Parallel Computing 5 (1987) 127–139
North-Holland

Parallel implementation of the block conjugate gradient algorithm

Dianne P. O'LEARY *

Computer Science Department and Institute for Advanced Computer Studies, University of Maryland, College Park, MD 20742, U.S.A.

Abstract. The conjugate gradient algorithm is well-suited for vector computation but, because of its many synchronization points and relatively short message packets, is more difficult to implement for parallel computation. In this work we introduce a parallel implementation of the block conjugate gradient algorithm. In this algorithm, we carry a block of vectors along at each iteration, reducing the number of iterations and increasing the length of each message. On machines with relatively costly message passing, this algorithm is a significant improvement over the standard conjugate gradient algorithm.

Keywords. Conjugate gradient algorithm, parallel implementation, message passing architectures.

1. Introduction

During the past 10 years, the conjugate gradient algorithm has become a standard method for the iterative solution of sparse linear systems of equations. Several developments contributed to this; analysis and experimentation identified the most stable versions of the algorithm [17] and [16], produced an understanding of its error propagation [15], and provided effective preconditioning strategies to accelerate convergence of the algorithm on a wide class of applications (e.g., [2,3,5,12]). These ideas had their roots in much earlier work (e.g., [8,22]) but part of the reason for their success at this time was that the preconditioned algorithms were well matched to supercomputing technology (e.g., vector processing) and to several important problem classes (e.g., network problems and discretizations of two- and three-dimensional partial differential equations).

The conjugate gradient algorithm is not as naturally suited for parallel computation as it is for vector computation, in part, because of its many synchronization points and its relatively short, but numerous, message packets. Two techniques can reduce the number of synchronization points and increase the size of the message packets: effective preconditionings and the use of a block of vectors in the iteration. In this paper, we consider the second technique.

In Section 2, we review the mathematics of the block preconditioned conjugate gradient algorithm. Section 3 concerns the parallel implementation of the algorithm. A coarse grained parallel version is introduced for networks of processors communicating by message passing. Some analysis and simulation results for medium grained parallelism on message passing architectures are also presented. Section 4 lists some conclusions and open questions.

* This work was supported by the Air Force Office of Scientific Research under Grant AFOSR-82-0078.

2. The block preconditioned conjugate gradient algorithm

Consider the problem of solving the linear system of equations $AX = B$, where A is an $n \times n$ symmetric matrix. We do not assume that A is positive definite. Suppose that M is an $n \times n$ symmetric positive definite matrix, the preconditioner, which is in some sense a good approximation to A^{-1}. Given and initial vector z_1 satisfying $z_1^T M z_1 = 1$, it is possible to express the solution vector X^* as $X^* = \bar{Z}_k \psi_k$, where ψ_k is a k-vector and $\bar{Z}_k$ satisfies

$$A\bar{Z}_k = \bar{Z}_k \bar{T}_k + \tilde{Z}_k,$$

with $\bar{T}_k$ a tridiagonal matrix and $\tilde{Z}_k$ an $n \times k$ matrix that is nonzero in its last column only. The matrices $\bar{Z}_k$ have M-orthonormal columns:

$$\bar{Z}_k^T M \bar{Z}_k = I,$$

and first column equal to z_1. This construction of $\bar{Z}_k$ is the well-known Lanczos process [10] and has been exploited by Paige and Saunders [16] in solving linear systems.

This algorithm has a weakness (for parallel processing) that there are a great many inner products in relation to the remainder of the computational work. For efficiency, these inner products must be spread among processors, and introduce synchronization points at each iteration of the algorithm. For this reason, the *block* version of the algorithm becomes attractive: fewer iterations will be required if we let each $\bar{Z}_k$ be a block of b n-vectors. $\bar{T}_k$ will then become a matrix with $2b + 1$ nonzero bands. This version of the algorithm was studied independently by O'Leary and Underwood in the late 1970's, although, sadly, Richard Underwood's results were never published due to his death. This work was motivated by work of Cullum and Donath [6], Golub and Underwood [7], and Underwood [21] on the block Lanczos algorithm. More details on the block conjugate gradient algorithm and its properties may be found in [13].

The block version of the algorithm may be considered in two contexts: if b right-hand sides are to be processed, then the algorithm simultaneously produces a solution vector for each one, and probably in less work than applying the single vector algorithm b times. If only one linear system is to be solved, then the $b - 1$ extra vectors are a tool, like the matrix M, intended to reduce the number of iterations.

The block preconditioned form of the conjugate gradient algorithm can be described as follows:

Initialization: Given B and X_0, let $R_0 = B - AX_0$, $\hat{X}_0 = X_0$, $Z_0 = 0$, and $Z_1 = R_0 \nu_1$, where ν_1 is defined so that $Z_1^T M Z_1 = I$. Let $\rho_0 = 0$, $\rho_1 = Z_1^T MAMAMZ_1$, $\bar{L}_{1,1} = \rho_1$, and $V_1 = I_{2b \times 2b}$.

Step k $(k = 1, 2, \ldots)$

1. Form the new set of columns $\tilde{Z}_{k+1}$:

$$\tilde{Z}_{k+1} = AMZ_k - Z_k \rho_k - Z_{k-1} \nu_k^{-T}.$$

Form a 'QR' factorization of $\tilde{Z}_{k+1}$, yielding a set of new columns Z_{k+1} and a triangular set of coefficients ν_{k+1}:

$$Z_{k+1} = \tilde{Z}_{k+1} \nu_{k+1},$$

where $Z_{k+1}^T M Z_{k+1} = I$.

