JOURNAL OF COMPUTATIONAL PHYSICS 100, 396–401 (1992)

# Performance of the Conjugate Gradient Method on VICTOR

RAMESH NATARAJAN AND PRATAP PATTNAIK

*IBM Thomas J. Watson Research Center, P. O. Box 704, Yorktown Heights, New York 10598*

Received November 29, 1990; revised June 17, 1991

We describe an implementation of the conjugate gradient method on VICTOR, a 256-node message-passing MIMD parallel computer based on the T800 transputer, developed at IBM, Yorktown Heights. A detailed performance analysis is also presented for this algorithm on a simple but representative application (Poisson's equation in a square region discretized using bilinear finite elements). 

## 1. INTRODUCTION

We describe an implementation of the conjugate gradient method for symmetric, positive-definite matrix systems on a message-passing multiprocessor. The matrices are assumed to be derived from the uniform finite element discretization of second-order, elliptic partial differential equations on two-dimensional rectangular domains. This assumption is primarily made to simplify the programming and analysis, but we note that by using standard tools such as conformal mapping and coordinate stretching, more general polygon domains and graded discretizations can also be treated by the present approach.

Performance measurements on the VICTOR parallel computer are presented and analyzed for a representative self-adjoint model problem (Poisson's equation on a square region discretized by bilinear finite elements) as described in further detail below. For the nonsymmetric matrices obtained from non-self-adjoint problems, another algorithm (the conjugate gradient squared algorithm of Sonneveld [1]) has also been implemented and analyzed. However, the low-level details for the nonsymmetric case do not raise any new significant implementation or performance issues, and we therefore omit many of these details here for the sake of brevity.

## 2. ARCHITECTURAL DESCRIPTION

VICTOR is a 256-node, message-passing MIMD computer developed in the Computer Sciences department at IBM, Yorktown Heights (Wilcke *et al.* [4]). The node processor is the INMOS T800 transputer, which is a 32-bit microprocessor with a 20-MHz clock cycle, 4 Mbytes of on-chip RAM, and a built-in FPU. Each node transputer is connected through four 20-Mbit/s serial links to other nodes in a mesh topology, except for designated nodes on the edge of this mesh which are connected to disks, graphics devices, and host computers. Multiple user programs can execute without interference or performance penalty on distint subpartitions of the $16 \times 16$ processor mesh. For example, this means that four different users can be simultaneously space-sharing the machine, with each using a distinct $8 \times 8$ subpartition. A non-intrusive monitoring tool provides a visual display of memory and link activity in the nodes during program execution, which is particularly helpful for detecting runtime execution errors.

## 3. PROGRAMMING ENVIRONMENT

The Express programming environment in the so-called "Cubix" mode is used for program development (Fox *et al.* [5]). Here, the same program executes on all the nodes, but although individual nodes are programmed homogeneously, they can have different execution paths through this program, synchronizing with other processors only for message passing (either directly, or indirectly as a intermediate node for routing messages).

The node programs are written in FORTRAN, which is augmented by subroutine calls to Express library functions for message-passing. In addition to the basic point-to-point message-passing routines, Express provides various optimized high-level subroutine utilities that can facilitate program development, of which two deserve special mention in the context of the present application. The first is a "combining" function for performing global operations on data in the individual processors (for operators that are commutative and associative). The second is a set of utilities for embedding multidimensional grid topologies in the actual hardware, which allows programs to be written in terms of the logical geometric connectivity of the problem without concern for the actual hardware topology, while retaining full program portability.

0021-9991/92 $5.00

## 4. ALGORITHM DETAILS

The basic (preconditioned) conjugate gradient algorithm for solving matrix systems of the form $Ax = b$ is shown below [6]

conjugate gradient (CG):
$x_0$ = initial solution guess
$r_0 = b - Ax_0$
$s_0 = M^{-1} r_0$
$p_0 = s_0$
$\rho_0 = s_0^T \cdot r_0$
**for** $k = 0, 1, \ldots,$ **until convergence do**
**begin**
$q_k = Ap_k$
$\sigma_k = p_k^T \cdot q_k$
$\alpha_k = \rho_k / \sigma_k$
$x_{k+1} = x_k + \alpha_k p_k$
$r_{k+1} = r_k - \alpha_k q_k$
$s_{k+1} = M^{-1} r_{k+1}$
$\rho_{k+1} = s_{k+1}^T \cdot r_{k+1}$
$\beta_k = \rho_{k+1} / \rho_k$
$p_{k+1} = s_{k+1} + \beta_k p_k$
**enddo**

