# The Parallel Iterative Methods (PIM) package for the solution of systems of linear equations on parallel computers

Rudnei Dias da Cunha
*National Supercomputing Centre and Mathematics Institute*
*Universidade Federal do Rio Grande do Sul, Brazil*
rudnei@cesup.ufrgs.br

Tim Hopkins
*Computing Laboratory*
*University of Kent at Canterbury, U.K.*
trh@ukc.ac.uk

**Abstract**

We present a collection of public-domain Fortran 77 routines for the solution of systems of linear equations using a variety of iterative methods. The routines implement methods which have been modified for their efficient use on parallel architectures with either shared- or distributed-memory. PIM was designed to be portable across different machines. Results are presented for a variety of parallel computers.

## 1 Introduction

The solution of systems of linear equations arises in connection with many applications. Frequently such systems are sparse and of the order of thousands or millions of equations and require the use of parallel computers if they are to be solved in a reasonable time.

Being sparse, direct methods, like Gaussian elimination, are not practical for two main reasons. First, these methods often require large amounts of memory due to fill-in which may destroy the sparsity and second, many applications do not require a solution to machine precision. Iterative methods like Conjugate-Gradients may then be more effective since the user can control the accuracy of the solution obtained, and the sparsity pattern of the system is not altered, thus minimizing storage requirements and, hopefully, considerably reducing the number of floating-point operations required.

Using iterative methods to solve such systems on parallel computers requires some care especially on distributed-memory architectures where the problem of global operations (for example, the sum of scalar values scattered across the processors) is crucial to their performance. We have addressed this problem in PIM by reducing the number of global operations needed by an iterative method while at the same time maintaining its convergence properties.

In section 2 we provide an overview of the package and a number of examples describing the use of PIM are given in section 3. Through a case study in section 4 we look in some detail at how to design efficient routines for particular parallel architectures and provide results obtained on several parallel machines. We conclude, in section 5, with some comments on forthcoming improvements to PIM.

## 2 Overview of the package

PIM had its origins in a set of similar routines which were developed, in `occam2`, by one of the authors (see [5] and [6] for further details). These routines did not however offer all the functionality of PIM, neither were they portable.

To introduce PIM, consider the solution of a non-singular system of $n$ linear equations

$$Q_1 A Q_2 x = b \tag{1}$$

where $A$ is the coefficient matrix, $x$ the solution being sought, $b$ the independent vector and $Q_1$ and $Q_2$ are preconditioners. To solve (1) PIM offers a number of iterative methods, including

- Conjugate-Gradients (CG) [12],
- Bi-Conjugate-Gradients (Bi-CG) [10],
- Conjugate-Gradients Squared (CGS) [20],
- the stabilised version of Bi-Conjugate-Gradients (Bi-CGSTAB) [21],
- generalised minimal residual (GMRES) [19],
- generalised conjugate residual (GCR) [9],
- Conjugate-Gradients for normal equations with minimisation of the residual norm (CGNR) [15],
- Conjugate-Gradients for normal equations with minimisation of the error norm (CGNE) [4],
- transpose-free quasi-minimal residual (TFQMR) [11] and Chebyshev acceleration [14].

The coefficient matrix may be either real or complex; we provide a set of the PIM routines for each, in both single and double precision.

The user may want to use preconditioners to accelerate or obtain convergence. The routines allow the use of left-, right- or symmetric-preconditioning; the system can also be solved without preconditioning.

Seven different stopping criteria are available, these include the usual scaled residual of the iteration vector at the $k$-th iteration $x_k$, i.e., $||\, b - Ax_k \,|| < \varepsilon ||\, b \,||$ and the absolute difference between two successive iteration vectors $||\, x_k - x_{k-1} \,|| < \varepsilon$.

PIM was developed with two main goals

1. to allow the user complete freedom with respect to matrix storage, access, and partitioning,
2. to achieve portability across a variety of parallel architectures and programming environments.

These goals are achieved by hiding from the PIM routines the specific details concerning the computation of the following three linear algebra operations

1. matrix-vector (and transpose matrix-vector) product,

2. preconditioning step,

3. inner-products and vector norm.

