-j)$ multiplications. To this end the current matrix is kept in a factored form as DA where D is an $m \times m$ diagonal matrix. (In the actual implementation D is stored as a vector.) Consider rows a_1 and a_2 of A as in (4). Instead of computing the entries of the new rows in (4), we first compute $\tau = \frac{s}{c}$. Then

$$\begin{bmatrix} ca_1 + sa_2 \\ -sa_1 + ca_2 \end{bmatrix} = \begin{bmatrix} c & 0 \\ 0 & c \end{bmatrix} \begin{bmatrix} a_1 + \tau a_2 \\ -\tau a_1 + a_2 \end{bmatrix}$$

This allows savings on the number of multiplications performed by postponing the scalar-vector products in (4) until the very end. We need to see how D should be updated when a new rotation on a row is performed. It suffices to consider only the two rows that are altered. Solving for μ' and ν' in

$$\begin{bmatrix} \mu' & 0 \\ 0 & \nu' \end{bmatrix} \begin{bmatrix} a' \\ b' \end{bmatrix} = \begin{bmatrix} c & s \\ -s & c \end{bmatrix} \begin{bmatrix} \mu & 0 \\ 0 & \nu \end{bmatrix} \begin{bmatrix} a \\ b \end{bmatrix}$$

where a, a', b, and b' are row vectors results in the equations

$$\mu' a' = c\mu a + s\nu b$$

$$\nu' b' = -s\mu a + c\nu b$$

For new values, it is possible to choose either

$$\mu' = s\nu, \quad \nu' = s\mu$$

or

$$\mu' = c\mu, \quad \nu' = c\nu.$$

We choose the former if $|s| \geq |c|$, and the latter otherwise. These two possibilities result in type 1 and type 2 fast-Givens rotations, and the appropriate choice serves to keep the diagonal elements of D as far from 0 as possible [8]. Each time a row is involved in a rotation, the absolute value of the corresponding term in the diagonal matrix could decrease by a factor of $1/\sqrt{2}$. Since a row may be involved in up to $2n$ rotations, the corresponding diagonal term can change by a factor of as much as 2^{-n}. This imposes a limit on the number of columns we can safely have without overflow unless we pay the price of monitoring the growth of D by appropriate scaling [8].

3 Householder reflectors

Householder reflectors are orthogonal matrices of the form

$$E = I - 2ww^T$$

where $w \in \mathbb{R}^{m \times 1}$ is a unit vector. They are used to introduce zeros to more than one entry of a column vector of A simultaneously by a single premultiplication, by a suitable choice of w. To make the entries of a vector $x \in \mathbb{R}^{m \times 1}$ below its kth component 0 while leaving the ones above it unchanged, form the vector

$$u = (0, \cdots, 0, x_k + sign(x_k)\alpha, x_{k+1}, \cdots, x_m)^T$$

where

$$\alpha = (x_k^2 + \cdots + x_m^2)^{\frac{1}{2}}.$$

The Householder transformation matrix corresponding to u is formed by setting

$$E = I - \frac{2}{u^T u} u u^T .$$

E is an $m \times m$ orthogonal matrix which is also called an elementary reflection. It can be verified that the vector Ex has the desired properties. In addition, computing Ey for an arbitrary vector y is simple, for

$$Ey = Iy - \frac{2}{u^T u} u u^T y \ ,$$

and $u^T y$ is a scalar. Thus we can determine EA for given matrix A relatively easily. If E_j denotes the Householder reflector that fills the subdiagonal entries in column j of the current matrix with zeros, then the Householder reduction algorithm for A proceeds as given in Figure 3. This successively introduces

for $j = 1$ **to** n **do**
  Compute E_j
  $A := E_j A$
end

Figure 3: Reduction to upper triangular form (QR) by Householder reflectors.

zeros in subdiagonal entries of columns of A and reduces A to upper triangular form. The Householder reduction algorithm as described above requires about $mn^2 - \frac{n^3}{3}$ multiplications.

