Algorithmic Bombardment for the Iterative Solution of Linear Systems: A Poly-Iterative Approach

Richard Barrett[*], Michael Berry[†], Jack Dongarra[†‡], Victor Eijkhout[§] and Charles Romine[‡].

April 10, 1996

Abstract

Many algorithms employing short recurrences have been developed for iteratively solving linear systems. Yet when the matrix is nonsymmetric or indefinite, or both, it is difficult to predict which method will perform best, or indeed, converge at all. Attempts have been made to classify the matrix properties for which a particular method will yield a satisfactory solution, but "luck" still plays large role. This report describes the implementation of a poly-iterative solver. Here we apply three algorithms simultaneously to the system, in the hope that at least one will converge to the solution. While this approach has merit in a sequential computing environment, it is even more valuable in a parallel environment. By combining global communications, the cost of three methods can be reduced to that of a single method.

1 Introduction

Many iterative methods for solving real nonsymmetric linear systems

$$Ax = b \tag{1}$$

have been proposed. Even though theoretically certain statements concerning the convergence of such methods hold, in practice we often cannot choose a 'best' method in advance, for a variety of reasons.

For instance, the popular choice of the GMRES method [17], in the absence of rounding error, guarantees convergence in n steps for an order n matrix, but its memory requirements often rule out solving large systems. Variations for reducing the length of the recurrences have been proposed (using restart or truncation), but this compromises convergence theory[1]. Also, in a distributed memory environment, the communication requirements of the increasing number of inner products that must be performed can severely slow the time to solution[2].

Recent developments have made less memory and communication intensive algorithms more viable. Each offers theoretical justification for its convergence properties, and if the user has certain

*Distributed Computing Group, Los Alamos National Laboratory, Los Alamos, NM 87545
†Dept. of Computer Science, University of Tennessee, Knoxville, TN 37996
‡Mathematical Sciences Section, Oak Ridge National Laboratory, Oak Ridge, TN 37831-8083
§Department of Mathematics, University of California at Los Angeles, Los Angeles, CA 90024

[1] Indeed, there are examples in which the convergence of GMRES with *any* length recurrence less that n will stagnate.

[2] It is possible to combine the communication of these inner products (using the unmodified Gram-Schmidt), but this is often unstable.

information concerning the spectrum of the matrix, it is possible to select a method which should work well. However, when this is not the case, time (and expense) may be wasted when an algorithm terminates without convergence. For example, one might hope that for a sequence of similar problems the same method will consistently outperform the others, but that is not necessarily the case. Small variations in the PDE coefficients, or choosing a different grid size for the same problems, is often enough to reverse the relative ranking of two iterative methods.

In this paper we propose a simple strategy for combining iterative methods that increases the chance of finding the solution in a reasonable amount of time. The Poly-Iterative approach (informally called 'Algorithmic Bombardment' since it unleashes multiple methods on a single problem) consists of

- Choosing a number of iterative methods that are a priori suited for the problem at hand;
- Applying these methods simultaneously (or more precisely, interleaved; this will be discussed in detail later) on the dataset;
- Removing methods from the process that break down;
- Terminating this process when one method has converged.

Although the number of operations per iterative step equals the sum of the operations of the individual methods, we believe that when knowledge of matrix properties is lacking or incomplete, the extra floating point computation and memory requirements are outweighed by three factors:

1. An increased probability of finding the solution.
2. An efficient parallel implementation. By iterating in lock-step, i.e. the algorithms are always on the same iteration count, we gain time savings by combining overlapping communication (inner products, matrix–vector products, preconditioner solves).
3. Increased floating point performance. Depending upon the structure of the matrix, an efficient matrix-vector product may be constructed so as to make use of data locality. This may also be true for preconditioning.

2 The Algorithms

In this section we give a brief description of the methods that make up our implementation of the algorithmic bombardment algorithm.

When the coefficient matrix of the linear system is a symmetric positive definite matrix, the traditional iterative algorithm of choice is the Conjugate Gradient (CG) method [15]. However, when the coefficient matrix is nonsymmetric, CG typically fails to find the solution. The Biconjugate Gradient method [9], [16], rather than relying on a single sequence of residuals (as does CG), creates another sequence $\{\tilde{r}\}_{j=0}^{n}$ using A^T, which is orthogonal to $\{r\}_{j=0}^{n}$, as follows:

$$\tilde{r}_j = \tilde{r}_{j-1} - \alpha_j A^T \tilde{p}_j,$$

where

$$\tilde{p}_j = \tilde{r}_{j-1} + \beta_{j-1}\tilde{p}_{j-1}.$$

The biorthogonality requirements between r_j with $\tilde{r}_j$ and p_j with $\tilde{p}_j$ (with respect to the A inner product) are enforced by choosing

$$\alpha_j = \frac{\tilde{r}_{j-1}^T r_{j-1}}{\tilde{p}_j^T A \tilde{p}_j}, \text{ and } \beta_j = \frac{\tilde{r}_j^T r_j}{\tilde{r}_{j-1}^T r_{j-1}}.$$

2.1 Quasi-Minimal Residual (QMR)

BiCG can be erratic in practice, making no progress towards the solution for several iterations. QMR is designed to smooth out this problem, and make progress even when BiCG stalls. This algorithm was initially developed for complex symmetric linear systems [10], then later adapted to nonsymmetric systems [11].

Whereas GMRES constructs and solves an upper Hessenberg matrix consisting of an orthogonal Krylov subspace, the biorthogonality property of BiCG yields a tridiagonal matrix. Solving it in a least squares sense provides a quasi-minimization of the residual, which can overcome the instability that often occurs in BiCG, allowing for smoother convergence, while maintaining three term recurrences.

Further research into this algorithm has resulted in a number of improvements. A two term recurrence version has been developed [12]. Furthermore, van der Vorst has developed a relatively inexpensive recurrence relation for the computation of the residual vector, as well as a reduction in the number of preconditioning steps (from three to two) [2].

Note that as we have implemented it, QMR may breakdown.[3]

2.2 Conjugate Gradient Squared (CGS)

The goal of QMR is to further reduce the residual when the BiCG iteration stalls. In the case of convergence for BiCG, both $\|r_j\|$ and $\|\tilde{r}_j\|$ converge to zero, yet only the convergence of r_j is exploited. Sonneveld showed [18] that by concentrating the effort on the r_j, the speed of BiCG convergence could be doubled.

