32

Parallel Computing 16 (1990) 99–112
North-Holland

Practical aspects and experiences

# Parallel QR factorization by Householder and modified Gram-Schmidt algorithms

Dianne P. O'LEARY [†]

*Computer Science Department and Institute for Advanced Computer Studies*

Peter WHITMAN [†]

*Computer Science Department University of Maryland, College Park, MD 20742, USA*

Received October 1988

**Abstract.** In this paper, the parallel implementation of two algorithms for forming a QR factorization of a matrix is studied. We propose parallel algorithms for the modified Gram-Schmidt and the Householder algorithms on message passing systems in which the matrix is distributed by blocks of rows. The models that predict performance of the algorithms are validated by experimental results on several parallel machines.

**Keywords.** QR factorization, Gram-Schmidt algorithm, Householder algorithm, Message passing systems.

## 1. Introduction

The programming of parallel algorithms can be undertaken from either a global or a local approach. In the global approach we begin with a sequential algorithm and look for tasks which can be executed concurrently, such as those in a loop. The approach adopted by O'Leary and Stewart [19] is a local one, writing algorithms from the point of view of transformations on a particular matrix element or matrix block. The execution of such an algorithm with a particular set of data is performed by a *computational node*. Such nodes can communicate with a simple protocol of *send* and *request*, pausing until all requests for information have been satisfied. A network of such nodes can be distributed across a set of communicating processors in a parallel machine and the network can be proven to operate deterministically, regardless of the particular assignment of nodes to processors and regardless of the particular scheduling algorithm [20]. Further, by analyzing the graph describing the flow of data among the computational nodes, we can determine ways to assign nodes to processors in order to maximize the speed-up, and can calculate lower bounds on the execution times of the algorithm on a multiprocessor [18].

[†] This work was supported by the Air Force Office of Scientific Research under Grants 82-0078 and 87-0158.

In this paper we use this formalism to analyze medium grained parallel algorithms for computing the QR factorization of a matrix using either Householder transformations or the modified Gram-Schmidt algorithm. This continues work described in [18].

There are three popular algorithms for computing the QR factors of a matrix: Givens transformations, Householder transformations, and Gram-Schmidt orthogonalization. It is relatively easy to devise an effective implementation of any of these algorithms on a parallel machine that permits efficient execution of *fine-grained* algorithms; in such a machine, the cost of sharing short messages between two computational nodes is roughly equivalent to the cost of a floating point computation. Unfortunately, unless a parallel computer has shared memory or systolic synchronization, the overhead for passing messages between computational nodes is quite high, and fine-grained algorithms are not well matched to currently available message passing architectures. Thus, it is important to develop efficient *medium-grained* algorithms, in which both the amount of computation between bursts of communication, and the length of the messages, is relatively high.

We study medium grained algorithms for computing a factorization of a given $m \times n$ matrix $A$ into the product of an orthogonal $m \times n$ matrix $Q$ and a right triangular $n \times n$ matrix $R$. Often the matrix $Q$ must be computed explicitly, either because an orthogonal basis for the column space of $A$ is required, or because of efficiency considerations in the later use of $Q$ in the parallel machine, and we focus in particular on problems in which the number of rows $m$ is much larger than the number of columns $n$. We consider both Householder and modified Gram-Schmidt algorithms. Our primary consideration will be the time taken by the algorithm. Of course, criteria other than execution time should influence the choice of algorithm. For example, Björck [2] has shown that $Q^TQ$ can differ from the identity by a matrix of size proportional to machine precision when the Householder algorithm is used, but by a matrix of size proportional to machine precision times the condition number of the original matrix when the modified Gram-Schmidt algorithm is used. The choice of algorithm must be guided by the computer architecture and by the intended use for the matrix factors.

Algorithms for QR factorizations on parallel processing machines have been studied in many contexts. A survey is given in [6]. Systolic algorithms based on Givens rotations are given in Barlow and Ipsen [1], Bojanczyk [3], Heller and Ipsen [11], Ipsen [12], and Luk [15]. Givens algorithms on shared memory machines are studied in Sameh and Kuck [25], Cosnard, Muller and Robert [7], Dongarra, Sameh, and Sorensen [8], Modi and Clarke [16] and Lord [14]. Givens algorithms for distributed memory machines are given in Elden [9] for one and two dimensional meshes and the torus, Pothen and Raghavan [22] for ring and broadcast communication, and in Chamberlain and Powell [4], Pothen, Somesh, and Vemulapati [23], and Chu and George [5] for a hypercube. Algorithms based on Householder reflections are given in Dongarra, Sameh, and Sorensen [8] and Katholi and Suter [13] for shared memories, in Eldén [9] for one and two dimensional meshes and the torus, and in Pothen and Raghavan [22] for column partitioning with ring or broadcast communication. Parallel implementation of the modified Gram-Schmidt algorithm does not seem to have been studied.

In Section 2 we will discuss parallel algorithms for the modified Gram-Schmidt and QR factorization algorithms when the matrix is distributed among processors by blocks of rows. In Section 3 we will present experimental and simulation results validating the models of execution time. In Section 4 we will summarize the results and present conclusions.

## 2. QR factorization algorithms

We have two major algorithmic constraints. Since the $m \times n$ matrix $Q$ is to be computed explicitly, methods which form an implicit version of the complete $m \times m$ orthogonal matrix are at a disadvantage. Since the problems have many more rows than columns, partitioning the matrix by columns permits only limited parallelism, and we focus in this paper on algorithms for partitionings of the matrix into blocks of rows. We will assume that we have $p$ processors, each containing the matrix elements from $b$ rows of the matrix $A$. For simplicity we will assume that $p$ divides $m$, so that $b = m/p$; the results are not essentially different for the case that some processors have one more row than others.