4 A distributed memory Givens implementation

First we consider basic schedules for parallel implementation of the Givens reduction algorithm.

4.1 Scheduling for Givens rotations

A Givens rotation affects only two rows of A at a time. If successive rotations were to affect only disjoint pairs of rows, then they could be done simultaneously without affecting the final result. Otherwise, a suitable ordering (schedule) needs to be specified on the execution of individual Givens rotations that avoids common rows. More formally, suppose $T(j,k)$ is the time step at which the (j,k)th entry of the matrix is annihilated (becomes 0), and $S(j,k)$ is the index of the row that is used along with row j to annihilate the (j,k)th entry. Any such pair of functions defined for $k < j$ satisfying the following requirements is an acceptable schedule for the parallel execution of Algorithm 2 :

1. Concurrent operations act on different rows. In other words, if $T(j,k) = T(j',k')$, and $(j,k) \neq (j',k')$, then the sets $\{j, S(j,k)\}$ and $\{j', S(j',k')\}$ are disjoint.

2. If $T(j,k) = t$ and $S(j,k) = i$, then $T(j,l) < t$ and $T(i,l) < t$ for all $l < k$. This ensures that all the previous columns of the two rows involved at time step t have already been annihilated.

A particular schedule satisfying these two properties was developed by Sameh and Kuck [13], in which

$$\begin{aligned} T(j,k) &= m - j + 2k - 1 \ , \\ S(j,k) &= j - 1 \ . \end{aligned}$$

As an example, Figure 4 (a) gives the Sameh-Kuck annihilation pattern $T(j,k)$ for a 10×10 matrix. Only the subdiagonal entries of the matrix are shown. The numbers indicate the stage at which the corresponding entry is annihilated. If $m > n$, the last annihilation occurs for the $(n+1,n)$th element

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

(a)

	9								
$p=1$	8	11							
	7	10	12						
	6	9	12	15					
$p=2$	5	8	11	14	17				
	4	7	10	13	16	18			
	3	6	9	12	15	18	21		
$p=3$	2	5	8	11	14	17	20	23	
	1	4	7	10	13	16	19	22	24

(b)

Figure 4: (a) Sameh-Kuck annihilation pattern, (b) Block pattern for $P = 3$ processors, for a 10×10 matrix.

at stage $m + n - 2$. Thus for an $m \times n$ matrix, approximately $m + n$ stages need to be performed.

It is clear that given n processors, a single Givens rotation can be performed in small constant time. Therefore the computation time of the overall Givens reduction Algorithm 2, given n processors is ideally of order of $m + n$. If $nm/2$ processors are available, then up to $m/2$ rotations can be performed simultaneously. However for execution on a cluster of workstations, a large number of machines would be needed even for modest values of m and n. The problem of efficient scheduling becomes interesting if the number of available processors is limited. Special cases of this for various architectures were discussed in [11], [10]. In the limited processor case, the work of many processors in the original schedule needs to be assigned to a single processor p. On an ideal shared memory computer this distribution could be done without overhead since there would be no communication costs involved. In practical situations however, rows of A need to be distributed across the processors, compatible with the annihilation pattern in Figure 4 (a), so that the overall interprocessor movement of the rows is minimized.

Figure 4 (b) gives the annihilation pattern used in our implementation of the Givens reduction algorithm when $P = 3$ processors are available for a 10×10 matrix.

4.2 The algorithm

We view each p as a *process* instead of a processor, since in some instances it may be necessary to create more than one process to run on a physical processor. For the implementation of the algorithm, we create P processes, each represented by a distinct identifier in $\{1, 2, \ldots, P\}$. A block of adjacent rows are assigned to a single process, with each process p keeping approximately the same number of rows of A as indicated in 4 (b) for $P = 3$. Each process annihilates entries beginning with the first column of the matrix. In each column, annihilation starts with the last row in the block of rows assigned to the process. Annihilation of the entries in row a_i is performed using row a_{i-1}. Note that in order to annihilate an entry in the first row of its block, process p has to access the last row of process $p - 1$. In addition, process p cannot begin to annihilate an entry in its last row until it has received the modified version

