P_SPARSLIB Working Note No. 2
The Iterative Solvers Module *

Yousef Saad and Kesheng Wu
University of Minnesota
Department of Computer Science

November 17, 1993

Abstract

P_SPARSLIB is a library of portable FORTRAN routines for sparse matrix computations. The current thrust of the library is in iterative solution techniques. In this note we present the 'accelerators' part of the library, which consists of the best known of Krylov subspace techniques. This iterative solution module is implemented in reverse communication mode so as to allow any preconditioned to be combined with the package. In addition, this mechanism allows us to ensure portability, since the communication calls required in the iterative solution process are hidden in the dot product and the matrix-vector product and preconditioning operations.

1 Introduction

Solving linear systems

$$Ax = b$$

*Work supported in part by ARPA under grant number NIST 60NANB2D1272, in part by NSF under grant number NSF/CCR-9214116, and in part by AIPCRC (University of Minnesota) under Army Research Office grant number DAAL03-89-C-0038
is one of most frequently performed tasks in many scientific and engineering applications. It is not uncommon large application generate matrices of dimensions on the order of millions or even billions. With large matrices like these, it is a common practice to take advantage of the sparsity of the systems. Since most of the iterative linear system solvers can take advantage of the sparsity of the matrices, iterative solvers have become important building blocks of new scientific computation software packages. The iterative solvers discussed in this report are part of parallel sparse library (P-SPARSLIB).

In the last three decades, a large number of iterative solvers have been developed. In implementing this module of P-SPARSLIB we have a rich selection of algorithms to choose from. However, the typical performance of these algorithms can be rather well represented by a smaller set consisting a few representatives from each of the large classes of methods. We have selected the following subset for this module:

- The Conjugate Gradient (CG) method,
- Conjugate Gradient on Normal Residual equation (CGNR),
- Bi-Conjugate Gradient (BCG),
- Direct implementation of Lanczos-BCG (DBCG),
- Bi-Conjugate Gradient Stabilized (BCGSTAB),
- Generalized Minimum Residual (GMRES),
- Flexible GMRES (FGMRES),
- Direct implementation of Quasi-GMRES (DQGMRES),

The set of iterative solvers includes some of the most commonly used ones and some of most recently developed ones. Most of them have been shown to be robust under different conditions.

All of these methods are Krylov subspace methods, a common characteristic of which is that they only require the coefficient matrices for performing matrix-vector multiplications. This makes it possible to arrange the iterative solvers in a way that is independent of the specific sparse matrix storage
format used. This obviates the need for any considerations regarding data
structures in the iterative solvers module.
When designing this module we attempted to achieve a few of the basic
qualities of standard software systems, namely,

- convenience of use,
- reliability,
- flexibility,
- good performance.

Often these requirements are in conflict with each other. However, al-
though we tried not to make big sacrifices in performance, we feel that
the current implementation, which emphasizes modularity via the reverse
communication mechanism, will be far more amenable to incorporate any
improvements that will be obtained in the other modules, such as the pre-
conditioning techniques, or the basic computational modules.

In the rest of this report, we will discuss in some detail each of the iterative
solvers, the protocol used for the reverse communication and the stopping
criteria used for them. We will also briefly discuss the performance of the
iterative solvers implemented.

2 Algorithms

We first give a brief introduction of the iterative solvers implemented in the
iterative solvers module. For reference, we will call this module ITERS in
what follows. Some of these algorithms have been published fairly recently
and some are currently now widely used. In the second half of this section
we will briefly discuss the complexity of all algorithms implemented.

2.1 The algorithms

Conjugate Gradient (CG). This widely used algorithm was developed
by Hestenes and Stiefel (1952). It is the only one we implemented for solving
Hermitian positive definite linear systems. For references on this algorithm
see, for example [1].
CGNR. This algorithm is intended for solving linear systems as well as least-squares problems. It consists of solving the linear system, $A^T Ax = A^T b$ by a CG method. Since $A^T A$ is always positive semi-definite, it is guaranteed, in theory, to always converge to a solution. CGNR may be a good approach for highly indefinite matrices. For example if the matrix is unitary, then it can solve the linear system in just one step, whereas most of the other Krylov subspace projection methods will typically converge slowly. For typical problems arising from the discretization of partial differential equations, CGNR converges more slowly than CG or BCG and so this approach is not as popular in this particular context.

Bi-Conjugate Gradient (BCG) is a variation of the Lanczos algorithm for solving linear systems [5] and was also developed in its best known form by Fletcher [2]. It is a generalization of the Conjugate gradient algorithm for solving nonhermitian linear systems. For references, see for example [7, 3, 6, 9]. Implicitly the Lanczos algorithm is a projection technique onto a Krylov subspace, such that the representation of the matrix is tridiagonal. In BCG the projected tridiagonal system is solved progressively as the tridiagonalization is performed. Since no pivoting is performed during the solution of the tridiagonal system, the algorithm may break down, or more generally the approximation thus obtained may be very inaccurate in some cases.

Direct implementation of BCG (DBC) is a variant of BCG where the tridiagonal system mentioned above is solved with partial pivoting. In the original BCG algorithm the tridiagonal matrix generated from the Lanczos procedure is solved by Gauss elimination without pivoting. The pivoting increases the stability of the iterative solver, eliminates some of the breakdowns in the BCG algorithm. The benefit of the partial pivoting is important when the linear system is 'hard'.