We use a very simple model of execution time for parallel algorithms. We model floating point computations by defining the time necessary to execute a statement such as

$$y[i] = a*x[i] + y[i]$$

to be a unit known as a *flop*. We will count a floating point addition, division, or multiplication as half of a *flop*, realizing that this may or may not be a good model for a given machine. We model communication time by saying that each message requires some start-up time $\sigma$ for packaging the message, getting another processor's attention, etc., and some per-word time $\tau$. We denote the sending of a message of length $k$ between two neighboring processors by *send*$(k)$, and allot it time $\sigma + k\tau$.

## 2.1 The parallel modified Gram-Schmidt algorithm

The Gram-Schmidt algorithm constructs an orthogonal set of vectors $q_i$ from the columns $a_i$ of the given matrix through an iterative process of considering each vector in turn, subtracting the component of the vector in the direction of each previous vector, and normalizing the remainder to length one. The modified Gram-Schmidt algorithm, which we consider here, rearranges these operations in order to improve stability. Each vector in turn is normalized to length one, and the subsequent vectors are updated by subtracting a multiple of the given vector in order to produce orthogonality. For a discussion of the stability of the two algorithms, see Björck [2] and Golub [10].

The $i$th step of the modified Gram-Schmidt algorithm is

$$r_{ij} = \frac{a_i^{\mathrm{T}} a_j}{\sqrt{a_i^{\mathrm{T}} a_i}}, \quad j = i, \ldots, n,$$

$$a_j \leftarrow a_j - r_{ij} a_i, \quad j = i+1, \ldots, n,$$

$$a_i \leftarrow a_i / r_{ii},$$

where $a_i$ is column $i$ of matrix $A$. The final matrix $A$ is $Q$; the elements of $R$ are the multipliers $r_{ij}$.

In the parallel algorithm, the norm of each vector and its inner product with subsequent vectors must be computed through a process of locally accumulating the inner products for the blocks, accumulating the inner products across processors, making the inner products known globally, and then doing the local modifications of the vector blocks.

From the point of view of each individual processor, the $i$th step of the algorithm $(i = 1, \ldots, n)$ is as follows:

DOTPRD $[i]$: Compute the dot products of the local block of column $i$ with columns $i, \ldots, n$. This consists of $n - i + 1$ dot products of length $b$. TIME $= (n - i + 1)b$ *flops*.

ACCUM $[i]$: Wait for the $n - i + 1$ dot products from a 'preceding' processor, if there is one, and add on the local dot products. TIME $= (n - i + 1)$ *adds*.

PASS $[i]$: Send the revised $n - i + 1$ dot products to a 'successor' processor, if there is one. TIME $= 1$ *send*$(n - i + 1)$.

SCALE $[i]$: The last processor takes a square root to get the norm of the $i$th column, and scales the other dot products by that norm. TIME $= 1$ *sqrt* $+ 1$ *division* $+ (n - i)$ *multiplications*.

| | | | |
|---|---|---|---|
| *DOTPRD [1] | DOTPRD [1] | DOTPRD [1] | DOTPRD [1] |
| *PASS [1] | ← | | |
| | *ACCUM [1] | | |
| | *PASS [1] | ← | |
| | | *ACCUM [1] | |
| | | *PASS [1] | ← |
| | | | *ACCUM [1] |
| | | | *SCALE [1] |
| ← | | | *BRDCAST[1] |
| *BRDCAST[1] | ← | | UPDATE [1] |
| *UPDATE [1] | BRDCAST[1] | ← | DOTPRD [2] |
| *DOTPRD [2] | UPDATE [1] | UPDATE [1] | |
| | DOTPRD [2] | DOTPRD [2] | |
| *PASS [2] | ← | | |
| | *ACCUM [2] | | |
| | *PASS [2] | ← | |
| | | *ACCUM [2] | |
| | | *PASS [2] | ← |
| | | | *ACCUM [2] |
| | | | *SCALE [2] |
| ← | | | *BRDCAST[2] |
| *BRDCAST[2] | ← | | UPDATE [2] |
| *UPDATE [2] | BRDCAST[2] | ← | DOTPRD [3] |
| *DOTPRD [3] | UPDATE [2] | UPDATE [2] | |
| | DOTPRD [3] | DOTPRD [3] | |
| *PASS [3] | ← | | |
| | *ACCUM [3] | | |
| | *PASS [3] | ← | |
| | | *ACCUM [3] | |
| | | *PASS [3] | ← |
| | | | *ACCUM [3] |
| | | | *SCALE [3] |
| ← | | | *BRDCAST[3] |
| *BRDCAST[3] | ← | | UPDATE [3] |
| UPDATE [3] | *BRDCAST[3] | ← | |
| | UPDATE [3] | *UPDATE [3] | |

Fig. 1. Time course for modified Gram-Schmidt algorithm on a ring of processors. Each column gives the tasks executed by a different processor. If a processor cannot proceed until completion of a task on a neighboring processor, this is indicated by an arrow pointing to that task.

BRDCAST[$i$]: Wait for the $n-i+1$ accumulated dot products from a 'preceding' processor, if there is one, and pass them on to a 'successor' processor, if there is one. TIME = 1 *send*$(n-i+1)$.

UPDATE [$i$]: Scale the local block of column $i$ by the norm of the $i$th vector, and update the local block of columns $i+1$ through $n$ by subtracting from them multiples of the local block of column $i$ (determined by the dot products) to make those columns orthogonal to column $i$. TIME = $(n-i)b$ *flops* + $b$ *multiplications*.

The times above ignore wait times. In order to find the elapsed time for the algorithm, we must find a critical path of steps through the algorithm which must be performed sequentially, and add the times for these sequential steps to the wait times.

From the local point of view, a processor will fall into a pattern of performing BRDCAST[$i$], UPDATE [$i$], and DOTPRD [$i+1$], and then wait before ACCUM [$i+1$] and PASS [$i+1$], and wait again before BRDCAST[$i+1$]. The length of the wait will depend on the connectivity of the processors: if the processor network is a ring, then the processor which is the $k$th to receive the dot products will wait for $k-1$ processors preceding it to perform PASS [$i+1$] before its own ACCUM [$i+1$], and then will wait for $p-k-1$ processors after it to perform PASS [$i+1$] and the $k$ preceding processors to perform BRDCAST[$i+1$] before its own BRDCAST[$i+1$]. As an example, the time course for $n=3$ $p=4$ would be as indicated in *Fig. 1*. The algorithm could be improved slightly at the cost of increased program complexity by letting the assignment of the processor which performs SCALE [$i$] rotate counterclockwise by one each step.