In this paper we only consider the unpreconditioned version of the algorithm, for which in each iteration, the arithmetic work is comprised of one matrix-vector multiplication to obtain $q_k$, three saxpy operations to obtain $x_{k+1}$, $r_{k+1}$, $p_{k+1}$, and two dot-product operations to obtain $\sigma_k$ and $\beta_k$, respectively. The storage requirements beyond that for the matrix $A$ and the right-hand side vector $b$, are the four vectors for the most recent values of $x$, $r$, $q$, $p$ which are overwritten on each iteration. Another vector for the most recent value of $s$ is also required if preconditioning is used, in addition to any storage for the preconditioner itself.

## 5. IMPLEMENTATION DETAILS

The use of preconditioning in the conjugate gradient algorithm reduces the number of iterations required for convergence, and the trade-off between the cost of applying the preconditioner and the improved convergence due to using it must then be considered. For the unpreconditioned algorithm, however, it is sufficient to consider the cost per iteration, which is simply given by

$$T_{CG} = T_{MV} + 3T_S + 2T_D, \tag{4.1}$$

where $T_{MV}$, $T_S$, $T_D$, are the costs of a matrix-vector multiplication, a saxpy, and a dot-product, respectively (in units of seconds). The cost of these three primitives will depend on certain problem and implementation details, as discussed in detail below.

The basic strategy for partitioning the problem among the processors is domain decomposition. We partition the domain into $P$ open, connected, and disjoint (non-overlapping) regions, denoted by $\Omega_i$, so that $\Omega = \bigcup_i^P \Omega_i$ and $\Omega_i \cap \Omega_j = \phi$, $i \neq j$, where $P$ is the number of processors. Furthermore, we assume that the boundaries between any two adjacent subdomains, say $\Omega_i$ and $\Omega_j$ (denoted by $\Gamma_{ij}$) correspond to the edges or faces of a conforming finite element mesh on $\Omega$. Note that $\bigcup \Gamma_{ij} - \bigcap \Gamma_{ij} = \partial\Omega$, $\Gamma_{ij} = \Gamma_{ji}$, and $\Gamma_{ij} = \phi$ unless there are some nodal points that belong to the interfaces of $\Omega_i$ and $\Omega_j$. The underlying assumption here is that the finite element mesh partitioning is much finer than the subdomain partitioning. Now each subdomain is assigned to a processor that is responsible for performing the computations involving the nodal variables and equations that belong to it. The nodal variables and equations defined at interfaces will be shared between two or more processors, which will require some information exchange between these "neighboring" processors at various points in the algorithm.

Within each subdomain, the interior nodal variables and equations are numbered first, followed by the boundary nodal variables and equations, so that after the subdomain finite element assembly is performed we obtain the stiffness matrix appropriately partitioned as

$$\begin{bmatrix} A_i & B_i \\ C_i & D_i \end{bmatrix} \begin{bmatrix} x_i \\ y_i \end{bmatrix} = \begin{bmatrix} b_i \\ c_i \end{bmatrix}. \tag{4.2}$$

Here, the submatrices $A_i$, $B_i$ and the load vector $b_i$ are fully assembled, representing the contributions to the rows of the global stiffness matrix from test functions that are nonzero only within the interior of the subdomain in question. On the other hand, the submatrices $C_i$ and $D_i$ and the load vector $c_i$ are only partially assembled at this stage and have to be merged with the equivalent submatrices obtained on the subdomain with which it shares nodal grid points. For simplicity, we write (4.2) in the aggregated form

$$\hat{A}_i \hat{x}_i = \hat{b}_i. \tag{4.3}$$

The global finite element assembly can now be performed with these subdomain matrices to yield

$$\hat{A} = \sum_i^P T_i \hat{A}_i T_i^T, \qquad \hat{b} = \sum_i^P T_i \hat{b}_i, \tag{4.4}$$

where $\hat{A}$ is the global stiffness matrix and the matrices $T_i$ are permutation matrices that contain the mapping between the subdomain node numbering and the global node numbering. In practice, however, there is no need to explicitly assembly the matrix $\hat{A}$, since the CG algorithm only requires the action of $\hat{A}$ with a given vector, which can

be computed by using a "subdomain-by-subdomain" algorithm,

$$\hat{A}y = \sum_{i}^{P} T_i \hat{A}_i T_i^T y = \sum_{i}^{P} T_i \hat{A}_i y_i, \tag{4.5}$$

where $y_i$ denotes the restriction of the vector $y$ to the subdomain. Here the individual matrix-vector products on each subdomain can be performed independently, following which the required summation for the nodal unknowns on the boundaries is carried out by routing the data between the processors that share interface nodes. In this way each processor will accumulate the global sum corresponding to the interface variables on its subdomain.