Bi-Conjugate Gradient Stabilized (BCGSTAB) This algorithm was developed mostly to eliminate the need to use $A^T$ in the BCG algorithm [9]. The Conjugate Gradient Squared (CGS) [8] is the first of this class of techniques referred to as transpose-free variants of the biconjugate gradient method. CGS is a technique that obtains iterates whose residual polynomial are the square of those of the standard BiCG. This has the effect of
accelerating convergence for the same amount of work, in the cases when convergence is fast. However, when the original iterates diverge or have a very erratic convergence behavior, then CGS may have a convergence a poor convergence behavior. We did not implement CGS because of these potential difficulties.

**Transpose-Free Quasi-Minimum Residual method (TFQMR)** This algorithm is another Transpose-Free variation of the Lanczos algorithm developed by R. Freund [4]. The underlying idea is the Quasi-Minimum Residual (QMR) algorithm [3]. Similarly to BiCG, QMR is based on Lanczos tridiagonalization, and requires the use of the transpose of \( A \). It attempts to produce residual norms that are ‘quasi-optimal’, namely some expression for the residual norms is minimized, pretending that the Lanczos vectors were orthonormal. Despite the lack of orthogonality of the Lanczos vectors, the residual vectors tend to behave smoothly, and to show an almost monotonic, decrease.

**Generalized Minimum Residual (GMRES)** GMRES differs from Lanczos-based methods in that it reduces the matrix to the Hessenberg form, instead of the tridiagonal form, and it uses the Givens rotation rather than the Gauss elimination to solve the resulting Hessenberg linear system generated. GMRES performs a minimization on the residual norm which is therefore guaranteed not to increase.

**Flexible GMRES (FGMRES)** is a variation of GMRES which has as an objective to allow variations in the preconditioning. This greatly enhances flexibility by allowing many combinations of preconditioners that can be used. For example any iterative solver can be used as a preconditioner.

**Direct implementation of Quasi-GMRES (DQGMRES)** This is another variation of GMRES which is based on incomplete orthogonalization. In order to limit the size of Krylov subspace, i.e., the work space required, GMRES and FGMRES restart the algorithm, i.e., after a certain number of steps a new outer iteration of GMRES is started with the initial guess set to the latest approximate solution obtained. On the other hand, DQGMRES keeps a fixed number of Krylov vectors, those that the most recent,
and constrains to be orthogonal to each other. As a result the Krylov basis
is not only ‘incompletely orthogonal’. This provides some continuity in the
solution process, and have the effect of reducing the stagnation phenomenon
often observed with GMRES for indefinite problems. In our implementation,
DQGMRES has a similar feature of flexibility of FGMRES.

2.2 Complexity of the algorithms

The main computational kernels of all the above methods are as follows:

1. SAXPY,
2. dot-product,
3. matrix-vector multiplication,
4. preconditioning solve.

In addition, some of the methods require some additional lower order com-
putations which are not represented here. In our implementation, only the
SAXPY and the dot-product operations are part of the iterative solvers.
The matrix-vector multiplications and preconditioning operations will be per-
formed by the caller. Details on this reverse-communication mechanism will
be discussed in Section 3. We must pass work space to the iterative solvers.
Here we will briefly discuss the work space requirements and the time com-
plexities of the iterative solvers. Table 1 lists the space and time complexities,
where $n$ is the dimension of the linear system. For both GMRES($m$) and
DQGMRES($k$), $m, k << n$. The time complexity shown in the table is the
number of floating-point operations required to update the solution averaged
over the number of matrix-vector multiplications called (assuming the num-
ber of iteration is much larger than $m$ and $k$). We should emphasis that
the time complexity only includes the SAXPY and dot-product operations.
In most real applications, the cost of the matrix-vector multiplications to-
gether with the preconditioning operations can be much higher than that of
SAXPY and dot-product, i.e. the FLOPS shown in table 1. Note that the
time complexity listed in the table assumes that the least costly stopping
tests are used. These stopping tests are based on residual norms, since most
of the method provide an inexpensive strategy for computing or estimating
the Euclidean norm of the residual.
<table>
<thead>
<tr>
<th>Method</th>
<th>Space</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>CG</td>
<td>5n</td>
<td>8n</td>
</tr>
<tr>
<td>CGNR</td>
<td>5n</td>
<td>4n</td>
</tr>
<tr>
<td>BCG</td>
<td>7n</td>
<td>7n</td>
</tr>
<tr>
<td>DBCG</td>
<td>11n</td>
<td>7n</td>
</tr>
<tr>
<td>BCGSTAB</td>
<td>9n</td>
<td>10n</td>
</tr>
<tr>
<td>TFQMR</td>
<td>11n</td>
<td>11n</td>
</tr>
<tr>
<td>GMRES(m)</td>
<td>(m+2)n</td>
<td>(2m+7)n</td>
</tr>
<tr>
<td>FGMRES(m)</td>
<td>(2m+2)n</td>
<td>(2m+7)n</td>
</tr>
<tr>
<td>DQGMRES(k)</td>
<td>(2k+1)n</td>
<td>6kn</td>
</tr>
</tbody>
</table>

Table 1: Complexity of the algorithms.

The data in table 1 is directly measured from the program. They are only accurate to the constant in front of the $n$. The exact amount of work space required may be slightly higher for GMRES and its variants, since these require some additional space to store the Hessenberg matrix.