Processor $p-1$ will be the last to finish, and we have marked a critical time-path through the time course with stars. The elapsed time for three or more processors will be

$$\begin{aligned}\text{TIME}_{GS,ring} = \sum_{i=1}^{n} \{&\text{DOTPRD}\ [i] + (p-1)\ \text{ACCUM}\ [i] + \text{SCALE}\ [i] \\ &+ (p-1)\ \text{PASS}\ [i] + 2\ \text{BRDCAST}[i] + \text{UPDATE}\ [i]\} \\ &+ (p-3)\ \text{BRDCAST}[n].\end{aligned}$$

We can determine the optimal number of processors by minimizing this expression with respect to $p$; the optimal number of processors is proportional to the square root of the number of rows, and the full expression is given in *Fig. 3*.

The time analysis for the execution of the algorithm on a hypercube (or any other architecture in which sums across processors can be accumulated and distributed in $\log_2 p$ steps) is performed in a similar fashion. In this case,

$$\begin{aligned}\text{TIME}_{GS,cube} = \sum_{i=1}^{n} \{&\text{DOTPRD}\ [i] + (\log_2 p - 1)\ \text{ACCUM}\ [i] + \text{SCALE}\ [i] \\ &+ \log_2 p\ \text{PASS}\ [i] + \log_2 p\ \text{BRDCAST}[i] + \text{UPDATE}\ [i]\}.\end{aligned}$$

The optimal number of processors is proportional to the number of rows $m$, and the full expressions are given in *Fig. 4*.

## *2.2 The parallel Householder algorithm*

The Householder factorization algorithm transforms the matrix $A$ by reducing one column at a time into upper triangular form. The $i$th step is defined by

$$\begin{aligned}&u_i = a_i + \gamma_i e_i, \\ &\gamma_i = sign(a_{ii})\sqrt{a_{i,i}^2 + \cdots + a_{i,n}^2}, \\ &\pi_i = \frac{1}{(a_{ii}+\gamma_i)\gamma_i}, \\ &A \leftarrow (I - \pi_i u_i u_i^{\mathrm{T}})A.\end{aligned}$$

The final transformed matrix $A$ is $R$.

The description of the parallel algorithm is complicated by the fact that at step $i$, only rows $i$ through $m$ of the matrix are active, so different processors may have different numbers of active rows, ranging from 0 to $b$. We describe the decomposition algorithm from the point of view of an individual processor. Processors which contain only rows numbered less than $i$ do not participate in step $i$. The $i$th step ($i=1,\dots,n$) is as follows:

DOTPRD [$i$]: Compute the dot products of the local block of column $i$ with columns $i,\dots,n$. Only use elements that are in matrix rows $i$ through $m$. TIME $= (n-i+1)\alpha$ *flops*, where $\alpha$ is an integer between 0 and $b$, depending on the step $i$ and the index of the processor.

ACCUM $[i]$: Wait for the $n-i+1$ dot products from a 'successor' processor, if there is one, and add on the local dot products. TIME $=(n-i+1)$ *adds*.

PASS $[i]$: Send the revised $n-i+1$ dot products to a 'preceding' processor, if there is one. TIME $=1$ *send*$(n-i+1)$.

SCALE $[i]$: The processor which contains $a_{ii}$ is the last to receive the revised dot products. It takes a square root to determine the scale factor $\gamma_i$, adds it to the main diagonal element, corrects the inner products for the addition of $\gamma_i$ to the $i$th element of $u_i$, and multiplies them by $\pi_i$. TIME $=1$ *sqrt* $+1$ *division* $+1$ *addition* $+(n-i+1)$ *flops* $+(n-i)$ *multiplications*.

BRDCAST$[i]$: Wait for the $n-i+1$ scale factors from a 'preceding' processor, if there is one, and pass them on to a 'successor' processor, if there is one. TIME $=1$ *send*$(n-i+1)$.

UPDATE $[i]$: Update the local block of columns $i$ through $n$ by subtracting multiples of the local block of column $i$. TIME $=(n-i)\alpha$ *flops*.

Note that this algorithm forms $u_i^{\mathrm{T}} A$ by computing $a_i^{\mathrm{T}} A$ and then adding $\gamma_i e_i^{\mathrm{T}} A$. So far we have computed an upper triangular matrix $R$ and vectors $u_i$ which determine the matrix $Q$ by the relation

$$(Q, Q_{\perp})=\left(I-\pi_1 u_1 u_1^{\mathrm{T}}\right)\cdots\left(I-\pi_n u_n u_n^{\mathrm{T}}\right),$$

and for many applications this is sufficient. In this paper, however, we are considering the problem of computing the matrix $Q$ explicitly, so a second phase of the algorithm is needed to construct $Q$. Since we are looking for the first $n$ columns of the product matrix, the product must be accumulated from right to left, unless we compute the entire matrix and throw away the last $m-n$ columns. This right to left accumulation means that the construction of $Q$ cannot be overlapped with the computation of $R$, and we will see that this essentially doubles the cost.