2. Compute

$$AMZ_{k+1} \quad \text{and} \quad \rho_{k+1} = Z_{k+1}^T MAMAMZ_{k+1}.$$

3. Compute two new blocks of L coefficients:

$$\left[L_{k+1,k-1}, \bar{L}_{k+1,k}\right] = \left[0, \nu_{k+1}^{-1}\right] V_k^T.$$

4. Given $\bar{L}_{k,k}$ and $\nu_{k+1}^{-\mathrm{T}}$, compute the QR factorization
$$[\bar{L}_{k,k}, \nu_{k+1}^{-\mathrm{T}}] V_{k+1}^{\mathrm{T}} = [L_{k,k}, 0].$$
5. Compute two new blocks of L coefficients:
$$[L_{k+1,k}, \bar{L}_{k+1,k+1}] = [\bar{L}_{k+1,k}, \rho_{k+1}] V_{k+1}^{\mathrm{T}}.$$
6. Compute
$$[W_k, \bar{W}_k] = [\bar{W}_k, MZ_{k+1}] V_{k+1}^{\mathrm{T}}.$$
7. Compute ψ_k from
$$L_{k,k}\psi_k = -(L_{k,k-2}\psi_{k-2} + L_{k,k-1}\psi_{k-1}).$$
8. Compute the new approximation to the solution from
$$\hat{X}_k = \hat{X}_{k-1} + W_k\psi_k.$$

Upon termination: Determine $\bar{\psi}_{k+1}$ and X_{k+1} from
$$\bar{L}_{k+1,k+1}\bar{\psi}_{k+1} - (L_{k+1,k-1}\psi_{k-1} + L_{k+1,k}\psi_k),$$
$$X_{k+1} = \hat{X}_k + \bar{W}_{k+1}\bar{\psi}_{k+1}.$$

If A is also assumed to be positive definite, then there is a well-known bound for the error at the kth step of the conjugate gradient algorithm (see, for example, [11]). Let the eigenvalues of MA be denoted by $\lambda_n \geq \lambda_{n-1} \geq \cdots \geq \lambda_1$. Then
$$E_k^{\mathrm{T}} A E_k \leq \left((1 - \sqrt{\kappa^{-1}})/(1 + \sqrt{\kappa^{-1}})\right)^{2k} c$$
where $E_k = X_k - X^*$ is the error after k iterations, $\kappa = \lambda_n/\lambda_1$ is the condition number of MA in the 2-norm, and c is a matrix of constants. This result also holds for the block conjugate gradient algorithm for each of the b error vectors [13]. In this case, however, κ is reduced to λ_n/λ_b. For special eigenvalue distributions, tighter convergence bounds can be derived for both algorithms by appealing to the optimal polynomial property of conjugate gradients as in [11].

3. Parallel block preconditioned conjugate gradient algorithms

3.1. Coarse grained algorithms and analysis

Figure 1 is a diagram of the per-iteration work of the block preconditioned conjugate gradient algorithm illustrating its coarse-grained parallelism; if O(1) processors are available, it might be practical to assign subsets of them to given tasks, performing, for example, Steps 2, 3, and 4 in parallel. The updating of the X matrix of solution vectors (Steps 3 through 8) may be largely decoupled from the generation of the Lanczos vectors (Steps 1 and 2), operating on a separate group of processors or in low-priority mode on the Lanczos processors, as long as there is a queuing mechanism for saving the Lanczos vectors until they have contributed to the X update. This coarse-grained algorithm is also well-suited for parallel computers with vector processors, since it preserves long vector operations within a single processor. Standard tricks for forming products of sparse matrices with vectors can be employed, and results can be pipelined between steps such as 6 and 8.

Another way to achieve coarse-grained parallelism is to assign one processor (or a small number of processors) per column of Z, since in general the number of columns is independent

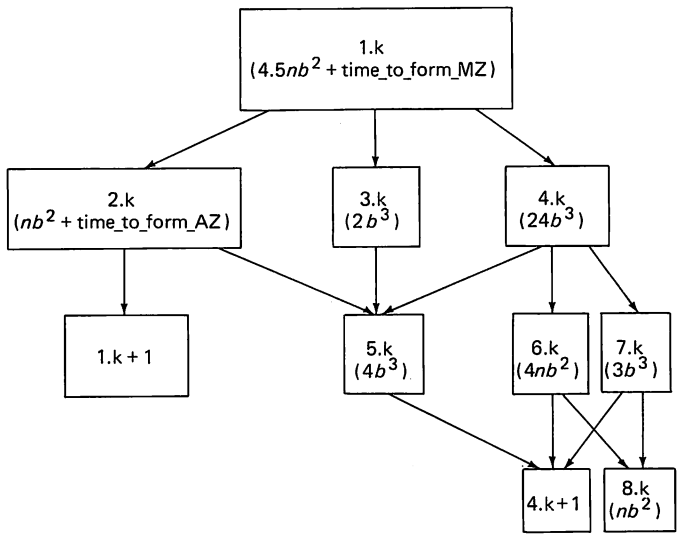

Fig. 1. Coarse-grain parallelism in the block conjugate gradient algorithm (time per step at the kth iteration, and the partial ordering of the steps).

of the size of the matrix. This scheme results in a great deal of inter-processor communication but good load balancing.

3.2. *Medium-grained algorithms and analysis*

If more than a constant number of processors are available, further speedups can be achieved by dividing the work within steps of the block conjugate gradient algorithm. From Fig. 1, we see that the most time-consuming steps of the algorithm are Steps 1 (updating the Lanczos vectors) and 2 (forming A times the block of vectors). If b, the number of vectors, is small, then the next most costly steps are 6 (forming a new block of auxiliary vectors) and 8 (updating the approximate solution vectors). In partitioning the work among a large number of processors, we have two choices:

(1) Partition the Z matrices into pieces containing one or more rows and all of the columns;

(2) Partition the Z matrices into pieces containing elements from one or more rows and columns.