3 Reverse Communication

The rather large number of different data structures that exist in sparse matrix techniques, makes it difficult to develop a library if iterative solvers which accomodates all possible cases in a unified calling sequence. One obvious solution would be to limit the number of possible data storage schemes and have users make the necessary transformations from their data structures to a few possible choices supported by the library, using transformations routines from libraries such as SPARSKIT. This has the disadvantage of providing a rigid architecture. An alternative which is becoming increasingly popular in software packages is to bypass data structure completely.

This mechanism consists of putting the iterative solution subroutine in the body of a FORTRAN loop which is run for as long convergence is not reached. Every time the subroutine is exited it means that it requesting that the user performs a type of operation as identified by a code (e.g. a matrix-vector product) on a given vector and place the result into a given vector.
Once this is done the control is given back to the solver which is called again. Thus, a communication loop in the simplest case would look as follows,

\[
\text{icod}e = 0 \\
\text{continue} \\
\text{call solver (n, param, iparm, wk1, wk2, icod) } \\
\text{if (icod eq. 1) then} \\
\text{call precon(n, wk1, wk2) \(<-- \text{preconditioning operation} \) } \\
\text{goto 1} \\
\text{else if (icod eq. 2) then} \\
\text{call matvec (n, wk1, wk2) \(<-- \text{matrix-vector product} \) } \\
\text{goto 1} \\
\text{endif}
\]

The correct control path inside the solver routine requires some additional care and can be implemented with the help of ASSIGN statements or computed GOTO statements.

Reverse communication allows us to move the matrix dependent operations outside of the iterative solvers, leaving only BLAS-1 operations and other simple operations inside the body of the iterative solvers. For the benefit of portability, we only require the user to provide a dot-product routine, since distributed versions or local versions of these may be needed depending on the environment. The other BLAS-1 type operations are coded directly in the solvers.

An advantage of using reverse communication is that it provides maximum flexibility for the iterative solvers. The user has complete control over the matrix storage format to use, as well as the implementation of the key matrix dependent operations, such as matrix-vector multiplications, and preconditioning.

On the negative side, reverse communication may impose an overhead to the program because of the increase of function calls. However, as can be expected for large problems this overhead is likely to be small. We verified this by performing some tests, and observed that on most machines considered, the overhead was at most very small percentage of the total execution time in the worst case.

Another concern is that since reverse communication gives responsibility to the user on the basic operations. The code does not take a matrix, and right-hand-side as actual inputs and as a result there is no way of checking
whether a failure resides in the method itself or in the user’s implementation of the matrix operations.

However, we note that there are subroutines, e.g., from SPARSKIT, that can be used to translate specified storage schemes for which well tested matrix—operations and preconditioners are available. This will help reduce the time develop or prototype a given iterative solver, from a number of building blocks from other packages. Some illustrations will be provided in later sections.

Before giving some details of the reverse communication protocol, we would like to list all the possible matrix operations that will be required in the course of the iteration process. When no preconditioner is applied, the iterative solvers will perform matrix-vectors with $A v = Au$ and possibly with $A^T v = A^T u$. With both left and right-preconditioning, the two matrix-vector multiplications will become $v = M_l^{-1} A M_c^{-1} u$ and $v = M_c^{-T} A^T M_l^{-T} u$. Therefore, there is a total of six possible operations, namely:

1. $v = Au$, $A$ matrix-vector operation,
2. $v = A^T u$, $A^T$ matrix-vector operation with $A^T$,
3. $v = M_l^{-1} u$, left preconditioning operation,
4. $v = M_c^{-T} u$, left preconditioning operation for $A^T$,
5. $v = M_l^{-1} u$, right preconditioning operation,
6. $v = M_c^{-T} u$, right preconditioning operation for $A^T$.

These operations will be performed outside of the solver routine. Upon return from the solver, the user determines which of these six operations is requested then performs the desired operation on specific vectors and recalls the solver. In addition, the solver may also return with error or termination conditions. Finally, in our current implementation, the user can also opt to apply his own set of convergence or termination criteria. In such cases, the iterative solver must also communicate requests to apply the stopping test to the user. As a result, we have opted to use the following return values for the reverse-communication status parameter,

1: request a matrix-vector multiplication with $A$,
2: request a matrix-vector multiplication with $A^T$, 
3: request a left preconditioner solve $M_l$, 
4: request a left preconditioner transposed solve $M_l^T$, 
5: request a right preconditioner solve $M_r$, 
6: request a right preconditioner transposed solve $M_r^T$, 
10: request the caller to perform a stopping test, 
0: normal termination of the solver, satisfied the convergence test, 
-1: termination because number of matrix-vector multiplications is greater than the preset limit, 
-2: return due to insufficient work space, 
-3: return due to anticipated break-down / divide by zero.

4 Stopping Criteria

To determine when to stop an iteration, we need to define a stopping criterion based on a set of stopping conditions. An iterative solver may stop for one of the following reasons:

- convergence test satisfied, 
- iterations count limit exceeded, 
- insufficient resources to perform/continue the algorithm, 
- algorithm breaks down, 
- external error.

We will address each of above items from the last one to the first. External errors are errors that are external to the body of the iterative solver itself. When a solver requests that the caller perform a given operation, and the
operation fails the reverse communication loop must stop, and the solver is no longer called. This type of error is left to responsibility of the user.