The algorithm for constructing $Q$ begins with an array of zeroes, with a main diagonal of ones, distributed among the processors. Step $i$ of the algorithm updates $Q$ by computing

$$Q_{i+1} \leftarrow Q_i-\pi_{n-i} u_{n-i}\left(u_{n-i}^{\mathrm{T}} Q_i\right), \quad i=0, \ldots, n-1.$$

Since the vectors $u_{n-i}$ have zeroes for their first $n-i+1$ entries, at step $i$, only matrix entries in rows $n-i$ through $m$ and columns $n-i$ through $n$ can change. Thus, only processors containing rows $n-i$ through $m$ need participate in the $i$th step. Further, the first $n-i$ columns of $Q_i$ are unit vectors, so the operations involving column $n-i$ are particularly simple. The operations are as follows, for $i=0, \ldots, n-1$:

DOTPRD $[i]'$: Compute the dot products of the local block of $u_{n-i}$ with the local block of the last $i$ columns of the current $Q$. TIME $=\alpha\beta$ *flops*, where $\alpha$ ranges between 1 and $b$ and $\beta$ ranges between 0 and $n$, depending on the step $i$ and the index of the processor.

ACCUM $[i]'$: Wait for the $i$ dot products from a 'preceding' processor, if there is one, and add on the local dot products. TIME $=i$ *adds*.

PASS $[i]'$: Send the revised $i$ dot products to a 'successor' processor, if there is one. TIME $=$ *send*$(i)$.

SCALE $[i]'$: The last processor to receive the dot products scales them (since the processors actually store an unnormalized multiple of $u_{n-i}$). TIME $=1$ *division* $+\, i$ *multiplications*.

BRDCAST$[i]'$: Wait for the $i+1$ scaled dot products from a 'preceding' processor, if there is one, and pass them on to a 'successor' processor, if there is one. TIME $=$ *send*$(i+1)$.

UPDATE $[i]'$: Add the multiples of the local block of $u_{n-i}$ to the local block of the last $i+1$ columns of $Q$. TIME $=\alpha\beta$ *flops*.

As an example, the time course for the first three column operations for computing $R$ on a ring of 4 processors with two rows each is given in *Fig. 2*. We assume here that the matrix has

| | | | |
|---|---|---|---|
| DOTPRD [1] | DOTPRD [1] | DOTPRD [1] | *DOTPRD [1] |
| | | → | *PASS [1] |
| | | *ACCUM [1] | |
| | → | *PASS [1] | |
| | *ACCUM [1] | | |
| → | *PASS [1] | | |
| *ACCUM [1] | | | |
| *SCALE [1] | | | |
| *BRDCAST[1] | ← | | |
| UPDATE [1] | *BRDCAST[1] | ← | |
| DOTPRD [2] | UPDATE [1] | *BRDCAST[1] | ← |
| | DOTPRD [2] | UPDATE [1] | *UPDATE [1] |
| | | DOTPRD [2] | *DOTPRD [2] |
| | | → | *PASS [2] |
| | | *ACCUM [2] | |
| | → | *PASS [2] | |
| | *ACCUM [2] | | |
| → | *PASS [2] | | |
| *ACCUM [2] | | | |
| *SCALE [2] | | | |
| *BRDCAST[2] | ← | | |
| UPDATE [2] | *BRDCAST[2] | ← | |
| | UPDATE [2] | *BRDCAST[2] | ← |
| | DOTPRD [3] | UPDATE [2] | *UPDATE [2] |
| | | DOTPRD [3] | *DOTPRD [3] |
| | | → | *PASS [3] |
| | | *ACCUM [3] | |
| | → | *PASS [3] | |
| | *ACCUM [3] | | |
| | *SCALE [3] | | |
| | *BRDCAST[3] | ← | |
| | UPDATE [3] | *BRDCAST[3] | ← |
| | . | UPDATE [3] | *UPDATE [3] |
| | . | . | . |
| | . | . | . |
| | | . | . |

Fig. 2. Time course for the first three steps of the Householder algorithm on a ring of four processors with two rows per processor.

been distributed by blocks of rows, so that processor 1 has rows 1, 2,..., $b$, etc. The time course for the accumulation of $Q$ is similar, although successive steps involve more and more processors rather than fewer and fewer.

Processor $p$ will be the last to finish, and we have denoted a critical time-path through the time course by stars. The elapsed time for computing $R$ will be

$$\sum_{i=1}^{n} \{\text{DOTPRD } [i] + (p_{current} - 1) \text{ ACCUM } [i] + \text{SCALE } [i]$$
$$+ (p_{current} - 1) \text{ PASS } [i] + (p_{current} - 1) \text{ BRDCAST}[i] + \text{UPDATE } [i]\},$$

where $p_{current}$ is the current number of active processors. The time to compute $Q$ is expressed

Modified Gram-Schmidt algorithm on ring of processors:

$$\text{TIME} = \frac{n^2p}{4} + \frac{mn^2}{p} + \frac{np}{4} + \frac{mn}{2p} + n\, sqrt$$

$$+ \left\{ \frac{n^2p}{2} + \frac{n^2}{2} + \frac{np}{2} + \frac{n}{2} + p - 3 \right\} \tau + \{ np + n + p - 3 \} \sigma$$

$$p_{opt} = \frac{\sqrt{2m(2n+1)}}{\sqrt{n(1+2\tau) + 2\tau + 4\sigma + 1 + 4(\tau+\sigma)/n}}$$

Householder algorithm with block mapping of rows on a ring of processors:

$$\text{TIME} = \frac{n^2p}{2} + \frac{2mn^2}{p} + \frac{np}{2} + \frac{mn}{p} + \frac{3n^2}{4} + \frac{7n}{4} - \frac{n^3p}{4m} - \frac{n^2p}{4m} + n\, sqrt$$

$$+ \left\{ 2n^2p + 3np - \frac{n^3p}{m} - \frac{3n^2p}{2m} - n^2 - \frac{3n}{2} \right\} \tau + \left\{ 4np - \frac{2n^2p}{m} - 2n \right\} \sigma$$

$$p_{opt} = \frac{\sqrt{2m(2n+1)}}{\sqrt{(1+4\tau)n + 6\tau + 8\sigma + 1 - \frac{2n^2\tau + 3n\tau + 4n\sigma + n^2/2 + n/2}{m}}}$$