The other computational steps in the CG algorithm are the saxpy and the dot-product. The distributed implementation of the saxpy operation is quite trivial, since it involves no communication. The dot-product, on the other hand, requires each processor to compute its portion of the dot-product (avoiding overlaps whenever a node belongs to a subdomain interface $\Gamma_{ij}$ shared by more than one processor). These individual sums are then combined to yield the global sum, which is then redistributed to each processor.

The implementation methodology outlined above is quite general, but we develop a detailed performance model for a specific application, which is Poisson's equation in two dimensions, using a bilinear finite element discretization. For simplicity, we restrict the present discussion to the consideration of a square region, with $\sqrt{n}$ elements on each edge of the domain, so that the number of nodal unknowns will be $(\sqrt{n}+1)^2$. We also assume that $m = n/P$ is a perfect square, so that each subdomain consists of $m$ elements. The processors can be thought of as being arranged in a $\sqrt{P} \times \sqrt{P}$ grid, to emphasize the communication connectivity requirements. For this partitioning, therefore, each processor will have $(\sqrt{m}+1)^2$ unknowns, with $(\sqrt{m}-1)^2$ in the interior, and $4\sqrt{m}$ on the interface.

Each processor is given an identifier, denoted *mypid*, using, say, a lexicographic assignment rule on our logical processor grid. It is also useful to have a special notation for the nearest neighbor processors on this logical grid. In particular, since *mypid* will share $\sqrt{m}+1$ grid points with each of the processors to its north, south, east, and west, we denote these by *mypid.n*, *mypid.s*, *mypid.e*, *mypid.w*, respectively. It will also share a single corner node with the processors in the intermediate directions, i.e., northwest etc., but it turns out not to be necessary to have special identifiers for these. For the processors on the boundaries of our logical grid, one or possibly two of these parameters values correspond to "dummy" nodes, a convention that simplifies the discussion.

We first consider the storage requirements in each node for the CG algorithm, which can be itemized as follows:

1. For bilinear finite elements, each nodal unknown is connected to nine others in the mesh. We use sparse matrix format for the subdomain matrix $\hat{A}_i$, in which only the nonzero entries and index information are stored. The storage requirement for it and for the right-hand side $b_i$ is then $15(\sqrt{m}+1)^2$.
2. The two-dimensional grid point coordinates, $2(\sqrt{m}+1)^2$.
3. Various bookkeeping tables and boundary condition data, $\frac{5}{2}(\sqrt{m}+1)^2 + 2m$.
4. Vectors used in the CG algorithm, $x$, $r$, $p$, and $q$, $4(\sqrt{m}+1)^2$.

Therefore, summing terms, we obtain for the overall storage requirement

$$\tfrac{47}{2}(\sqrt{m}+1)^2 + 2m. \tag{4.6}$$

Since each processor has sufficient storage for approximately 0.5 Mwords (double precision), this restricts the number of elements along the edge of each subdomain to roughly a maximum of 140. In practice, the maximum value is somewhat less than this, since storage is also required for the code and kernel routines.

We turn our attention to the computation costs, and for the saxpy it is easily seen that

$$T_S = \alpha_1(\sqrt{m}+1)^2. \tag{4.7}$$

The constant $\alpha_1$ depends on the node architecture and on the compiler, and its experimental value on VICTOR is $9.14 \times 10^{-6}$.

For the dot-product, each processor assumes the responsibility of computing the partial dot-products for the nodes on north and east edges. However, the processors on the south (resp. west) boundaries of our logical grid will compute the partial products for the north, east, and south edges (resp. the north, east, west edges). Finally, the processor on the southwest corner will compute the partial dot-product for all the edge nodes in it, and hence it will be the slowest to finish computing this phase. At this point, the values in the individual processors are additively combined (using the Express utility mentioned earlier), and the resulting value is then distributed to all processors. This combining function can be optimized by performing it on a spanning tree in the connectivity graph of the actual hardware. The overall cost is, therefore,