Most of the iterative solvers may break down in some circumstances. Often, this is due to a division by zero, and is easily detected in the iterative solvers. Some of the break-downs that occur may be so-called “happy break-downs” which are characterized by the solution being exact. In such instances, we will signal to the caller that the iterative solvers have satisfied the given convergence test rather than signal for a break-down.

Insufficient-resource errors occur when the iterative solver does not have enough work space to perform its operations. The size of work space is checked before the iteration starts. When this error is detected, the minimum required work space size is reported.

It is common in implementations of the iterative solvers, to limit the maximum number of iterations allowed, in order to prevent cases where the iterative solver makes no progress or a progress that is unacceptably slow towards the solution. In our implementation, the limit is imposed on the total number of matrix-vector multiplications.

The most complex and important stopping criterion is the convergence test. One can decide to stop the iteration based on the size of the residual norm, the estimated error norm, or the correction to the approximate solution at each update. Norms that are commonly used are the 1-norm, 2-norm, and the infinity-norm. The tests can be based on an absolute tolerance or a relative tolerance. More complicated converge tests may be devised using some backward error analysis. For the sake of simplicity, we decided to provide only four different convergence tests in our iterative solvers. If we define \( d x_i = x_i - x_{i-1} \), \( r_i = b - A x_i \), these tests are

\[
\begin{align*}
\|d x_i \| & \leq \tau_r \|b\| + \tau_a \\
\|d x_i \| & \leq \tau_r \|d x_i \| + \tau_a \\
\|r_i \| & \leq \tau_r \|b\| + \tau_a \\
\|r_i \| & \leq \tau_r \|r_0\| + \tau_a
\end{align*}
\]

in which \( \tau_a \) is the absolute tolerance and \( \tau_r \) is the relative tolerance, as supplied by the user. We only use the 2-norm in all the iterative solvers. The user can specify one of the above four convergence tests to be used in the iterative solvers. Alternately, the caller can also that the iterative solver should use a user-defined termination test implemented with reverse
communication as discussed earlier. In this situation, after each update of
the solution $x_i$, the iterative solver will ask the caller to perform a stopping
test. If no stopping criterion is specified, the iterative solver will select a
default stopping test, which is usually the most economical to apply for the
given method being used. In most cases, the default test is the fourth one in
the above list.

5 Using the Iterative Solvers

In this section we will show two sections of pseudo-code to illustrate the use of
the iterative solvers. Then we will show some results on a set of experiments.
In introducing the pseudo-code, we will also give more additional details on
the reverse communication protocol.

5.1 Two Examples

First we will give the prototype of the iterative solvers. All the iterative
solvers are programmed with the following standard calling sequence:

```fortran
solver(n, b, x, ipar, fpar, wk)
integer n, ipar(16)
real*8 b(n), x(n), fpar(16), wk(*)
```

where the $b$ is the right-hand side, $x$ is the solution $n$ is the dimension of the
linear system, $wk$ is the work space, and $ipar$, $fpar$ are two arrays used for
passing various parameters, including those used for reverse communication.
The complete description of the parameters is listed in the appendix. Here
we will give only enough information to explain the following segment of
pseudo-code.

```fortran
ipar(1) = 0
10: call solver(n, b, x, ipar, fpar, wk)
   if (ipar(1).eq.1) then
      call amux(n, wk(ipar(8)), wk(ipar(9)), a, ja, ia)
      goto 10
   else if (ipar(1).eq.2) then
      call atmux(n, wk(ipar(8)), wk(ipar(9)), a, ja, ia)
```

goto 10  
else if (ipar(1).eq.5) then  
call lusol(n, wk(ipar(8)), wk(ipar(9)), alu, jalu, iau)  
goto 10  
else if (ipar(1).eq.6) then  
call lutsol(n, wk(ipar(8)), wk(ipar(9)), alu, jalu, iau)  
goto 10  
else if (ipar(1).gt.0) then  
preconditioner or convergence test not implemented  
else  
solver terminated with code = ipar(1)  
endif

This segment of pseudo-code illustrates how the reverse communication is used to apply an actual iterative solver. It implements an iterative solver with right preconditioning. Here, we assume that the sparse matrix is stored in CSR format. Subroutines _amux_ and _atmux_ from SPARSKIT perform the operations \( v = Au \) and \( v = A^T u \) respectively. For the preconditioner, we assume that an ILU type preconditioner is being used, such as ILUT or ILUTP from SPARSKIT. The results of the incomplete factorization is stored in arrays _alu_, _jalu_, and _iau_. The subroutine _lusol_ solves the two triangular systems from the LU factorization, \( v = (LU)^{-1} u \), and _lutsol_ performs \( v = (LU)^{-T} u \). In the four matrix-vector multiplication and triangular solution routines, the second arguments in the calling sequences are the input vector \( u \), and the third arguments are the output vector \( v \). Before calling the iterative solvers, the first six elements of _ipar_ and _fpar_ must be specified. In this example, we only show the initial assignment to _ipar_(1), see the appendix for an example of assigning all eight input parameters. Three elements of the array _ipar_ are used here, the parameter in _ipar_(1) is used as the status variable discussed before. _ipar_(8) is the pointer to the input vector _u_, i.e. \( u(1) = wk(ipar(8)) \), \( u(2) = wk(ipar(8) + 1) \), \ldots, \( u(n) = wk(ipar(8) + n - 1) \). Finally, _ipar_(9) points the output vector _v_ from the user-supplied matrix operation—whether a matrix-vector or a preconditioning operation. In this example, a right preconditioner is used.