Householder algorithm with wrap mapping of rows on a ring of processors:

$$\text{TIME} = \frac{n^2p}{2} + \frac{2mn^2}{p} + \frac{3n^2}{2} + \frac{mn}{p} + \frac{np}{2} + 2n - \frac{n^3}{p} - \frac{n^2}{2p} + n\, sqrt$$

$$+ \{ 2n^2p + 3np - 2n^2 - 3n \} \tau + \{ 4np - 4n \} \sigma$$

$$p_{opt} = \frac{\sqrt{2m(2n+1) - n(2n+1)}}{\sqrt{(1+4\tau)n + 6\tau + 8\sigma + 1}}$$

Fig. 3. Elapsed time and optimal number of processors for the algorithms on a ring of processors. The times are normalized so that the time for a *flop* is one. If the formula for $p_{opt}$ gives a value outside the range $[1, m]$, the optimal number of processors is either 1 or $m$.

similarly as

$$\sum_{i=0}^{n-1} \{ \text{DOTPRD}\,[i]' + (p_{current} - 1)\, \text{ACCUM}\,[i]' + \text{SCALE}\,[i]'$$
$$+ (p_{current} - 1)\, \text{PASS}\,[i]' + (p_{current} - 1)\, \text{BRDCAST}[i]' + \text{UPDATE}\,[i]' \}$$

for a total time given in *Fig. 3*. The number of processors which gives minimal time is proportional to the square root of the number of rows.

The analogous calculations for the hypercube connections (assuming that every BRDCAST$[i]$ and PASS $[i]$ takes the full $\log_2 p$ stages) shows that the number of processors which gives minimal time is proportional to $m$.

An alternate distribution of the data is the *wrap mapping* of rows. Here, each processor contains every $p$-th row of the matrix, so that, for example, processor 1 has rows $1, p+1, 2p+1, \ldots, (b-1)p+1$. Under this mapping, all processors remain active throughout the algorithm as long as $m > n + p$. *Figs. 3* and *4* give the results, and the optimal number of processors is proportional to $\sqrt{m}$ for a ring and $m$ for a hypercube.

Modified Gram-Schmidt algorithm on a hypercube of processors:

$$\text{TIME} = \frac{mn^2}{p} + \frac{mn}{2p} + n\,sqrt + \left\{\frac{n^2}{4} + \frac{n}{4}\right\}\log_2 p$$
$$+ \left\{n^2 + n\right\}\tau \log_2 p + 2n\sigma \log_2 p$$
$$p_{opt} = \frac{2m(2n+1)\log_e 2}{(4\tau+1)n + 4\tau + 8\sigma + 1}$$

Householder algorithm with block mapping of rows on a hypercube of processors:

$$\text{TIME} = \frac{2n^2m}{p} + \frac{mn}{p} + \frac{n^2}{2} + \frac{3n}{2} + n\,sqrt + \frac{1}{2}\left\{n^2 + n\right\}\log_2 p$$
$$+ \left\{2n^2 + 3n\right\}\tau \log_2 p + 4n\sigma \log_2 p$$
$$p_{opt} = \frac{2m(2n+1)\log_e 2}{(1+4\tau)n + 6\tau + 8\sigma + 1}$$

Householder algorithm with wrap mapping of rows on a hypercube of processors:

$$\text{TIME} = \frac{2mn^2}{p} + \frac{mn}{p} + \frac{3n^2}{2} + 2n - \frac{n^3}{p} - \frac{n^2}{2p} + n\,sqrt + \frac{1}{2}(n^2 + n)\log_2 p$$
$$+ \left\{2n^2 + 3n\right\}\tau \log_2 p + 4n\sigma \log_2 p$$
$$p_{opt} = \frac{(2m-n)(2n+1)\log_e 2}{(1+4\tau)n + 6\tau + 8\sigma + 1}$$

Fig. 4. Elapsed time and optimal number of processors for the algorithms on a hypercube of processors. The times are normalized so that the time for a *flop* is one. If the formula for $p_{opt}$ gives a value outside the range $[1, m]$, the optimal number of processors is either 1 or $m$.

If the matrix $Q$ is not explicitly required, then similar calculations show that the high-order terms of arithmetic operations in the Householder algorithm (block or wrap mapping) agree with those of the modified Gram-Schmidt algorithm in *Figs. 3* and *4*. For a ring of processors, the high order terms for communication are twice as large; for a hypercube, they are the same as for modified Gram-Schmidt.

## 3. Experimental results

The algorithms were programmed and tested on several machines: a BBN Butterfly with 128 nodes and software floating point, a BBN Butterfly with 16 nodes using software floating point,

| Machine | *flop* | start-up $\sigma$ | per-word $\tau$ | relative $\sigma$ | relative $\tau$ |
|---|---|---|---|---|---|
| 16-Butterfly, software fp | 852 | 223 | 16 | .26 | .02 |
| 128-Butterfly, software fp | 1096 | 321 | 23 | .29 | .02 |
| 16-Butterfly, hardware fp | 27 | 223 | 16 | 8.26 | .59 |
| 16-McMoB, software fp | 447 | 4232 | 147 | 9.47 | .33 |

Fig. 5. Times (in microseconds) for the machines for floating point operations and communication, and relative communication times when the floating point time is normalized to 1.