Row partitioning has two advantages. The first is that only the QR factorization of Step 1 and the matrix multiplication and inner product of Step 2 involve communication among processors; all other steps require data which is local to its processor. (Since the L and V matrices are small, we let each processor which is responsible for a block of X update these matrices independently. This means that Steps 3, 4, 5, and 7 are performed by every one of these processors.) The second advantage is that it is significantly easier to program than the row and column partitioning algorithm.

Two orthogonal factorizations need to be performed at each iteration: the first finds an orthogonal basis for the $\tilde{Z}$ vectors (Step 1) and the other updates the factors of the matrix $\bar{T}$ (Step 4). The first of these is difficult for three reasons: the matrix is large ($n \times b$), the orthogonal matrix must be computed explicitly rather than as a product of factors, and it is important that the computed columns of this matrix really be orthogonal. If an algorithm such as Givens or Householder transformations is used for this QR factorization, the elementary matrices must be multiplied together in order to accumulate the first b columns of the Q

matrix, and this cannot be begun until the factorization is complete, unless we are willing to form the full $n \times n$ matrix. This fact effectively doubles the time required for either of these two algorithms. Because of this, the modified Gram–Schmidt algorithm is much faster in this computation. The disadvantage is that the computed columns may not be sufficiently close to orthogonal. Björck [4] 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 $\tilde{Z}$ when the modified Gram–Schmidt algorithm is used. Unfortunately, even the modified Gram–Schmidt algorithm is 'slow' for parallel processing since it requires $b-1$ passes through the matrix, with communication on each pass. The classical Gram–Schmidt algorithm, in which $\tilde{Z}^T\tilde{Z}$ is formed initially, and then subtraction and normalization of columns is done, is much more efficient in parallel computation. An error analysis has been given by Abdelmalek [1]. An alternative, proposed by Stewart [20], is to compute R as a Cholesky factor of $\tilde{Z}^T\tilde{Z}$ and form the rows of Q by solving $\tilde{Z} = QR$. This algorithm shares the efficiency of classical Gram–Schmidt. Both these algorithms and modified Gram–Schmidt will fail to produce a sufficiently orthogonal Q if $\tilde{Z}$ is ill-conditioned, and it is possible to iterate them ('reorthogonalize'). Ill-conditioning, however, indicates that the columns of $\tilde{Z}$ are close to rand-deficient, and we should drop a column and continue the algorithm on the reduced (and well-conditioned) block. The timing counts below assume that the classical Gram–Schmidt algorithm is used and that reorthogonalization is not necessary.

A lower bound on execution time for a parallel algorithm can be determined by the technique of studying a critical path through the computational graph for the algorithm as in [14]. If such a time can actually be achieved by an implementation, then it is optimal. This is the technique used to derive the results of this section.

The analysis of parallel algorithms depends on three parameters: the time for floating-point operations, the time for start-up of a message, and the per-word transmission time for messages. To normalize the results, we will assume that

- the time for a typical floating-point computation (a 'flop', defined to be an addition, a multiplication, and some indexing) is 1,
- start-up time is s,
- per-word transmission time is w.

We will use $p_R \times p_C$ processors, arranged either in a torus or a hypercube. The matrices Z are partitioned into $(n/p_R) \times (b/p_C)$ pieces, one per processor. The $b \times b$ matrices L, $\bar{L}$, and ψ^T, and the four blocks of V are stored in $(b/p_C) \times b$ pieces, with p_R copies of each. The matrices ρ, ρ^T, ν^{-1}, and ν^{-T} are stored in a similar way in $b \times (b/p_C)$ blocks.

On a ring of p processors, the time to cycle p vectors of length x, distributed one vector per processor, among all of the processors is $(p-1)s + (p-1)xw$. If the processors are connected in a hypercube, this can be done in time $2 \log p\, s + 2x(p-1)w/\log p$ (see, for example, [18]). Using these facts, we can calculate the time per step of each conjugate gradient iteration. Table 1 gives these times for matrices partitioned as described above. If $p_C = 1$, then the number of floating-point operations can be reduced as indicated by the numbers in Fig. 1.

There are many variants on implementations of the block conjugate gradient algorithm. The times for the variants differ in low-order terms only. The computation time for one variant is

$$\text{time}_{\text{comp}} = \frac{11nb^2 - nb}{p_R p_C} - \frac{nb^2}{p_R p_C^2} + \frac{51b^3 - 21b^2 + b/3}{2p_C} - \frac{16b^3}{3p_C^2} - \frac{b^3}{6p_C^3}$$

$$+ 4b^3 + 2b^2 + \tfrac{1}{3}b + \text{time\_to\_form\_}Z + \text{time\_to\_form\_}MZ. \qquad (1)$$

The total time per iteration on a torus of processors for one variant is the computation time