If we write $r_j = P_j(A)r_0$ and $\tilde{r}_j = P_j(A^T)\tilde{r}_0$, we see that

$$(r_j, \tilde{r}_i) \;=\; (P_j(A)r_0, P_i(A^T)\tilde{r}_0) = (P_i(A)P_j(A)r_0, \tilde{r}_0) \;=\; 0$$

for $i < j$. This implies that we could construct $\tilde{r}_j = P_j^2(A)r_0$. This is the basis for the Conjugate Gradient Squared Method (CGS). Note that the savings is not only that the $\tilde{r}$'s are not formed, but we also do not require the transpose of matrix A. The result is that the Krylov subspace is built up twice as fast as BiCG, theoretically doubling the speed of convergence. Because of the "squaring" of the polynomial, when the BiCG iterate makes progress towards the solution, CGS doubles that progress. However, when the BiCG iterate turns away from the solution, that error is also doubled. This explains the erratic behavior of the residual norm.

2.3 Biconjugate Gradient Stabilized (BiCGSTAB)

Van der Vorst [19] proposed that instead of building the basis vectors for the $i-th$ dimensional Krylov subspace $\mathcal{K}^i(\tilde{r}_0, A^T)$ using the same polynomial, i.e. $P_i(A)$, as does CGS, the residual could be smoothed using a different polynomial. He ruled out using Chebyshev polynomials since the optimal parameters were not easily obtainable. Instead, he selected a polynomial of the form $Q_i(A) = (1-\omega_1 A)((1-\omega_2 A)\cdots(1-\omega_i A)$, which gives an easy recurrence relation for updating Q. The choice of ω would be such that $r_i = Q_i(A)P_i(A)r_0$ is minimized. Experiments show that this often smoothes the peaks common to the residual norm in CGS, while maintaining the speed of convergence. Note that finite termination is maintained by the orthogonality property $(P_j(A)r_0, Q_i(A^T)\tilde{r}_0) = 0$, for $i < j$.

[3] A version of QMR that includes a "look-ahead" algorithm can avoid these problems, yet for simplicity we do not use it.

3 The Algorithmic Bombardment Algorithm

As indicated earlier, none of the algorithms above are guaranteed to find the solution. They can diverge, stall out, or break down. Thus, we are led to the idea of using all algorithms simultaneously, on the same problem. As soon as one method has converged we stop the overall iteration; if a method breaks down we drop it from the iterative scheme. The resulting poly-iterative algorithm takes more time to converge than the best method, but it has an improved chance of finding the solution.

Since the choice of methods depends on the specific problem, we really have a parameterized process

$$\texttt{PolyIt}(A, b, \texttt{method}_1, \texttt{method}_2, \ldots) \tag{2}$$

In this paper, we report results with $\texttt{PolyIt}(A, b, \texttt{CGS}, \texttt{BiCGstab}, \texttt{QMR})$ which uses three all-purpose methods that do not need a great deal of storage. By no means do we claim that this particular combination is the be-all and end-all of all iterative methods. For example, if the problem is indefinite, it would make sense to include `MINRES` among the methods; if core memory is not at a premium, `GMRES(`x`)` with x a large number would be appropriate for inclusion.

3.1 Parallel Implementation

The poly-iteration requires the sum of the floating point operations of the included algorithms, yet in the context of message-passing parallel computers, we can increase the efficiency of the approach with regard to the global data because of the high cost of communication.

The algorithms we consider are all based on some form of the conjugate gradient method, and thereby they have a very similar structure: they begin by computing an inner product, followed by vector updates, then a preconditioner solve, etc. The inner products, matrix-vector products, and preconditioner solves all require a communication stage. We make the poly-iterative method more efficient by *aligning* these methods at these operations and combining the communication stages. The other mathematical operations (vector updates, certain preconditioners, scalar operations, etc.) are computed in parallel, requiring no communication. The overall effect of the extra work is a function of the sparsity of the original coefficient matrix.

Figure 1 illustrates the poly-iterative idea. The operations listed in the circles take advantage of combined communication. A listing of the other operations each algorithm performs in parallel (left to right: CGS, BiCGSTAB, QMR) is also provided.

3.2 Structure of the iteration

The global structure of an iteration of the poly-iterative method is as follows:

- In each *parallel* region, that is, a part of the algorithm where there is no communication, let each processor perform in sequence the operations of the individual methods on its part of the data.
- At the start of a communication stage, pack the data of all methods that is to be transmitted in one buffer[4], then send this buffer in total.

Combining the communications amortizes the communication overhead over the methods. In the case of inner products where just a single floating point number per method is sent, this effectively divides the communication cost by the number of methods.

[4] In certain communication schemes such as PVM [13] this buffering is provided, in other schemes such as PICL [14] and the BLACS [6] it has to be implemented as part of the poly-iterative algorithm.

