To the Graduate Council:

I am submitting herewith a thesis written by Richard Frederick Barrett entitled "Algorithmic Bombardment for the Iterative Solution of Linear Systems: A Poly-Iterative Approach." I have examined the final copy of this thesis for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Master of Science, with a major in Computer Science.

Jack J. Dongarra, Major Professor

We have read this thesis
and recommend its acceptance:

Michael W. Berry

Victor Eijkhout

Charles H. Romine

Accepted for the Council:

Associate Vice Chancellor
and Dean of the Graduate School

STATEMENT OF PERMISSION TO USE

In presenting this thesis in partial fulfillment of the requirements for a Master's degree at The University of Tennessee, Knoxville, I agree that the Library shall make it available to borrowers under rules of the Library. Brief quotations from this thesis are allowable without special permission, provided that accurate acknowledgment of the source is made.

Permission for extensive quotation from or reproduction of this thesis may be granted by my major professor, or in his absence, by the Head of Interlibrary Services when, in the opinion of either, the proposed use of the material is for scholarly purposes. Any copying or use of the material in this thesis for financial gain shall not be allowed without my written permission.

Signature Richard F Barrett

Date 29 April 1994

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

A Thesis

Presented for the

Master of Science Degree

The University of Tennessee, Knoxville

Richard Frederick Barrett

May 1994

Dedication

This thesis is dedicated to Robert A. "Boh" Hrees, who showed me the thrill of mathematics. A man who ran to the window shouting, "it explains how sap runs up a tree, it put man on the moon!" A man who showed me the simple beauty of the unit circle and the Möbius strip. And who told me I must learn linear algebra.

Acknowledgments

This research was supported in part supported in part by ARPA and ARO under contract number DAAL03-91-C-0047, the National Science Foundation Science and Technology Center Cooperative Agreement No. CCR-8809615, the Applied Mathematical Sciences subprogram of the Office of Energy Research, under contract DE-AC05-84OR21400.

Abstract

Many algorithms exist 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. This approach has value in a sequential computing environment, yet it is in a parallel environment where it is even more valuable. Here we combine the global communication, effectively reducing these expensive operations to the cost of one method.

Contents

1 Introduction **1**

1.1 The Krylov Subspace 4

1.2 Preconditioning 6

2 The Algorithms **12**

2.1 Some background 12

2.1.1 Choosing the search directions 17

2.2 Nonsymmetric Linear Systems 20

2.3 Quasi-Minimal Residual (QMR) 23

2.3.1 Preliminary: GMRES 23

2.3.2 A Quasi-Minimization Method 25

2.4 Conjugate Gradient Squared (CGS) 28

2.5 Biconjugate Gradient Stabilized (BiCGSTAB) 31

2.6 Brief Summary 33

3 The Algorithmic Bombardment Algorithm **36**

3.1 Basic idea 36

3.2 Parallel implementation 36

3.3 Structure of the iteration 37

3.4 Cost Model 40

3.4.1 Combined Operations 41

4 Some Numerical Results **44**

4.1 Implementation Details 44

4.2 Sequential Examples 46

4.3 Effects of Different Preconditioning 49

4.4 Distributed Memory Parallel Processing Experiments 53

4.4.1 PVM experiments 57

5 Sharing Information Between Algorithms **67**

6 Conclusions **75**

6.1 Future Work 77

References **80**

Appendices **86**

A Notation **87**

B More Experiments **89**

C Example FORTRAN 77 code **94**

C.1 Combined Inner Product 94

C.2 The Algorithm 95

Vita **98**

List of Tables

1 Synchronized operations 38

2 Summary of operations and workspace requirements 40

3 Performance on Intel iPSC/860 55

4 Experiments: One converges, two do not 90

5 Experiments: Two converge, one does not 90

6 Experiments: All methods converge 91

7 Experiments: No algorithm converges 91

8 Experiments: IEEE versus IBM RS6000 arithmetic 92

9 Experiments: All results 93

List of Figures

1 Common behavior of each algorithm 34

2 Sequence of Operations 39

3 Sequential example: CGS wins 47

4 Sequential example: BiCGSTAB wins 48

5 Sequential example: QMR wins 49

6 Preconditioning effects 50

7 Preconditioning effects 51

8 Preconditioning effects 53

9 Parallel example: BiCGSTAB wins 58

10 Times to solution on the Intel i860 60

11 Times to solution on Sparc IPX workstations 61

12 Number of iterations to solution on the Intel i860 62

13 Number of iterations to solution on Sparc IPX workstations 63

14 Percentage communication time on the Intel i860 64

15 Percentage communication time on Sparc IPX workstations 65

16 Parallel example: CGS fails 66

17 Parallel example: BiCGSTAB wins 67

18 All methods converge. 69

19 Sharing information from BiCGSTAB 70

20 Sharing information from BiCGSTAB (once) 71

21 Sharing information from QMR 72

22 Sharing information from QMR (once) 73

23 Sharing information from CGS 74

1 Introduction

Many iterative methods for solving real nonsymmetric linear systems

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

have been proposed. A popular choice is GMRES, which 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 [1]. Variations for reducing the length of the recurrences have been proposed (using restarting or truncation), but this sacrifices the convergence theory. 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 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, this is often true with nonlinear problems, where many linear

[1] Indeed, there are examples where GMRES with *any* length recurrence less than the full 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.

systems are generated, and the properties of the matrix change. Also, in practice an algorithm may stall, diverge, or breakdown even when the matrix properties indicate the method should converge.

Moreover, in practice these properties may be so affected by rounding error that we lose the ability to predict their behavior. In fact, machines using different methods for performing the arithmetic may yield different results. For example, the IBM RS6000 performs the multiply and add before truncating, as opposed to say a Sparc ipx which uses IEEE arithmetic, truncating after the multiply and again after the add. While the IBM is designed to be more accurate, it is still finite arithmetic, and with these iterative solvers, any inaccuracy may alter the behavior of the algorithm. We should note that our experiments show that this is not always a negative thing. Rounding error may actually aid convergence. (For examples illustrating the phenomenon, see section 4).

In response to this problem, we propose applying many iteration schemes, in parallel using the same data set, in the hope that at least one will yield an acceptable approximation to the true solution. Although this means that the number of operations per iterative step equals the sum of the operations of the individual methods, we believe that when unsure of the matrix properties, the extra floating point work and memory requirements are outweighed by three factors:

1. 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) (discussed in section 4.4).

2. 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.

3. An increased probability of finding the solution. (Examples in section 4.)

The high cost of communication is the weak link in distributed memory architectures, so by combining the distribution of global data, we can improve upon the performance of the poly-iterative approach. For example, since each method is operating on the same matrix, and each matrix vector operation requires communication to update the product vector, we may be able to save time by combining the k product vectors, where k is the number of methods whose matrix-vector product operands we can combine into one buffer for communication. Additionally, when the structure of a matrix allows it, we can multiply an $n-$dimensional matrix by a $n \times k$ matrix, rather than a single vector, thereby improving the floating point efficiency of the computation [5]. Moreover, we can combine other communication, such as that required for the computation and distribution of inner products and residual norms. This is

discussed in more detail in section 3.4.1.

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 matrix.

For our implementation, we have chosen three algorithms which approach the problem differently [3].

- Quasi-Minimal Residual (QMR),
- Conjugate Gradient Squared (CGS), and
- Biconjugate Gradient Stabilized (BiCGSTAB).

1.1 The Krylov Subspace

Before we continue, it is worth discussing a theoretical characteristic of the algorithms dealt with here. Each method begins with an initial guess x_0, then improves upon it using the span of $\{r_0, Ar_0, A^2r_0, \ldots\}$, where $r_i = b - Ax_i$. That is, they generate iterates in a *Krylov* subspace,

$$x_n \in x_0 + \mathcal{K}_n(r_0, A), \text{ for } n = 1, 2, \ldots$$

[3] While all are based on the idea of biconjugate gradients, and therefore theoretically converge or diverge together, experiments have shown that one may converge much faster than the others, may breakdown, or fail to converge.

We say that $\mathcal{K}_n(r_0, A)$ is the $n-th$ Krylov subspace generated by r_0 and A.

It is also instructive to view these iterates in terms of polynomials in A, i.e. $\mathcal{K}_n(r_0, A) = \{P(A)r_0 : P \in \mathcal{P}\}$, leading to the idea that

$$x_n \in x_0 + P(A)r_0 \text{ and } r_n = P_n(A)r_0, \text{ normalized so that } P(0) = 1.$$

This polynomial perspective is useful in understanding the strategies for developing the algorithms, as well as polynomial preconditioning.

We will find the approximate solution only by causing r_i to shrink to zero. Two basic strategies are:

1. Minimize the residual norm

$$\|r_n\|_2 = \min_{x \in x_0 + \mathcal{K}_n(r_0, A)} \|b - Ax\|_2, \tag{2}$$

or

2. Force a Galerkin-type condition:

$$s^T r_n = 0 \text{ for } s \in \mathcal{S}_n \subset \mathbf{R}^N. \tag{3}$$

Either of these conditions can be met through some construction of a basis set

in the Krylov subspace and $\mathcal{S}_n$. Algorithms based on (1) are called minimal residual methods (e.g. GMRES), and algorithms based on (2) are called orthogonal residual methods. The problem is that it has been shown [15] that for nonsymmetric matrices, neither of these conditions can be met using short recurrences, meaning that work and memory requirements will increase linearly with the number of iterations. So methods have been developed which attempt to mimic these conditions using short recurrences. We outline some of them here, but first we discuss a way to speed up convergence by altering the spectrum of the matrix.

1.2 Preconditioning

The goal is to solve $Ax = b$, but speed of convergence, (or even convergence at all) depends on the the spectrum of the matrix and how the method exploits those properties. Preconditioning gives us a tool to *attempt* to manipulate the spectrum.

Basic linear algebra lets us rewrite equation (1) as follows: For some nonsingular $M \in \mathbf{R}^{n \times n}$

$$M_1^{-1} A M_2^{-1} y = M^{-1} b, \quad \text{where} \quad M = M_1 M_2 \quad \text{and} \quad y = Mx. \tag{4}$$

When $M = M_1$ (i.e. $M_2 = I$) this is called *left* precondtioning, and when $M = M_2$ (i.e. $M_1 = I$) this is called *right* preconditioning. The full formulation 4 is called *full*,

or *left and right* preconditioning. Note that if A is symmetric positive definite and $M_1 = M_2^T$, then $M_1 A M_2 x$ is also symmetric positive definite. This property is not preserved for $M^{-1}A$ or AM^{-1}.

So we should try to find a matrix M so that equation (4) is "easier" to solve than equation (1). Some basic considerations for choosing such a *preconditioner* are apparent, and must be balanced in terms of cost and effect:

1. M should be "close" to A. This will force $M_1^{-1} A M_2^{-1}$ closer to the identity, causing fewer iterations to be required to find the solution. That is, M^{-1} should approximate A^{-1}. In terms of conditioning and the spectrum, we want

$$\kappa_2(M_1^{-1} A M_2^{-1}) < \kappa_2(A).$$

2. M should be "easy" to compute and store. Time spent forming the preconditioner is time that could be spent iterating. And the storage of the preconditioner limits the size of the system that may be solved.

3. M should be "easy" to invert. Since applying the preconditioner adds work to each iteration, that work must be balanced by reducing the number of iterations required of the algorithm.

Polar choices would be $M = A$ and $M = diag(A)$. The first gives "perfect" pre-

conditioning (in terms of convergence), but we are trying to avoid actually computing the inverse. The second satisfies all three of the above criteria, but may not reduce the condition number significantly.

It's worth noting that when applying the preconditioned algorithms to equation (1) we don't have to actually form $M^{-1}A$ or AM^{-1}. For example, with right preconditioning, by observing that the matrix A is directly involved in the algorithm only during the matrix-vector products, we compute $AM^{-1}x$ as follows:

1. $\tilde{x} = M^{-1}x$, then

2. $A\tilde{x} = AM^{-1}x$.

Since we are trying to approximate the inverse of matrix A, a reasonable starting point would be to look at how the exact inverse would be formed in practice, i.e. A is usually factored into LDU, using techniques of Gaussian elimination, where L, D, and U are lower, diagonal, and upper triangular factors [28], respectively. A problem with this method with respect to sparse matrices is the large amount of extra storage caused by "fill", i.e. nonzero entries that appear in the factorization corresponding to zero entries in the original matrix. Still, the ideas involved can be applied to sparse matrices using "incomplete" factorizations. We present some below.

- Diagonal scaling (also called Jacobi preconditioning):

- $M = diag(A) = a_{ii}$ for $i = 1, \ldots, n$ (i.e. $L = U = I$). This is most effective when the matrix is diagonally dominant, but since it is so inexpensive to form and apply (including in parallel), it is often used in conjunction with more sophisticated preconditioners.

- SSOR

  - $M = (L + D)D^{-1}(D + U)$, where $A = L + D + U$. Therefore, no memory is needed to store the preconditioner elements, only pointers to these submatrices. The cost of applying this preconditioner is proportional to the cost of the matrix-vector product. Computation can be reduced by applying the preconditioner using a technique developed by Eisenstat [13], which rewrites the factors L and U as $\tilde{L} = LD^{-1} + I$ and $\tilde{U} = I + D^{-1}U$, then reassociates and combines the various factors. (This is equivalent to applying one iteration of symmetric Gauss-Seidel [24].)

- Incomplete Factorizations:

  - If the inverse of A is the perfect preconditioner, then it may be reasonable to *approximate* the inverse by $A \approx \tilde{L}\tilde{U}$. The algorithm for forming $\tilde{L}$ and $\tilde{U}$ modifies the regular factorization $A = LU$ by ignoring the zero entries of A. (This explains the name "incomplete factorization".) Therefore, the

resulting structures of $\tilde{L}$ and $\tilde{U}$ are the same as that of A, and it is hoped that the information lost is offset by the practicality of forming, storing, and applying this preconditioner.

- A modification of the above ILU factorization is to allow some fill to the structures of $\tilde{L}$ and $\tilde{U}$. The idea is to get more information from A while controlling the resulting structure of the factors (this is notated $ILU(m)$)
- Block ILU. Matrix A is partitioned into *blocks*, and an incomplete factorization is performed on each block. This is often effective when the domain is divided using a cartesian product grid. In a distributed memory environment this has the advantage over regular incomplete factorizations as the application of the preconditioner is a local computation.
- Note that for a symmetric positive definite matrix A, an incomplete Choleski factorization is formed.