Table 1
Overhead time per iteration for the medium grained algorithm

Step	Floating-point operations (flops)	TORUS communication and accumulation time	HYPERCUBE communication and accumulation time
1	$\frac{5nb^2}{p_R p_C} + \frac{nb}{p_R p_C} - \frac{nb^2}{p_R p_C^2}$	$4(p_C-1)s + (p_R-1)s$	$7 \log p_C\, s = 2 \log p_R\, s$
	$+ \frac{3b^3-3b^2+b}{6p_C} - \frac{b^3}{6p_C^3}$	$+4(p_C-1)\frac{nbw}{p_R p_C}$	$+ \frac{8nb(p_C-1)}{p_R p_C \log p_C} w$
	+ time_to_form_MZ	$+(p_R-1)(w+1)\frac{b^2}{p_C}$	$+ \frac{b^2 \log p_R}{p_C}(2w+1)$
		$+(p_C-1)\left(\frac{b^2+b}{2p_C} - \frac{b^2}{2p_C^2}\right) w$	$+\left(\frac{b^2+b}{2p_C} - \frac{b^2}{2p_C^2}\right) w$
2	$\frac{nb^2}{p_R p_C} - \frac{b^2}{p_C}$ + time_to_form_AZ	$(p_C-1)s + (p_R-1)s$	$2 \log p_C\, s + 2 \log p_R\, s$
		$+(p_C-1)\frac{nb}{p_R p_C} w$	$+ \frac{2nb(p_C-1)}{p_R p_C \log p_C} w$
		$+(p_R-1)\frac{b^2}{p_C}(w+1)$	$+ \frac{b^2 \log p_R}{p_C}(2w+1)$
3	$\frac{2b^3}{p_C} - \frac{2b^2}{p_C}$	$(p_C-1)s + 2(p_C-1)\frac{b^2}{p_C} w$	$2 \log p_C\, s + \frac{4b^2(p_C-1)}{p_C \log p_C} w$
4	$\frac{16b^3-2b^2}{p_C} - \frac{16b^3}{3p_C^2}$	$3(p_C-1)s + 8(p_C-1)\frac{4b^2}{p_C} w$	$5 \log p_C\, s + 16\frac{b^2(p_C-1)}{p_C \log p_C} w$
	$+4b^3 + 2b^2 + \frac{1}{3}b$	$+(p_C-1)\left(\frac{2b^2+b}{p_C} - \frac{2b^2}{p_C^2}\right) w$	$+\left(\frac{2b^2+b}{p_C} - \frac{2b^2}{p_C^2}\right) w$
5	$\frac{4b^3}{p_C} - \frac{2b^2}{p_C}$	$(p_C-1)s + 4(p_C-1)\frac{b^2}{p_C} w$	$2 \log p_C\, s + 8\frac{b^2(p_C-1)}{p_C \log p_C} w$
6	$\frac{4nb^2}{p_R p_C} - \frac{2nb}{p_R p_C}$	$(p_C-1)s + 2(p_C-1)\frac{nb}{p_R p_C} w$	$2 \log p_C\, s + \frac{4nb(p_C-1)}{p_R p_C \log p_C} w$
7	$\frac{3b^3}{p_C} - \frac{3b^2}{p_C}$	$2(p_C-1) + 3(p_C-1)\frac{b^2}{p_C} w$	$4 \log p_C\, s + 6\frac{b^2(p_C-1)}{p_C \log p_C} w$
8	$\frac{nb^2}{p_R p_C}$	$(p_C-1)s + (p_C-1)\frac{nb}{p_R p_C} w$	$2 \log p_C\, s + 2\frac{nb(p_C-1)}{p_C p_R \log p_C} w$

above plus the portion of the communication and accumulation time which cannot be overlapped with computation, which is given by

$$\text{time}_{\text{comm}} = \frac{8nbw}{p_R} - \frac{8nbw}{p_R p_C} + \frac{4b^2 p_R + 4b^2 p_R w - 48b^2 w - 3bw - 4b^2}{2p_C} + \frac{5b^2 w}{2p_C^2} + \frac{39b^2 w}{2} + \frac{3bw}{2} + 2p_R s + 14p_C s - 16s, \tag{2}$$

and from this we can determine, given the machine parameters s and w and the problem parameters n and b, the optimal number of processors. We define the *overhead time* to be the

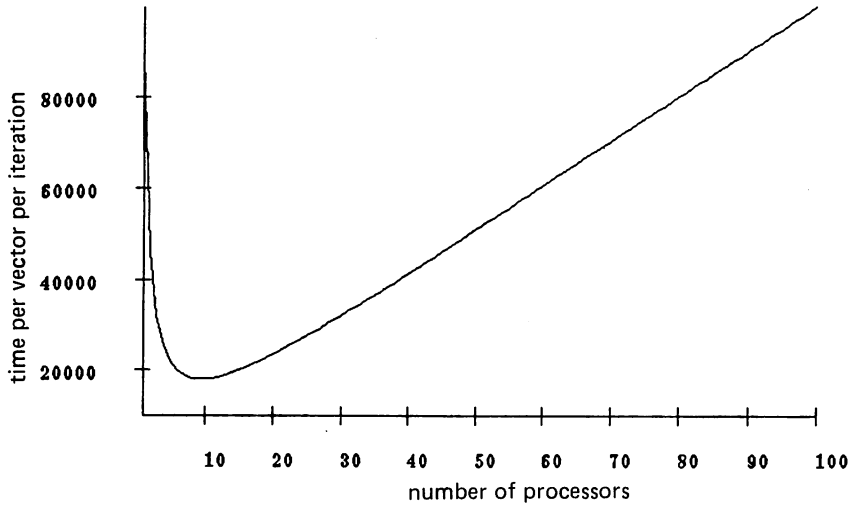

Fig. 2. Time per iteration for a torus of processors; matrix size $n = 10000$, blocksize $b = 1$, start-up time $s = 500$, per-word transmission time $w = 0.01$.

total time, but not including the time to form AZ or MZ. In order to minimize this overhead time per iteration, p_R should be chosen by the rule

$$p_R^2 = \frac{11nb^2p_C - nbp_C - nb^2 + 8nbwp_C^2 - 8nbwp_C}{2b^2p_C + 2b^2wp_C + 2sp_C^2}. \tag{3}$$