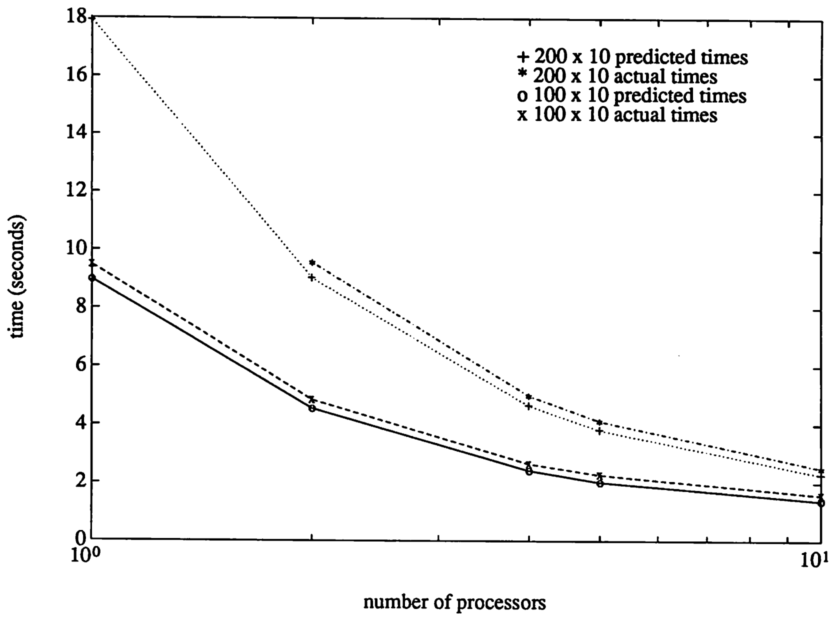

Fig. 6. Results of modified Gram-Schmidt timings on 16-node Butterfly, software floating point.

a BBN Butterfly with 16 nodes using hardware floating point, and the 16 node McMob. The values for the parameters $\sigma$, $\tau$, and the time for one *flop* are given in *Fig. 5*.

The Butterfly is a shared memory machine. Each processor has a direct access path to its own memory module, and there is a $\log_2 p$ level switch between the $p$ processors and the $p$ memory modules to allow access to other modules. McMob is a machine designed and built at the University of Maryland consisting of MC-68000 processors connected by a slotted ring,

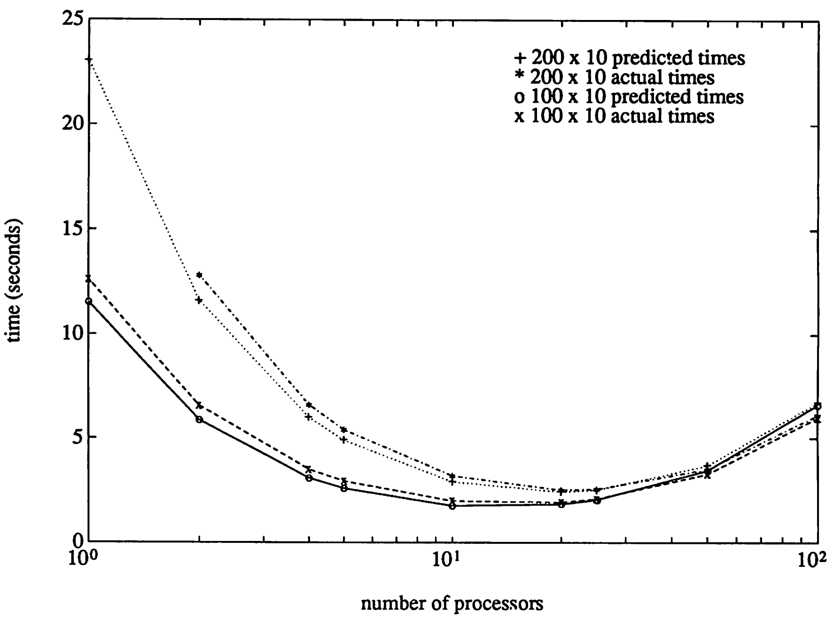

Fig. 7. Results of modified Gram-Schmidt timings on 128-node Butterfly.

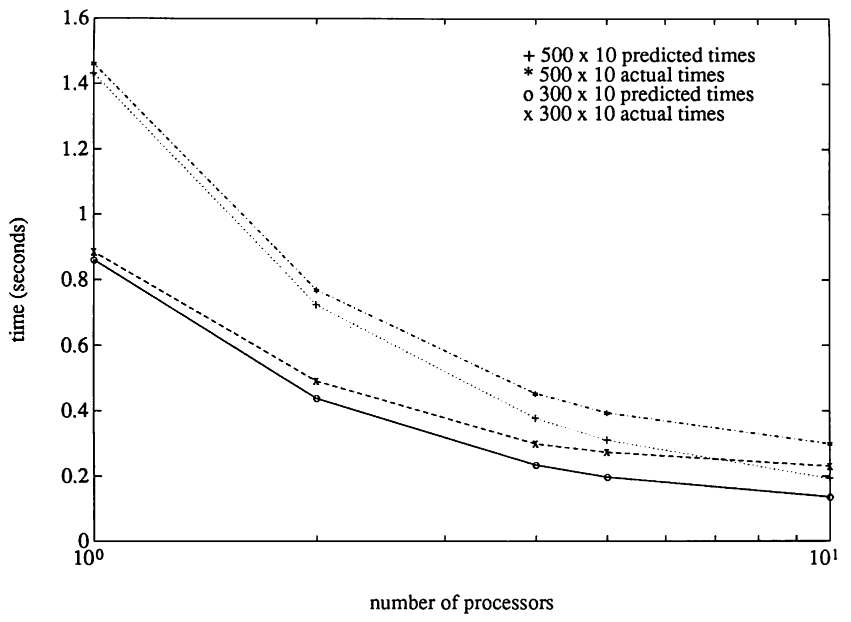

Fig. 8. Results of modified Gram-Schmidt timings on 16-node Butterfly, hardware floating point.

permitting direct communication between any pair of processors without assistance or interference from other processors. See [24] for further details on the architecture.

We did not use the full communication capability of any of the machines. McMob and the Butterflys were programmed using the DOMINO communication system [21] as if they were rings of processors, and the Butterfly memory was used as a message passing medium rather than as a resource shared among processors.