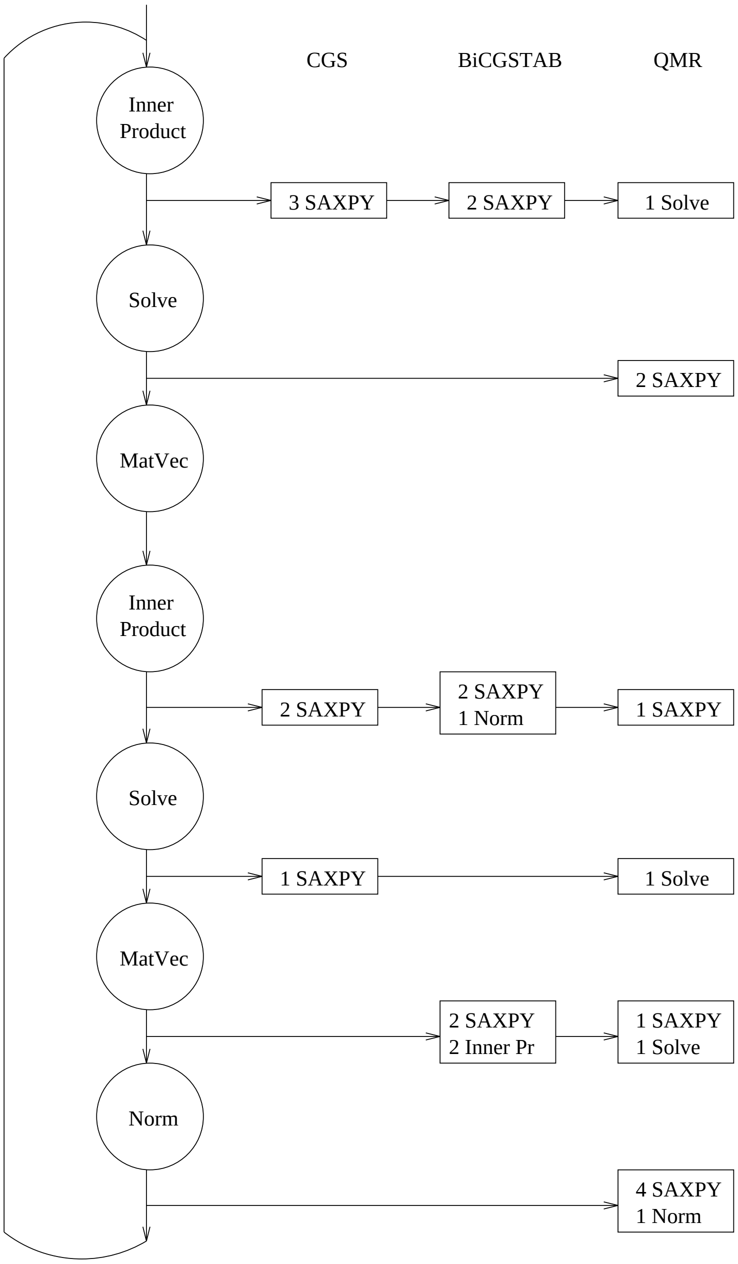

Figure 1: Sequence of Operations
This figure illustrates the sequence of mathematical operations as performed by our implementation of algorithmic bombardment. The operations in the circles combine the communication required of all three methods into one message. The operations in the rectangles are performed in parallel (left to right: CGS, BiCGSTAB, and QMR).

3.3 Cost Model

Obviously this approach requires the combined floating point operations and workspace of each method [5]. This limits the size of the linear system that may be solved, although the actual impact is a function of the sparsity of the matrix. Table 1 lists the computational requirements for each method included in our implementation. Table 2 lists the communication requirements of each method, as well as for bombardment, plus the storage requirements for each[6].

Table 1: Summary of operations for a single iteration. "1/1" means an iteration requires both a matrix times vector and matrix transpose times vector operation.

Method	Amount of Work/Iteration			
	$\alpha \leftarrow x^T y$	$y \leftarrow \alpha x + y$	$y \leftarrow Ax$	$x \leftarrow M^{-1}y$
CGS	2	6	2	2
Bi-CGSTAB	4	6	2	2
QMR	2	$8{+}4^{ab}$	1/1	2/2

[a] True SAXPY operations + vector scalings

[b] Less for implementations that do not recursively upddate the residual.

Table 2: Summary of communication requirements for a single iteration. We assume the preconditioning steps require communication, which may not be the case.

Method	Number of Communications/Iteration				Storage
	$\alpha \leftarrow x^T y$	$y \leftarrow \alpha x + y$	$y \leftarrow Ax$	$x \leftarrow M^{-1}y$	Requirements
CGS	2	0	2	2	matrix + $6n$
Bi-CGSTAB	3	0	2	2	matrix + $6n$
QMR	2	0	2	2	matrix + $16n^b$
Bombardment	3	0	2	3	matrix + $26n^b$

The scalar cost of an iteration of the poly-iterative method equals the sum of the costs of the individual methods. In the case where one method is more expensive than the others and this method is not the first to converge, we incur a relatively high cost. On the other hand, when using only one method, and that method fails to converge, the cost is magnified by the number of iterations performed until it is abandoned. The cost of the subsequent algorithms will be accumulated in a similar fashion.

In addition to the storage of matrix A, bombardment requires 26 workspace vectors of length n. GMRES with restart parameter m uses $(m+5)n = 5n + mn$, so the amount of workspace is equal when the restart parameter is 21. The problem is that restarting voids the guaranteed convergence property of GMRES.

Another consideration is the amount of work per iteration. GMRES performs one matrix-vector product and one preconditioner solve per iteration compared to two each for each algorithm in bombardment. However, the number of inner products per iteration for GMRES grows linearly with the restart parameter, whereas bombardment requires three (in terms of communication). While it is true that it is possible to compute the GMRES inner products independently, this is known to cause a loss of stability [5].

[5] less $2n$ since the right-hand-side vector b need only be stored once.

[6] excluding scalar storage

Parallel architectures require global communication, and this remains the over-riding factor in the performance of these algorithms. For example, the computation of an inner product is an order n operation, but each processor requires the global result. This requires the communication of a single scalar. Each processor computes and sends its local result to all other processors, and receives the partial sums from all other processor. Our method computes three inner products locally, then packs them into one message for the same communication requirement as the single case. Our implementation performs three combined inner products per iteration (one being a Euclidean norm; also the two consecutive inner products in BiCGSTAB are combined, as they could be in an individual implementation).

To perform the matrix-vector product Ax, we first collect the global multiplier vector x on each processor [7], then the resulting local product stays on that processor. This means that we combine the communication here by packing the three multiplier vectors x_1, x_2, and x_3 into one buffer x which is broadcast to all participating processors. When the transpose of the matrix is explicitly stored, this is the same procedure for performing $A^T x$.

When the transpose is not stored, we can still combine communication as follows. First perform the local matrix-vector product $x^T A$. This results in a *partial sum* of the global product. Each processor needs the partial sums of the rows it is responsible for from each processor, so this is packed in the above buffer for collecting the global multiplier.

That is,

1. $x^T A$ is performed in parallel,
2. the buffer is packed and broadcast, then
3. the local matrix-vector products Ax are performed.

By combining these operations where possible, the bombardment scheme requires eight communications per iteration, compared with five for CGS, seven for BiCGSTAB, and six for QMR (ignoring preconditioning). The savings involved in the preconditioning step is a function of the structure of the preconditioner. For example, if we apply diagonal scaling, no global communication is required, so there is no savings. However, if an incomplete factorization is used, a relative savings will occur, depending on the requirements of the solve.

4 Some Numerical Results

In this section we present some examples as justification for the bombardment approach. For comparison purposes, we define the *best* algorithm as the one that computes the solution in the least amount of elapsed time.

4.1 Implementation Details

- Software
  - All codes were written in ANSI standard Fortran 77.
  - The *Distributed Iterative Linear System Solvers* [8] research software was adapted to the bombardment algorithm.
  - Also, we have adapted the PIM package [4]. In addition to writing the bombardment algorithm, we changed the communication interface to the BLACS [6]. This allows for

[7] Actually, the structure of the matrix determines how much of the global multiplier vector is needed. For example, if the matrix is block tridiagonal, such as arises in five-point discretization methods, only nearest-neighbor information may be needed.

portability of the code among the various platforms, while giving optimized communication patterns (especially useful for the global sums required by the inner products), at a negligible cost due to the added programming layer [21].

- Hardware
  - Executed on an Intel iPSC860 Gamma [7] at Oak Ridge National Laboratory (ORNL).
  - Virtual parallel machines were formed using Sun SPARCstation IPX workstations using PVM [13] over ethernet.

For stopping criteria we use a tolerance `TOL >` $\|r_k\|/\|b\|$. Since we use the initial guess $x_0 = 0$, this is equivalent to `TOL >` $\|r_k\|/\|r_0\|$, i.e. we require that the initial residual is sufficiently reduced. We note that this is not necessarily the optimal stopping criteria since the actual accuracy of the reported solution is dependent upon the relationship between the norms of the matrix, the right-hand-side and the true solution. However, for the examples we offer here, this is a reasonable choice. For an overview of stopping criteria, see [2]. For the right hand side we use the unit vector $b = [1, \ldots, 1]^T$.