Routines to compute these operations need to be provided by the user. Many vendors supply their own optimised linear algebra routines which the user may want to use. A number of packages for the iterative solution of linear systems already exist including ITPACK [13] and NSPCG [17]. PIM differs from these packages in three main aspects. First, while ITPACK and NSPCG may be used on a *parallel vector supercomputer* like a Cray Y-MP, there are no versions of these packages available for *distributed-memory parallel* computers. Second, there is no debugging support; this is dictated by the fact that in some multiprocessing environments parallel I/O is not available. The third aspect is that we do not provide a collection of preconditioners but leave the responsibility of providing the appropriate routines to the user.

In this sense, PIM has many similarities to Ashby and Seager's proposed standard for iterative linear solvers [2]. In that proposal, the user supplies the matrix-vector product and preconditioning routines. We believe that their proposed standard satisfies many of the needs of the scientific community as, drawing on its concepts, we have been able to provide software that can be used in a variety of parallel environments. However, PIM does not always follow the proposal, specifically with regard to the lack of debugging support and the format of the matrix-vector product routines.

Due to the openness of the design of PIM, it is also possible to use it on a sequential machine. In this case, the user can take advantage of the BLAS [8] to compute the three required linear algebra operations. This characteristic is important for testing purposes; once the user is satisfied that the selection of preconditioners and stopping criteria are suitable, the computation may be accelerated by using an appropriate parallel implementation of these operations.

PIM has already been used in a variety of applications, for example, as part of a finite-element code for designing gas turbines (Pratt&Whitney, Canada), the modelling of flux in porous media (ARCO Oil & Gas) and the modelling of geo-electromagnetic induction in the Earth [1].

## 2.1 Parallel programming model

PIM uses the *Single Program, Multiple Data* (SPMD) programming model. The main implication of using this model is that certain scalar values are needed in each processing element (PE). Two of the user-supplied routines, to compute a global sum and a vector norm, must provide for this; distributed-memory computers like the Intel Paragon XP and parallel programming systems like p4 and TCGMSG offer routines which can be used for this purpose.

## 2.2 Data partitioning

With PIM, the iterative method routines have no knowledge of the way in which the user has chosen to store and access either the coefficient or the preconditioning matrices. We thus restrict ourselves to partitioning the vectors.

The assumption made is that each PE knows the *number of elements* of each vector stored in it and that *all* vector variables in a processor have the *same* number of elements. This is a broad assumption that allows us to accommodate many different data partitioning schemes, including contiguous, cyclic (or wrap-around) and scattered partitionings. We are able to make this assumption because the vector-vector operations used – vector accumulations, assignments and copies – are *disjoint element-wise*. The other operations used, involving matrices and vectors, which may require knowledge of the individual indices of vectors, are the responsibility of the user.

PIM requires that the elements of vectors must be stored locally starting from position 1; thus the user has a *local* numbering of the variables which can be translated to a *global* numbering if required. For example, if a vector of 8 elements is partitioned in wrap-around fashion among 2 processors, using blocks of length 1, then the first processor stores elements 1, 3, 5 and 7 in the first four positions of an array; the second processor then stores elements 2, 4, 6 and 8 in positions 1 to 4 of its array. We stress that in most cases this translation is not necessary or can be computed with very few operations, depending on the partitioning scheme used.

### 2.3 Increasing the parallel scalability of iterative methods

A major cause of the poor scalability of implementations of iterative methods on distributed-memory computers is the need to compute inner-products, $\alpha = u^T v = \sum_{i=1}^{n} u_i v_i$, where $u$ and $v$ are vectors distributed across $p$ processors (without loss of generality assume that each processor holds $n/p$ elements of each vector). This computation can be divided in three parts

1. the *local computation* of partial sums of the form $\beta_j = \sum_{i=1}^{n/p} u_i v_i$, within each processor,
2. the *accumulation* of the $\beta_j$ values, where these values travel across the processors in some efficient way (for instance, as if traversing a binary-tree) and are summed during the process. At the end, the value of $\alpha = \sum_{j=1}^{p} \beta_j$ is stored in a single processor,
3. the *broadcast* operation to send $\alpha$ to all processors.

Parts 2. and 3. are usually implemented as a single operation called a *global sum*.

During parts 2. and 3. a number of processors are idle for some time. A possible strategy to reduce this idle time, and thus increase the scalability of the implementation, is to re-arrange the operations in the algorithm so that parts 2. and 3. accumulate a number of partial sums corresponding to several inner-products. Some of the algorithms available in PIM, including CG, CGEV, Bi-CG, CGNR and CGNE have been rewritten using the approach suggested by D'Azevedo and Romine [7]. Others, like Bi-CGSTAB, restarted GCR and restarted GMRES have not been re-arranged but some or all of their inner-products can be computed with a single global sum operation.