• D-ILU [24]

- This preconditioner is a variant of SSOR that adds information from the matrix to the diagonal matrix D as follows:

$$d_{ii} = a_{ii} - \sum_{k=0}^{i-1} l_{ik} d_{kk} u_{ki}, \quad \text{for} \quad i = 1, \ldots, n.$$

The computation cost is the same as for SSOR (the Eisenstat trick still applies), but storage increases by n elements since $D \neq D_A$. Note that the structure of the matrix plays a role in the interpretation of this preconditioner. For example, if A is tridiagonal (common in 1-D device simulation problems), this is an *exact* factorization. For single variable finite difference discretizations, this is equivalent to $ILU(0)$.

- Polynomial preconditioning
  - If the spectral radius of matrix B is less than one, where $A = I - B$, then $A^{-1} = \sum_{k=0}^{\infty} B^k$ (Neumann series). Therefore, A^{-1} may be approximated by truncating this sum. This lends itself well to an analysis of acceleration properties when viewing the iterative methods as polynomials involving the matrix applied to the initial residual (for example [9], [1], [2],[22], [25]).
- Application "driven". Often the physical problem implies a way to affect the spectrum. Also, in time-stepping problems, where subsequent matrices may be perturbations of previous matrices, it may be possible to determine information from previous linear system solves for use in subsequent preconditioners.

Finally, we should point out that a poor choice of preconditioner can in fact hinder progress, i.e. $AM^{-1}y = b$ may be more difficult to solve than $Ax = b$, for a particular algorithm. Our experiments will show that the outcome of our bombardment scheme

may change drastically with the application of preconditioning (section 4). On the other hand, if enough information is available to the user concerning a preconditioner, it may be possible to select an optimal algorithm, or any algorithm may work. That is, the existence of a good preconditioner may preclude the need for polyiteration.

2 The Algorithms

In this section we give a brief description of the methods that make up our implementation of the algorithmic bombardment algorithm. First, however, we discuss the basis upon which these algorithms are built, the method of conjugate gradients. Then we show how this method for solving symmetric positive definite systems has been adapted for solving nonsymmetric and/or indefinite systems.

2.1 Some background

We begin with the method of steepest descent, which will be subsequently improved through the development of other algorithms.

If we view the linear system in terms of an energy function $\phi(x)$ and assume the matrix is symmtric positive definite, the solution is found at a point of stable

equilibrium, i.e. where the derivatives are

$$\frac{d\phi}{dx} = Ax - b = 0 \text{ and } \frac{d^2\phi}{dx^2} > 0.$$

The energy function is the antiderivative of $d\phi/dx$:

$$\phi(x) = \frac{1}{2}x^T Ax - x^T b.$$

We can relate the minimization of this energy function to the solution of the linear system, with the added benefit of a geometric viewpoint. One way to show that this is in fact the minimum is to assume that there is another point y such that $\phi(y) < \phi(x)$, and find a contradiction. So assume

$$\begin{aligned} 0 > \phi(y) - \phi(x) &= \frac{1}{2}y^T Ay - y^T b - \frac{1}{2}x^T Ax - x^T b \\ &= \frac{1}{2}y^T Ay - y^T b + \frac{1}{2}x^T Ax \\ &= \frac{1}{2}(y-x)^T A(y-x). \end{aligned}$$

But this contradicts the positive definiteness of A, so it must be true that $\phi(y) > \phi(x)$.

We get a geometric representation of the problem by realizing that $\phi(x)$ describes a parabloid, with minimum point $x = A^{-1}b$. In order to find this minimum x, we

choose an initial guess x_0. This puts us at point $\phi(x_0)$ on the parabloid, which we assume is not equal to $\phi(x) = -\frac{1}{2}b^T A^{-1}b$. So we need a way to move down the parabloid to the actual minimum, thereby solving the linear system. We will do this using the gradient of $\phi(x)$, which will show us the path of greatest change (in a local sense).

Given a "current" position x_c, the gradient $-\bigtriangledown \phi(x_c) = b - Ax_c$ points in the direction of the largest decrease of ϕ. The corresponding residual vector is $r_c = b - Ax_c$, which if nonzero implies that there exists some $\alpha > 0$ such that $\phi(x_c + \alpha r_c) < \phi(x_c)$. We minimize $\phi(x_c + \alpha r_c)$ by setting $\alpha = r_c^T r_c / r_c^T A r_c$. So the algorithm is as follows:

Steepest Descent Algorithm

If $A \in \mathbf{R}^{n \times n}$ is symmetric positive definite and $b \in \mathbf{R}^n$, this algorithm generates approximations x_i for $x \in \mathbf{R}^n$ such that $Ax = b$.

Compute $r_0 = b - Ax_0$ for some initial guess x_0

for $i = 1, 2, \ldots$

$\alpha_i = r_i^T r_i / r_{i-1}^T A r_{i-1}$

$x_i = x_{i-1} + \alpha_i r_i$

$r_i = r_{i-1} - Ax_i$

check convergence; continue if necessary

end

The problem with steepest descent is that unless the matrix is well conditioned, the gradients will be fairly flat, and we won't move down the paraboloid very fast. And since the search directions are based on the residual vectors, we can't expect this to improve. Therefore, we must consider using another set of search directions.

We begin as before, yet replace r_k with p_k, an as yet unnamed search direction. Then $\alpha_k = p_k^T r_{k-1}/p_k^T A p_k$ minimizes $\phi(x_{k-1} + \alpha p_k)$. Then $\phi(x_{k-1} + \alpha_k p_k)$ can be rewritten

$\phi(x_{k-1}) - \frac{1}{2}(p_k^T r_{k-1})^2 p_k^T A p_k$. Note that this shows we will need to prevent p_k and r_{k-1} from becoming orthogonal. Inserting this choice of α into the steepest descent algorithm leads to $x_k = x_{k-1} + \alpha_k p_k$, i.e. x_k is in the span of $\{p_i\}_{i=1}^k$.

Success requires that the true solution x be in the range of $P_k = \{p_i\}_{i=1}^k$. Thus writing $x = P_{k-1} y + \alpha p_k$ leads to

$$\phi(x) = \phi(P_{k-1} y) + \alpha y^T P_{k-1}^T A p_k + \frac{\alpha^2}{2} p_k^T A p_k - \alpha p_k^T b.$$

Requiring

$$P_{k-1}^T A p_k = 0 \tag{5}$$

would greatly simplify minimizing this functional since all previous search directions

would be decoupled from the current p_k. Defining x_{k-1} in the range of P_{k-1} by $\phi(x_{k-1}) = \min_y \phi(P_{k-1}y)$ and $\alpha_k = p_k^T b / p_k^T A p_k$ leads to

$$min_{\alpha,y}\phi(P_{k-1}y + \alpha_k p_k) = \min_y \phi(P_{k-1}y) + \min_\alpha \left\{ \frac{\alpha^2}{2} p_k^T A p_k - \alpha p_k^T b \right\},$$

which is solved using $P_{k-1}y = x_{k-1}$ and $\alpha = \alpha_k$. The numerator of α can be rewritten $p_k^T(Ax_{k-1}) + r_{k-1} = p_k^T r_{k-1}$ since x_{k-1} is a linear combination of P_{k-1}.

This gives us a better idea on how to choose p_k:

1. p_k should be in the span of the orthogonal set $\{Ap_i\}_{i=1}^{k-1}$ such that

2. $p_k r_{k-1} \neq 0$.

Using the fact that $x_{k-1} \in span\{p_i\}_{i=1}^{k-1}$, $r_{k-1} = b - Ax_{k-1} = r_{k-1} = b - AP_{k-1}z$ for some $z \in \mathbf{R}^{k-1}$, we multiply through by p, getting

$$p^T r_{k-1} = p^T b - p^T A P_{k-1} z.$$

This shows that b must also be in the span of $\{Ap_i\}_{i=1}^{k-1}$. So $x = A^{-1}b$ must be in the space spanned by $\{p_i\}_{i=1}^{k-1}$, implying that $x = x_{k-1}$. Equivalently, $r_{k-1} = 0$, which if not true, i.e. we have not yet found the solution, means we can still find a nonzero search direction p_k.

So far we've shown that such search directions can be found under our desired

conditions, and that p_k should be "close" to r_k. It still remains to be shown exactly *how* to choose p_k.

2.1.1 Choosing the search directions

In order to see how to form better search directions, let's look at the problem another way. We are attempting to solve $Ax = b$. If we can transform this system into $Ty = f$, where T is tridiagonal, the system will be easy to solve ($T = LDL^T$ if T is symmetric positive definite).

The *Lanczos* algorithm gives a way of generating such a sequence of vectors.

Set $q_0 = 0, \beta_0 = 1$, and $r_0 = b - Ax_0$ for some initial guess x_0; $j = 0$

while $\beta_j \neq 0$:

$q_{j+1} = r_j/\beta_j$

$\alpha_j = q_j^T A q_j$

$r_j = (A - \alpha_j I)q_j - \beta_{j-1} Pj - 1 q_{j-1}$

$\beta_j = \|r_j\|_2$

end

We apply this orthonormal sequence of Lanczos vectors $Q = [q_1, \ldots, q_n]$ to the

linear system as follows:

$$\begin{aligned} Ax = b \;\; &\Rightarrow \;\; Q^T Ax = Q^T \Rightarrow Q^T A(QQ^T)x = Q^T b \Rightarrow (Q^T AQ)Q^T x = Q^T b \\ &\Rightarrow \;\; Ty = f, \quad \text{for} \quad T = Q^T AQ \quad \text{and} \quad f = Q^T b. \end{aligned}$$

Additionally, setting $r_0 = b$ causes $f = [\|b\|, 0, \ldots, 0]^T$, allowing for an even easier solution.

So far we have assumed that T will be completely formed, requiring n iterations. But since T builds up with each iteration, we can recursively update y by solving $T_j y_i = f_j$. That is, y_j is updated from y_{j-1}. If we choose the "correct" inner product, we can force orthogonality upon this sequence of updates, resulting in finite termination. Selecting the T-inner product, we get

$$(y_{j+1} - y_j)^T T(y_{j+1} - y_j) = 0.$$

Using the relations $Q^T AQ = T$ and $x = Qy$, we see that this is equivalent to

$$(x_{j+1} - x_j)^T A(x_{j+1} - x_j) = 0.$$

This is to say that the direction from x_j to x_{j+1} is A-orthogonal to *all* previous directions, the improvement we wanted from steepest descent.

To obtain the solution x, we want

$$x = \alpha_1 p_1 + \ldots + \alpha_n p_n.$$

The A-orthogonality condition reduces this to

$$p_j^T A x = \alpha_j p_j^T A p_j,$$

giving the formula for computing α_j. The steepest descent algorithm would select r_j as the search direction p_j, but in order to enforce conjugacy upon the previous search directions, we add a multiple of the previous search direction to it, i.e. $p_j = r_j + \beta_j p_{j-1}$, where β_j is a scalar correction that forces the A-inner product to 0.

The algorithm is as follows:

Conjugate Gradient Algorithm

If $A \in \mathbf{R}^{n \times n}$ is symmetric positive definite and $b \in \mathbf{R}^n$, then the following algorithm, using symmetric positive definite preconditioner M, generates approximations x_i to $x \in \mathbf{R}^n$ of $Ax = b$.

Compute $r_0 = b - Ax_0$ for some initial guess x_0

for $i = 1, 2, \ldots$

solve $Mz_{i-1} = r_{i-1}$

$\rho_{i-1} = r_{i-1}^T z_{i-1}$

if $i = 1$

$p_1 = z_0$

else

$\beta_{i-1} = \rho_{i-1}/\rho_{i-2}$

$p_i = z_{i-1} + \beta_{i-1} p_{i-1}$

endif

$q_i = Ap_i$

$\alpha_i = \rho_{i-1}/p_i^T q_i$

$x_i = x_{i-1} + \alpha_i p_i$

$r_i = r_{i-1} - \alpha_i q_i$

check convergence; continue if necessary

end

2.2 Nonsymmetric Linear Systems

When A is not symmetric positive definite, the orthogonality of the residual vectors cannot be ensured, causing the algorithm to prematurely terminate. (The denomi-

nators of α or β may become zero.) Therefore, this problem must be overcome if we are to solve nonsymmetric systems.

The Biconjugate Gradient method [23], [16], rather than relying on a single sequence of residuals, 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

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

Biconjugate Gradient Algorithm

For $A \in \mathbf{R}^{n \times n}$ and $b \in \mathbf{R}^n$, the following algorithm, using preconditioner M, generates approximations x_i to $x \in \mathbf{R}^n$ of $Ax = b$.

Compute $r_0 = b - Ax_0$ for some initial guess x_0.

Choose $\tilde{r}_0$ (for example, $\tilde{r}_0 = r_0$).

for $i = 1, 2, \ldots$

solve $Mz_{i-1} = r_{i-1}$

solve $M^T \tilde{z}_{i-1} = \tilde{r}_{i-1}$

$\rho_{i-1} = z_{i-1}^T \tilde{r}_{i-1}$

if $\rho_{i-1} = 0$, **breakdown**

if $i = 1$

$p_i = z_{i-1}$

$\tilde{p}_i = \tilde{z}_{i-1}$

else

$\beta_{i-1} = \rho_{i-1}/\rho_{i-2}$

$p_i = z_{i-1} + \beta_{i-1} p_{i-1}$

$\tilde{p}_i = \tilde{z}_{i-1} + \beta_{i-1} \tilde{p}_{i-1}$

endif

$q_i = Ap_i$

$\tilde{q}_i = A^T \tilde{p}_i$

$\alpha_i = \rho_{i-1}/\tilde{p}_i^T q_i$

$x_i = x_{i-1} + \alpha_i p_i$

$r_i = r_{i-1} - \alpha_i q_i$

$\tilde{r}_i = \tilde{r}_{i-1} - \alpha_i \tilde{q}_i$

check convergence; continue if necessary

end

2.3 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 [17], then later adapted to nonsymmetric systems [18]. It adapts ideas used in another method, GMRES, so we first give a brief outline of this algorithm.

2.3.1 Preliminary: GMRES

The Generalized Minimal Residual method (GMRES) [26] concentrates its effort on minimizing the residual (equation 2). It is based on the Arnoldi process, which produces an upper Hessenberg matrix rather than the Lanczos generated tridiagonal matrix.

The Arnoldi process

Choose $v_1 \in \mathbf{R}^n$ such that $\|v_1\|_2 = 1$.

for $n = 1, 2, \ldots$

for $k = 1, 2, \ldots, n$

$h_{k,n} = v_k^H A v_n$

Set $\tilde{v}_{n+1} = Av_n - \sum_{k=1}^{n} h_{k,n} v_k$

Compute $h_{n+1,n} = \|\tilde{v}_{n+1}\|_2$

if $h_{n+1,n} = 0$, **stop**;

otherwise $v_{n+1} = \tilde{v}_{n+1}/h_{n+1,n}$.

end

Rewriting the inner loop, we get $AV_n = V_{n+1} H_n^{(e)}$, where the columns of V are orthogonal, and $H^{(n+1)\times n}$ is a full rank upper Hessenberg matrix. In order to use this to solve linear systems, set $v_1 = r_0/\|r_0\|_2$, resulting in Krylov iterates

$$x_n = x_0 + V_n z_n. \tag{6}$$

The minimization of the residual is accomplished via least squares:

$$\|d_n - H_n^{(e)} z_n\|_2 = \min_{z \in \Re^n} \|d_n - H_n^{(e)} z\|_2,$$

where $d_n = [\|r_0\|_2 \ 0 \cdots \ 0]^T \in \mathbf{R}^{n+1}$.