of it from process $p+1$. High level steps of the algorithm executed by each process $p \in \{1, 2, \ldots, P\}$ are given in Figure 5. The function $rotate(\cdot, \cdot)$ used in Figure 5 operates in the same manner as the one mentioned earlier, except that it also modifies the corresponding elements of the vector b. In order not to clutter up the algorithm description, we assume that whenever a message is to be sent to or received from a process whose number is not in the range $\{1, 2, \ldots, P\}$, no action is taken.

Input: $A \in \mathbb{R}^{m \times n}$ and $b \in \mathbb{R}^m$
Output: $Q^T A = R$ and $Q^T b$

Step 1: Process p is assigned the block of rows indexed $m(p-1)/P + 1$ (first) through mp/P (last)

Step 2: Send last row to process $p+1$

Step 3: **for** ($j = 1$ **to** $min(\text{last} - 1, n)$) **do**

(a) Get back modified last row from process $p+1$
(b) **if** p has more than a single row **then**
  i. $rotate(last, j)$
  ii. Send last row to process $p+1$
(c) $rotate(i, j)$ for all p's interior rows with index $i > j$
(d) **if** index of first row $> j$ **then**
  i. Receive last row of process $p-1$
  ii. $rotate(first, j)$
  iii. Send process $p-1$'s modified last row back to it
  iv. **if** p has only one row **then** send it to process $p+1$
(e) Scale entries of D corresponding to the block

Step 4: Multiply partial matrix by diagonal elements and output result.

Figure 5: Main steps of the distributed memory Givens algorithm.

Next we give an informal justification of the correctness of this algorithm: First of all each row is assigned to exactly one process. We can argue that if each process executes all the statements, the result obtained will be the desired one. Subsequently we can show that all the statements will indeed be executed.

The main calculation occurs in Step 3. For each value of j, the first row, interior rows (whose indices are between first and last), and last row are annihilated in column j. Besides, when (b) is executed, the process already has its last row from the next process, received in (a). When p modifies its first row in (d)(ii), it has already received the last row of process $p-1$ in step (d)(i). Step 3 (c) ensures that only, and all, subdiagonal elements are annihilated. Since j varies over all the required columns, the reduced matrix is upper triangular. In Step 3 (e), the diagonal elements of D corresponding to the rows assigned to p are scaled.

To show that that all the steps are actually executed we need to make sure that the required messages are actually sent by the responsible processes. In Step 2, each process sends its last row. Therefore, each process will have the last row of the previous process. In particular, process P will have the last

row of process $P-1$ and will be able to execute Step 3. Note that it does not need to execute Step 3 (a). P returns $P-1$'s last row in (d)(iii). Thus, $P-1$ in turn can complete its execution for this value of j. In general, a process needs the last row of $p-1$ after annihilating column $j-1$ and its last row after process $p+1$ has used it in order to annihilate column j. Step (b)(ii) ensures the former, and step (d)(iii) the latter. Thus, eventually all processes execute to completion. Finally, Step 4 is evident.

4.3 Time complexity

To consider the time taken for computation we need to consider only Step 3. Let us assume $m/P \gg n$ to simplify the analysis, for otherwise equal subdivision of rows to processes does not result in near equal loads. For each j, $rotate(i,j)$ requires about $2(n-j)$ multiplications using fast-Givens rotations. This is performed for at most m/P rows. Thus, Step 3 requires about $2m(n-j)/P$ multiplications. Summing this over all j gives mn^2/P multiplication steps for each process. However the processes first have to wait in order to start, and also for the messages needed for intermediate computations. The initial wait is only for one column, and so is of the order of mn and does not add to the higher order term for computation. The total number of messages sent is $2Pn$ and each message consists of at most n floating point numbers that make up a row of A. If T_1 is the fixed startup cost of sending and receiving a message relative to a multiplication, and T_2 is the relative cost of sending a floating point number across the network, then the time complexity of the algorithm is no worse than

$$mn^2/P + 2T_1Pn + 2T_2Pn^2 \ .$$

This expression is an upper bound since it does not take into account computation/communication overlap or communication/communication overlap. The latter should play an important role in a communication network such as a grid or a hypercube, when multiple messages can be sent simultaneously from sources to destinations.

