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 at Santa Barbara
Santa Barbara, CA 93106
omer@cs.ucsb.edu ashok@cs.ucsb.edu

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. Givens rotations enable communication hiding and take greater advantage of parallelism than Householder reflectors, provided the matrices are sufficiently large.

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 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, i.e. $m \gg n$. The unknown x is required to minimize 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.

*Supported in part by the NASA grant NAGW-3888, and UCSB Committee on Research grant 8-562503-19900-7.

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 (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 [6, 7], [11]. 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]. 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 the sequential algorithm of choice for dense QR. However, in a parallel setting this advantage is offset by the higher communication requirements of the Householder reduction algorithm.

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 the back substitution step is not performed as it is identical for both algorithms.

First we present an overview of Givens and fast-Givens transformations in Section 2, and Householder transformations in Section 3. Detailed analyses of these methods can be found in [6], [7], [9], [11], [12]. In Section 4 we present our distributed memory implementation of the fast-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

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 an additional zero can be introduced in the k-th 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. We can define a rotation in the (i, j) plane in $\mathbb{R}^n$ similarly. 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. Each rotation uses $4(n - j)$ multiplications, and a sequence of these can be applied to reduce $A \in \mathbb{R}^{m \times n}$ to upper triangular form using a total of $2n^2(m - n/3)$ multiplications. This can be halved by using *fast-Givens* (or fast-scaled) transformations: 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 $\tau = s/c$. Then

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

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

$$\mu' = s\nu, \quad \nu' = s\mu \ ; \qquad \text{or} \qquad \mu' = c\mu, \quad \nu' = c\nu.$$

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 [7].

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 k-th 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

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

E is an $m \times m$ orthogonal matrix which is 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

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

and $u^T y$ is a scalar. 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 successively introduces zeros in subdiagonal entries of columns of A and reduces A to upper triangular form, using a total of about $mn^2 - n^3/3$ multiplications.

4 A distributed memory fast-Givens implementation

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 [12]. 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 Givens rotations:

1. Concurrent operations act on different rows. In other words, if $T(j,k) = T(j',k')$, and $(j,k) \neq (j',k')$, then $\{j, S(j,k)\} \cup \{j', S(j',k')\} = \phi$.
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 given by Sameh and Kuck [10], in which

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

A single Givens rotation can be performed in small constant time with n processors. Therefore the computation time of the overall Givens reduction with 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. For the limited processor case, efficient scheduling for various architectures are discussed in [8]. Figure 1 gives the annihilation pattern $T(j,k)$ used in our implementation of the Givens reduction algorithm when $P = 3$ processors are available for a 10×10 matrix.

	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

Figure 1: Block annihilation pattern for $P = 3$ processors, 10×10 matrix.

4.1 The algorithm

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 1 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 2. The function $rotate(\cdot,\cdot)$ 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.

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. 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.

In Step 2, each process sends its last row to the next 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

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 $m(p-1)/P+1$ (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**
1. Receive modified last row from $p+1$
2. **if** p has more than a single row **then**
i. $rotate(last, j)$
ii. Send last row to process $p+1$
3. $rotate(i, j)$ for all p's interior rows with index $i > j$
4. **if** index of first row $> j$ **then**
i. Receive last row of process $p-1$
ii. $rotate(first, j)$
iii. Send $p-1$'s modified last row back to it
iv. **if** p has only one row **then** send it to process $p+1$
5. Scale entries of D corresponding to the block
Step 4: Multiply partial matrix by diagonal elements and output result.

Figure 2: Distributed memory fast-Givens.

$p+1$ has used it in order to annihilate column j. Step (b)(ii) ensures the former, and step (d)(iii) the latter. Finally, Step 4 is evident.

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 $mn^2/P + 2T_1Pn + 2T_2Pn^2$ for $m/P \gg n$.

5 Comparison with Householder Reductions

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 [4, 5]. ScaLAPACK uses block-cyclic data distribution as its primary data decomposition method, and uses Householder transformations for QR factorization. The characteristics and the performance of various ScaLAPACK factorization routines, including QR factorization on the Intel family of parallel computers is reported in [5].

5.1 Test platform

The parallel fast-Givens algorithm was compared with the QR factorization routine of ScaLAPACK. PVM [3] 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 for 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 [13] 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 as high as the number of columns.

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 3, 4, and 5.