Next we will show an example of using an iterative solver as a preconditioner to another iterative solver.

\[ \text{ipar}(1) = 0 \]
10: call dqgmres(n, b, x, ipar, fpar, wk)
if (ipar(1).eq.1) then
  call amux(n, wk(ipar(8)), wk(ipar(9)), a, ja, ia)
goto 10
else if (ipar(1).eq.5) then
  jpar(1) = 0
20: call cgnr(n, wk(ipar(8)), wk(ipar(9)), jpar, gpar, wk2)
if (jpar(1).eq.1) then
  call amux(n, wk2(jpar(8)), wk2(jpar(9)), a, ja, ia)
goto 20
else if (jpar(1).eq.2) then
  call atmux(n, wk2(jpar(8)), wk2(jpar(9)), a, ja, ia)
goto 20
endif
  goto 10
endif

This example shows how DQGMRES uses CGNR as preconditioner, this type of set-up is also known as inner-outer iteration. The inner iteration, CGNR, uses its own set of parameters jpar, gpar and its own work space. Notice that the inner loop is exactly like in the previous example, and it is possible to precondition the inner iteration as well. We may use any iterative solver in place of CGNR, except DQGMRES itself, since otherwise the local variables that are saved, using the Fortran SAVE command, will be lost when passing from the outer to the inner call. The same set of local variables cannot be used to solve two different systems at the same time. Note that we can in this outer-inner iteration scheme, alternate among several preconditioners, or dynamically adjust the parameters to the preconditioners. An example of this type will be shown later.

5.2 Experiments

We have performed a set of experiments on a couple of different architectures to demonstrate the capabilities of the iterative solvers. We will first report on the speed (in megaflops) of the solvers and then show some data on the convergence behaviors of the iterative solvers. In one of the examples, we an ILUT preconditioner, and in another one we mix two types of preconditioners.
In our first experiment we will not use any preconditioning. All the timings are obtained for the parameters $\tau_a = 0$, $\tau_r^1 = 2.2 \times 10^{-16}$. The default, internally chosen, stopping criteria is used. In all the experiments the iterative solvers are made to run for a maximum of 100 iterations.

The tests have been performed on a SPARC-2, a SPARC-10 and a CRAY C90. Since the types of machines have vastly different capabilities, we tested different size problems. On the SPARC's, the test problem is constructed from a matrix in Harwell/Boeing collection named SHERMAN5. It is a matrix arising from a block oil simulator on a $16 \times 23 \times 3$ grid. The matrix has 3312 rows and 20793 non-zero elements. The matrix used on the CRAY is an exact Jacobian from a 2-D airfoil simulation. It has 38744 rows and 1771722 non-zero elements.

**NAME OF MATRIX = ???????**

Table 2 shows the speed of the major components of the iterative solvers, namely SAXPY, dot-product and matrix-vector multiplication routines. The matrices are in CSR format. The matrix-vector multiplication routine, AMUX is from SPARSKIT. It is constructed from dot-product operations. Due to indirect addressing and the short inner loops, the performance of this matrix-vector multiplication is far from optimal on the CRAY. This also make the program unsuitable for running with more than one PE. Note that since our goal here is to compare the overall speed of the iterative solvers with that of the major components, we have not attempted to optimize the performance of the AMUX routine on the CRAY and have reported on results using only 1 PE.

From table 3 we can see that most of the overall speeds of the iterative solvers are much close to those of the matrix-vector multiplication routine. The GMRES and its variants have more BLAS-1 type operations in addition to matrix-vector multiplications, which explains the higher overall speeds compared with other solvers.

Figure 1 shows the convergence of solving SHERMAN5 using ILUT(3, $10^{-3}$) as preconditioner. This example only uses four out of the nine solvers. Since the preconditioner is fixed, FGMRES behavior exactly like GMRES, so we do reproduce the experiment. The rest of iterative solvers generally converges slower than these four. Figure 2 shows an example of the flexible preconditioning. The solver used is DQGMRES, the preconditioner alter-
Table 2: Speed (megaflops) of the major components of the iterative solvers.

<table>
<thead>
<tr>
<th></th>
<th>SPARC 2</th>
<th>SPARC 10</th>
<th>C90 1 PE</th>
</tr>
</thead>
<tbody>
<tr>
<td>AMUX</td>
<td>2.0</td>
<td>7.7</td>
<td>96</td>
</tr>
<tr>
<td>SAXPY</td>
<td>4.5</td>
<td>10.9</td>
<td>797</td>
</tr>
<tr>
<td>dot-product</td>
<td>5.5</td>
<td>22.5</td>
<td>830</td>
</tr>
</tbody>
</table>

Table 3: Overall speed (megaflops) of the iterative solvers.