5 Comparison with Householder for LLS

Block-partitioned QR factorization algorithm using Householder reflectors is implemented in the LAPACK [1, 2], which is a software library for performing dense and banded linear algebra computations on vector machines and shared memory computers. LAPACK makes use of block-partitioned algorithms to utilize fast matrix-vector and matrix-matrix operations on data that reside in higher levels in hierarchical memories. An extension of the LAPACK to distributed memory concurrent computers is the ScaLAPACK library [5, 6]. ScaLAPACK uses block-cyclic data distribution to processors as its primary data decomposition method for matrices to improve overall performance and scalability. The characteristics and the performance of various ScaLAPACK factorization routines, including QR factorization on the Intel family of parallel computes is reported in [6]. ScaLAPACK makes use of the LAPACK Householder transformations for QR factorization.

Householder reduction is more difficult to implement efficiently on a distributed memory system because each annihilation requires access to not only an entire row, but also an entire column. Consequently, irrespective of how one allocates the matrix elements to the processors, interprocess communication at each step of the annihilation is necessary. Since we have to compute the norm of a column at each step, dividing rows among processors would involve extensive of inter-process communication at each step just for this computation. On the other hand, in LLS problems with $m \gg n$, it is not possible to take advantage of parallelism by using columns either, since the number of columns is not significant compared to the number of rows. In other words, the annihilation pattern for the elements of matrix

A for a distributed memory implementation of Householder reduction essentially reflect the pattern of the sequential algorithm. The advantage of parallelism comes mainly from processors making parallel updates on their local data using annihilating reflectors received from other processors.

5.1 Test platform

The parallel fast Givens algorithm was compared with the QR factorization routine of ScaLAPACK. PVM version 3.3.7 was used for message passing in the fast Givens algorithm. ScaLAPACK version 1.1, was used to test the ScaLAPACK algorithm, using the PVM version of BLACS message passing. The tests were conducted on a cluster of Sun SPARCStation LX workstations connected over an Ethernet LAN, for matrices of three different sizes, using matrices from [16] and the parallel double precision matrix generator of ScaLAPACK.

The sizes of the matrices used for the test were 1000×10, 2000×50, and 5000×50. The tests were performed on up to 20 processors. ScaLAPACK allows the user to choose the processor grid structure and the block size. Due to the high communication cost involved in the Ethernet, it was found that having one column of processors gave the best results for a fixed number of processors. Therefore, results are reported only for this situation. It was found that bigger block sizes are better for a PVM implementation. In accordance with this, the block size was taken to be at least as high as the number of rows.

The measure of time was taken to be the "wall clock time" for the actual calculations, ignoring the time taken for generating the test matrix and for the output of the factors. At least four experiments were performed for each value of the parameter set, and the average of these is reported in Figures 6, 7, and 8. The datasets labeled Givens-1 and Givens-2 correspond to two different scaling mechanisms used in the fast-Givens implementation as described below.

5.2 Scaling of D

In the fast-Givens method the diagonal matrix D needs to be periodically scaled to avoid its elements from becoming too small. In the annihilation of each column, the absolute value of the elements may be reduced by up to half. After each annihilation, we check to ensure that this value is ≥ 0.5. If not, we multiply it by 2 and divide each element of the corresponding row by 2. In our initial implementation of the fast-Givens transformations, we performed these operations using floating point arithmetic. The results are reported as Givens-1 in Figures 6, 7, and 8.