Since, in general, the blocksize b will be a small number, and since p_C is bounded above by b, equations (1), (2), and (3) above can be evaluated for all possible p_C values to determine the best. If $p_C = 1$, (3) reduces to

$$p_R^2 = \frac{10nb^2 - nb}{2(b^2 + b^2w + s)},$$

giving a per-iteration time proportional to $\sqrt{n}$. The constant of proportionality can be quite high, however. In Fig. 2 we graph the overhead time per iteration vs. the number of processors for a matrix of size 10000 on a machine with start-up time $s = 500$ and per-word time $w = 0.01$. Only 9 processors can be used effectively, giving a minimal time per iteration of 18030. On a machine with $s = 1$, many more processors can be utilized, as shown in Fig. 3. Here a time per iteration of about 1200 can be achieved with 145 processors. The time per iteration on a single processors is 80013.

Figures 4 and 5 analogous results for the block algorithm with a blocksize of 8. With $s = 500$, 160 processors can be used to give a time per iteration per vector of 13623, much better than the single vector algorithm. At $s = 1$, 1728 processors can be used, but the time is 1462, larger than for the single vector iteration. The time per vector per iteration for this algorithm on a single processor is 711436.

Table 2 illustrates the minimal time per iteration achievable under the constraint that the algorithm be at least 50% efficient, i.e., time per iteration given p processors must be bounded by $2/p$ times the time per iteration for a single processor. As start-up time decreases, the number of usable processors increases to a limit of 183 when $w = 0.01$ and the matrix dimension is 10000. On a smaller problem, $n = 1000$, this limit is 56, as shown in Table 3.

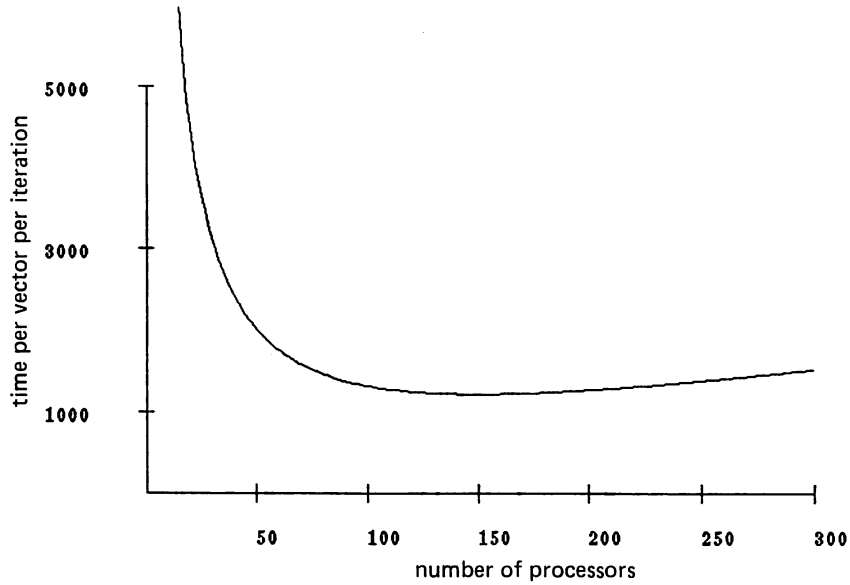

Fig. 3. Time per iteration for a torus of processors; matrix size $n = 10000$, blocksize $b = 1$, start-up time $s = 1$, per-word transmission time $w = 0.01$.

Similar results can be derived for a hypercube of processors. If p_C is a power of two, then communication across processors with the same 'row' index is confined to a hypercube of dimension $\log p_R$, and similarly for the 'column' indices. Communication and accumulation can be performed in time

$$\text{time}_{\text{comm}} = \frac{4b^2 \log p_R\, w}{p_C} - \frac{16nbw}{p_R p_C \log p_C} + \frac{16nbw}{p_R \log p_C} - \frac{34b^2 w}{p_C \log p_C} + \frac{34b^2 w}{\log p_C} + \frac{5b^2 w + 3bw + 4b^2 \log p_R}{2p_C} - \frac{5b^2 w}{2p_C^2} + 4 \log p_R\, s + 26 \log p_C\, s, \qquad (4)$$

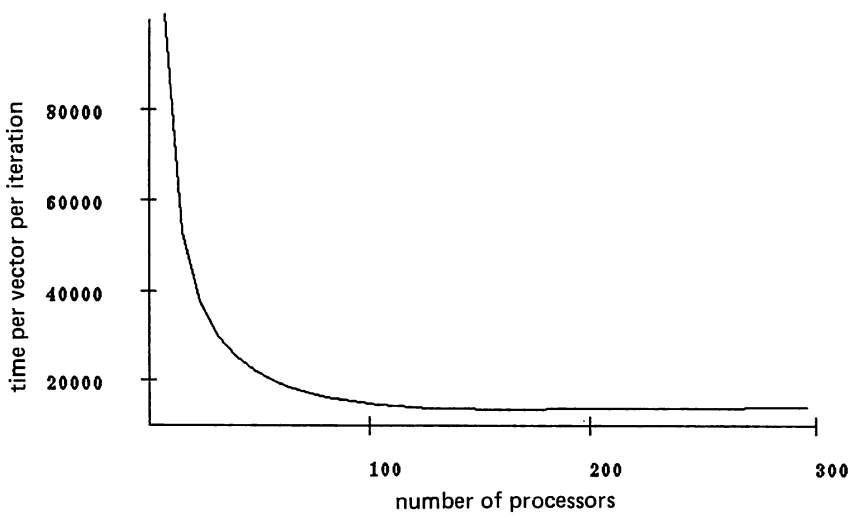

Fig. 4. Time per iteration per vector for a torus of processors; matrix size $n = 10000$, blocksize $b = 8$, start-up time $s = 500$, per-word transmission time $w = 0.01$.