4.2 Distributed Memory Parallel Processing Experiments

In a distributed memory parallel processing environment, we can combine the communication of the three algorithms required for the matrix-vector products, preconditioner solvers, and inner products. The actual time savings depends on the structure of the matrix and preconditioner, and the resulting efficiency of the matrix-vector multiplier and preconditioner solver, as well as the latencies involved with message passing. The following experiments were run on the Intel iPSC860 multiprocessor machine at Oak Ridge National Laboratory [7] and clusters of workstations which communicate over ethernet using PVM. The overhead and latency of other machines, as well as floating point performance, will affect these results. Note that *time*, unless otherwise noted, refers to *wall clock* time.

Example 1: Random sparse matrix

Our first example involves a matrix with random sparsity so that an efficient matrix-vector product cannot be designed, and so that no method will converge or breakdown. This allowed us to perform the algorithm for a fixed number of iterations (5000), and compare the times for each method individually and the time for the poly-iterative method. For example, executing on eight processors of the Intel iPSC860, the respective times per iteration for CGS, BiCGSTAB, and QMR are 0.0274, 0.0276, and 0.0282 seconds. Bombardment took 0.0298 seconds per iteration, only 8.8% longer than CGS, 8.0% longer than BiCGSTAB, and 5.7% longer than QMR. These timings in some sense may be interpreted as the best case for bombardment since each processor must communicate with all the others, and the messages sent during the matrix-vector products are as long as they would ever be. Subsequent examples involve well-structured matrices so that the matrix-vector product can be optimized in order to minimize communication.

Example 2: The Poisson Problem

Mathematicians have spent, and are spending, a great deal of time trying to identify the properties for which a particular method is optimal. For example

- CGS tends to quickly diverge when the initial guess is *close* to the exact solution. Therefore, this method should probably be avoided when solving time-dependent problems.

- BiCGSTAB tends to breakdown when the imaginary parts of the eigenvalues are large relative to the real parts.
- QMR is designed to avoid the breakdown situations that may arise with CGS and BiCGSTAB, but we have found that it is prone to stall.

Yet mysteries still remain, and careful analysis of the coefficient matrix may or may not provide clues as to which method to use. Additionally, even small perturbations may change these properties so that the method that worked well before no longer works at all. And even with this analysis, rounding errors may alter our prediction.

We illustrate this problem using the 2-D Poisson problem. In its basic form, the resulting symmetric positive definite matrix is easily solved by all three methods. Yet if we perturb the basic PDE, so that symmetry or definiteness is altered, a method that previously worked well may breakdown, stall, or diverge. Mathematical reasons could probably be found to explain this behavior, but when a user just wants the solution, the extra time and workspace needed by algorithmic bombardment may be justified. Below are some experiments run on distributed memory parallel machines as well as networks of work stations. They involve perturbations of the 2-D Poisson equation, solved using central differences on square grids. The goal of these experiments is to illustrate two things:

1. the difficulty in selecting the *best* algorithm, and
2. the use of the bombardment scheme is not much more expensive than using an individual routine.

We first consider the effects of the problem size on elapsed time. Suppose we wish to solve [8]

$$-\epsilon\left(\frac{\partial^2 u}{\partial x^2}+\frac{\partial^2 u}{\partial y^2}\right)+\cos(\alpha)\frac{\partial u}{\partial x}+\sin(\alpha)\frac{\partial u}{\partial y}=0. \tag{3}$$

with $\epsilon=-1/10, \alpha=-\pi/6$, on square grids ranging from dimension 100 (order 10,000 matrix) to 400 (order 160,000 matrix) on eight processors of the following parallel machines:

- The Intel iPSC860 (60 Mflop/s per node[9]) and
- SUN SPARCstation IPX workstations using PVM over ethernet.

As expected, as the size of the problem increases (and thus the number of floating point operations increases), the difference between executing the *best* algorithm (BiCGSTAB) and the bombardment algorithm increases (see Figures 2 and 3).

Because QMR takes many more iterations to converge than BiCGSTAB (see Table 3), the time to solution for QMR is greater than the time to find the solution using bombardment. This difference is of course more pronounced for the PVM implementation.

Again, the *best* algorithm is the one that gives us an accurate solution in the shortest amount of time, regardless of the number of iterations performed. This means the best algorithm could change based on the computing environment. For example, CGS takes more iterations to find the solution for these examples than does BiCGSTAB, and at first glance it appears that these two algorithms require about the same amount of work to perform an iteration. But BiCGSTAB requires an extra global communication step to accomplish the two extra inner products per iteration it must perform. In the Intel environment, where communication latencies are not high, BiCGSTAB is the fastest algorithm. However when the individual nodes are connected via ethernet, as is the case with the PVM experiments, the extra communication becomes significant. The gap closes, and in fact

[8] This problem was used in Sonneveld's paper presenting CGS [18].
[9] Millions of floating point operations per second.

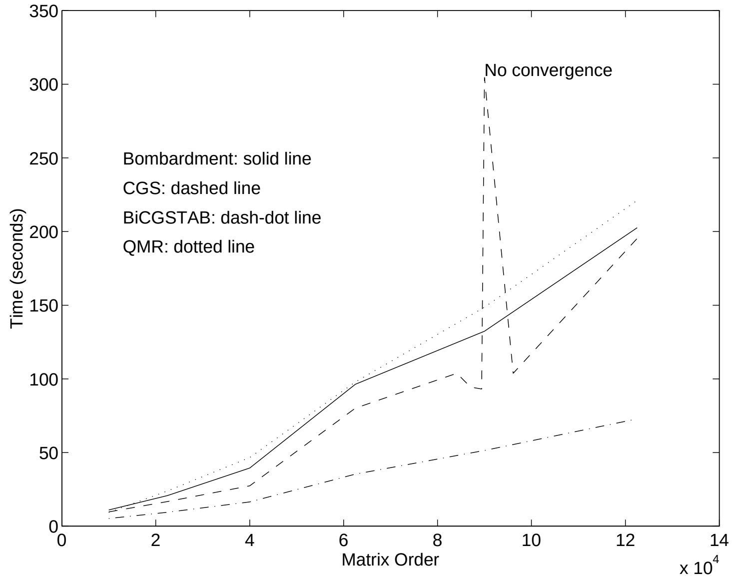

Figure 2: Times to solution on the Intel i860
Using 8 processors of the Intel iPSC860, with no preconditioning, we apply the individual algorithms and the bombardment algorithm to $-\frac{1}{10}\left(\frac{\partial^2 u}{\partial x^2}+\frac{\partial^2 u}{\partial y^2}\right)+\cos\left(\frac{-\pi}{6}\right)\frac{\partial u}{\partial x}+\sin\left(\frac{-\pi}{6}\right)\frac{\partial u}{\partial y}=0$, discretized on a square grid.