$$T_D = \alpha_2(\sqrt{m}+1)^2 + \beta_2 \operatorname{diam}(P). \tag{4.8}$$

The first term here denotes the time taken by the slowest processor in the first phase. The experimental value for $\alpha_2$ on VICTOR is $6.56 \times 10^{-6}$. This is better than $\alpha_1$ because of

the special nature of the register set on the FPU of the T800 transputer, on which the saxpy requires 3 loads, 2 ops, and a store for each result, whereas, equivalently, for the dot-product 2 loads and 2 ops suffice. The second term contains the diameter of the processor network (or, equivalently, the height of the minimum spanning tree). A least-squares analysis of the experimental measurements gives a good fit with $\beta_2 = 1.7 \times 10^{-3}$ and $\text{diam}(P) = P^{0.4}$, for values of $P$ greater than 16. The power law dependence of the combining overhead on $P$ can be ascertained from the linear graph in a log–log plot of the experimental data in Fig. 1.

We now consider the cost of computing the distributed matrix-vector product $T_{\text{MV}}$. First, we need the cost of message-passing between two neighboring processors, which we model in the form $L + r/B$, where $L$ is the message latency, $B$ is the bandwidth, and $r$ is the number of transferred words. Again experimental measurements on VICTOR indicate the values $L = 1.408 \times 10^{-3}$ s and $B = 2.26 \times 10^4$ words/s (double precision). In comparative terms, the startup latency is equivalent to the interprocessor transfer of about 32 words and to roughly 440 arithmetic operations at each node processor.

The implementation of the matrix-vector product in the CG algorithm is carried out as follows:

1. Each processor first computes the subdomain matrix-vector product. Using a row-oriented sparse matrix format for $\hat{A}$ and an inner-product algorithm, this cost can be written as $9\alpha_3(\sqrt{m}+1)^2$.

2. Then each processor executes the following two-phase communication algorithm:

- Phase I. East-West communication.
  - — sends its partial sum for the east interface nodes to *mypid.e*
  - — receives a partial sum from *mypid.w* and adds these to the current west interface values
  - — sends the updated west interface values to the *mypid.w*
  - — finally receives the updated values from the *mypid.e* and overwrites the current partial sum.

  The total cost for this, ignoring lower order computational terms and assuming no overlap of sends and receives, is given by $4(L + (\sqrt{m}+1)/B)$.

- Phase II. Similarly for the North–South directions, $4(L + (\sqrt{m}+1)/B)$.

We remark on two aspects of this particular algorithm. First the "corner" nodal unknowns in each subdomain which are shared between four processors are correctly summed by this two-phase algorithm. Second, if the underlying hardware supports bidirectional transfer on each link (as is the case with the transputer links on VICTOR then a savings of possibly up to a factor of 2 in the communication costs can be obtained by simultaneously interchanging the boundary values during both the communication phases. Then both processors involved in this exchange can separately compute the updated values, making the final "send" unnecessary. In any event, for the particular implementation outlined above, we obtain

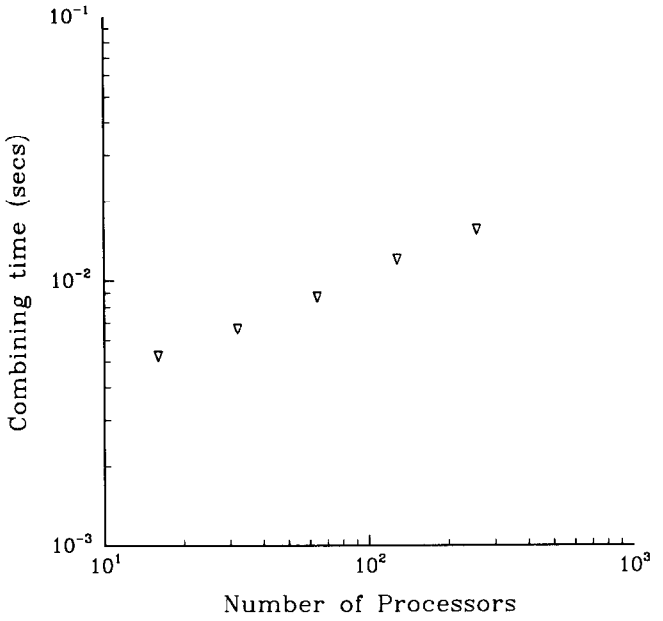


**FIG. 1.** Combining overhead in the distributed dot-product.

$$T_{\text{MV}} = 9\alpha_3(\sqrt{m}+1)^2 + 8[L + (\sqrt{m}+1)/B]. \quad (4.9)$$

The experimental value for $\alpha_3$ on VICTOR is $1.805 \times 10^{-5}$. This is worse than $\alpha_1$ and $\alpha_2$ because of the additional overhead of indirect addressing and index bound testing in the inner loop of the algorithm.