<table>
<thead>
<tr>
<th></th>
<th>SPARC 2</th>
<th>SPARC 10</th>
<th>C90 1 PE</th>
</tr>
</thead>
<tbody>
<tr>
<td>CG</td>
<td>2.3</td>
<td>7.6</td>
<td>106</td>
</tr>
<tr>
<td>BCG</td>
<td>2.0</td>
<td>6.6</td>
<td>115</td>
</tr>
<tr>
<td>DBCG</td>
<td>2.0</td>
<td>6.7</td>
<td>115</td>
</tr>
<tr>
<td>CGNR</td>
<td>2.0</td>
<td>6.9</td>
<td>112</td>
</tr>
<tr>
<td>BCGSTAB</td>
<td>2.2</td>
<td>7.6</td>
<td>106</td>
</tr>
<tr>
<td>TFQMR</td>
<td>2.2</td>
<td>7.3</td>
<td>112</td>
</tr>
<tr>
<td>GMRES</td>
<td>3.0</td>
<td>9.3</td>
<td>129</td>
</tr>
<tr>
<td>FGMRES</td>
<td>2.9</td>
<td>9.1</td>
<td>129</td>
</tr>
<tr>
<td>DQGMRES</td>
<td>2.9</td>
<td>9.1</td>
<td>171</td>
</tr>
</tbody>
</table>

nates between one of the regular solvers and SSOR. The number of matrix-vector multiplications shown on the horizontal axis are the total number of the matrix-vector multiplications from both the outer and inner iterations. The number of iterations taken by DQGMRES is not very different from that shown in figure 1.

6 Concluding Remarks

Although we have not shown any experiments on multiprocessor systems, we would like to emphasize we are currently using these techniques on the CM5, a massively parallel computer. We should stress that our main motivation for the use of reverse communication was to ensure maximum portability and
Figure 1: SHERMAN5 solved with ILUT(3, 10^{-3}) preconditioner.
Figure 2: SHERMAN5 solved by the inner-outer iteration schemes with pre-conditioner alternating between SSOR and an iterative solver.
flexibility. Because the tasks requiring communication have been isolated out of the solver itself, there is no difference between a code that is run on the cm5 or the sun. Communication is performed within the distributed dot-product function and the matrix-vector and preconditioning operations. This module of the P-SPARSLIB library can be used on a variety of architectures to solve general sparse linear systems.

A module of iterative solvers without preconditioners would not be very successful. We have also developed a set of preconditioners which will be part of the 'preconditioners' module. Unfortunately, assembling a collection of preconditioners is far more complex than that of developing a module of conjugate gradient-like solvers. The preconditioners must depend on the architecture, data structure used, and the type of problem. Our work in this direction will be developed in a future working note.
References


### The solvers and the calling sequence

The iterative solution techniques provided in this module are the following:

<table>
<thead>
<tr>
<th>Solver</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CG</td>
<td>Conjugate Gradient Method</td>
</tr>
<tr>
<td>CGNR</td>
<td>Conjugate Gradient Method (Normal Residual equation)</td>
</tr>
<tr>
<td>BCG</td>
<td>Bi-Conjugate Gradient Method</td>
</tr>
<tr>
<td>DBCG</td>
<td>BCG with partial pivoting</td>
</tr>
<tr>
<td>BCGSTAB</td>
<td>BCG stabilized</td>
</tr>
<tr>
<td>TFQMR</td>
<td>Transpose-Free Quasi-Minimum Residual method</td>
</tr>
<tr>
<td>GMRES</td>
<td>Generalized Minimum RESidual method</td>
</tr>
<tr>
<td>FGMRES</td>
<td>Flexible version of Generalized Minimum RESidual method</td>
</tr>
<tr>
<td>DQGMRES</td>
<td>Direct versions of Quasi Generalize Minimum RESidual method</td>
</tr>
</tbody>
</table>

These routines all have the following calling sequence:

```fortran
subroutine solver(n, rhs, sol, ipar, fpar, w)
integer n, ipar(16)
real*8 rhs(n), sol(n), fpar(16), w(*)
```

In which:

1. 'solver' is one of the above method names
2. 'n' is the size of the linear system,
3. 'rhs' is the right-hand side of the linear system,
4. 'sol' is the solution to the linear system,
5. 'ipar' is an integer array containing all parameters
6. 'fpar' is an floating-point array storing all parameter to and from the iterative solvers.
7. 'w' is the work space (size is specified in ipar)

The preconditioners can be applied from either from left or right or both (specified by ipar(2), see below).
**Work space required** The work space required by each of the iterative solvers is as follows.

- CG \(== 5 \times n\)
- CGNR \(== 5 \times n\)
- BCG \(== 7 \times n\)
- DBCG \(== 11 \times n\)
- BCGSTAB \(== 9 \times n\)
- TFQMR \(== 11 \times n\)
- GMRES \(== (n+3)(m+2) + (m+1)m/2\) \((m = \text{ipar}(5), \text{default } m=15)\)
- FGMRES \(== n(2m+1) + (m+1)m/2 + 3m + 2\) \((m = \text{ipar}(5), \text{default } m=15)\)
- DQGMRES \(== n + lb \times (2n+4)\) \((lb=\text{ipar}(5)+1, \text{default } lb = 16)\)

**The dot product operation** ALL iterative solvers require a user-specified DOT-product routine named DISTDOT. The prototype of the function DISTDOT is as follows:

```fortran
real*8 function distdot(n,x,ix,y,iy)
integer n, ix, iy
real*8 x(1+(n-1)*ix), y(1+(n-1)*iy)
```

Thus, the interface to DISTDOT is exactly the same as that of DDOT (or SDOT if real \(== \text{real*8}\)) from BLAS-1. It should have the same functionality as DDOT on a single processor machine. On a parallel/distributed environment, each processor can perform DDOT on the data it has, then perform a summation on all the partial results.