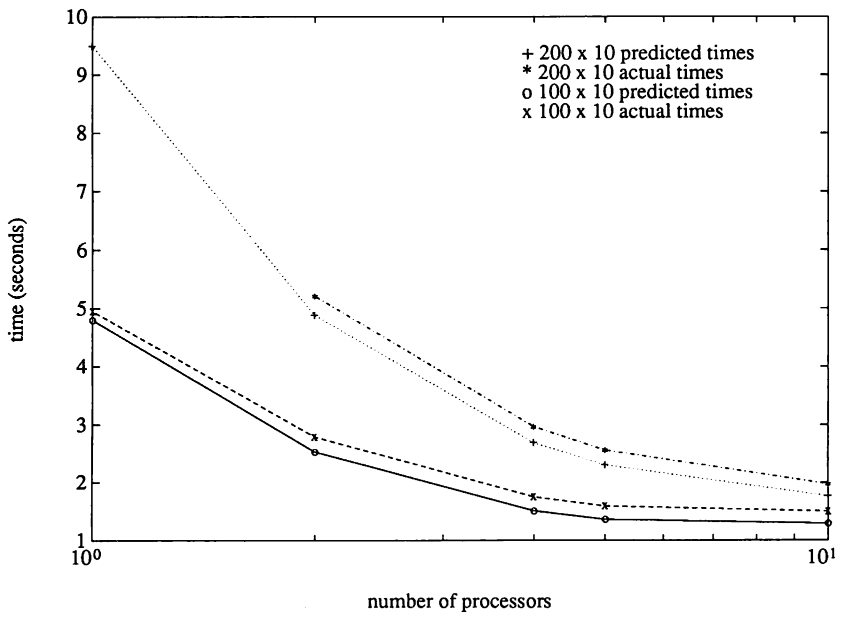

Fig. 9. Results of modified Gram-Schmidt timings on 16-node McMob.

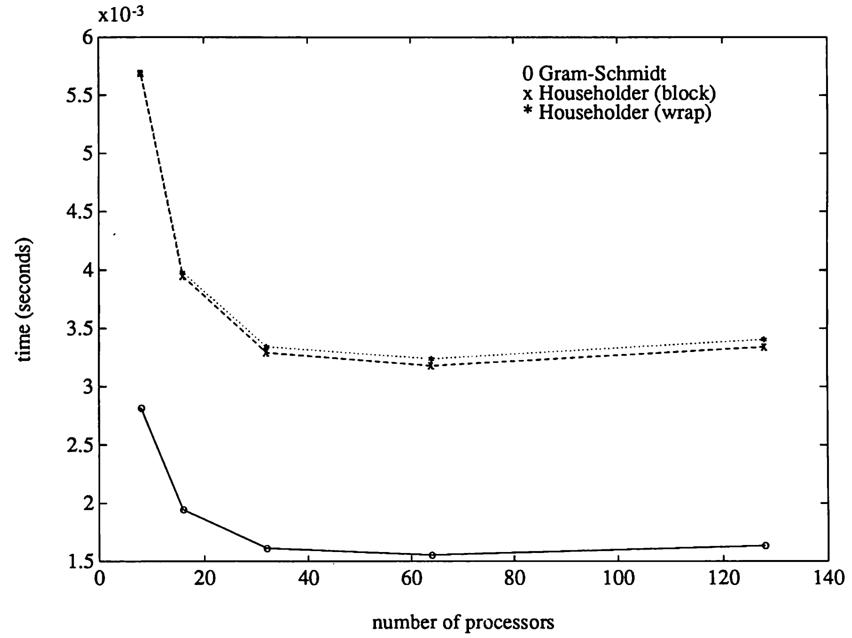

Fig. 10. Model of algorithm performance on hypercubes for a matrix of dimension 256 × 8.

The modified Gram-Schmidt program implemented the algorithm of Section 2.1 except that an axpy $(ax + y)$ operation was used for ACCUM $[i]$, adding $n(n+1)(p-1)/4$ operations. This causes the curves to bend up sooner than they might. The actual times and the results predicted by the model of *Fig. 3*, with the correction for ACCUM $[i]$, are given in *Figs. 6–9*. The actual times are an average of 5–10 runs. The variation in the McMob data was negligible, but the

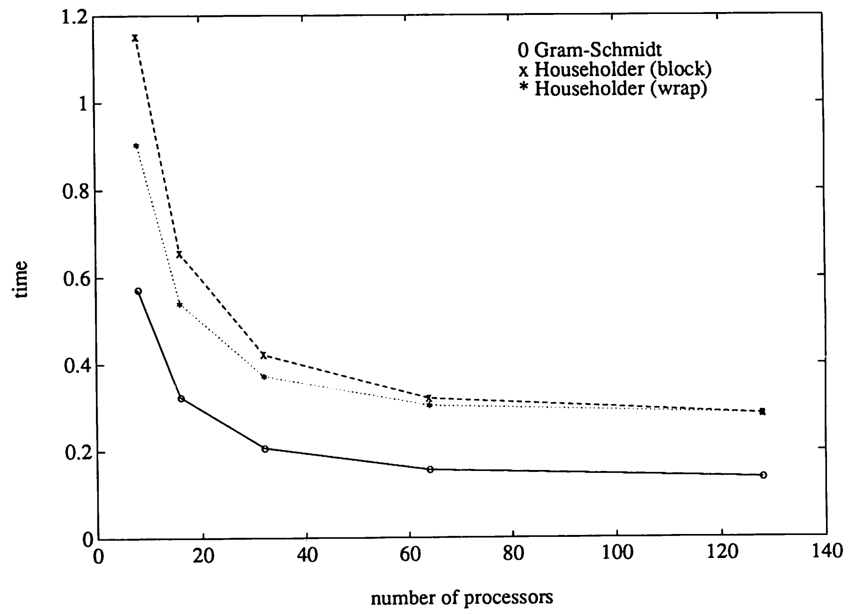

Fig. 11. Model of algorithm performance on hypercubes for a matrix of dimension 256 × 128.

Butterfly timings varied by as much as 10% for two runs using the same configuration of processors and 20% if a different choice of processors was made. On the Butterfly, even if a single processor was used, the reported timings varied by as much as 0.05 sec., which can be attributed to system-level interrupts and variability in the manufacturer-provided timing routines. When multiple processors are used, there is an additional variability due to interference in the switching network for memory accesses. As indicated in the figure captions, most examples were $100 \times 10$ or $200 \times 10$, but larger matrices were used on the hardware floating point Butterfly in order to reduce the variability in the timings.