Because H is upper Hessenberg, the minimization is inexpensive since an updated QR factorization may be used.

2.3.2 A Quasi-Minimization Method

In theory, GMRES is guaranteed to converge to the solution in n steps. However, the length of the recurrences grows linearly with the number of iterations, often too great a memory requirement for solving large systems. While it is possible to restart or truncate the algorithm every m iterations (GMRES(m)), which limits space requirements, this loses the guarantee of convergence. Freund and Nachtigal decided to apply the ideas involved in GMRES to the Biconjugate Gradient method. The result is a method that sets up a BiCG iterate, then attempts to further reduce the residual using a quasi-minimization technique.

But where 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 [19]. 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) [3].

Quasi-Minimal Residual Algorithm

For $A \in \mathbf{R}^{n\times n}$ and $b \in \mathbf{R}^n$, the following algorithm, using preconditioner M, generates approximations x_i to $x \in \mathbf{R}^n$ of $Ax = b$.

Compute $r_0 = b - Ax_0$ for some initial guess x_0

$\tilde{v}_1 = r_0$; **solve** $M_1 y = \tilde{v}_1$; $\rho_1 = \|y\|_2$

Choose $\tilde{w}_1$, for example $\tilde{w}_1 = r_0$

solve $M_2^T z = \tilde{w}_1$; $\xi_1 = \|z\|_2$

$\gamma_0 = 1, \eta_0 = -1$

for $i = 1, 2, \ldots$

if $\rho_i = 0$ or $\xi_i = 0$ **method fails**

$v_i = \tilde{v}_i/\rho_i$; $y = y/\rho_i$

$w_i = \tilde{w}_i/\xi_i$; $z = z/\xi_i$

$\delta_i = z^T y$; **if** $\delta_i = 0$ **method fails**

solve $M_2 \tilde{y} = y$

solve $M_1^T \tilde{z} = z$

if $i = 1$

$p_1 = \tilde{y}$

$q_1 = \tilde{z}$

else

$p_i = \tilde{y} - (\xi_i \delta_i / \epsilon_{i-1}) p_{i-1}$

$q_i = \tilde{z} - (\rho_i \delta_i / \epsilon_{i-1}) q_{i-1}$

endif

$\tilde{p} = A p_i$

$\epsilon_i = q_i^T \tilde{p}$;**if** $\epsilon_i = 0$ **method fails**

$\beta_i = \epsilon_i / \delta_i$; **if** $\beta_i = 0$ **method fails**

$\tilde{v}_{i+1} = \tilde{p} - \beta_i v_i$

solve $M_1 y = \tilde{v}_{i+1}$; $\rho_{i+1} = \|y\|_2$

$\tilde{w}_{i+1} = A^T q_i - \beta_i w_i$

solve $M_2^T z = \tilde{w}_{i+1}$; $\xi_{i+1} = \|z\|_2$

$\theta_i = \rho_{i+1} / (\gamma_{i-1} |\beta_i|)$; $\gamma_i = 1/\sqrt{1 + \theta_i^2}$; **if** $\gamma_i = 0$ **method fails**

$\eta_i = -\eta_{i-1} \rho_i \gamma_i^2 / (\beta_i \gamma_{i-1}^2)$

if $i = 1$

$d_1 = \eta_1 p_1$; $s_1 = \eta_1 \tilde{p}$

else

$d_i = \eta_i p_i + (\theta_{i-1} \gamma_i)^2 d_{i-1}$

$s_i = \eta_i \tilde{p} + (\theta_{i-1} \gamma_i)^2 s_{i-1}$

endif

$x_i = x_{i-1} + d_i$

$r_i = r_{i-1} - s_i$

check convergence; continue if necessary

end

Note again that as we have implemented it, QMR may breakdown.[4]

2.4 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 [27] 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 $\tilde{r}_j = P^2(A)r_0$. This is the basis for the Conjugate Gradient Squared Method (CGS). Note that the savings is not only

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

that the $\tilde{r}$'s are not formed, but we also avoid needing A^T. The result is that the Krylov subspace is built up twice as fast as BiCG, theoretically doubling the speed of convergence.

Conjugate Gradient Squared Algorithm

For $A \in \mathbf{R}^{n \times n}$ and $b \in \mathbf{R}^n$, this algorithm, using preconditioner M, generates approximations x_i to $x \in \mathbf{R}^n$ of $Ax = b$.

Compute $r_0 = b - Ax_0$ for some initial guess x_0

Choose $\tilde{r}$ (for example, $\tilde{r} = r_0$)

for $i = 1, 2, \ldots$

$\rho_{i-1} = \tilde{r}^T r_{i-1}$

if $\rho_{i-1} = 0$ **method fails**

if $i = 1$

$u_1 = r_0$

$p_1 = u_1$

else

$\beta_{i-1} = \rho_{i-1} / \rho_{i-2}$

$u_i = r_{i-1} + \beta_{i-1} q_{i-1}$

$p_i = u_i + \beta_{i-1}(q_{i-1} + \beta_{i-1} p_{i-1})$

endif

solve $M\hat{p} = p_i$

$\hat{v} = A\hat{p}$

$\alpha_i = \rho_{i-1}/\tilde{r}^T\hat{v}$

$q_i = u_i - \alpha_i\hat{v}$

solve $M\hat{u} = u_i + q_i$

$x_i = x_{i-1} + \alpha_i\hat{u}$

$r_i = r_{i-1} - \alpha_i A\hat{u}$

check convergence; continue if necessary

end

Notice that this method can prematurely terminate, or "break down". This occurs when the residual vectors r_k and $\tilde{r}$ become orthogonal [5].

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.

[5] In fact in finite arithmetic we terminate the iteration if $r_k^T\tilde{r} \approx 0$ ($r_k^T\tilde{r} <$ `BREAK_TOL`).

2.5 Biconjugate Gradient Stabilized (BiCGSTAB)

Van der Vorst proposed that instead of building the basis vectors for $\mathcal{K}^i(\tilde{r}_0, A^T)$ 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 $r_n = Q_i(A)P_i(A)$ 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 $i < j$.

Biconjugate Gradient Algorithm

For $A \in \mathbf{R}^{n \times n}$ and $b \in \mathbf{R}^n$, the following algorithm, using preconditioner M, generates approximations x_i to $x \in \mathbf{R}^n$ of $Ax = b$.

Compute $r_0 = b - Ax_0$ for some initial guess x_0

Choose $\tilde{r}$ (for example, $\tilde{r} = r_0$)

for $i = 1, 2, \ldots$

$\rho_{i-1} = \tilde{r}^T r_{i-1}$

if $\rho_{i-1} = 0$ **method fails**

if $i = 1$

$p_i = r_{i-1}$

else

$\beta_{i-1} = (\rho_{i-1}/\rho_{i-2})(\alpha_{i-1}/\omega_{i-1})$

$p_i = r_{i-1} + \beta_{i-1}(p_{i-1} - \omega_{i-1}v_{i-1})$

endif

solve $M\hat{p} = p_i$

$v_i = A\hat{p}$

$\alpha_i = \rho_{i-1}/\tilde{r}^T v_i$

$s = r_{i-1} - \alpha_i v_i$

check norm of s; if small enough: set $x_i = x_{i-1} + \alpha_i \hat{p}$ and stop

solve $M\hat{s} = s$

$t = A\hat{s}$

$\omega_i = t^T s/t^T t$

$x_i = x_{i-1} + \alpha_i \hat{p}_i + \omega_i \hat{s}$

$r_i = s - \omega_i t$

check convergence; continue if necessary

for continuation it is necessary that $\omega_i \neq 0$

end

Note the two checks for convergence. As BiCGSTAB incorporates steepest descent into BiCG, we can view this algorithm as BiCG $\times$ GMRES(1), i.e. GMRES(m) with restart parameter $m = 1$, so after performing a BiCG iterate, we can check for "early" convergence. Also, as with CGS, BiCGSTAB may breakdown. But another possibility for failure is introduced: if vectors s and t become orthogonal, it is useless to continue.

2.6 Brief Summary

Figure 1 illustrates the common behavior of the algorithms when they performs well. QMR smooths out the erratic behavior of BiCG, CGS converges erratically yet faster, and BiCGSTAB smoothly converges. Note that this in no way suggests BiCGSTAB is the superior algorithm. Experiments in section 4 will show examples of each outperforming the others.

Below we summarize some of the properties of the three algorithms used in our algorithmic bombardment routine.

1. Quasi-Minimal Residual (QMR)

    - Applicable to nonsymmetric and indefinite matrices.
    - Designed to avoid the irregular convergence behavior of BiCG, it avoids one of the two breakdown situations of BiCG.

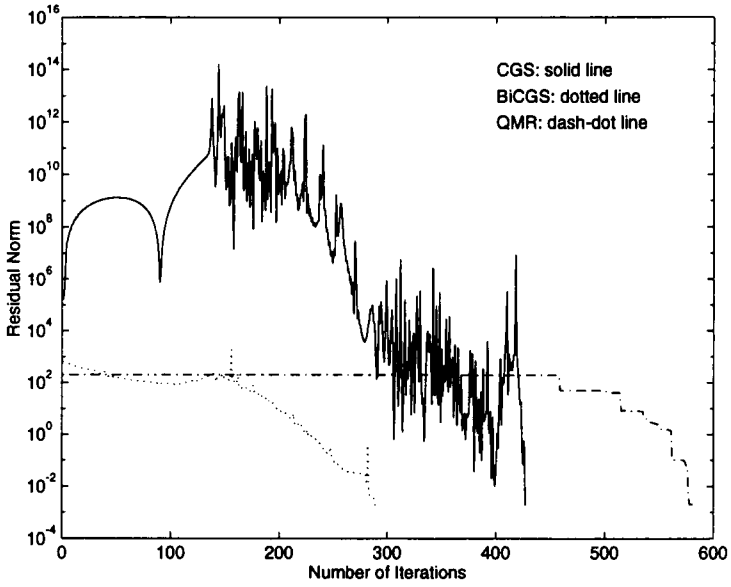


Figure 1: Common behavior of each algorithm
This graph shows the convergence behavior of CGS, BiCGSTAB, and QMR when applied to $-\frac{1}{10}\left(\frac{\partial^2 u}{\partial x^2}+\frac{\partial^2 u}{\partial y^2}\right)+cos\left(\frac{-\pi}{6}\right)\frac{\partial u}{\partial x}+sin\left(\frac{-\pi}{6}\right)\frac{\partial u}{\partial y}=0$, discretized on a 200×200 uniform grid (no preconditioning).

- If BiCG makes significant progress in one iteration step, then QMR delivers about the same result at the same step. But when BiCG temporarily stagnates or diverges, QMR may still further reduce the residual, albeit very slowly.
- Computation costs per iteration are similar to BiCG, but slightly higher. The method requires the transpose matrix-vector product.

2. Conjugate Gradient Squared (CGS)

   - Applicable to nonsymmetric and indefinite matrices, although often outperforms

     BiCGSTAB on nonsymmetric matrices.

   - Converges (diverges) typically about twice as fast as BiCG, although often quite irregularly, which may lead to a loss of accuracy in the updated residual. This is especially likely to occur when the initial guess is close to the true solution, a common occurrence in time dependent problems and some nonlinear solvers.

   - Computation costs per iteration are similar to BiCG, but the transpose matrix-vector product is not needed.

3. Biconjugate Gradient Stabilized (BiCGSTAB)

   - Applicable to nonsymmetric and indefinite matrices, although often outperforms CGS on indefinite matrices.

   - An alternative for CGS that avoids the irregular convergence patterns of CGS while maintaining about the same speed of convergence; as a result we often observe less loss of accuracy in the updated residual.

- Similar costs per iteration as BiCG and CGS, but again, the transpose matrix-vector product is not needed.

3 The Algorithmic Bombardment Algorithm

3.1 Basic idea

As we have indicated, 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.

3.2 Parallel implementation

The poly-iteration requires the sum of the floating point operations of the included algorithms, yet in a context of message-passing parallel computers, we can increase the efficiency of the algorithm.

The algorithms we consider including are all based on some form of the conjugate gradient method. Thus they have a very similar structure: they begin by computing an inner product, then some 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 'lining the methods up' at these operations and combining the communication stages. Table 1 lists the operations in the methods, "lined up" in order to take advantage of parallelism and combined message passing.

Figure 2 further illustrates the poly-iterative idea. The operations listed in the circles take advantage of combined communication. To the right of, and in between the combined operations is a listing of the other operations each algorithm performs in parallel (left to right: CGS, BiCGSTAB, QMR).

3.3 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[6], then send this buffer in total.

[6] In certain communication schemes such as PVM [20] this buffering is provided, in other schemes such as PICL [21] and the BLACS [8] it has to be implemented as part of the poly-iterative algorithm.

Table 1: Synchronized operations

The columns of this table show the mathematical operations that the individual algorithms perform. The left-most column indicates the synchronization points: "P" where the poly-iterative scheme combines message passing, and "B" and "Q" indicate synchronization points for BiCGSTAB and QMR, respectively. (Note that inner is an inner product, norm is a vector norm, axpy $= \alpha x + y$, solve is a preconditioner solve, and matvec' is a transpose-matrix times a vector.)