An important point to make is that we have chosen modifications to the iterative methods that reduce the number of synchronization points while at the same time maintaining their convergence properties and numerical qualities. This is the case of the D'Azevedo and Romine modification; also, in the specific case of GMRES, which uses the Arnoldi process (a suitable reworking of the modified Gram-Schmidt procedure) to compute a vector basis, the computation of several inner-products with a single global sum does not compromise the numerical stability of the Arnoldi process.

For instance, the restarted GMRES algorithm involves the computation of $j$ inner-products of the form $V_i^T V_j$, $i = 1, 2, \ldots, j$. It is thus possible to arrange for each processor to compute $j$ partial sums using the BLAS routine `_DOT` and store these in an array. Then in a single call to a global sum routine, these arrays are communicated among the processors and their individual elements are summed. On the completion of the global sum, the array containing the $j$ inner-products is stored in a single processor and is then broadcast to the remainder. The CGS and TFQMR implementations available on PIM do not benefit from this approach.

### 2.4 Portability

One of the main driving forces behind the design of PIM was to make the package portable across different parallel architectures. As mentioned previously this was achieved by concentrating machine dependencies in a small number of the common operations found in the iterative methods considered. For instance, by having an external routine to compute the global accumulation referred to in §2.3 it is possible to port the code to another parallel machine by just replacing calls to this external routine by the appropriate calls to the native global sum routine. Another reason for having separate routines is that, in many applications, writing an efficient matrix-vector product routine requires exploiting the structure of either the physical problem, or the modelling technique used to derive the linear system.

The PIM routines are almost completely portable; the only deviation from the ANSI Fortran 77 Standard is the use of routine names with more than six characters. However, the sequential external routines, (for example, the matrix-vector product, preconditioning step and global accumulation/vector norm) need to be replaced by code that performs the interprocessor communications by calls to native routines. In some cases, this process is very simple: Pindor in [18] mentions that porting PIM to the Kendall Square Research KSR1, using the auto-parallelisation software tools available was achieved with little intervention by the user.

## 3 Examples

In this section, we introduce some examples which show how to use PIM. The examples all call the double-precision version of the routines; only changes to the types of the parameters are required to use the other implementations.

### 3.1 Calling a PIM iterative method routine

With the exception of the Bi-CG, CGNR and CGNE methods, all the implemented methods have the same parameter list as CG. The argument list for the double-precision implementation of the CG method is

```
      SUBROUTINE PIMDCG(A,Q1,Q2,X,B,WRK,IPAR,DPAR,
     +                  MATVEC,PRECONL,PRECONR,PDSUM,PDNRM)
```

and for Bi-CG (it is the same for CGNR and CGNE)

```
      SUBROUTINE PIMDBICG(A,Q1,Q2,X,B,WRK,IPAR,DPAR,
     +                    MATVEC,TMATVEC,PRECONL,PRECONR,PDSUM,PDNRM)
```

Note in the example above that, contrary to the proposal in [2], PIM uses separate routines to compute the matrix-vector and transpose matrix-vector products.

### 3.2 External routines

As stated earlier, the user is responsible for supplying certain routines to be used internally by the iterative method codes. One of the characteristics of PIM is that if external routines are not required by an iterative method routine they are not called. Thus the user only needs to provide those subroutines that will actually be called by a particular routine. Depending on the selection of method, preconditioners and stopping criteria, dummy parameters may be passed in place of those that are not used. Some compilers may require the presence of all routines used in the program during the linking phase of the compilation; in this case the user may need to provide stubs for the dummy routines.

The external routines have a fixed parameter list to which the user must adhere. Because no knowledge about the matrix storage format and access is available to a PIM routine, the matrices are passed both to the PIM routine and from there to the external routines using the Fortran 77 *assumed-size array declaration*. From these external routines the user is then able to call routines that perform the actual computation and in which the matrices are referenced according to their actual declaration (see [2]).