To use this set of routines under SPMD/MIMD program paradigm, several items are to be noted:

- 'n' should be the number of vector elements of 'rhs' that is present on the local processor.

- if RHS(i) is on processor j, it is expected that SOL(i) will be on the same processor, i.e. the vectors are distributed to each processor in the same way.
• the preconditioning and stopping criteria specifications have to be the same on all processor involved, ipar and fpar have to be the same on each processor.

• DISTDOT should be replaced by a distributed dot-product function.

**Reverse Communication Protocol**  When a reverse-communication routine returns, it could either stop because the routine has terminated or it simply requires the caller to perform one matrix-vector multiplication. The possible matrices that involve in the matrix-vector multiplications are:

\[
\begin{align*}
A & \quad \text{(the matrix of the linear system)}, \\
A^T & \quad \text{(A transposed)}, \\
ML^{-1} & \quad \text{(inverse of the left preconditioner)}, \\
ML^{-T} & \quad \text{(inverse of the left preconditioner transposed)}, \\
Mr^{-1} & \quad \text{(inverse of the right preconditioner)}, \\
Mr^{-T} & \quad \text{(inverse of the right preconditioner transposed)}. \\
\end{align*}
\]

For all the matrix vector multiplications, \(v = Au\), the input and output vectors are part of the work space 'w', and the starting positions of these vectors are stored in ipar(8:9), see below. The array 'ipar' is used to store the information about the solver. Below is the list of what each element represents:

**ipar(1)** – status of the call/return. a call to the solver with \(ipar(1) = 0\) will initialize the iterative solver. On return from the iterative solver, \(ipar(1)\) carries the status flag which indicates the condition of solver on return. The status information is divided into two categories,

1. a positive value indicates the solver requires a matrix-vector multiplication,
2. a non-positive value indicates termination of the solver.

Here is the current definition:

\[
\begin{align*}
1 & \quad \text{== request a matvec with } A, \\
2 & \quad \text{== request a matvec with } A^T, \\
3 & \quad \text{== request a left preconditioner solve (} ML^{-1}){,} \\
4 & \quad \text{== request a left preconditioner transposed solve (} ML^{-T}){,} \\
5 & \quad \text{== request a right preconditioner solve (} Mr^{-1}){,} \\
6 & \quad \text{== request a right preconditioner transposed solve (} Mr^{-T}){,} \\
\end{align*}
\]
10  == request the caller to perform stopping test,
0   == normal termination of the solver, satisfied the stopping
     criteria,
-1  == termination because iteration number is greater than the
     preset limit,
-2  == return due to insufficient work space,
-3  == return due to anticipated break-down / divide by zero,

ipar(2) - status of the preconditioning
0   == no preconditioning
1   == left preconditioning only
2   == right preconditioning only
3   == both left and right preconditioning

ipar(3) - stopping criteria (details of this will be
         discussed later)

ipar(4) - number of elements in the array 'w'. if this is less
         than the desired size, it will be over-written with the minimum
         requirement. In which case the status flag ipar(1) = -2.

ipar(5) - size of the Krylov subspace (used by GMRES and its
         variants), e.g. GMRES(ipar(5)), FGMRES(ipar(5)),
         DQGMRES(ipar(5))

ipar(6) - maximum number of matrix-vector multiplies, if not a
         positive number the iterative solver will run till convergence
         test is satisfied

ipar(7) - current number of matrix-vector multiplies. It is
         incremented after each matrix-vector multiplication. If there
         is preconditioning, the counter is incremented after the
         preconditioning associated with each matrix-vector multiplication.

ipar(8) - pointer to the input vector to the requested matrix-
         vector multiplication
ipar(9) – pointer to the output vector of the requested matrix-vector multiplication
to perform \( v = A \cdot u \), it is assumed that \( u \) is stored as \( w(ipar(8);ipar(8)+n-1) \)
and \( v \) is stored as \( w(ipar(9);ipar(9)+n-1) \)

ipar(10) – the return address (used to determine where to go to after the caller has performed the requested services)

ipar(11) – the result of the external convergence test
On final return from the iterative solvers, this value will be reflected by ipar(1) = 0 (details discussed later)

ipar(12) to ipar(16) are NOT defined, they are NOT USED by any iterative solver at this time.

Information about the error and tolerance are stored in the array FPAR. So are some internal variables that need to be saved from one iteration to the next one. Since the internal variables are not the same for each routine, we only define the common ones.

The first two are input parameters:
fpar(1) – the relative tolerance
fpar(2) – the absolute tolerance (details discussed later)

When the iterative solver terminates,
fpar(3) – initial residual/error norm
fpar(4) – target residual/error norm
fpar(5) – current residual norm (if available)
fpar(6) – current residual/error norm
fpar(7) – convergence rate

fpar(8:10) are used by some of the iterative solvers to save some internal information.

fpar(11) – number of floating-point operations. The iterative
solvers will add the number of FLOPS to this variable, but they do NOT initialize it, nor add the number of FLOPS due to matrix-vector multiplications (since matvec is outside of the iterative solvers). To insure the correct FLOPS count, the caller should set fpar(11) = 0 before invoking the iterative solvers.

fpar(12:16) are not used in the current implementation.