	CGS	BiCGSTAB	QMR
P	inner	inner	inner
	axpy	axpy	
	axpy	axpy	
	axpy		
P	solve	solve	solve
			axpy
			axpy
P	matvec	matvec	matvec
P	inner	inner	inner
	axpy	axpy	axpy
B	axpy	norm	
P	solve	solve	solve
	axpy		norm
P	matvec	matvec	matvec'
B	axpy	inner(2)	
		axpy	
		axpy	
		axpy	
P	norm	norm	norm
			axpy
			axpy
			axpy
			axpy
Q			norm

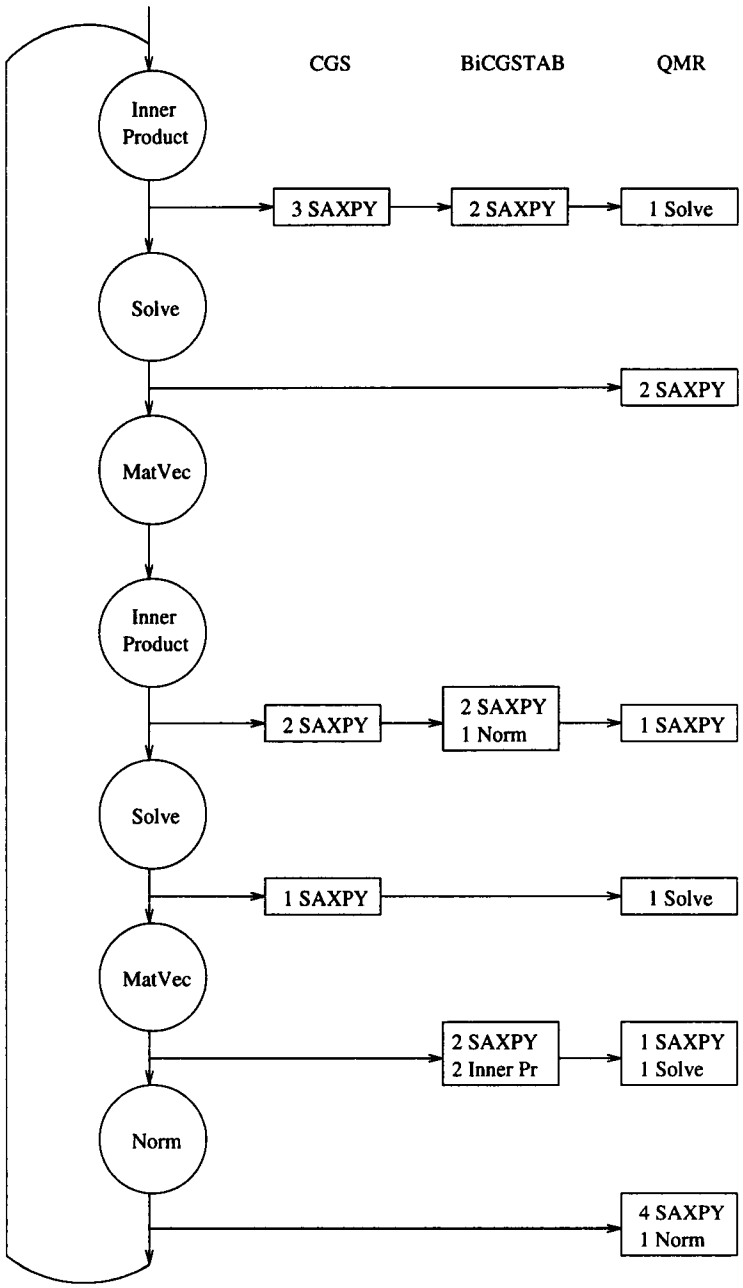


Figure 2: Sequence of Operations

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).

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.

3.4 Cost Model

Obviously this approach requires the combined floating point operations and workspace of each method [7]. This limits the size of the system that may be solved, although the actual impact is a function of the sparsity of the matrix. Table 2 lists the requirements for each method, as well as for bombardment (excluding preconditioning). Note that we ignore scalar storage.

Table 2: Summary of operations and workspace requirements
Summary of Operations for Iteration i. "1/1" means iteration requires both a matrix times vector and matrix transpose times vector operation. The operations counts for algorithmic bombardment reflect the combining of communication. For example, when we are able to combine the computation of inner products, we count this as one inner product.

Method	Amount of Work/Iteration				Storage
	$\alpha \leftarrow x^T y$	$y \leftarrow \alpha x + y$	$y \leftarrow Ax$	$x \leftarrow M^{-1}y$	Requirements
CGS	2	6	2	2	matrix $+ 9n$
Bi-CGSTAB	4	6	2	2	matrix $+ 9n$
QMR	2	5	1/1	1/1	$2 *$ matrix $+ 13n$
Bombardment	3	17	2	2	$2 *$ matrix $+ 29n$

The scalar cost of an iteration of the poly-iterative method equals the sum of the

[7] less $2n$ since the right hand side need only be stored once

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. Then the cost of the next algorithm used must be added to that. And so on.

In addition to the matrix, bombardment requires 29 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 24. 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 for GMRES grows linearly with the restart parameter, whereas bombardment requires eight. While it is true that it is possible to compute the GMRES inner products independently, this is known to cause a loss of stability.

3.4.1 Combined Operations

In this section we describe some of the details involved in performing the combined inner products and matrix-vector products.

Parallel architectures require global communication, and this remains the overriding 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). (FORTRAN-77 code fragments are shown in appendix C.1.)

To perform the matrix-vector product Ax, we first collect the global multiplier vector x on each processor [8], 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.

[8] 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.

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 eleven communications per iteration, compared with seven for CGS, nine for BiCGSTAB, and nine for QMR (ignoring preconditioning). The savings involved in the preconditioning step is a function of the structure of the preconditioner. For example, if we assume 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. First we offer a definition we will need for some comparisons.

- The algorithm that finds the solution in the least amount of time we call the "best" algorithm.

4.1 Implementation Details

All codes were written in ANSI standard Fortran 77.

Sequential version

- Uses the BLAS [6] for performing the vector operations.
- Executed on Sparc IPX workstations, with no optimization. This allows for IEEE arithmetic.
- Executed on IBM RS6000 350 (the BLAS are optimized). This machine does not use IEEE arithmetic, and experiments show results that differ, sometimes greatly, from the Sparc. Here the multiply and add steps are combined, and then the result is truncated. This results in greater numerical accuracy than the IEEE version, which truncates after the multiply, then again after the add.

Parallel versions

- Distributed Iterative Linear System Solvers [12].
- Also, we have adapted the PIM package [4]. In addition to writing the bombardment algorithm, the communication is accomplished using the BLACS [8]. This allows for 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 [7].
- Executed on an Intel iPSC860 Gamma at ORNL [11].
- Virtual parallel machines were formed using Sparc IPX using PVM [20] over ethernet.

For stopping criteria we use `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 [3]. For the right hand side we use $1 = [1, \ldots, 1]^T$.

4.2 Sequential Examples

These examples are designed to illustrate the unpredictability of the various methods. Not only is this a function of the linear system being solved, but also the arithmetic the particular machine uses. The examples come from various sources. Some are from the Harwell-Boeing collection of test matrices [10], while others have been used in previous papers on iterative methods ([29], [31]).

Example 1: CGS wins

The nonsymmetric linear system comes from the discretization, using a central difference scheme on a $12 \times 12 \times 12$ grid, of the three-dimensional elliptic partial differential equation

$$\Delta u + 100xu_x - yu_y + zu_z + 100\frac{x+y+z}{xyz}u = F$$

on the unit cube. (The exact solution is $u = e^{xyz}\sin(\pi x)\sin(\pi y)\sin(\pi z)$).

Using block ILU preconditioning, CGS quickly converges (58 iterations), while BiCGSTAB and QMR fail to converge (see figure 3).

For more examples, see table 4 in appendix B.

Example 2: BiCGSTAB wins

A nonsymmetric matrix from the Harwell-Boeing set provides an example of

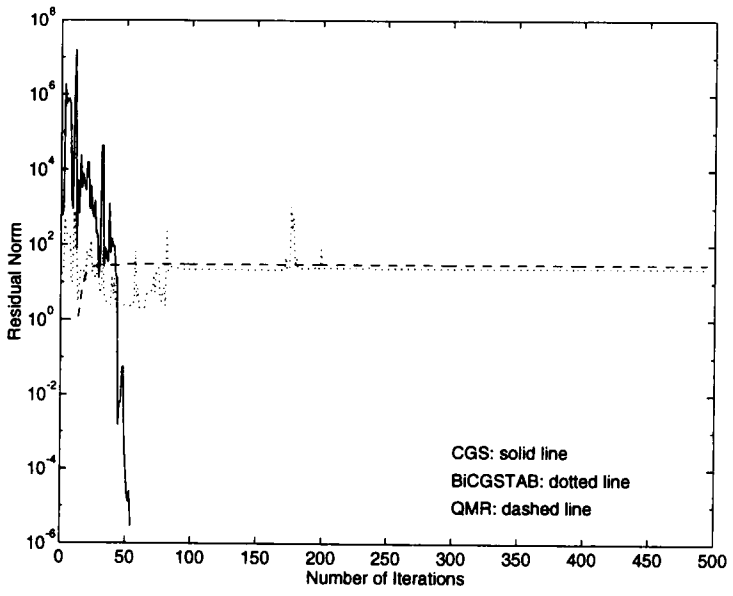


Figure 3: Sequential example: CGS wins
Example 1: The residual norm history of each algorithm applied to $\Delta u + 100xu_x - yu_y + zu_z + 100\frac{x+y+z}{xyz}u = F$ on the unit cube, discretized on a $12 \times 12 \times 12$ grid.

BiCGSTAB finding the solution, while CGS and QMR do not. The 3 dimensional "sherman1" example comes from a black oil simulator, using a $10 \times 10 \times 10$ grid, resulting in an order 1000 matrix with 3750 nonzeros.

Figure 4 shows the path of the residual norms of each method. Note that the results are different when executing on the IBM RS6000 (see table 8 in appendix B). The "more accurate" arithmetic causes BiCGSTAB to need more iterations to find the solution, but now QMR also converges (CGS still does not converge).

For more examples, see table 4 in appendix B.

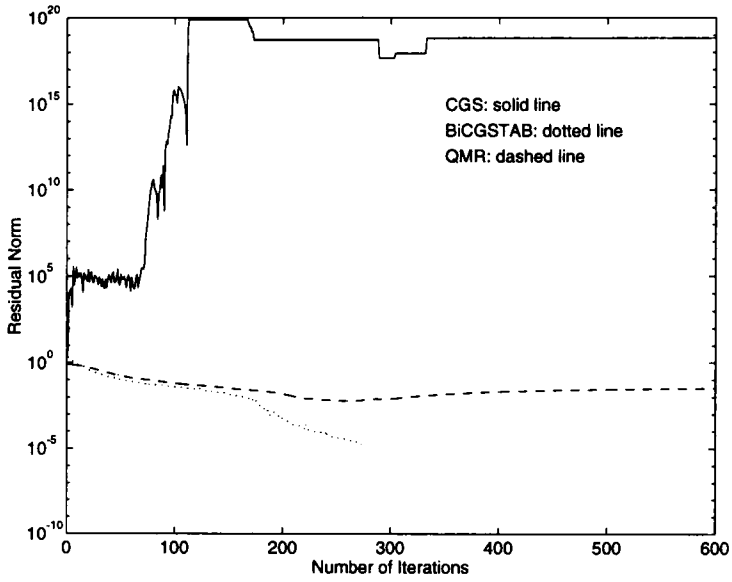


Figure 4: Sequential example: BiCGSTAB wins
Example 2: The residual norm history of each algorithm applied to the sherman1 matrix from the Harwell-Boeing test set (order 1000, no preconditioning).

Example 3: QMR wins

The nonsymmetric linear system comes from the discretization, using a $12 \times 12 \times 12$ grid, of the three-dimensional elliptic partial differential equation

$$(e^{xyz}u_x)_x + (e^{-xyz}u_y)_y + (e^{-xyz}u_z)_z - 250(x+y+z)u_x - 250[(x+y+z)u]_x - (1+x+y+z)^{-1}u = F$$

on the unit cube. (The exact solution is $u = e^{xyz}\sin(\pi x)\sin(\pi y)\sin(\pi z)$.) Note that this problem has positive definite symmetric part $(A + A^T)/2$ for any mesh size [14].

Using block ILU, none of individual algorithms can find the solution. Yet with no

preconditioning, QMR finds the solution in 382 iterations, while CGS and BiCGSTAB still do not converge (see figure 5).

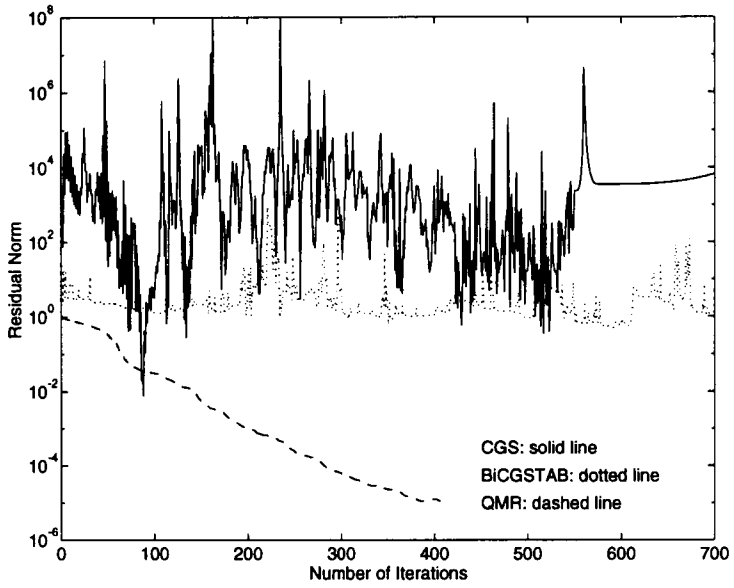


Figure 5: Sequential example: QMR wins
Example 3: the residual norm history of each algorithm applied to $(e^{xyz}u_x)_x + (e^{-xyz}u_y)_y + (e^{-xyz}u_z)_z - 250(x+y+z)u_x - 250((x+y+z)u)_x - (1+x+y+z)^{-1}u = F$. (No preconditioning.)

For more examples, see table 4 in appendix B.

4.3 Effects of Different Preconditioning

The above examples illustrate that the choice of preconditioner plays an important role in the performance of the algorithm. The following three examples further illustrate this.

Example 4:

In example 1, CGS converges while the others do not. But if we do not apply preconditioning, QMR becomes the fastest algorithm, with CGS a close second, while BiCGSTAB now breaks down (see figure 6).

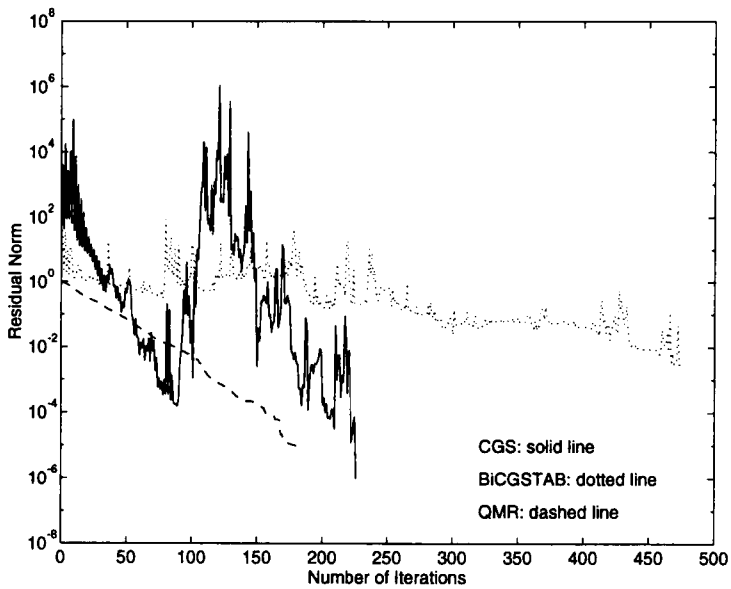


Figure 6: Preconditioning effects
Example 4: The residual norm history of each unpreconditioned algorithm applied to $\Delta u + 100xu_x - yu_y + zu_z + 100\frac{x+y+z}{xyz}u = F$. Compare this with the preconditioned experiment (example 1).