Since division and multiplication by 2 are shifts and can be performed by changing only the exponents involved, in the later version of the implementation we took advantage of this by using the C library function *ldexp* to perform integer operations on the exponents. The resulting algorithm Givens-2 has significantly reduced sequential running time over Givens-1. However as the number of processors increases, the computational costs of scaling become less important and Givens-1 and Givens-2 behave similarly.

5.3 Test results

It was found that the sequential version of the QR factorization using Householder reflections of ScaLAPACK was much faster than the fast-Givens algorithm on a single processor. This could be partly due to the innate nature of the algorithm and also possibly to a difference in the quality of the compilers in optimization, since the fast Givens algorithm was written in C whereas the Householder algorithm was in $FORTRAN$. It was found that for the smaller 1000×10 matrix, the sequential Householder

algorithm was much faster than its parallel version for $P \leq 20$, processors tested, probably due to the high communication overhead. Similarly for the parallel Givens algorithm, while there was a slight speedup with increasing P, it was not significant for the smaller matrix (Figure 6). For the medium size 2000×50 matrix, moderate speedup was obtained using the Householder algorithm for a small number of processors, substantial speedup was obtained using the Givens algorithm across the whole range of processors tested. These results are shown in Figure 7. For the 5000×50 case shown in Figure 8, the Householder algorithm outperformed the fast-Givens implementation for small number of processors only. After about $P = 6$, the Givens implementation performed better.

The performance of the algorithms can be explained if we observe that both algorithms have comparable number of floating point operations, and the number of messages sent is also about the same. However, much of the communication cost can be hidden in the fast-Givens algorithm, since computations can be performed while communication is taking place. Therefore the observed communication cost is approximately $2n + P$ messages, as opposed to $2Pn$. It does not seem possible to hide the communication cost in the Householder algorithm. This may account for its poor speedup when the communication network used is slow.

5.4 Conclusions and future work

Givens and Householder transformations were implemented on a cluster of workstations using PVM for the solution of LLS problems. Some of the experimental results are conclusive: the parallel algorithm for fast-Givens transformations can significantly reduce the time complexity for an $m \times n$ coefficient matrix with $m \gg n$. It appears that the Givens transformation is better suited to distributed memory machines with significant communication costs than the Householder transformation, at least when dealing with large matrices. Furthermore, the analysis giving the time complexity of the parallel algorithms can be expected to be valid even when implemented in different environments.

We intend to generalize the implementation of the parallel fast-Givens algorithm so that a row of processors is used to perform updates of the rows in parallel by using a rectangular array of processors. We can also adapt the block-cyclic data distribution scheme of ScaLAPACK to the Givens case. In place of a cluster of workstations, we plan to conduct our next tests on a parallel computer (Meiko CS-2) which has a fast interconnection network. In this setup, the communication costs incurred are expected to be much lower than for the Ethernet delays that we have observed. It will also be possible to to determine the sizes of the matrices for which the communication hiding ability of the Givens algorithm gives a higher speedup than the Householder algorithm.

References

[1] Anderson, E., Z. Bai, C. Bischof, J. Demmel, J.J. Dongarra, J. DuCroz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen, "LAPACK: A Portable Linear Algebra Library for High-Performance Computers," in *Proc. Supercomputing '90*, pp.1–10, IEEE Press, 1990.

[2] Anderson, E., Z. Bai, J. Demmel, J.J. Dongarra, J. DuCroz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov, and D. Sorensen, *LAPACK Users Guide*, SIAM Press, Philadelphia, PA, 1992.

[3] Beguelin, A, J.J. Dongarra, G.A. Geist, R. Manchek, and V.S. Sunderam, A users' guide to PVM parallel virtual machine, ORNL/TM11826, 1991.

[4] Bertsekas, D.P., and J.N. Tsitsiklis, *Parallel and Distributed Computation, Numerical Methods*, Prentice Hall, Englewood Cliffs, NJ, 1989.