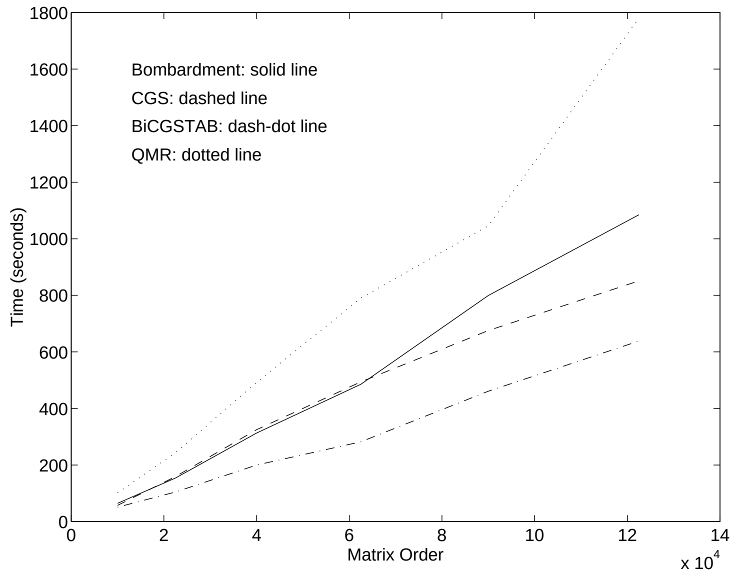

Figure 3: Times to solution on Sun SPARCstation IPX workstations
Using a parallel machine consisting of 8 SPARC IPX workstations connected with ethernet using PVM, with no preconditioning, we apply the individual algorithms and the bombardment algorithm to $-\frac{1}{10}\left(\frac{\partial^2 u}{\partial x^2}+\frac{\partial^2 u}{\partial y^2}\right)+\cos\left(\frac{-\pi}{6}\right)\frac{\partial u}{\partial x}+\sin\left(\frac{-\pi}{6}\right)\frac{\partial u}{\partial y}=0$, discretized on a square grid.

Table 3: Number of Iterations to Solution

This table lists the number of iterations required to find the solution to $-\frac{1}{10}\left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}\right) + \cos\left(\frac{-\pi}{6}\right)\frac{\partial u}{\partial x} + \sin\left(\frac{-\pi}{6}\right)\frac{\partial u}{\partial y} = 0$, discretized on a square grids (the order of the resulting matrix is the square of the grid size). The first line for a grid size is from 8 processors of the iPSC860 and the second line is from 8 Sparc IPX workstations connected with ethernet using PVM. "*" denotes a failure to converge. (However, for grid size 295, CGS converges in 970 iterations and for grid size 310, it converges after 1081 iterations.) "-" denotes that the data would not fit in the memory of the machine for that grid size.

Grid	CGS	BiCGSTAB	QMR
100	181	145	296
	181	156	296
150	356	205	459
	271	220	479
200	427	289	581
	480	269	587
250	1034	361	765
	1157	377	765
300	*	532	1164
	658	423	911
350	1529	557	1142
	1589	522	1287
400	-	-	-
	1247	597	1246

CGS converges faster for some matrix sizes. Although this result may be attributed to other network traffic, it is the nature of ethernet message passing. The time spent in communication provides the insight into why this is happening. Figures 4 and 5 show the proportion of the time to solution spent in message passing as opposed to floating point computation. As expected, the gap is a function of the interconnection network. As expected, the floating point operation requirements increase as the problem size increases, although the startup time to send a message remains constant.

We particularly note the difference in required iterations on the different machines (see Table 3). This is due to the way the arithmetic is performed by the floating point unit. The SPARCstation IPX uses IEEE arithmetic while the i860 does not. The i860 chip is designed to produce more accurate computations, but since these iterative solvers are not self-correcting, any inexact arithmetic alters convergence patterns, and *more accurate* does not necessarily correlate with fast convergence. In fact, experiments have shown that an algorithm may converge on one machine yet fails to converge on another [3]. This is illustrated here. For a grid size of 300, the IPX finds the solution, while the iPSC does not. (However, the iPSC does converge for grid sizes slightly smaller and slightly larger than 300.)

These experiments involved only 8 processors of the Intel machine so that results could be compared with a network of workstations. It is of interest, however, to see how our implementation performs on much larger problems, so we performed this experiment using 128 processors of the Intel iPSC860.

Again, we will apply bombardment to Equation (3) on square grids, ranging from dimension 400 (order 160,000 matrix) to 1500 (order 2,250,000 matrix). The time to solution of the bombardment algorithm as well as the individual algorithms are shown in Figure 6.

We see that there are no surprises with respect to time to the solution for the winning algorithm

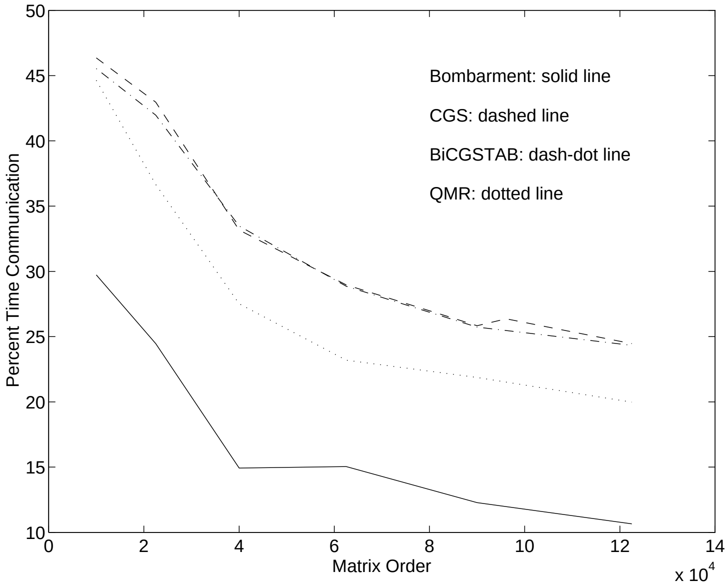

Figure 4: Percentage communication time on the Intel i860

Using 8 processors of the Intel iPSC860, with no preconditioning, we apply the individual algorithms and the bombardment algorithm to $-\frac{1}{10}\left(\frac{\partial^2 u}{\partial x^2}+\frac{\partial^2 u}{\partial y^2}\right)+\cos\left(\frac{-\pi}{6}\right)\frac{\partial u}{\partial x}+\sin\left(\frac{-\pi}{6}\right)\frac{\partial u}{\partial y}=0$, discretized on a square grid.