Example 5:

The nonsymmetric linear system comes from the discretization, using a $12 \times 12 \times 12$

grid, of the three-dimensional elliptic partial differential equation

$$\Delta u + 1000e^{xyz}(u_x + u_y + u_z) = F$$

on the unit cube. (The exact solution is $u = x + y + z$.)

- With no preconditioning, CGS and QMR converge, but BiCGSTAB does not (see figure 7).

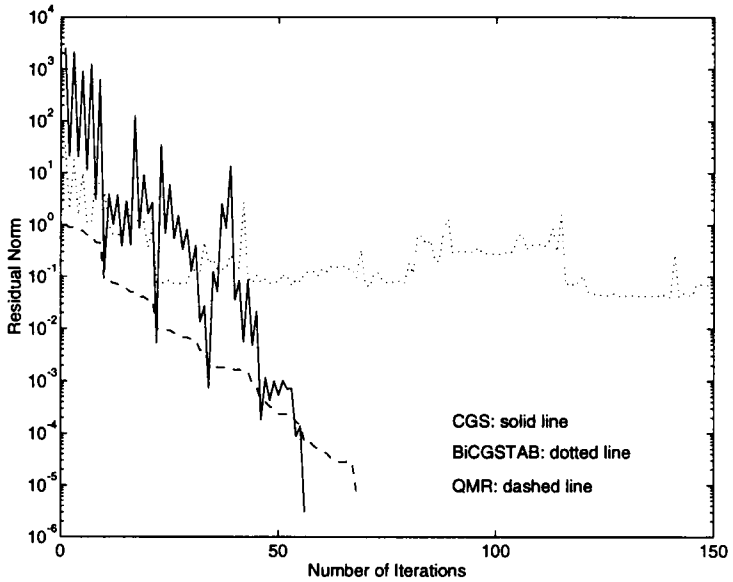


Figure 7: Preconditioning effects

Example 5: The residual norm history of each algorithm applied to $\Delta u + 1000e^{xyz}(u_x + u_y + u_z) = F$ on the unit cube. Here we apply no preconditioning. Compare this with the block ILU preconditioned experiment.

- Applying block ILU preconditioning, CGS and BiCGSTAB converge, but now

QMR does not.

Example 6:

Yet another experiment adds to the confusion of how to attack a problem. Consider the nonsymmetric linear system that comes from the discretization of the two-dimensional elliptic partial differential equation

$$-\Delta u + 10u_x + 10u_y = F$$

on the unit square. All three algorithms quickly find the solution if *no* preconditioning is used (see figure 8), yet by applying ILU preconditioning with diagonal scaling, CGS converges 500% faster than the unpreconditioned version, but BiCGSTAB and QMR fail to converge at all.

Other Results

It is not unexpected that other combinations of outcomes can occur. All may converge, although at different rates (appendix B, table 6), none may converge (appendix B, table 7), or two of the three (appendix B, table 5) may converge. Included in the tables are the outcomes of applying GMRES and GMRES(24) to the systems. It's not unexpected that GMRES almost always finds the solution (when workspace is available), but restarted GMRES is quite often outperformed by at least one of the algorithms in the bombardment scheme. And we found no examples where GMRES(24)

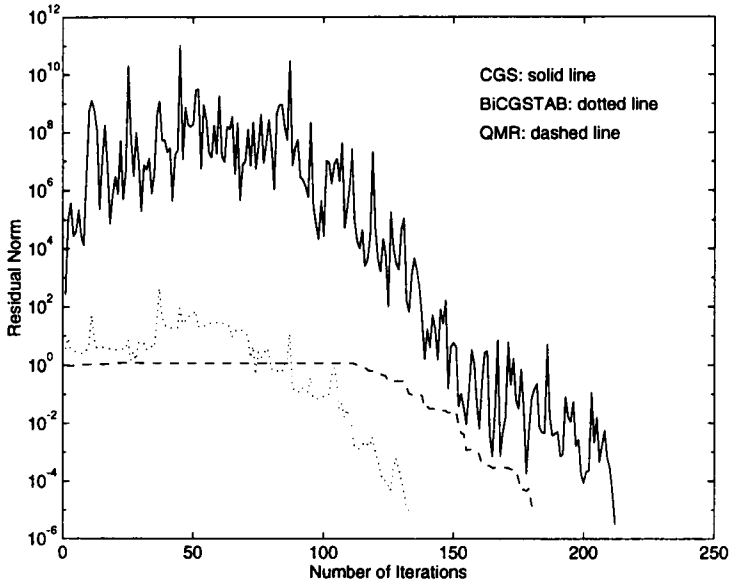


Figure 8: Preconditioning effects

Example 6: The residual norm history of each algorithm applied to $-\Delta u + 10u_x + 10u_y = F$ when preconditioning is not used. Compare this with the preconditioned experiment discussed above.

converged while the others did not, though examples may exist.

4.4 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 the message

passing. The experiments were run on the Intel iPSC/860 multiprocessor machine at Oak Ridge National Laboratory [11] and clusters of workstations which communicate over an ethernet using PVM. The overhead and latency of other machines, as well as floating point performance, will affect these results.

Example 7: Random sparse matrix

First we offer an example using 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 allows us to perform the algorithm for a fixed number of iterations, and compare the times for each method individually and the time for the poly-iterative method. 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.

Table 3 lists the time to perform 5000 iterations on 8 processors of the Intel iPSC/860. Individually the methods take about the same amount of time per iteration, yet when performing all three in parallel, the time increases only 5.6% over the fastest algorithm.

Table 3: Performance on Intel iPSC/860
Time (in seconds) to solution for solving perturbations of the Poisson equation. * means convergence not achieved given 5000 iterations.

Example	CGS	BiCGSTAB	QMR	Bombardment
7	1.37e2	1.38e2	1.41e2	1.49e2
8	4.01e2	5.02e2	*	8.96e2
9	*	4.10e1	*	8.77e1
10	*	*	7.98e0	1.32e1

The Poisson Problem

The problem with the sequential examples is the matrices are very different from one another. 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, for example, 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, yet we have found that it is prone to stall.

Yet mysteries still remain, and regardless, this requires careful analysis of the matrix that 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 can be justified. Below are some experiments.

We want to solve the following equation for $u(x, y)$ using various values for α and β:

$$\frac{\partial}{\partial x}\left(\frac{\partial u}{\partial x}\right) + \frac{\partial}{\partial y}\left(\frac{\partial u}{\partial y}\right) + \alpha\left(\frac{\partial u}{\partial x} + \frac{\partial u}{\partial y}\right) + \beta u = 1$$

on $(0,1) \times (0,1)$, with Dirichlet boundary conditions. The following examples are discretized using a uniform mesh. Times to solution are listed in table 3.

Example 8: QMR wins

On a 100×100 grid (10,000 variables) and using no preconditioning, we perturb β so that the matrix is indefinite. Here QMR converges, yet CGS and BiCGSTAB

fail to converge. (See table 3 for timings.)

Example 9: CGS wins

On a 400×400 grid (160,000 variables) and using block ILU preconditioning, we perturb β so that the matrix loses positive definiteness. BiCGSTAB converges, but CGS converges faster. But QMR no longer finds the solution (even after 5000 iterations). (See table 3 for timings.) Note that block ILU preconditioning requires no communication.

Example 10: BiCGSTAB wins

On a 400×400 grid (160,000 variables) and using block ILU preconditioning, we perturb α so that symmetry is lost. BiCGSTAB converges while neither CGS or QMR converge. (See table 3 for timings. Also, the residual norm histories are shown in figure 9.)

4.4.1 PVM experiments

Here we compare PVM timings on Sparc IPX workstations with the Intel iPSC860. As expected, the time difference between the best algorithm and the bombardment algorithm is greater on the Intel machines.

This set of examples solves the Poisson equation perturbed in another way:

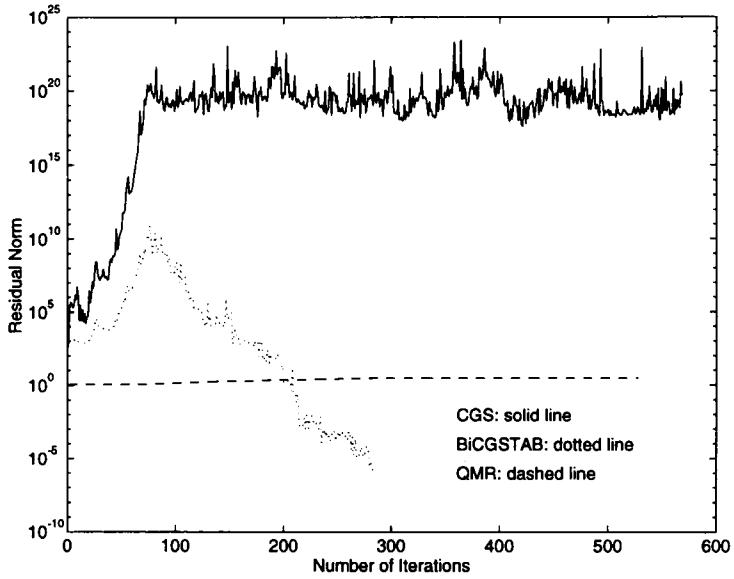


Figure 9: Parallel example: BiCGSTAB wins
Example 10: The residual norm history of each algorithm, using block ILU preconditioning, applied to the dimension 400 Laplacian skewed symmetric problem. (16 processors of Intel i860 gamma at ORNL.)

$$-\epsilon\left(\frac{\partial^2 u}{\partial x^2}+\frac{\partial^2 u}{\partial y^2}\right)+cos(\alpha)\frac{\partial u}{\partial x}+sin(\alpha)\frac{\partial u}{\partial y}=0. \tag{7}$$

Again we will solve this equation on the unit square, using a five point difference stencil, with various values of ϵ.

First we illustrate the time effects of the size of the problem. We'll solve[9]

[9]This problem was used in Sonneveld's paper presenting CGS [27]. A graph showing the path of the residual norms is shown in figure 1.

$$-1/10\left(\frac{\partial^2 u}{\partial x^2}+\frac{\partial^2 u}{\partial y^2}\right)+cos(-\pi/6)\frac{\partial u}{\partial x}+sin(-\pi/6)\frac{\partial u}{\partial y}=0$$

on square grids ranging from dimension 100 (order 10,000 matrix) to 200 (order 40,000 matrix) on eight processors of the parallel machines:

- The Intel iPSC860 (60 Mflops per node) and
- Sparc IPX workstations (located in a student computer lab at the University of Tennessee) using PVM over ethernet.

As expected, as the size of the problem increases (and thus the number of required floating point operations increases), the difference between executing the "best" algorithm (BiCGSTAB) and the bombardment algorithm increases (see figures 10 - 11).

But the significant outcome is that because for this example QMR takes many more iterations to converge than BiCGSTAB, the time to solution for QMR is greater than the time to find the solution using bombardment.

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

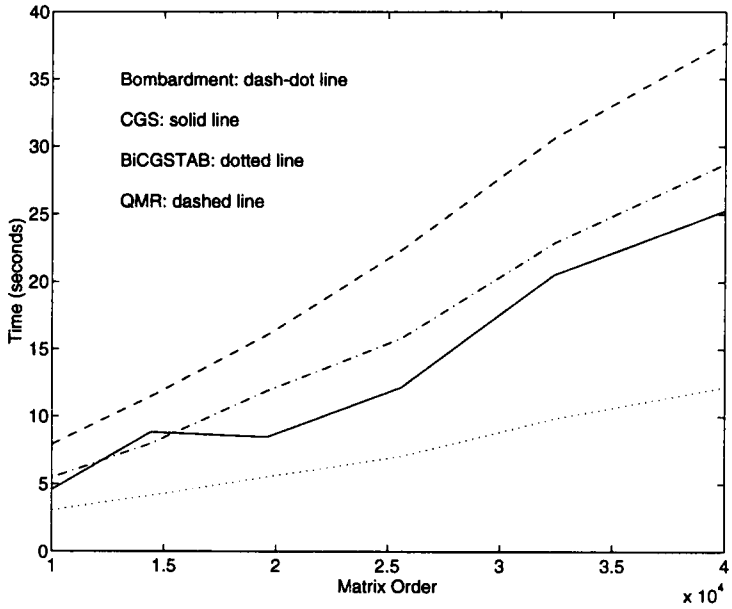


Figure 10: Times to solution on the Intel i860

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 grids. (8 processors)

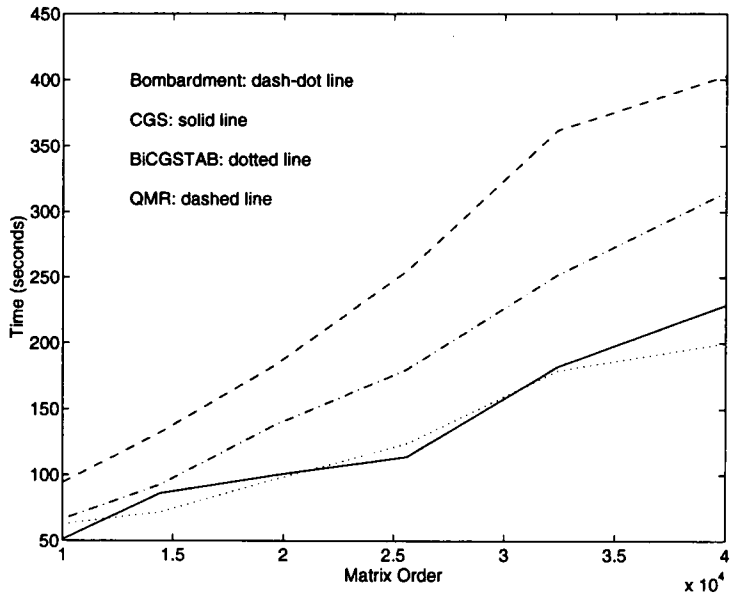


Figure 11: Times to solution on Sparc IPX workstations

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 grids. (8 processors)

example, CGS takes more iterations to find the solution for these examples than does BiCGSTAB (see figures 12 and 13), and at first glance it appears that these two algorithms require about the same amount of time to perform an iteration.