[5] Choi, J., J.J. Dongarra, R. Pozo, and D.W. Walker, "A Scalable Linear Algebra Library for Distributed Memory Concurrent Computers," In *Proceedings of Fourth Symposium on the Frontiers of Massively Parallel Computation (McLean, Virginia)*. IEEE Computer Society Press, Los Alamitos, CA , 1992.

[6] Choi, J., J.J. Dongarra, S. Ostrouchov, A.P. Petitet, D.W. Walker, R.C. Whaley, "The Design and Implementation of the ScaLAPACK LU, QR, and Cholesky Factorization Routines," Oak Ridge National Laboratory, TM-12470, September 1994.

[7] Golub, H. G., and J.M. Ortega, *Scientific Computing an Introduction with Parallel Computing*, Academic Press, Inc., San Diego, 1993.

[8] Golub, H. G., and C. F. Van Loan, *Matrix Computations*, 2nd Edition, The Johns Hopkins University Press, Baltimore, 1990.

[9] Hamming, R.W., *Numerical Methods for Scientists and Engineers*, 2nd Edition, Dover Publications, Inc., New York, 1986.

[10] Kowalik, J., S. Kumar, and E. Kamgnia, "An Implementation of the fast-Givens Transformations on a MIMD Computer," *Appl. Math. Polish Academy of Science*, (?).

[11] Lord, R., J. Kowalik, and S. Kumar, "Solving Linear Algebraic Equations on a MIMD Computer," *Proc. 1980 Int. Conf. Par. Proc.*, pp. 205–210.

[12] Parlett, B. N., *The symmetric eigenvalue problem*, Prentice-Hall, Englewood Cliffs, NJ, 1980.

[13] Sameh, A., and D. Kuck, "On Stable Parallel Linear System Solvers," *J. ACM* 1978, 25, pp. 81–91.

[14] Wilkinson, J. H, *The Algebraic Eigenvalue Problem*, Oxford University Press, Oxford, 1965.

[15] Ortega, J.M., and R.G. Voigt, *Solution of Partial Differential Equations on Vector and Parallel Computers*, SIAM, Philadelphia, 1985.

[16] Gregory, T., and D.L. Karney, *A collection of matrices for testing computational algorithms*, Wiley-Interscience, New York, 1969.