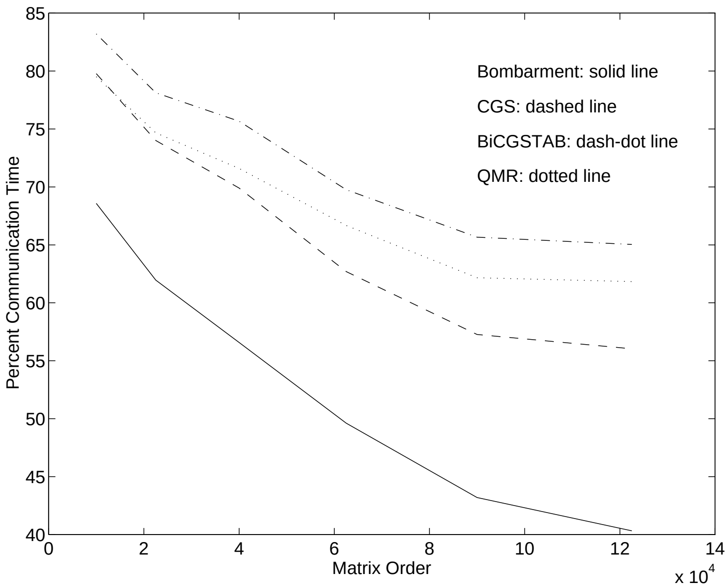

Figure 5: Percentage communication time on SPARC IPX workstations

Using a parallel machine consisting of 8 SPARC IPX workstations connected with ethernet using PVM, with no preconditioning we apply the individual algorithms and the bombardment algorithm to $-\frac{1}{10}\left(\frac{\partial^2 u}{\partial x^2}+\frac{\partial^2 u}{\partial y^2}\right)+\cos\left(\frac{-\pi}{6}\right)\frac{\partial u}{\partial x}+\sin\left(\frac{-\pi}{6}\right)\frac{\partial u}{\partial y}=0$, discretized on a square grid.

(in this case BiCGSTAB) and bombardment. As the problem size increases, BiCGSTAB remains about twice as fast as bombardment. Again, the time spent in communication provides the insight into why this is happening. Figure 7 shows the proportion of the time to solution spent in message passing as opposed to floating point operations.

Notice the effects of the grid size upon convergence of CGS, which fails for these finer meshes. Looking back at the coarser meshes in the 8 processor experiments, this is not completely unexpected. For comparison purposes, we iterated for 2500, 5000, and 10000 iterations [10] for grid sizes of $500 \times 500, 1000 \times 1000$, and 1500×1500, respectively.

Next, we present some experiments in which bombardment finds the solution but one or more of the included algorithms fail.

Example 3: BiCGSTAB is preferable.

Solving Equation (3) on a 200×200 grid (40,000 variables) and no preconditioning[11], we set $\epsilon = 1/100$ and $\alpha = -\pi/6$. BiCGSTAB converges while neither CGS nor QMR converge. (See Table 4 for timings, and residual norm histories in Figure 8.)

Table 4: Performance on Intel iPSC/860
Time (in seconds) to solution for solving perturbations of the Poisson equation. * means convergence not achieved.

Example	CGS	BiCGSTAB	QMR	Bombardment
1	1.37e2	1.38e2	1.41e2	1.49e2
3	*	4.78e1	*	1.26e2
4	*	*	7.98e0	1.32e1
5	4.01e2	5.02e2	*	8.96e2

The next two examples perturb

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} + \alpha(\frac{\partial u}{\partial x} + \frac{\partial u}{\partial y}) + \beta u = 1. \tag{4}$$

Example 4: QMR is preferable.

On a 100×100 grid (10,000 variables) and using no preconditioning, we perturb β ($\alpha = 0$). QMR converges, yet CGS and BiCGSTAB fail to converge. (See Table 4 for timings.)

Example 5: CGS is preferable.

Solving Equation (4) on a 400×400 grid (160,000 variables) and using block ILU preconditioning [12][1], we perturb β ($\alpha = 0$). BiCGSTAB converges, but CGS converges faster. QMR fails to determine the solution (even after 5000 iterations). (See Table 4 for timings.)

5 Sharing Information Between Algorithms

Since the individual algorithms are iterating in lock-step, it is tempting to *share* the information from the methods that appears to be working best with the others. For example, if CGS is closer to

[10] Actually, we iterated for far more iterations to convince ourselves that convergence would not be achieved.
[11] The result is similar when D-ILU preconditioning is used.
[12] Block ILU preconditioning requires no communication.

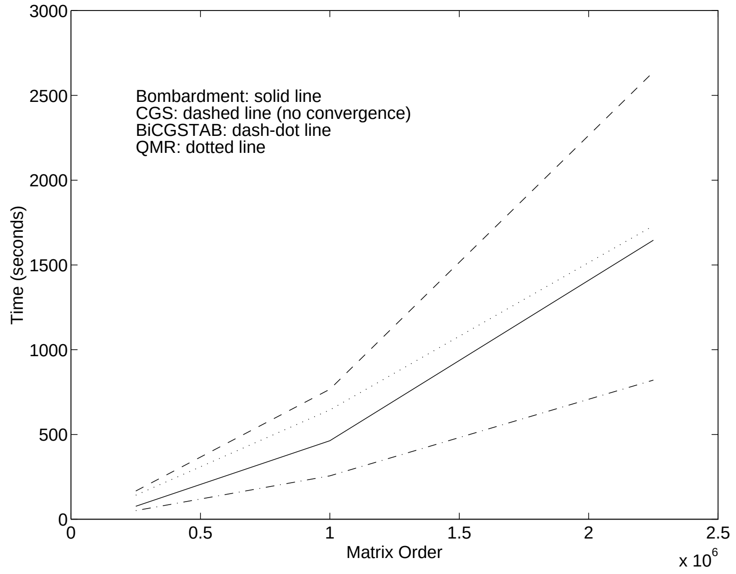

Figure 6: Times to solution on the Intel i860
Using 128 processors of the Intel iPSC860, with no preconditioning, we apply the individual algorithms and the bombardment algorithm to $-\frac{1}{10}\left(\frac{\partial^2 u}{\partial x^2}+\frac{\partial^2 u}{\partial y^2}\right)+\cos\left(\frac{-\pi}{6}\right)\frac{\partial u}{\partial x}+\sin\left(\frac{-\pi}{6}\right)\frac{\partial u}{\partial y}=0$, discretized on square grids ranging from dimension 400 to 1500.