Substituting from (4.7)–(4.9) into (4.1) and setting $m = n/P$, we obtain

$$T_{\text{CG}}(n, P) = (3\alpha_1 + 2\alpha_2 + 9\alpha_3)(\sqrt{n/P}+1)^2 + 2\beta_2 \,\text{diam}(P) + 8[L + (\sqrt{n/P}+1)/B]. \quad (4.10)$$

The number of parameters in (4.10) may possibly be reduced on some other architectures. Many uniprocessor architectures have independent parallel functional units, pipelined execution, and multiported register files, and on these we may expect that $\alpha_1 = \alpha_2 = \alpha_3 = \alpha$, say, to a good approximation. For operations on an embedded spanning tree one can set $\beta_2 = 2L + \alpha$, with the factor of 2 coming from the combining and distribution phases of the operation. For a mesh, $\text{diam}(P)$ is proportional to $P^{1/2}$ (and, equivalently, to $\log P$ for a hypercube).

For the special case $P = 1$, the communication terms drop out of (4.10) and we obtain

$$T_{CG}(n, 1) = (3\alpha_1 + 2\alpha_2 + 9\alpha_3)(\sqrt{n} + 1)^2. \quad (4.11)$$

Finally, we note that in Sonneveld's conjugate gradient squared algorithm, each iteration requires 2 matrix-vector multiplications, 7 saxpy's, and 2 dot-products. Therefore, similar to (4.1), the time per iteration for this algorithm is given by

$$T_{CGS} = 2T_{MV} + 7T_S + 2T_D. \quad (4.12)$$

The values of $T_{MV}$, $T_S$, and $T_D$ are identical to those computed above for the CG algorithm, and therefore we obtain for $T_{CGS}(n, P)$ an expression whose form is equivalent to (4.10), except for the different values of the coefficients.

## 6. PERFORMANCE ANALYSIS

The parallel efficiency of the CG implementation is given by the quantity $T_{CG}(n, 1)/PT_{CG}(n, P)$. In Fig. 2, we show the experimental values for the parallel efficiencies on VICTOR for a mesh of $100 \times 100$ bilinear elements (which, as mentioned earlier, is just about the largest problem that will fit on a single node). As expected, the efficiency drops off as the number of processors is increased, with values of 0.75 and 0.50 for 24 and 64 processors, respectively. Also shown in Fig. 2 is the experimental parallel efficiency for the CGS implementation, which drops off to 0.66 on 64 processors. One reason for this dropoff is the increase in the communication to computation ratio of the application, due to the fewer elements in each subdomain, as the number of processors is increased. The other reason is the increased cost of the combining operation, which also explains the higher parallel efficiency for the CGS algorithm, since comparing one iteration of CGS to two iterations of CG, we see that in the former there is one more saxpy, but two fewer dot-product operations.

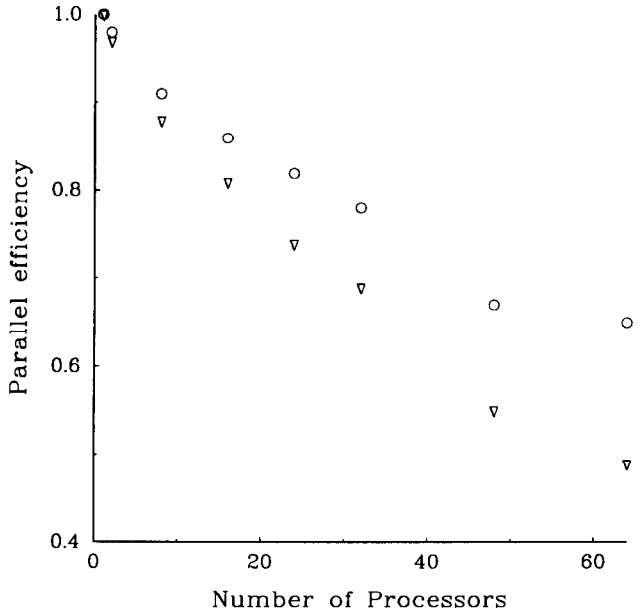


**FIG. 2.** Parallel efficiency of the conjugate gradient (CG) implementation (∇) on VICTOR ($100 \times 100$ bilinear elements mesh). Also shown are the equivalent parallel efficiencies for Sonneveld's conjugate gradient squared (CGS) algorithm (○) an another model problem.