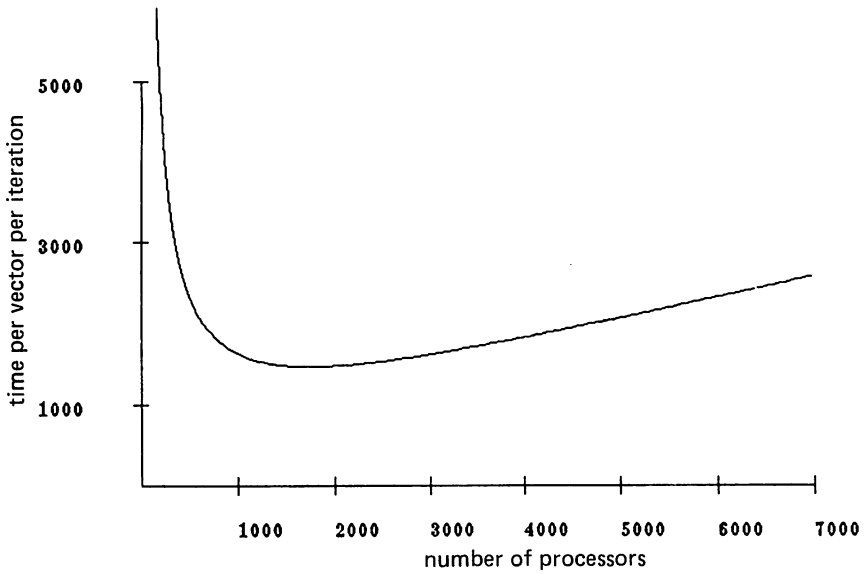

Fig. 5. Time per iteration per vector for a torus of processors; matrix size $n = 10000$, blocksize $b = 8$, start-up time $s = 1$, per-word transmission time $w = 0.01$.

Table 2
50% efficiency times for various simulated torus machines, $n = 10000$

s	w	$b = 1$		$b = 8$	
		Processors	Time	Processors	Time
500	0.01	8	18278	88	15660
50	0.01	26	6026	328	4293
5	0.01	76	2100	688	2058
0.5	0.01	150	1064	808	1753
0.05	0.01	178	894	824	1719
0	0.01	183	873	832	1710
0.5	0.10	146	1094	728	1947

Table 3
50% efficiency times for various simulated torus machines, $n = 1000$

s	w	$b = 1$		$b = 8$	
		Processors	Time	Processors	Time
500	0.01	3	5018	16	7959
50	0.01	8	1853	48	2858
5	0.01	24	666	96	1460
0.5	0.01	46	345	112	1268
0.05	0.01	55	291	112	1261
0	0.01	56	285	112	1260
0.5	0.10	45	355	104	1388

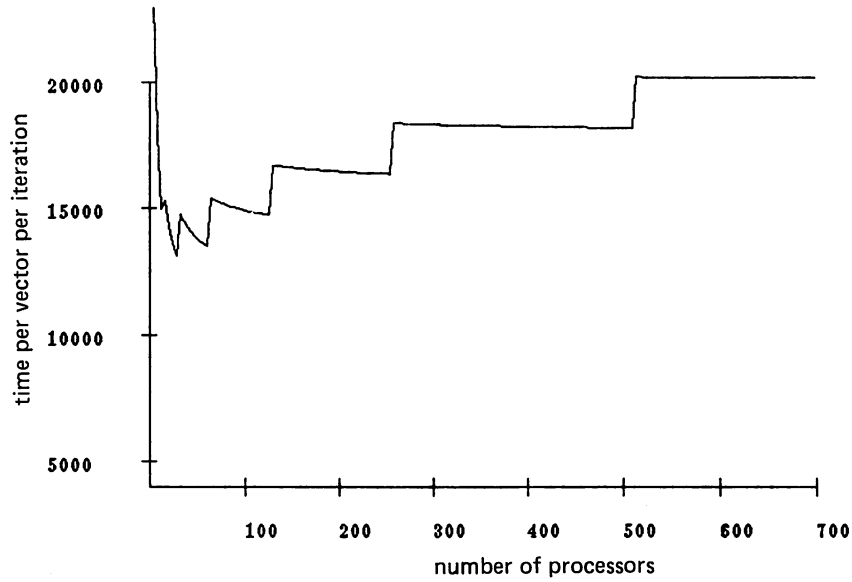

Fig. 6. Time per iteration for a hypercube of processors; matrix size $n = 10000$, blocksize $b = 1$, start-up time $s = 500$, per-word transmission time $w = 0.01$.

where the $\log p_C$ terms should be omitted if $p_C = 1$, and similarly for $\log p_R$. Analytical minimization of this formula plus (1) yields

$$p_R = \frac{11nb^2 - nb - nb^2/p_C - 16nbw/\log p_C + 16nbwp_C/\log p_C}{(4b^2w + 2b^2 + 4sp_C)\log e},$$