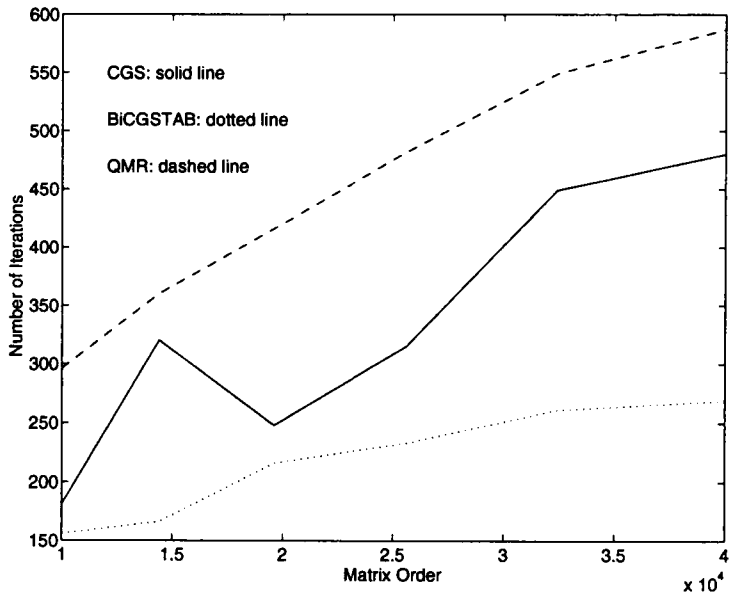


Figure 12: Number of iterations to solution on the Intel i860
Using 8 processors of the Intel iPSC860, with no preconditioning, we apply the individual algorithms and the bombardment algorithm to $-\frac{1}{10}\left(\frac{\partial^2 u}{\partial x^2}+\frac{\partial^2 u}{\partial y^2}\right)+cos\left(\frac{-\pi}{6}\right)\frac{\partial u}{\partial x}+sin\left(\frac{-\pi}{6}\right)\frac{\partial u}{\partial y}=0$, discretized on a square grids. (8 processors)

But BiCGSTAB requires an extra global communication step to accomplish the two extra inner products is must perform. In the Intel environment, where communication latencies are not high, this doesn't affect the outcome much in terms of time, and therefore BiCGSTAB is the fastest algorithm. But when the individual nodes are connected with ethernet, as is the case with these PVM experiments, the extra

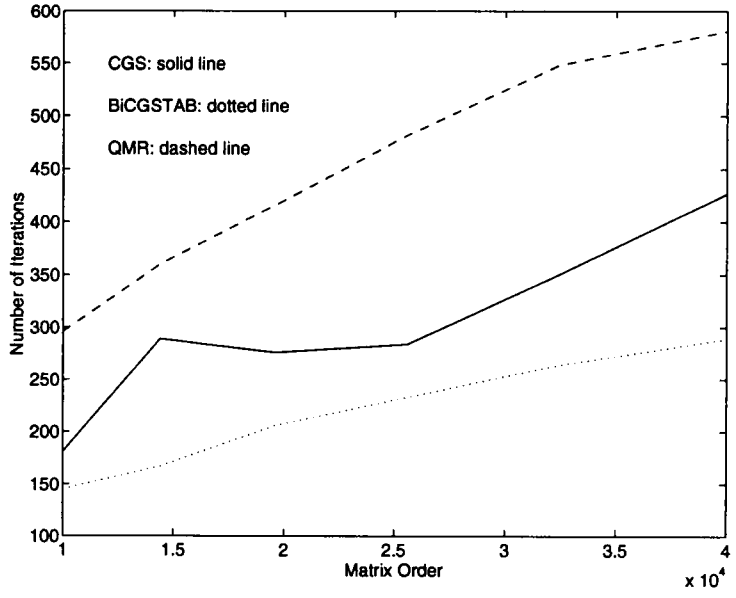


Figure 13: Number of iterations to solution on Sparc IPX workstations
Using a parallel machine consisting of 8 Sparc IPX workstations connected with ethernet using PVM, with no preconditioning, we apply the individual algorithms and the bombardment algorithm to $-\frac{1}{10}\left(\frac{\partial^2 u}{\partial x^2}+\frac{\partial^2 u}{\partial y^2}\right)+cos\left(\frac{-\pi}{6}\right)\frac{\partial u}{\partial x}+sin\left(\frac{-\pi}{6}\right)\frac{\partial u}{\partial y}=0$, discretized on a square grids. (8 processors)

communication becomes significant. The gap closes, and in fact CGS converges faster for some matrix sizes. It may be that this is attributable to other network traffic, but that is the nature of ethernet message passing.

The time spent in communication provides the insight into why this is happening. Figures 14 - 15 show the proportion of the time to solution spent in message passing as opposed to floating point operations.

As expected, the gap is a function of the interconnection network. And as the problem size increases, the floating point operation requirements increase, but the

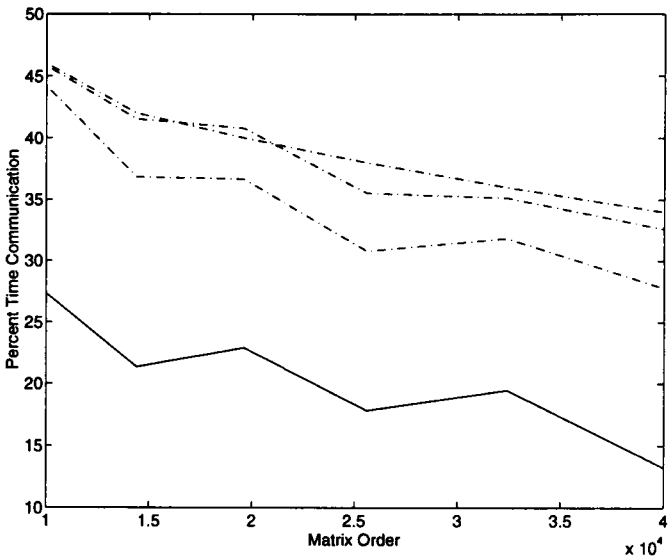


Figure 14: Percentage communication time on the Intel i860
Using 8 processors of the Intel iPSC860, with no preconditioning, we apply the individual algorithms and the bombardment algorithm to $-\frac{1}{10}\left(\frac{\partial^2 u}{\partial x^2}+\frac{\partial^2 u}{\partial y^2}\right)+cos\left(\frac{-\pi}{6}\right)\frac{\partial u}{\partial x}+sin\left(\frac{-\pi}{6}\right)\frac{\partial u}{\partial y}=0$, discretized on a square grids.

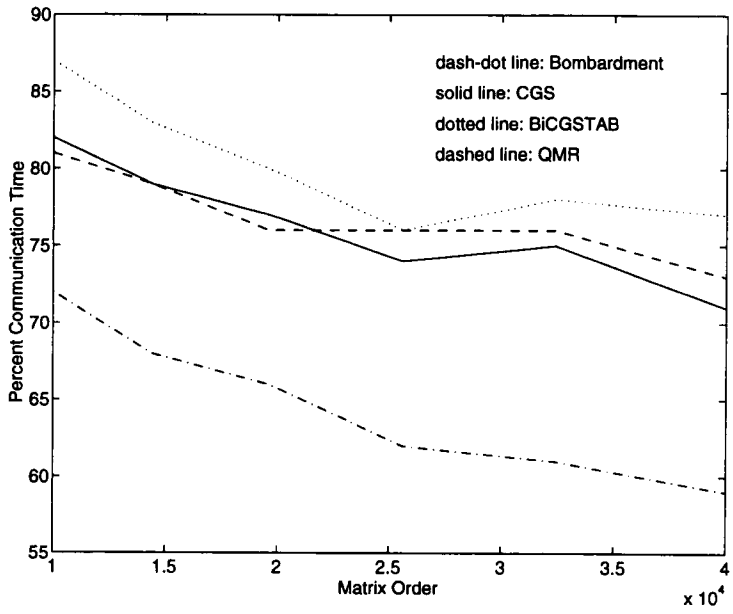


Figure 15: Percentage communication time on Sparc IPX workstations
Using a parallel machine consisting of 8 Sparc IPX workstations connected with ethernet using PVM, with no preconditioning we apply the individual algorithms and the bombardment algorithm to $-\frac{1}{10}\left(\frac{\partial^2 u}{\partial x^2}+\frac{\partial^2 u}{\partial y^2}\right)+cos\left(\frac{-\pi}{6}\right)\frac{\partial u}{\partial x}+sin\left(\frac{-\pi}{6}\right)\frac{\partial u}{\partial y}=0$, discretized on a square grids. (8 processors)

startup time to send a message remains constant.

Example 11: CGS fails

On a 200×200 grid (40,000 variables) and using no preconditioning, we set $\epsilon = 10^4$ and $\alpha = -\pi/6$. CGS fails to converge while both BiCGSTAB and QMR converge. Using D-ILU preconditioning, CGS will converge, but takes much longer than BiCGSTAB or QMR to do so. (See table 3 for timings. Also, the residual norm histories are shown in figure 16.)

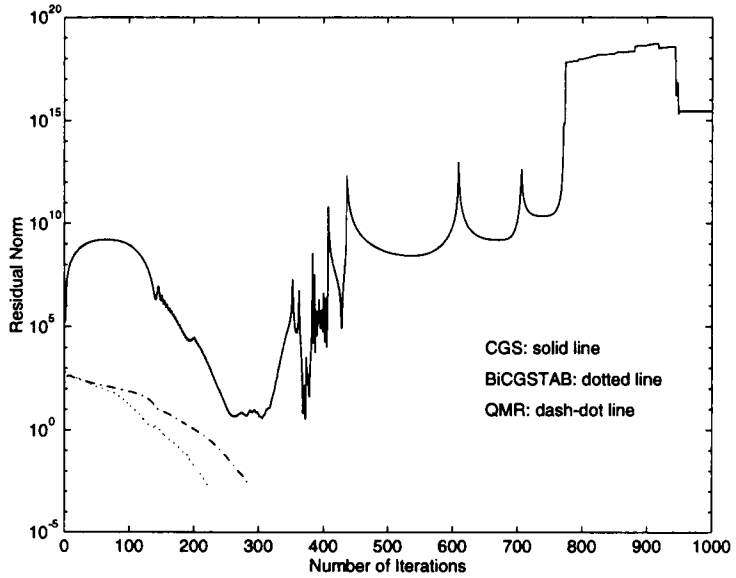


Figure 16: Parallel example: CGS fails

Example 11: The residual norm history of each algorithm, using no preconditioning, applied to $10^5 \left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}\right) + cos\left(\frac{-\pi}{6}\right) \frac{\partial u}{\partial x} + sin\left(\frac{-\pi}{6}\right) \frac{\partial u}{\partial y} = 0$, discretized on a 200 grid. (8 processors of Intel i860 gamma at ORNL.)

Example 12: BiCGSTAB wins

On a 200×200 grid (40,000 variables) and using D-ILU preconditioning, we set $\epsilon = 1/100$ and $\alpha = -\pi/6$. BiCGSTAB converges while neither CGS or QMR converge. (See table 3 for timings. Also, the residual norm histories are shown in figure 17.) The result is similar when no preconditioning is used.

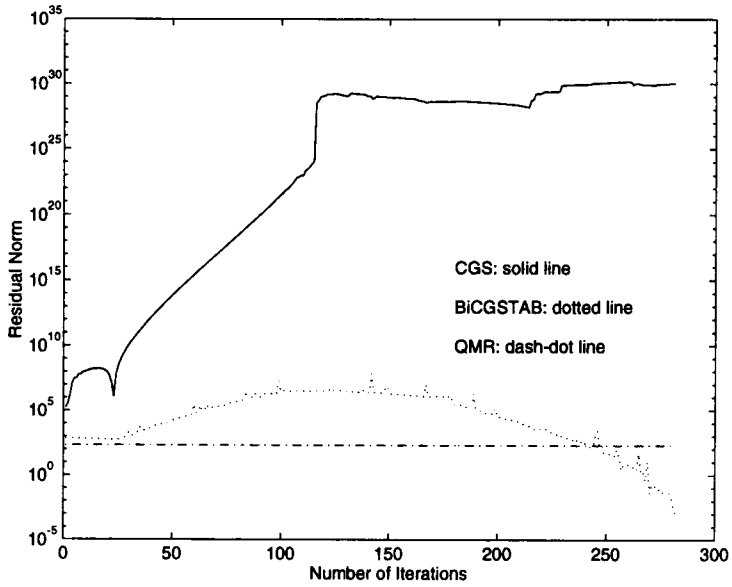


Figure 17: Parallel example: BiCGSTAB wins

Example 12: 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.)

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. That is, if for example CGS is closer to the solution that BiCGSTAB and QMR, why not "restart" them using the current iterate and residual from CGS? This would have the effect of projecting the iterate of one algorithm onto the Krylov subspace of another. Algorithmic Bombardment provides an easy way to do this, so we tried it.

Following are some experiments[10].

- A matrix provided by van der Vorst [30], which results from a 5-point discretization, is easily solved by each method, fastest by BiCGSTAB (figure 18). We try sharing information from the "winning" algorithm after 20 iterations, and every 20 thereafter. After 20 iterations, BiCGSTAB has the better solution approximation of the three. BiCGSTAB continues to have the best approximation, and therefore converges as before. However, CGS and QMR are irreparably damaged. In fact, immediately after receiving new information, CGS diverges quickly, then attempts to recover, but then is fed new information, interfering again. QMR merely stalls (figure 19).

  Since CGS attempts to converge after temporarily diverging, we try giving it information only once (after 20 iterations). We see that CGS and QMR are still irreparably damaged (figure 20). In fact, this shows that by continuing to give CGS "better" information, we hid the fact that it's convergence pattern was destroyed. So information from BiCGSTAB is of no value to CGS and QMR. Next we try sharing information from QMR.

[10]Note that for experimental purposes, we continue until either all algorithms converge or some maximum number of iterations are performed. Of course, when an algorithm finds the solution to tolerance, we drop it entirely from the scheme.

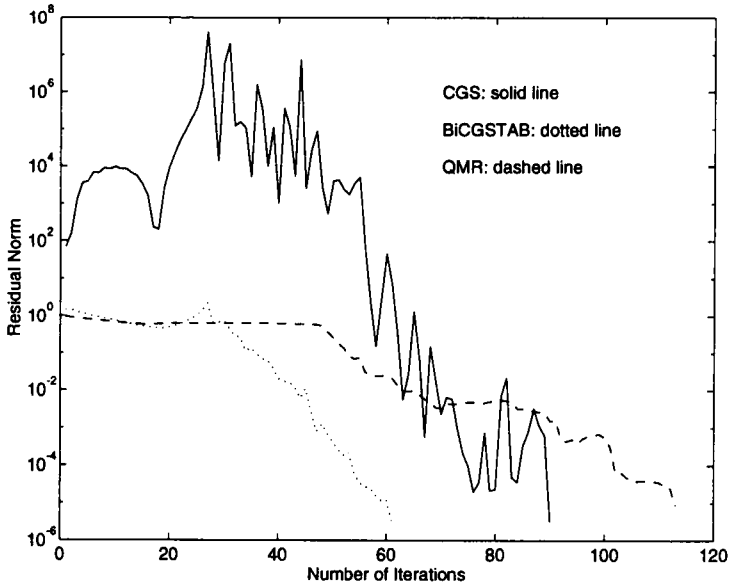


Figure 18: All methods converge.