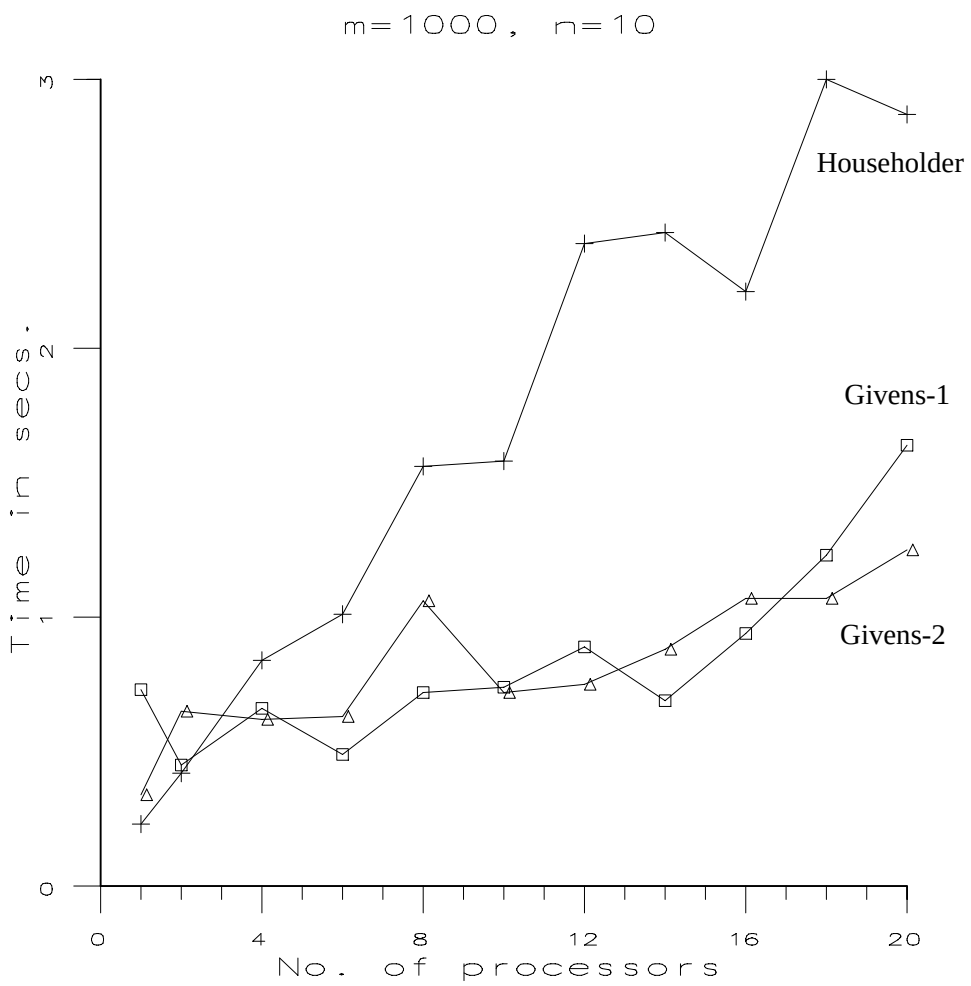

Figure 6: Comparison of Givens and Householder reductions for LLS: $m = 1000, n = 10$.