We also implemented a Householder algorithm on a ring of processors, and results were as predicted by the corresponding model.

*Figs. 10* and *11* give predictions of the relative performance of the modified Gram-Schmidt and Householder algorithms on hypercubes using the models of *Fig. 4* for matrices of size $256 \times 8$ and $256 \times 128$. For these figures, the floating point time was taken to be 1, the relative start-up time was $\sigma = 10$, and the transmission time was $\tau = 0.5$. The block-mapped algorithm is faster than the wrapped-mapped algorithm for $p > n$ and slower for $p < n$ regardless of the machine parameters. As the number of rows in the matrix increases, keeping the number of columns and the number of processors constant, the two mappings for the Householder algorithm give results that are indistinguishable. Recall, however, that the model neglects the savings in ACCUM $[i]$ and BRDCAST$[i]$ as fewer processors remain active in each algorithm.

## 4. Conclusions

Models for the execution time of a modified Gram-Schmidt and a Householder algorithm for computing the QR factorization of rectangular matrices have been presented and validated. If an explicit representation of the $Q$ matrix is needed, then the modified Gram-Schmidt algorithm can be much more efficient, and in any case is not slower. The choice of algorithm for a given application must also be based on stability properties of the two algorithms and the intended application.

## References

[1] J.L. Barlow and I.C.F. Ipsen, Parallel scaled Givens rotations for the solution of linear least squares problems, *SIAM J. Sci. Stat. Comput.* 8 (1987) 716–733.

[2] Å. Björck, Solving linear least squares problems by Gram-Schmidt orthogonalization, *BIT* 7 (1967) 1–21.

[3] A. Bojanczyk, R.P. Brent and H.T. Kung, Numerically stable solution of dense systems of linear equations using mesh-connected processors, *SIAM J. Sci. Stat. Comput.* 5 (1984) 95–104.

[4] R.M. Chamberlain and M.J.D. Powell, QR factorization for linear least squares problems on the hypercube, Technical Report CCC 86/10, Chr. Michelsen Institute, Bergen, Norway, 1986.

[5] E. Chu and A. George, QR factorization of a dense matrix on a hypercube multiprocessor, *Parallel Comput.* 11 (1989) 55–71.

[6] M. Cosnard, M. Daoudi, J.M. Muller and Y. Robert, On parallel and systolic Givens factorizations of dense matrices, in: M. Cosnard et al., eds., *Parallel Algorithms and Architectures*, (Elsevier Science Publishers B.V., North-Holland, 1986) 245–258.

[7] M. Cosnard, J. Muller and Y. Robert, Parallel QR decomposition of a rectangular matrix, *Numer. Math.* 48 (1986) 239–249.

[8] J.J. Dongarra, A.H. Sameh and D.C. Sorensen, Implementation of some concurrent algorithms for matrix factorization, *Parallel Comput.* 3 (1986) 25–34.

[9] Lars Eldén, A parallel QR decomposition algorithm, Technical Report LiTH-MAT-R-1988-02, Linköping University, Sweden, 1988.

[10] G.H. Golub, Numerical methods for solving linear least squares problems, *Numer. Math.* 9: 139–148, 1966.

[11] D.E. Heller and I.C.F. Ipsen, Systolic networks for orthogonal decompositions, *SIAM J. Sci. Stat. Comput.* 4 (1983) 261–269.

[12] I.C.F. Ipsen, A parallel QR method using fast Given's rotations, Technical Report YALEU/DCS/RR-299, Computer Science Dept., Yale Univ., 1984.

[13] C.R. Katholi and B.W. Suter, A parallel algorithm for computing the QR factorization of a rectangular matrix, Technical Report TR-88-07, Dept. of Computer and Information Sciences, University of Alabama at Birmingham, 1988.

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

[15] F.T. Luk, A rotation method for computing the QR-decomposition, *SIAM J. Sci. Stat. Comput.* 7 (1986) 452–459.

[16] J.J. Modi and M.R.B. Clarke, An alternate Givens ordering, *Numer. Math.* 43 (1984) 83–90.

[17] D.P. O'Leary, Fine and medium grained parallel algorithms for matrix QR factorization, in: H.J.J. te Riele, Th.J. Dekker, and H.A. van der Vorst, eds., *Algorithms and Applications on Vector and Parallel Computers* (Elsevier Science Publishers B.V., North-Holland, 1987) 347–349.

[18] D.P. O'Leary and G.W. Stewart, Assignment and scheduling in parallel matrix factorization, *Linear Algebra Appl.* 77 (1986) 275–299.

[19] D.P. O'Leary and G.W. Stewart, Dataflow algorithms for parallel matrix computation, *Comm. ACM* 28 (1985) 840–853.

[20] D.P. O'Leary and G.W. Stewart, From determinacy to systaltic arrays, *IEEE Trans. Comput.* C-36 (1987) 1355–1359.

[21] D.P. O'Leary, G.W. Stewart and R. van de Geijn, DOMINO: a message passing environment for parallel computations, Technical Report TR-1648, Computer Science Dept., University of Maryland, 1986.

[22] A. Pothen and P. Raghavan, Distributed orthogonal factorizations: Givens and Householder algorithms, *SIAM J. Sci. Stat. Comput.* 10 (1989) 1113–1134.

[23] A. Pothen, J. Somesh and U. Vemulapati, Orthogonal factorization on a distributed memory multiprocessor, in: M.T. Heath, ed., *Proc. Hypercube Multiprocessors 1987*, SIAM, Philadelphia (1987) 587–596.

[24] Chuck Rieger, Zmob: hardware from a user's viewpoint, in: *Proc. IEEE Computer Society Conf. Pattern Recognition and Image Processing* (1981) 484–521.

[25] A.H. Sameh and D.J. Kuck, On stable parallel linear system solvers, *J. Assoc. Comput. Mach.* 25 (1978) 81–91.