All methods easily find the solution even without preconditioning. This example will be used to show the effects of sharing information from algorithm with the current closest approximation.

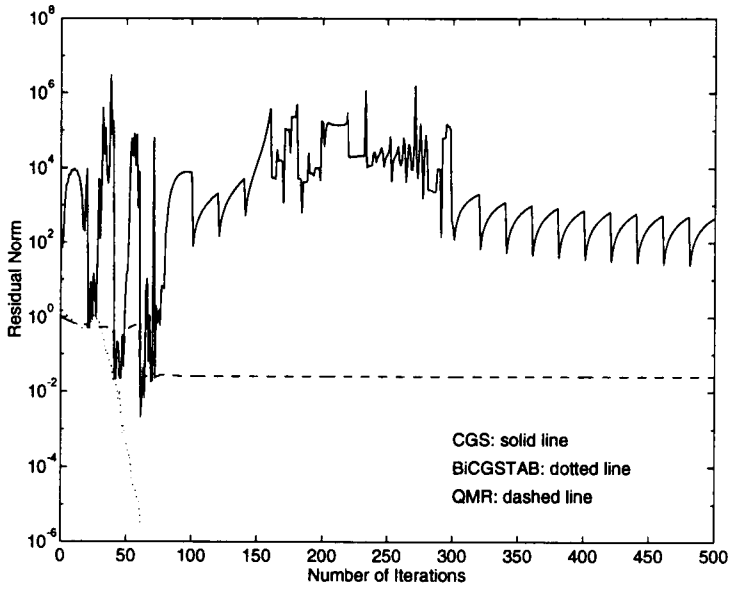


Figure 19: Sharing information from BiCGSTAB
Since BiCGSTAB is closest to the solution after 20 iterations, we give $x^{BiCGSTAB}$ and $r^{BiCGSTAB}$ to CGS and QMR. Then every 20 iterations we continuing sharing information.

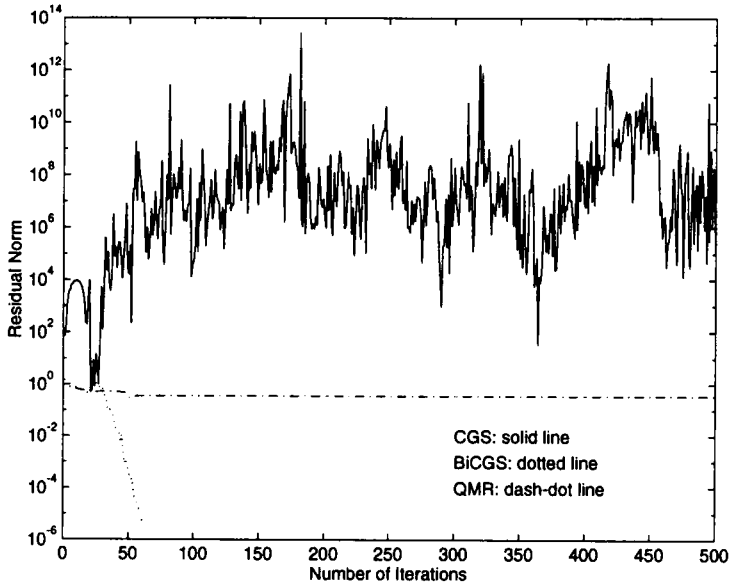


Figure 20: Sharing information from BiCGSTAB (once)
Since BiCGSTAB is closest to the solution after 20 iterations, we give $x^{BiCGSTAB}$ and $r^{BiCGSTAB}$ to CGS and QMR. But we share information only once.

- Returning to the graph of the residual histories for example 3, we see that QMR steadily converges, while CGS and BiCGSTAB wander about, making no significant progress towards the solution. So every 20 iterations we decide to share information (figure 21). Again, the immediate effect is for CGS and BiCGSTAB to diverge, then turn and try to converge. But since neither gets below QMR's residual, they are again restarted, and the pattern repeats. So we try restarting only once (figure 22). The effects are similar to the experiment where no information is exchanged, except that CGS never dips below QMR as

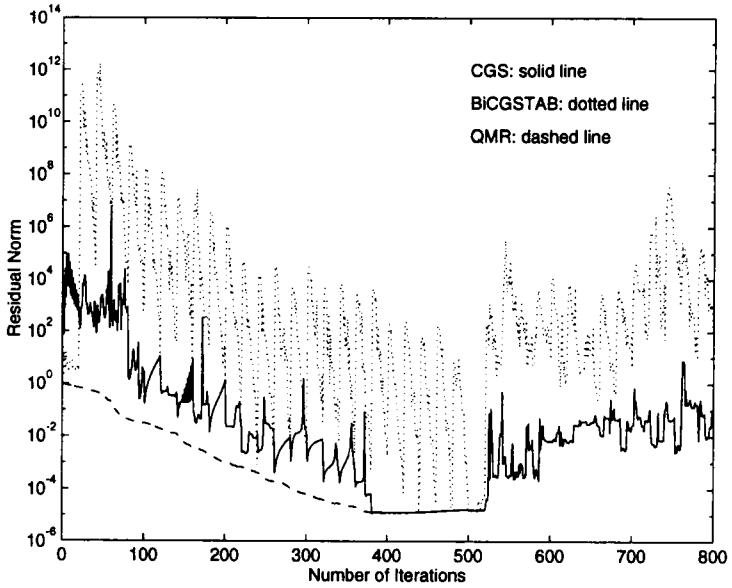


Figure 21: Sharing information from QMR

Since QMR is closest to the solution after 20 iterations, we give x^{QMR} and r^{QMR} to CGS and BiCGSTAB. (This is the same linear system used in example 1.)

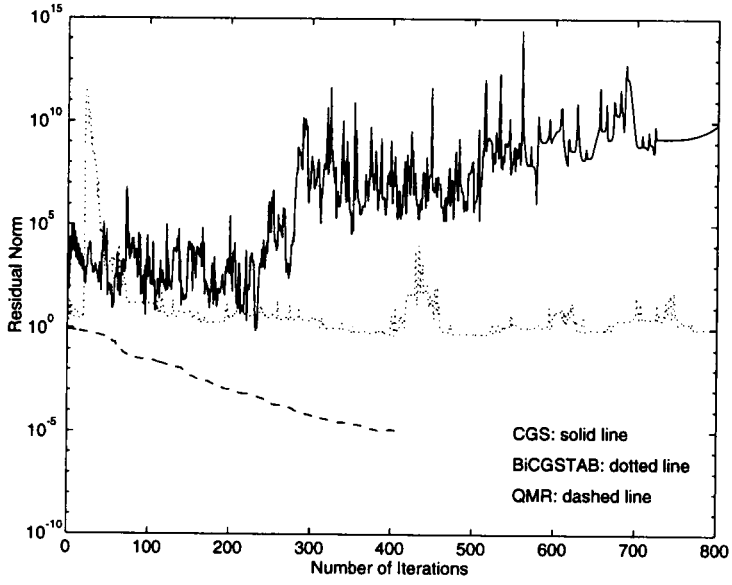


Figure 22: Sharing information from QMR (once)
Since QMR is closest to the solution after 20 iterations, we give x^{QMR} and r^{QMR} to CGS and BiCGSTAB. (This is the same linear system used in example 1.)

before. So information from QMR is of no value to CGS and BiCGSTAB. Next we try sharing information from CGS.

- The residual history for example 3 shows that after 88 iterations, CGS makes a sharp dip below QMR, temporarily having the best approximation. Here we give BiCGSTAB and QMR this information. BiCGSTAB gets more erratic, while QMR stalls. Then as before, CGS jumps above QMR, so QMR has the best approximation, but as we saw above, this doesn't help CGS or BiCGSTAB. And because QMR stalls, this situation doesn't change (see figure 23).

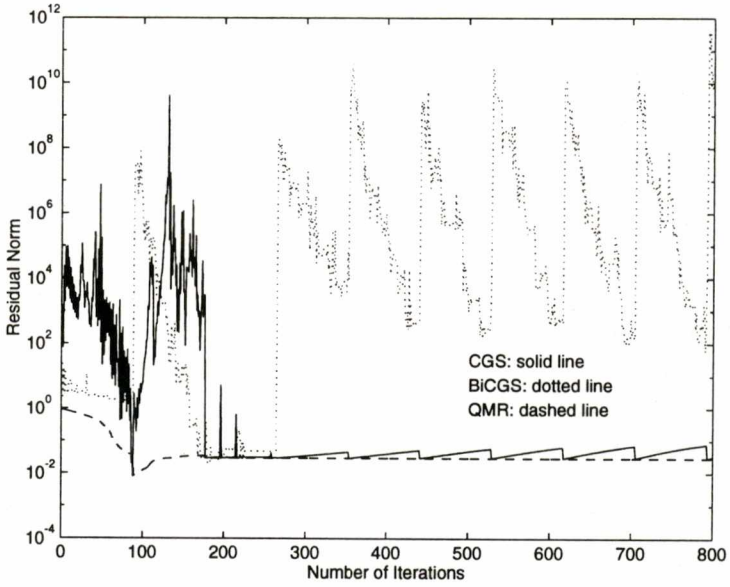


Figure 23: Sharing information from CGS
Since CGS is closest to the solution after 88 iterations, we give x^{CGS} and r^{CGS} to BiCGSTAB and QMR. (This is the same linear system used in example 3.)

There are two ways to look at this problem. First, we could examine the effects this sharing has on the various parameters of the algorithms. The problem is most easily seen in CGS and 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 that 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. Regardless, since the 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" 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.

Perhaps some sort of sharing among the algorithms could be done in some other way, which will be left for future research.

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 [20] exhibit high latency, and are dependent upon the traffic interconnection network, often the internet. Regardless, the increased probability of convergence may justify using poly-iteration.

6.1 Future Work

For future work we suggest

- More experiments. For example, some applications require solving many linear systems. It has been suggested that perhaps bombardment could be used during one such solve, then the "winning" algorithm would be used for the next few solves, then back to bombardment, and so on.

- incorporating more GMRES concepts into the poly-iteration. Since we originally ruled out using this valuable algorithm due to its linearly increasing workspace requirements, we must find a way to overcome this limitation while still gaining performance. BiCGSTAB is actually the combination of BiCG and GMRES(1). Recent work [30] shows that increasing the effects of GMRES can be worthwhile, such as combining GMRES(2) or GMRES(4) with BiCG.

- investigating other algorithms. New algorithms are being developed. Perhaps one or more will add something to the current set. Or perhaps an existing algorithm should be included, such as CG on the normal equations. This algorithm was kept out of the current implementation because of its slow convergence properties (the condition number is effectively squared), but perhaps it could replace QMR as the most robust algorithm.

- investigating preconditioning. Here we use the same preconditioner for each algorithm. Perhaps this is not optimal. For example, the performance of QMR was often degraded by ILU.

- investigating asynchronous communication. The current implementation forces the three individual algorithms into lock-step, which means for example that the CGS is perhaps unnecessarily slowed waiting for the other algorithms to cathch up. For example, QMR requires communication that cannot be combined with others. But while it performs this extra work, the other algorithms are blocked from continuing. Perhaps "pipelining" the communication phases can mask some communication costs.

- further investigating matrix-vector products. 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 $O(n^2)$ 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

References

[1] Steven F. Ashby, Thomas A. Manteuffel, and Paul E. Saylor. Adaptive polynomial preconditioning for hermitian indefinite linear systems. *BIT*, 29:583–609, 1989.

[2] Owe Axelsson and Gunhild Lindskog. On the eigenvalue distribution of a class of preconditioning matrices. *Numer. Math.*, 48:479–498, 1986.

[3] 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.

[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] J.J. Dongarra, J. DuCroz, I. Duff, and S. Hammerling. A set of level 3 basic linear algebra subprograms. *ACM Trans.on Math. Soft.*, 16:1–17, 1990.

[6] J.J. Dongarra, J. DuCroz, S. Hammarling, and R. Hanson. An extended set of fortran basic linear algebra subprograms. *ACM Trans.on Math. Soft.*, 14:1–32, 1988.

[7] J.J. Dongarra, Robert A. van de Geijn, and R. Clint Whaley. Basic linear algebra communication subprograms: Analysis and implementation across multiple parallel architectures. Technical report, Computer Science Department, University of Tennessee, 1993.

[8] J.J. Dongarra, Robert A. van de Geijn, and R. Clint Whaley. Lapack working note: A users' guide to the blacs. Technical report, Computer Science Department, University of Tennessee, 1993.

[9] D.F. Dubois, A. Greenbaum, and G.H. Rodrigue. Approximating the inverse of a matrix for use in iterative algorithms on vector processors. *Computing*, 22:257–268, 1979.

[10] I.S. Duff, R.G. Grimes, and J.G. Lewis. Sparse matrix test problems. *ACM Trans. Math. Software*, 15:1–14, 1989.

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

[12] Victor Eijkhout. Distributed iterative linear system solvers users' guide (in preparation). Technical report, Computer Science Department, University of Tennessee, 1994.

[13] Stanley C. Eisenstat. Efficient implementation of a class of preconditioned conjugate gradient methods. *SIAM J. Sci. Stat. Comput.*, 2:1–4, 1981.

[14] Howard C. Elman. Iterative methods for large, sparse nonsymmetric systems of linear equations. Technical report, Yale University, 1982.

[15] V. Faber and T. Manteuffel. Necessary and sufficient conditions for the existence of a conjugate gradient method. *SIAM J. Numer. Anal.*, 21:352–s62, 1984.

[16] 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.

[17] Roland W. Freund. Conjugate gradient type methods for linear systems with complex symmetric coefficient maitrices. Technical Report 89.54, RIACS, 1989.

[18] Roland W. Freund and Noël M. Nachtigal. QMR: a quasi-minimal residual method for non-Hermitian linear systems. to appear.

[19] Roland W. Freund and Noël M. Nachtigal. An implementation of the QMR method based on coupled two- term recurrences. Technical Report 92.15, RIACS, 1992.

[20] G. A. Geist, A. L. Beguelin, J. J. Dongarra, R. J. Manchek, and V. S. Sunderam. PVM 3.0 user's guide and reference manual. Technical Report ORNL/TM-12187, Oak Ridge National Laboratory, Oak Ridge, Tennessee, 1993.