As is well known, measuring speedups for fixed size problems on distributed memory processors is somewhat unrealistic since much of the memory is not utilized as the number of processors is increased. One alternative is to increase the problem size with the number of processors, so that the memory utilization per node remains fixed. The difficulty here is that increasing the problem size is equivalent to mesh refinement, which changes the condition number of the matrix and the convergence properties of the overall algorithm, so that even this does not seem to be a reasonable methodology.

A more revealing quantity is the so-called "floating point utilization" defined by Moler [7], which estimates the megaflop rate for the largest problem that can be solved on a given number of processors, relative to the megaflop rate for the same algorithm on a hypothetical single processor with infinite memory. One difficulty here is that the latter quantity cannot be measured experimentally, although it can be estimated with a good model (or conservatively, replaced by the peak uniprocessor megaflop rate). In the present case, however, a good model is available, and we can write for the floating point utilization in the form $T_{CG}(mP, 1)/PT_{CG}(mP, P)$, and use the definitions in (4.10) and (4.11). This quantity is expected to provide a more realistic measure of the parallel processing overheads in an actual "production-size" application. For this purpose, we consider an $800 \times 800$ bilinear element discretization (with 641,601 nodal unknowns) on 64 processors, for which the estimated floating point utilization for CG is 0.953 (similarly for CGS we obtain 0.958). Therefore, for each iteration of the CG algorithm, only about 4.7% of the time is lost to parallel processing overheads.

Experimental results for this computation were also obtained on a 64-node VICTOR subpartition for the model problem, and starting from a zero initial guess, the residual was reduced by a factor of $10^{-5}$ in 399 iterations. The overall solution time was 13.87 min and the time per iteration 2.086 s (including the time for residual computation, terminal I/O etc., which are not accounted for in the analysis). Similarly, a time per iteration of 4.267 s was obtained for an equivalent computation using the CGS algorithm on a simple model convection-diffusion problem.

## 7. FUTURE WORK

The implementation described here can be extended in a number of ways that might improve either the overall convergence of the algorithm, or its parallel efficiency. We list some of these possible extensions below.

1. The interior nodal unknowns in each subdomain, which are fully summed upon subdomain matrix assembly, can be eliminated by substructuring. This substructuring is a fully local operation which has the advantage of confining the conjugate gradient iteration to the consideration of the interface unknowns, while providing a "Schur-complement" preconditioning for the iteration. The node storage requirements, however, are increased, and this aspect requires that careful attention be paid to the nodal ordering within each subdomain. In any case, this will decrease the size of the largest problem that can be solved on a given number of processors, below that in the present method.

2. For very large processor networks the primary overhead is in the dot-product instruction which requires a global synchronization and information exchange. The use of block algorithms [8] might be helpful in reducing the number of such global synchronizations and in maximizing the amount of information exchanged during each such synchronization.

3. The use of polynomial preconditioners [9] could prove attractive for message-passing parallel computers because of their low communication requirements in comparison with other preconditioning methods.

## REFERENCES

1. P. Sonneveld, *SIAM J. Sci. Stat. Comput.* **10**, 36 (1986).
2. *IMS T800 Architecture*, Technical Note 6 (INMOS Limited, Bristol, UK, 1986).
3. *Express Reference Manual* (Parasoft Corporation, Pasadena, CA, 1990).
4. W. W. Wilcke, R. C. Booth, D. Brown, D. G. Shea, F. Tong, and D. Zukowski, *Design and Application of an Experimental Multiprocessor*, IBM Research Report RC 56722 (1987).
5. G. Fox *et al.*, *Solving Problems on Concurrent Processors*, Vol. 1 (Prentice-Hall, Englewood Cliffs, NJ, 1988).
6. G. H. Golub and C. F. Van Loan, *Matrix Computations* (John Hopkins, Baltimore, 1989).
7. C. Moler, in *Hypercube Multiprocessors 1986*, edited by M. T. Heath (SIAM, Philadelphia, 1986).
8. D. P. O'Leary, *Linear Algebra Appl.* **29**, 293 (1980).
9. O. G. Johnson, C. A. Micchelli and G. Paul, *SIAM J. Numer. Anal.* **20**, 362 (1983).