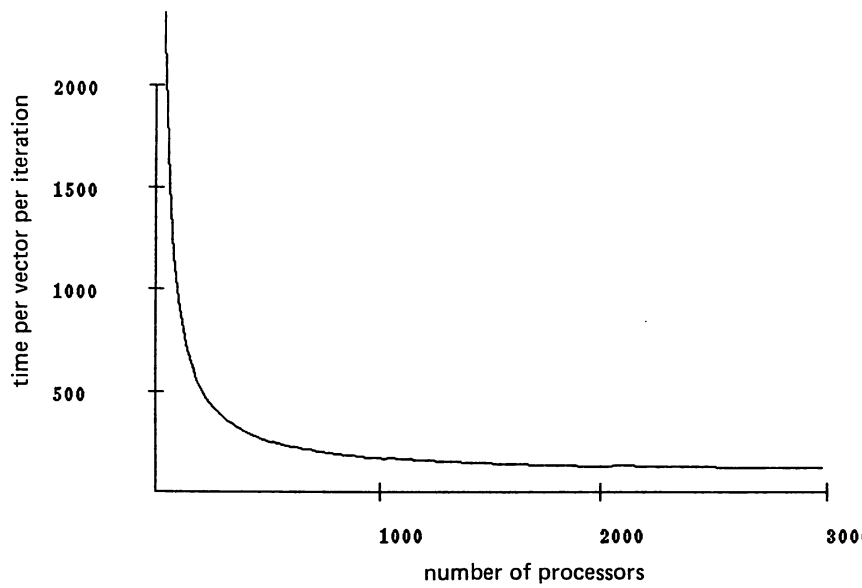

Fig. 7. Time per iteration for a hypercube of processors; matrix size $n = 10000$, blocksize $b = 1$, start-up time $s = 1$, per-word transmission time $w = 0.01$.

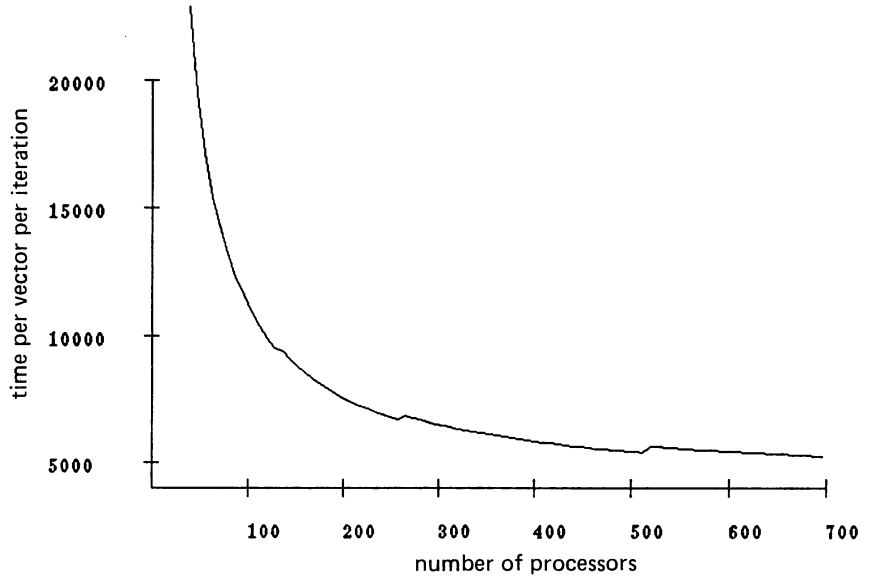

Fig. 8. Time per iteration per vector for a hypercube of processors; matrix size $n = 10000$, blocksize $b = 8$, start-up time $s = 500$, per-word transmission time $w = 0.01$.

and, when $p_C = 1$ this gives a formula for the optimal number of processors of

$$p_R = \frac{10nb^2 - nb}{(4b^2w + 2b^2 + 4s)\log e}.$$

Figures 6 through 9 give rsults analogous to Fig. 2 through 5, but for the hypercube rather than the torus. When $s = 500$, the optimal number of processors is now 29 rather than 9, and the time drops to 13128 from 18030. When $s = 1$, it is possible to effectively utilize 3000

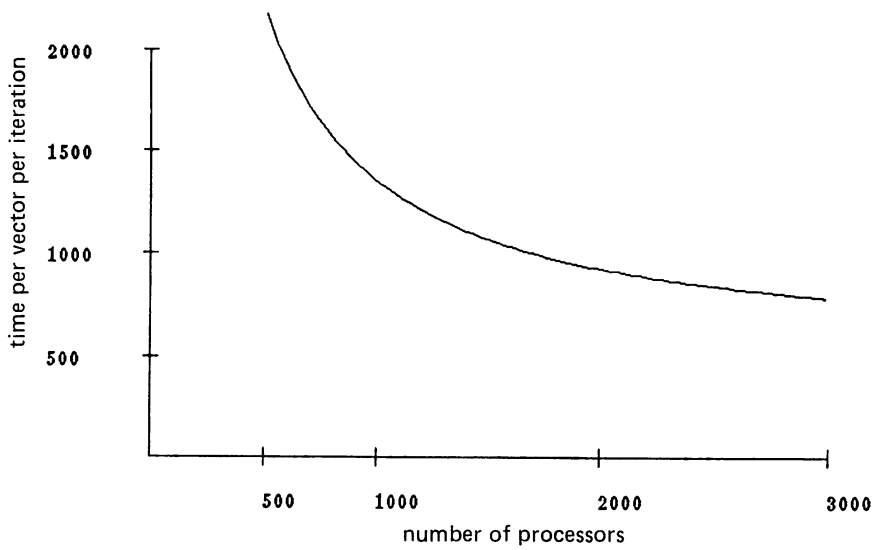

Fig. 9. Time per iteration per vector for a hypercube of processors; matrix size $n = 10000$, blocksize $b = 8$, start-up time $s = 1$, per-word transmission time $w = 0.01$.