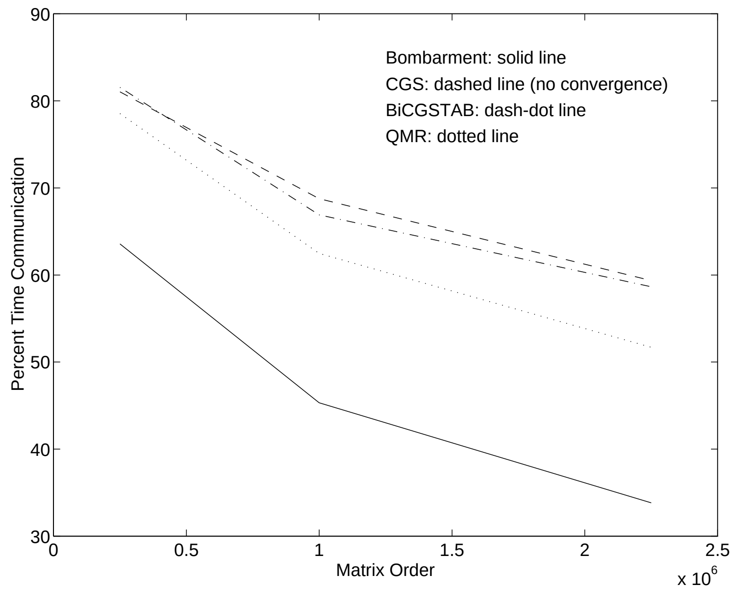

Figure 7: Percentage communication time on the Intel i860
Using 128 processors of the Intel iPSC860, with no preconditioning, we apply the individual algorithms and the bombardment algorithm to $-\frac{1}{10}\left(\frac{\partial^2 u}{\partial x^2}+\frac{\partial^2 u}{\partial y^2}\right)+\cos\left(\frac{-\pi}{6}\right)\frac{\partial u}{\partial x}+\sin\left(\frac{-\pi}{6}\right)\frac{\partial u}{\partial y}=0$, discretized on square grids ranging from dimension 400 to 1500.

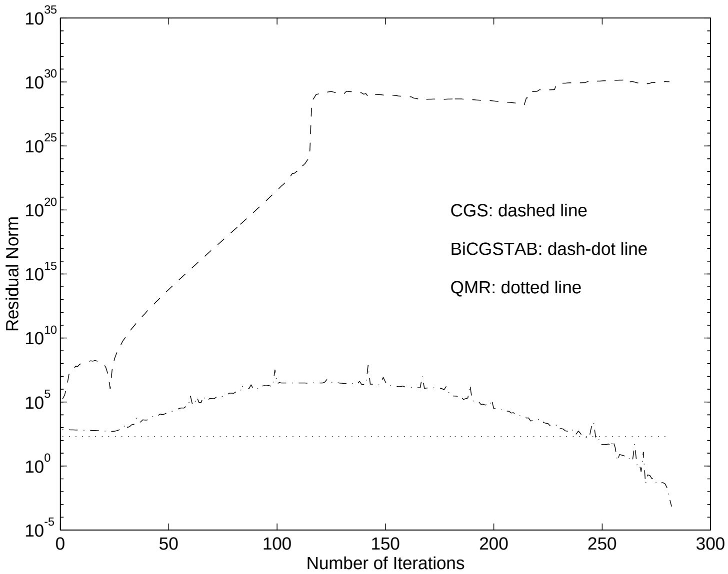

Figure 8: Parallel example: BiCGSTAB wins
The residual norm history of each algorithm, using D-ILU preconditioning, applied to $-\frac{1}{100}\left(\frac{\partial^2 u}{\partial x^2}+\frac{\partial^2 u}{\partial y^2}\right)+\cos\left(\frac{-\pi}{6}\right)\frac{\partial u}{\partial x}+\sin\left(\frac{-\pi}{6}\right)\frac{\partial u}{\partial y}=0$, discretized on a 200 grid. (8 processors of Intel i860 gamma at ORNL.)

the solution than BiCGSTAB and QMR, why not *restart* them using the current iterate and residual from CGS? This idea fails, because the methods depend on the full Krylov space built up during the iterative process. Restarting causes this space to be cut short, and in effect the iterative process to start anew. In fact, a restart too close to the solution may cause divergence of some methods.

This would have the effect of projecting the iterate of one algorithm onto the Krylov subspace of another. Because Algorithmic Bombardment facilitaties the sharing of information between algorithms, we experimented with this idea.

There are two ways to understand why this idea fails. First, we could examine the effects on the various parameters of the algorithms. The problem is most easily seen in CGS with the computation of $\beta = \rho_{i-1}/\rho_{i-2} = \tilde{r}^T r_{i-1}/\tilde{r}^T r_{i-2}$. In the step immediately following the restart, β will be smaller than it would have been without a restart. This causes a smaller than expected change in p and q, carrying down to a smaller updating of the solution and residual. It is during the next step where the big problem occurs. Now the denominator in the computation of β is smaller than expected, while the numerator is about the same size as it was during the previous step, causing β to become *too large*. This cascades down to the approximation, where the updating overshoots the solution. Since the choice of an initial guess doesn't matter, we might expect the algorithm to settle down and begin converging again. But these algorithms use information from *all* previous search directions, so the root cause of the problem is that we have interfered with that process, contaminating all previous work. Each algorithm builds up a *different* Krylov subspace in an attempt to find the solution, and while it is true that each algorithm operates on the same matrix, they do so in different ways.

This illustrates two important characteristics of these algorithms:

1. The initial guess doesn't matter (except with CGS, which is likely to *diverge* if x_0 is *too close* to the true solution, and

2. each method must build up, and remain in, its own Krylov subspace, based upon the algorithm and the spectrum of the matrix.

6 Conclusions

Many algorithms have been developed for solving large sparse nonsymmetric linear systems which use short recurrences. The downside is that convergence is no longer guaranteed, nor predictable in practice. Therefore we have incorporated three of these algorithms into a poly-iterative scheme, so that we may apply them simultaneously to the same data set. We have shown through various experiments that this increases the chance of finding the solution, and in a parallel environment this does not increase the time to solution threefold. In fact, even when all three algorithms would have found the solution, bombardment may be faster than the slowest of the three.

The expected performance of a given application is dependent upon the combination of the structure of the matrix (sparsity, structure, etc.), the data structure used, the preconditioner, and these effects upon the performance of the matrix-vector product and preconditioner solver. For example, if the matrix is well-structured, a matrix-vector product can usually be implemented that requires a small amount of communication. Also, if the matrix has a large number of nonzeros, it may be possible to reduce the effects of the indirect addressing of the matrix-vector product.

Ultimately, the performance of the computing environment determines the performance. The new Cray T3D is expected to have much lower communication overhead and latency than the Intel iPSC/860. On the other hand, workstation clusters connected using PVM [13] exhibit high latency, and are dependent upon the traffic interconnection network, often the Internet. Regardless, the increased probability of convergence should justify using poly-iteration.