[21] 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, Oak Ridge National Laboratory, Oak Ridge, Tennessee, 1990.

[22] Olin G. Johnson, Charles A. Micchelli, and George Paul. Polynomial preconditioning for conjugate gradient calculation. *SIAM J. Numer. Anal.*, 20:362–376, 1983.

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

[24] Claude Pommerell. *Solution of large unsymmetric systems of linear equations.* PhD thesis, Swiss Federal Institute of Technology, Zürich, 1992.

[25] Yousef Saad. Practical use of polynomial preconditionings for the conjugate gradient method. *SIAM J. Sci. Stat. Comput.*, 6:865–881, 1985.

[26] 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.

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

[28] G.W. Stewart. *Introduction to Matrix Computations.* Academic Press, 1973.

[29] 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.

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

[31] U. Meier Yang. Preconditioned conjugate gradient-like methods for nonsymmetric linear systems. Technical Report CSRD 1210, Ctr. for Supercomputing Res. and Develop., U. Ill., 1992.

Appendices

A Notation

In this section, we present some of the notation we use throughout this report. We have tried to use standard notation that would be found in any current publication on the subjects covered.

Throughout, we follow several conventions:

- Matrices are denoted by capital letters.
- Vectors are denoted by lowercase letters.
- Lowercase greek letters usually denote scalars.

We define matrix A of dimension $m \times n$ as follows:

$$A = \begin{bmatrix} a_{1,1} & \cdots & a_{1,n} \\ \vdots & & \vdots \\ a_{m,1} & \cdots & a_{m,n} \end{bmatrix} \qquad a_{i,j} \in \mathbf{R}.$$

We define the transpose matrix A^T of dimension $n \times m$ of matrix A of dimension $m \times n$ as follows:

$$A^T = \begin{bmatrix} a_{1,1} & \cdots & a_{1,m} \\ \vdots & & \vdots \\ a_{n,1} & \cdots & a_{n,m} \end{bmatrix} \qquad a_{i,j} \in \mathbf{R}.$$

We define vector x of dimension n as follows:

$$x = \begin{bmatrix} x_1 \\ \vdots \\ x_n \end{bmatrix} \qquad x_i \in \mathbf{R}.$$

- Matrix A is *symmetric* if $a_{i,j} = a_{j,i}$ for all $i \neq j$; that is, $A = A^T$.

- Matrix A is *positive definite* if, for every $x \neq 0, x^T Ax > 0$

B More Experiments

In this appendix we show more results of experiments using the individual methods. Unless otherwise noted, we use IEEE arithmetic. We also include GMRES and GMRES(24) (this restart parameter was selected since the workspace required to perform this algorithm is equal to the requirements of our poly-iterative scheme.)

Symbols:

- $*$ No convergence
- † not enough memory available
- ‡ from the Harwell – Boeing test set
- § executed on an IBM RS6000 350 (no IEEE arithmetic)

Table 4: Experiments: One converges, two do not
We include GMRES(24) and GMRES for comparison. GMRES(24) fails to converge for any of these problems, while GMRES, if workspace is available, finds the solution fastest.

Linear System	Preconditioner	QMR	CGS	BiCGSTAB	GMRES(24)	GMRES
Henk11	Jacobi	3319	*	*	*	†
nos1‡	Jacobi	464	*	*	*	183
nos6‡	None	903	*	*	*	149
nos7‡	None	2811	*	*	4724	127
Henk5	Block ILU	*	58	*		
Henk14	Block ILU	*	54	*		
bcsstm27‡	None	*	*	474	*	†

Table 5: Experiments: Two converge, one does not
We include GMRES(24) and GMRES for comparison. In one case, GMRES(24) would be the best algorithm, yet in the others, it fails to converge at all. If workspace is available, GMRES finds the solution, although for bcsstk09, BiCGSTAB beats it.

Linear System	Preconditioner	QMR	CGS	BiCGSTAB	GMRES(24)	GMRES
fs_541_3‡	None	2067	1559	*	*	261
meier2	None	80	68	*	76	68
bcsstk09‡	None	177	*	154	*	175
bcsstk09‡	Jacobi	172	157	149	*	175
meier2	Jacobi	42	*	66	80	70
nos1‡§	Jacobi	450	*	996	-	-
nos6‡	Jacobi	93	*	65	16	-
nos7‡	Jacobi	79	*	51	14	-
qcgstab2	None	216	*	373	*	†
qcgstab2	Jacobi	215	*	369	*	†
1138_bus‡	Jacobi	965	*	1387	*	545
1138_bus‡§	None	971	*	2234	-	-
meier2	Block ILU	*	19	20	-	-

Table 6: Experiments: All methods converge

Linear System	Preconditioner	QMR	CGS	BiCGSTAB	GMRES(24)	GMRES
bcsstk28‡	None	518	302	284	*	†
bcsstk28‡	Jacobi	25	15	14	20	-
jpwh_991‡	Jacobi	36	22	19	29	29
nos6‡,§	Jacobi	93	382	65	7	-
qcgstab7	Jacobi	1357	1501	1723	*	†
qcgstab7	None	1341	1663	1874	*	†
sherman4‡	Jacobi	79	67	50	140	67
steam1‡	Jacobi	19	7	42	5	-
steam3‡	Jacobi	23	14	12	10	-

Table 7: Experiments: No algorithm converges

Linear System	Preconditioner	QMR	CGS	BiCGSTAB	GMRES(24)	GMRES
meier5	None	*	*	*	*	866
meier5	Jacobi	*	*	*	*	385
plat1919‡	Jacobi	*	*	*	*	†
steam1‡	None	*	*	*	*	240
steam3‡	None	*	*	*	*	80
westXXX‡	None	*	*	*	*	*N*
westXXX‡	Jacobi	*	*	*	*	*N*
zenios‡	None	*	*	*	*	†
zenios‡	Jacobi	*	*	*	*	†

Table 8: Experiments: IEEE versus IBM RS6000 arithmetic
The first entry for an experiment are results from execution on a Sparc ipx worksations which use IEEE arithmetic. The second results are from an IBM RS6000 350 workstation, which doesn't use IEEE. In most experiments, the differences between the two methods of arithmetic were small. Here are some examples where the differences ranged from 14a difference between not finding the solution and finding it.

Linear System	Preconditioner	QMR	CGS	BiCGSTAB	GMRES(24)	GMRES
nos1‡	Jacobi	464	*	*	*	183
		450	*	996	-	-
nos6‡	Jacobi	93	*	65	16	-
		93	382	65	7	-
qcgstab7	None	1341	1663	1874	*	†
		1354	1457	1691	*	†

Table 9: Experiments: All results

Linear System	Preconditioner	QMR	CGS	BiCGSTAB	GMRES(24)	GMRES
bcsstk09‡	None	177	*	154	*	175
bcsstk09‡	Jacobi	172	157	149	*	175
bcsstk28‡	None	518	302	284	*	†
bcsstk28‡	Jacobi	25	15	14	20	-
bcsstm27‡	None	*	*	474	*	†
fs_541_3‡	None	2067	1559	*	*	261
Henk5	Block ILU	*	58	*		
Henk11	Jacobi	3319	*	*	*	†
Henk14	Block ILU	*	54	*		
jpwh_991‡	Jacobi	36	22	19	29	29
meier2	Block ILU	*	19	20	-	-
meier2	Jacobi	42	1111	66	80	70
meier2	None	80	68	*	76	68
meier5	None	*	*	*	*	866
meier5	Jacobi	478	*	*	*	385
nos1‡	Jacobi	464	*	*	*	183
nos1‡§	Jacobi	450	*	996	-	-
nos6‡	None	903	*	*	*	149
nos6‡	Jacobi	93	*	65	16	-
nos6‡,§	Jacobi	93	382	65	7	-
nos7‡	None	2811	*	*	4724	127
nos7‡	Jacobi	79	*	51	14	14
plat1919‡	Jacobi	*	*	*	*	†
qcgstab2	Jacobi	129	92	66	174	†
qcgstab7	Jacobi	295	347	472	*	†
sherman4‡	Jacobi	79	67	50	140	67
steam1‡	None	*	*	*	*	240
steam1‡	Jacobi	19	7	42	5	-
steam2‡	None	93	67	228	*	118
steam2‡	Jacobi	8	3	6	2	-
steam3‡	None	*	*	*	*	80
steam3‡	Jacobi	23	14	12	10	-
westXXX‡	None	*	*	*	*	N
westXXX‡	Jacobi	*	*	*	*	N
zenios‡	None	*	*	*	*	†
zenios‡	Jacobi	*	*	*	*	†
1138_bus‡	Jacobi	965	*	1387	*	545
1138_bus‡§	None	971	*	2234	*	678

C Example FORTRAN 77 code

C.1 Combined Inner Product

Here is an example of how the inner products are accomplished so that the required communication of the three algorithms is equal to one. Here we use a master-slave model, but more efficient communication patterns should be used for most implementations.

```
      SUBROUTINE ALLDDOT( N, ACTIVE, X1, Y1, DOT1, X2, Y2, DOT2,
     $                    X3, Y3, DOT3 )
*
*     .. Executable Statements ..
*
      IF ( ACTIVE(CGS) )   DOTS(CGS)   = DDOT( N, X1, 1, Y1, 1 )
      IF ( ACTIVE(BICGS) ) DOTS(BICGS) = DDOT( N, X2, 1, Y2, 1 )
      IF ( ACTIVE(QMR) )   DOTS(QMR)   = DDOT( N, X3, 1, Y3, 1 )
*
*     Global update using master-slave model.
*
      CALL SEND( 3, DOTS, 0 )
*
      CALL RECV( 3, DOTS, 0 )
*
      DOT1 = DOTS(CGS)
      DOT2 = DOTS(BICGS)
      DOT3 = DOTS(QMR)
```

C.2 The Algorithm

Here is a fragment of an implementation of the poly-iterative idea. First, all methods perform an inner product, then each method performs some vector updates (in parallel), then a combined preconditioner solve is performed.

```
*
*        ---------------- Combined Inner Product  ------------------
*
         CALL ALLDDOT( N, ACTIVE, CGSWS(1,RTLD), CGSWS(1,R), RHOCGS,
     $                 BICGSWS(1,RTLD), BICGSWS(1,R), RHOBICGS,
     $                 QMRWS(1,Z), QMRWS(1,Y), DELTAQMR )
*        ------------------------------------------------------------
*
*        CGS work in parallel.
*
         IF ( ACTIVE(CGS) ) THEN
            IF ( ABS( RHOCGS ).LT.RHOTOLCGS ) THEN
               ACTIVE(CGS) = .FALSE.
               WRITE(*,*) 'BREAKDOWN: RHO, CGS; ITERATION =', ITER
               INFO(CGS) = -1
               ALGITER(CGS) = ITER
               ACTIVEMETHODS = ACTIVEMETHODS - 1
               IF ( ACTIVEMETHODS.EQ.0 ) GO TO 100
            ENDIF
            IF ( ITER.GT.1 ) THEN
*
*              Compute U.
*
               BETACGS = RHOCGS / RHO1CGS
               CALL DCOPY( N, CGSWS(1,R), 1, CGSWS(1,U), 1 )
               CALL DAXPY( N, BETACGS, CGSWS(1,Q), 1, CGSWS(1,U),1)
*
*              Compute P.
*
               CALL DSCAL( N, BETACGS**2, CGSWS(1,P), 1 )
               CALL DAXPY( N, BETACGS, CGSWS(1,Q), 1, CGSWS(1,P), 1 )
               CALL DAXPY( N, ONE, CGSWS(1,U), 1, CGSWS(1,P), 1 )
            ELSE
               CALL DCOPY( N, CGSWS(1,R), 1, CGSWS(1,U), 1 )
```

```
               CALL DCOPY( N, CGSWS(1,R), 1, CGSWS(1,P), 1 )
            ENDIF
         ENDIF
*
*        BiCGSTAB work in parallel.
*
         IF ( ACTIVE(BICGS) ) THEN
            IF ( ABS( RHOBICGS ) .LT. RHOTOLBICGS ) THEN
               ACTIVE(BICGS) = .FALSE.
               WRITE(*,*) 'BREAKDOWN: RHO, BICGS; ITERATION =', ITER
               INFO(BICGS) = -1
               ALGITER(BICGS) = ITER
               ACTIVEMETHODS = ACTIVEMETHODS - 1
               IF ( ACTIVEMETHODS.EQ.0 ) GO TO 100
            ENDIF
*
*           Compute vector P.
*
            IF ( ITER.GT.1 ) THEN
*
               BETABICGS = ( RHOBICGS / RHO1BICGS ) *
     $                     ( ALPHABICGS / OMEGABICGS )
               CALL DAXPY( N, -OMEGABICGS, BICGSWS(1,V), 1,
     $              BICGSWS(1,P), 1 )
               CALL DSCAL( N, BETABICGS, BICGSWS(1,P), 1 )
               CALL DAXPY( N, ONE, BICGSWS(1,R), 1, BICGSWS(1,P), 1 )
            ELSE
               CALL DCOPY( N, BICGSWS(1,R), 1, BICGSWS(1,P), 1 )
            ENDIF
         ENDIF
*
*        QMR work in parallel.
*
         IF ( ACTIVE(QMR) ) THEN
            IF ( ABS( DELTAQMR ) .LT. DELTATOLQMR ) THEN
               ACTIVE(QMR) = .FALSE.
               WRITE(*,*) 'BREAKDOWN: DELTA, QMR; ITERATION =', ITER
               INFO(QMR) = -3
```

```
               ALGITER(QMR) = ITER
               ACTIVEMETHODS = ACTIVEMETHODS - 1
               IF ( ACTIVEMETHODS.EQ.0 ) GO TO 100
            ENDIF
            CALL PSOLVE( N, QMRWS(1,YTLD), QMRWS(1,Y) )
         ENDIF
*
*        ------------  Combined Preconditioner Solve ------------
*
         CALL ALLPSOLVE( N, ACTIVE,
     $                   'NOTRANS', CGSWS(1,PHAT), CGSWS(1,P),
     $                   'NOTRANS', BICGSWS(1,PHAT), BICGSWS(1,P),
     $                   'TRANS', QMRWS(1,ZTLD), QMRWS(1,Z) )
*        ------------------------------------------------------------
*
```

VITA

Richard Frederick Barrett was born June 20, 1959 in Morgantown, West Virginia. He graduated from Huntington High School, Huntington, West Virginia in 1977 and received a Bachelor of Science degree in Business Administration from Indiana University in Bloomington, Indiana in 1985. He received the Master of Science degree in Mathematics from the University of Tennessee in 1990. He is married to Susan Deel Barrett. In January 1991 he entered the Computer Science program at the University of Tennessee and was awarded the Master of Science degree in Computer Science in May 1994.