In our initial implementation of the distributed fast-Givens transformations, we performed scaling of D using floating point arithmetic. The results are reported as Givens-1 in Figures 3, 4, and 5. Since division and multiplication by 2 are integer operations on the exponent, in another version of the implementation we used the C library function *ldexp* for scaling. The resulting faster algorithm is reported as Givens-2.

5.2 Test results

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 $P \leq 20$, probably due to the high communication overhead. Similarly for the parallel fast-Givens algorithm, while there was a slight speedup with increasing P, it was not significant for the smaller matrix (Figure 3). 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 4. For the 5000×50 case shown in Figure 5, the Householder algorithm outperformed the fast-Givens implementation for small number of processors only. After about $P = 6$, fast-Givens 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. In fact 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.

5.3 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 parallel time. 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.

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 should be much lower than for the Ethernet delays that we have observed.

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] 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.

[5] 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.

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

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

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

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

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

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

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

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

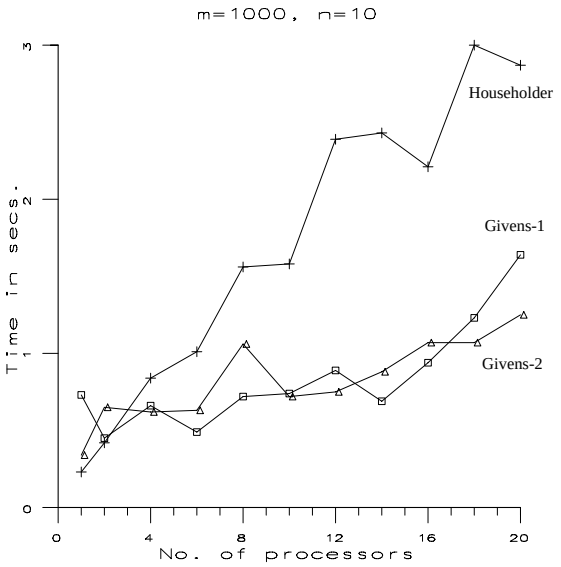

Figure 3: Comparison of Givens and Householder reductions for LLS: $m = 1000, n = 10$.

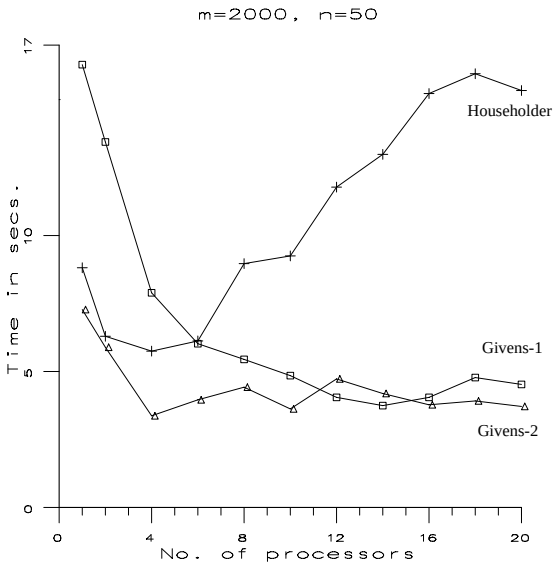

Figure 4: Comparison of Givens and Householder reductions for LLS: $m = 2000, n = 50$.

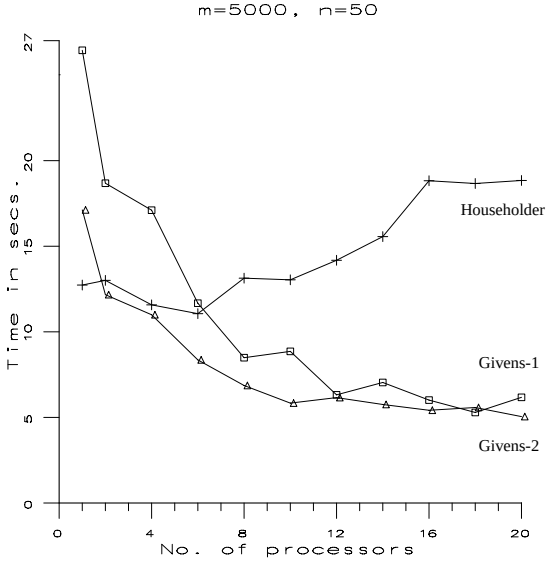

Figure 5: Comparison of Givens and Householder reductions for LLS: $m = 5000, n = 50$.