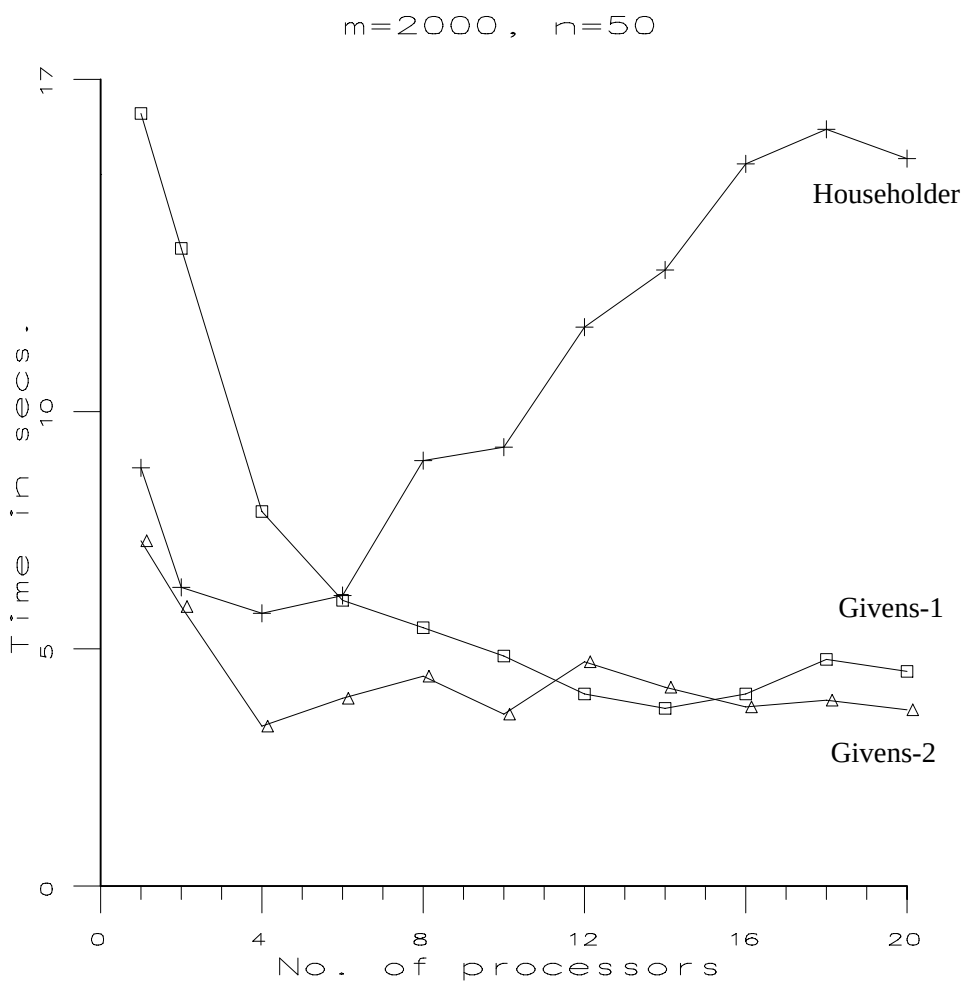

Figure 7: Comparison of Givens and Householder reductions for LLS: $m = 2000, n = 50$.

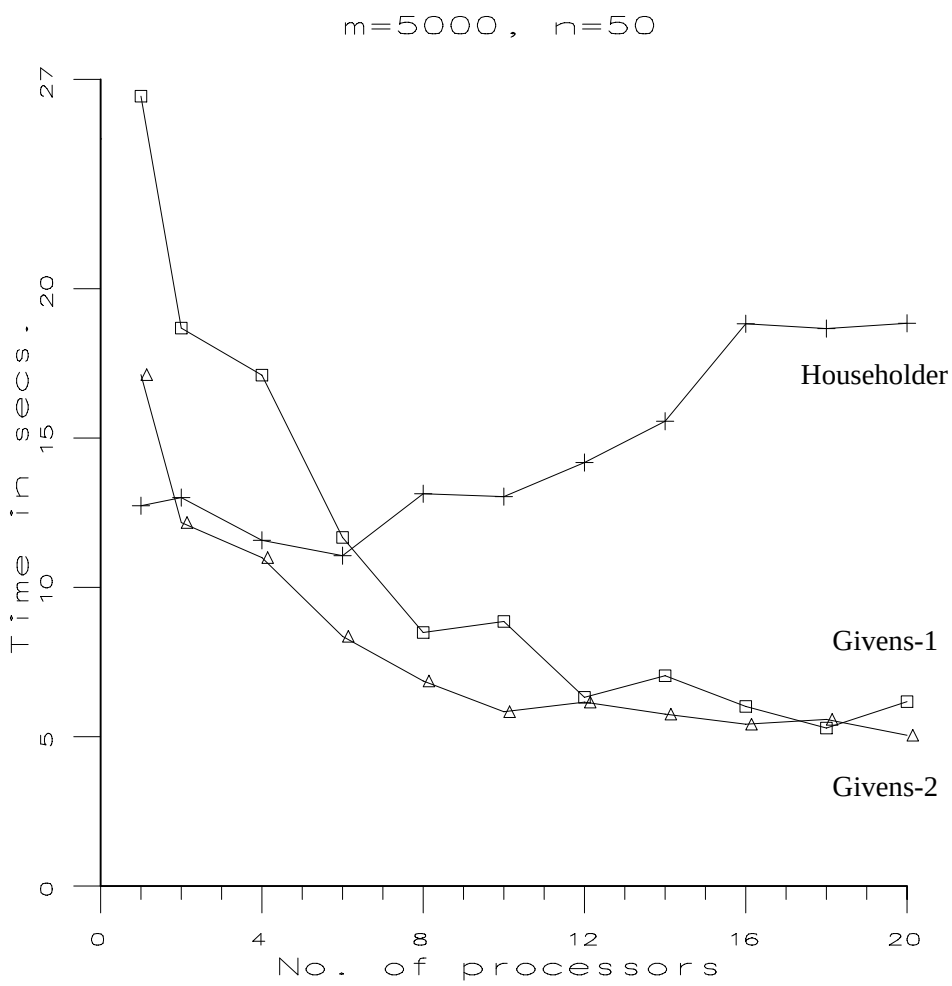

Figure 8: Comparison of Givens and Householder reductions for LLS: $m = 5000, n = 50$.