Table 4
50% efficiency times for various simulated hypercube machines, $n = 10000$

s	w	$b=1$		$b=8$	
		Processors	Time	Processors	Time
500	0.01	8	17271	168	8301
50	0.01	57	2806	496	2852
5	0.01	329	486	1024	1385
0.5	0.01	1187	134	1152	1234
0.05	0.01	1807	88	1168	1217
0	0.01	1922	83	1168	1216
0.5	0.10	1114	143	1064	1337

Table 5
50% efficiency times for various simulated hypercube machines, $n = 1000$

s	w	$b=1$		$b=8$	
		Processors	Time	Processors	Time
500	0.01	2	6517	16	7471
50	0.01	8	1746	56	2571
5	0.01	47	338	104	1358
0.5	0.01	151	106	120	1194
0.05	0.01	221	72	120	1189
0	0.01	232	69	120	1189
0.5	0.10	141	113	112	1285

processors. For the slow start-up machine, blocking reduces the overhead, but there is increased overhead time for the $s = 1$ machine. Tables 4 and 5 give the optimal number of processors under the 50% efficiency constraint. The number of usable processors increases to a limit of 1922 for $n = 10000$ and 232 for $n = 1000$, as the start-up time decreases.

4. Summary and open questions

The block conjugate gradient algorithm can:

(1) Significantly reduce conjugate gradient overhead time on machines with minimal connectivity (e.g., torus) and on machines with high communication costs;

(2) Give faster convergence, and thus a lower overall cost, on matrices with certain eigenvalue distributions, even on hypercubes with low communications costs;

(3) Solve several problems simultaneously;

(4) Reduce the number of accesses to the matrices A and M, causing a speed-up if these matrices must be generated or brought in from secondary storage each time they are used.

This work has not considered the related questions of efficient formation of matrix-vector products and the choice of preconditioners well suited for parallel computation. Much work has already been done on these questions (e.g., [19,9]), but much more remains.

References

[1] N.N. Abdelmalek, Round off error analysis for Gram–Schmidt method and solution of linear least squares problems, *BIT* **11** (1971) 345–367.

[2] O. Axelsson, On preconditioning and convergence acceleration in sparse matrix computations, Report 74-10, CERN, Geneva, 1974.
[3] O. Axelsson, Solution of linear systems of equations: iterative methods, in: V.A. Barker, ed., *Sparse Matrix Techniques* (Springer, New York, 1977) 1–11.
[4] Å. Björck, Solving linear least squares problems by Gram–Schmidt orthogonalization, *BIT* **7** (1967) 1–21.
[5] P. Concus, G.H. Golub and D.P. O'Leary, A generalized conjugate gradient method for the numerical solution of elliptic partial differential equations, in: J.R. Bunch and D.J. Rose, eds., *Sparse Matrix Computations* (Academic Press, New York, 1976) 309–322.
[6] J. Cullum and W.E. Donath, A block Lanczos algorithm for computing the q algebraically largest eigenvalues and a corresponding eigenspace of large, sparse, symmetric matrices, *Proc. 1974 IEEE Conference on Decision and Control* (1974) 505–509.
[7] G.H. Golub and R. Underwood, The block Lanczos method for computing eigenvalues, in: J.R. Rice, ed., *Mathematical Software III* (Academic Press, New York, 1977) 361–377.
[8] M.R. Hestenes, The conjugate gradient method for solving linear systems, *Proc. Symposium on Applied Mathematics* **6** (1956) 83–102.
[9] C. Kamath and A. Sameh, The preconditioned conjugate gradient algorithm on a multiprocessor, Manuscript, University of Illinois Computer Science Department, 1984.
[10] C. Lanczos, An iteration method for the solution of the eigenvalue problem of linear differential and integral operators, *J. Res. National Bur. Standards* **45** (1950) 255–282.
[11] D.G. Luenberger, *Linear and Nonlinear Programming* (Addison-Wesley, Reading, MA, 1984).
[12] J.A. Meijerink and H.A. van der Vorst, An iterative solution method for linear systems of which the coefficient matrix is a symmetric M-matrix, *Math. Comput.* **31** (1977) 148–162.
[13] D.P. O'Leary, The block conjugate gradient algorithm and related methods, *Linear Algebra Appl.* **29** (1980) 293–322.
[14] D.P. O'Leary and G.W. Stewart, Assignment and scheduling in parallel matrix factorization, *Linear Algebra Appl.* **77** (1986) 275–300.
[15] C.C. Paige, Computational variants of the Lanczos method for the eigenproblem, *J. Inst. Math. Appl.* **10** (1972) 373–381.
[16] C.C. Paige and M.A. Saunders, Solution of sparse indefinite systems of linear equations, *SIAM J. Numer. Anal.* **12** (1975) 617–629.
[17] J.K. Reid, On the method of conjugate gradients for the solution of large sparse systems of linear equations, in: J.K. Reid, ed., *Large Sparse Sets of Linear Equations* (Academic Press, New York, 1971) 231–254.
[18] Y. Saad and M.H. Schultz, Data communication in hypercubes, Computer Science Department Report 428, Yale University, 1985.
[19] M.K. Seager, Parallelizing conjugate gradient for the CRAY X-MP, *Parallel Comput.* **3** (1986) 35–47.
[20] G.W. Stewart, Communication in parallel algorithms: An example, *Proc. 18th Symposium on the Interface between Computer Science and Statistics* (American Statistical Association, Washington, DC, 1986) 11–14.
[21] R. Underwood, An iterative block Lanczos method for the solution of large sparse symmetric eigenproblems, Computer Science Department Report STAN-CS-75-496, Stanford, CA, 1975.
[22] E.L. Wachspress, Extended application of alternating direction implicit iteration model problem theory, *J. SIAM* **11** (1963) 994–1016.