Whether the content of fpar(3), fpar(4) and fpar(6) are residual norms or error norms depends on ipar(3). If the requested convergence test is based on the residual norm, they will be residual norms. If the caller want to test convergence based the error norms (estimated by the norm of the modifications applied to the approximate solution), they will be error norms. Convergence rate is defined by (Fortran 77 statement) fpar(7) = log10(fpar(3) / fpar(6)) / ipar(7) If fpar(7) = 0.5, it means that approximately every 2 (1/0.5) steps the residual/error norm decrease by a factor of 10.

Stopping criteria   An iterative solver may be terminated due to (1) satisfying convergence test; (2) exceeding iteration limit; (3) insufficient work space; (4) break-down. Checking of the work space is only done in the initialization stage, i.e. when called with ipar(1) == 0. A complete convergence test is done after each update of the solutions. Other conditions are monitored continuously.

With regard to the number of iteration, when ipar(6) is positive, the current iteration number will be checked against it. If current iteration number is greater the ipar(6) than the solver will return with status -1. If ipar(6) is not positive, the iteration will continue until convergence test is satisfied.

Two things may be used in the convergence tests, one is the residual 2-norm, the other one is 2-norm of the change in the approximate solution. The residual and the change in approximate solution are from the preconditioned system (if preconditioning is applied). The DQGMRES and TFQMR use two estimates for the residual norms. The estimates are not accurate, but they are acceptable in most of the cases. Generally speaking, the error of the TFQMR’s estimate is less accurate.
The convergence test type is indicated by ipar(3). There are four type
convergence tests:

(1) tests based on the residual norm;
(2) tests based on change in approximate solution;
(3) default internal criterion, the solver choose one from
above two on its own;
(4) caller will perform his own test, the solver should simply continue.

Here is the complete definition:

-2  == ||dx(i)|| <= rtol * ||rhs|| + atol
-1  == ||dx(i)|| <= rtol * ||dx(1)|| + atol
  0  == solver choosing the test
  1  == ||residual|| <= rtol * ||initialresidual|| + atol
  2  == ||residual|| <= rtol * ||rhs|| + atol
  999 == caller will perform the test

where dx(i) denote the change in the solution at the ith update.
||.|| denotes the 2-norm, rtol = fpar(1) and atol = fpar(2).

If the caller is to perform the convergence test, the outcome
should be stored in ipar(11).

ipar(11) = 0 - failed the convergence test, iterative sovler
should continue
ipar(11) = 1 - satisfied convergence test, iterative solver
should perform the clean up job and stop.

NOTE: the caller should allow the iterative solver to perform clean up
job, since some of the iterative solvers do not directly update the 'sol'
array, the update are accumulated at a different location and the convergence
information (e.g. fpar(3:7)), is only computed at the clean up stage.

Usage: To start solving a linear system, the user needs to specify first 6
elements of the ipar, and first 2 elements of fpar. The user may optionally set
fpar(11) = 0 if one wants to count the number of floating-point operations.
(Note: the iterative solvers will only add the floating-point operations inside
themselves, the caller will have to add the FLOPS from the matrix-vector
multiplication routines and the preconditioning routines in order to account for all the arithmetic operations.)

The following is an example:

\begin{verbatim}
ipar(1) = 0           always 0 to start an iterative solver
ipar(2) = 2           right preconditioning
ipar(3) = 1           use convergence test scheme 1
ipar(4) = 10000       array 'w' has 10,000 free locations
ipar(5) = 10          use *GMRES(10) (e.g. FGMRES(10))
ipar(6) = 100         use at most 100 matvec's
fpar(1) = 1.0E-6      relative tolerance 1.0E-6
fpar(2) = 1.0E-10     absolute tolerance 1.0E-10
fpar(11) = 0.0        clearing the FLOPS counter
\end{verbatim}

After the above specifications, one can start to call an iterative solver, say BCG. Here is a segment of pseudo-code showing how this can be done,

\begin{verbatim}
  10  call bcg(n,rhs,sol,ipar,jpar,w)
      if (ipar(1),eq,1) then
          call amux(n,w(ipar(8)),w(ipar(9)),a,ja,ia)
          goto 10
      else if (ipar(1),eq,2) then
          call atmux(n,w(ipar(8)),w(ipar(9)),a,ja,ia)
          goto 10
      else if (ipar(1),eq,3) then
          left preconditioner solver
          goto 10
      else if (ipar(1),eq,4) then
          left preconditioner transposed solve
          goto 10
      else if (ipar(1),eq,5) then
          right preconditioner solve
          goto 10
      else if (ipar(1),eq,6) then
          right preconditioner transposed solve
          goto 10
      else if (ipar(1),eq,10) then
\end{verbatim}
call my own stopping test routine
goto 10
else if (ipar(1),gt.0) then
    ipar(1) is an unspecified code
else
    the iterative solver terminated with code = ipar(1)
end if

This segment of pseudo-code assumes the matrix is in CSR format, AMUX and ATMUX are two routines from the SPARSKIT MATVEC module. They performs matrix-vector multiplications for CSR matrices. The input vectors are the second arguments, and the output vectors are the third arguments in the calling sequences. For simplicity, we did not show the name of the routine that performs the preconditioning solves or the convergence tests.