7 Future Work

The experiments presented above are frequently encountered in the scientific world, hence we believe the results justify our implementation of the poly-iterative idea, including the choice of algorithms as well as the scheme for performing the matrix-vector product and preconditioning. However, different approaches may be more appropriate depending upon the problem being solved. For example, some applications require solving many linear systems in a sequence of time steps. Since the matrix may not change significantly from one step to the next, it has been suggested that perhaps bombardment could be used during one such solve, then only the *winning* algorithm would be used for the next few solves, then back to bombardment, and so on.

Perhaps incorporating more GMRES concepts into the poly-iteration would be valuable in some cases. We originally ruled out using this valuable algorithm because of its linearly increasing workspace requirements, yet perhaps we can find a way to overcome this limitation while still gaining performance. BiCGSTAB is actually the combination of BiCG and GMRES(1). Recent work [20] shows that increasing the effects of GMRES can be worthwhile, such as combining GMRES(2) or GMRES(4) with BiCG.

There is limited freedom in varying the preconditioners over the methods. For any but totally parallel preconditioners we want to combine the communication step, which basically forces the same preconditioner structure on the methods. Still, if the preconditioner has some form of relaxation parameter, this can be varied independently for the different methods. Similarly, we could precondition one method with SSOR and another with ILU, since these have the same communication structure.

Further research into different matrix-vector product implementations may yield higher computational performance in some situations. For example, certain matrix structures may allow higher efficiency. One possibility would be to interleave the elements of the multiplier of the algorithms in order to force less indirect addressing, which slows the floating point performance. Dense matrix computations perform $O(n^3)$ operations on $O(n^2)$ data. But for sparse matrices, this is actually a vector-vector operation ($O(n)$ operations on $O(n)$ data), with the added degradation of indirect addressing. And since three such operations must be performed, the effect is magnified. This can be reduced in the bombardment scheme. Suppose the multipliers are $x^{CGS}, x^{BiCGSTAB}$, and x^{QMR}. The obvious way to compute $Ax^{CGS}, Ax^{BiCGSTAB}$, and Ax^{QMR} is to perform the operations sequentially. But the elements can be interleaved as

$$x = [x_1^{CGS}, x_1^{BiCGSTAB}, x_1^{QMR}, \ldots, x_n^{CGS}, x_n^{BiCGSTAB}, x_n^{QMR}]^T,$$

reducing the effects of indirect addressing threefold. Note that this scheme will cause indirect addressing of some vector updates, so its overall effect is dependent upon the number of nonzeros in the matrix.

References

[1] O. Axelsson. Incomplete block matrix factorization preconditioning methods. the ultimate answer? *J. Comp. Appl. Math.*, 12&13:3–18, 1985.

[2] Richard Barrett, Michael Berry, Tony F. Chan, James Demmel, June Donato, Jack Dongarra, Victor Eijkhout, Roldan Pozo, Charles Romine, and Henk van der Vorst. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods.* SIAM, 1994.

[3] Richard F. Barrett. Algorithmic bombardment for the iterative solution of linear systems: A poly-iterative approach. Master's thesis, University of Tennessee, 1994.

[4] Rudnei Dias da Cunha and Tim Hopkins. Pim 1.1: The parallel iterative methods package for systems of linear equations user's guide. Technical report, University of Kent at Canterburg, 1993.

[5] James Demmel, Michael T. Heath, and Henk A. van der Vorst. Parallel linear algebra. *Acta Numerica*, 2, 1993.

[6] J.J. Dongarra and R. Clint Whaley. Lapack working note 94: A users' guide to the blacs. Technical Report CS-95-281, Computer Science Department, University of Tennessee, 1995.

[7] T. H. Dunigan. Performance of the intel ipsc/860 hypercube. Technical Report ORNL/TM-11491, Oak Ridge National Laboratory, Oak Ridge, Tennessee, 1990.

[8] Victor Eijkhout. A library of distributed iterative linear system solvers. In *Proceedings of the 14th World Congress on Computation and Applied Mathematics*, 1994.

[9] R. Fletcher. Conjugate gradient methods for indefinite systems. In G.A. Watson, editor, *Numerical Analysis Dundee 1975*, pages 73–89, New York, 1976. Springer Verlag.

[10] Roland W. Freund. Conjugate gradient-type methods for linear systems with complex symmetric coefficient maitrices. *SIAM J. Sci. Stat. Comp.*, 13:425–448, Jan 1992.

[11] Roland W. Freund and Noël M. Nachtigal. QMR: a quasi-minimal residual method for non-Hermitian linear systems. *Numer. Math.*, 60:315–339, 1991.

[12] Roland W. Freund and Noël M. Nachtigal. An implementation of the QMR method based on coupled two- term recurrences. *SIAM J. Sci. Comp.*, 15(2):313–337, Mar 1994.

[13] G. A. Geist, A. L. Beguelin, J. J. Dongarra, R. J. Manchek, and V. S. Sunderam. *PVM: Parallel Virtual Machine; A Users' Guide and Tutorial for Networked Parallel Computing.* MIT, 1994.

[14] G. A. Geist, M. T. Heath, B. W. Peyton, P. H. Worley, and V. S. Sunderam. A users' guide to picl: a portable instrumented communication library. Technical Report ORNL/TM-11130, Oak Ridge National Laboratory, Oak Ridge, Tennessee, 1990.

[15] M.R. Hestenes and E. Stiefel. Methods of conjugate gradients for solving linear systems. *J. Res. Nat. Bur. Stand.*, 49:409–436, 1952.

[16] C. Lanczos. Solution of systems of linear equations by minimized iterations. *Journal of Research, Nat. Bu. Stand.*, 49:33–53, 1952.

[17] Yousef Saad and Martin H. Schultz. GMRes: a generalized minimal residual algorithm for solving nonsymmetric linear systems. *SIAM J. Sci. Stat. Comput.*, 7:856–869, 1986.

[18] Peter Sonneveld. CGS, a fast Lanczos-type solver for nonsymmetric linear systems. *SIAM J. Sci. Stat. Comput.*, 10:36–52, 1989.

[19] Henk van der Vorst. Bi-CGSTAB: a fast and smoothly converging variant of Bi-CG for the solution of nonsymmetric linear systems. *SIAM J. Sci. Stat. Comput.*, 13:631–644, 1992.

[20] Henk van der Vorst, Dec 1993. private communication.

[21] R. Clint Whaley. Lapack working note 73: Basic linear algebra communication subprograms: Analysis and implementation across multiple parallel architectures. Technical Report CS-94-234, Computer Science Department, University of Tennessee, 1994.