**Matrix-vector product** Consider as an example a dense matrix partitioned by contiguous columns among a number of processors. For illustrative purposes we assume that `N` is an integer multiple of `NPROCS`. The following code may then be used

```
      PROGRAM MATV
* SET UP PROBLEM SOLVING PARAMETERS FOR USE BY USER DEFINED ROUTINES
* LEADING DIMENSION OF A
      IPAR(1)=LDA
* NUMBER OF ROWS/COLUMNS OF A
      IPAR(2)=N
* NUMBER OF ELEMENTS STORED LOCALLY
      IPAR(4)=N/NPROCS
* CALL PIM ROUTINE
      CALL PIMDCG(A,Q1,Q2,X,B,WRK,IPAR,DPAR,MATVEC,PRECONL,PRECONR,PDSUM,PDNRM)
      STOP
      END

* MATRIX-VECTOR PRODUCT ROUTINE CALLED BY A PIM ROUTINE. THE
* ARGUMENT LIST TO THIS ROUTINE IS FIXED.
* U IS THE INPUT VECTOR, V THE RESULT OF A*U
      SUBROUTINE MATVEC(A,U,V,IPAR)
      DOUBLE PRECISION A(*),U(*),V(*)
      INTEGER IPAR(*)
      EXTERNAL USERMV
      CALL USERMV(IPAR(1),IPAR(2),IPAR(4),A,U,V)
      RETURN
      END

* USER-DEFINED ROUTINE TO IMPLEMENT A MATRIX-VECTOR PRODUCT ON A
* PARTICULAR PROCESSOR.
      SUBROUTINE USERMV(LDA,N,LOCLEN,A,U,V)
      DOUBLE PRECISION A(LDA,*),U(*),V(*)
      INTEGER LDA,N,LOCLEN
      ...
      RETURN
      END
```

The scheme above can be used for the transpose matrix-vector product as well. We note that many different storage schemes are available for storing sparse matrices; the reader may find it useful to consult Barrett *et al.* [3, pp. 57ff] where such schemes along with algorithms to compute matrix-vector products are discussed.

**Preconditioning** For the preconditioning routines, one may use the scheme outlined above for the matrix-vector product; this may not be necessary, for instance, when there is no need to operate with $A$ or the preconditioner is stored as a vector. An example is the diagonal (or Jacobi) left-preconditioning, where $Q_1 = diag(A)^{-1}$

```
      PROGRAM DIAGP
      EXTERNAL MATVEC,PRECON,PDUMR,PDSUM,PDNRM
      ...
      DO 10 I=1,N
        Q1(I)=1.0D0/A(I,I)
   10 CONTINUE
      ...
* SET LEFT-PRECONDITIONING
      IPAR(8)=1
      CALL DINIT(IPAR(4),0.0D0,X,1)
      CALL PIMDCG(A,Q1,DUMMY,X,B,WRK,IPAR,DPAR,MATVEC,PRECON,PDUMR,PDSUM,PDNRM)
      STOP
      END
      ...
      SUBROUTINE PRECON(A,Q,U,V,IPAR)
      DOUBLE PRECISION A(*),Q(*),U(*),V(*)
      INTEGER IPAR(*)
      EXTERNAL DCOPY,DVPROD
      CALL DCOPY(IPAR(4),U,1,V,1)
      CALL DVPROD(IPAR(4),Q,1,V,1)
      RETURN
      END
```

where `DVPROD` is a routine based on the BLAS Level 1 `DAXPY` routine that performs an element-by-element vector multiplication. This example also shows the use of dummy arguments.

**Eigenvalues estimation and Chebyshev acceleration** Consider the solution of a real linear system using Chebyshev acceleration. We can use a few iterations of the routine `PIMDRGMRESEV` to obtain estimates of the eigenvalues of $Q_1 A$ and then switch to `PIMDCHEBYSHEV` following a simple transformation on the extreme values on the real axis (see the code below for details).

In the following example, we use the Jacobi preconditioner as shown previously. Note that we may use the vector `X`, returned by `PIMDRGMRESEV`, as a possibly better estimate to the solution in `PIMDCHEBYSHEV`.

```
      PROGRAM CHBSOL
      EXTERNAL MATVEC,PRECON,PDUMR,PDSUM,PDNRM2
* SET LEFT-PRECONDITIONING
      IPAR(8)=1
* SET FEW ITERATIONS FOR RGMRESEV
      IPAR(10)=5
      CALL DINIT(IPAR(4),0.0D0,X,1)
      CALL PIMDRGMRESV(A,Q1,DUMMY,X,B,WRK,IPAR,DPAR,MATVEC,PRECON,PDUMR,
     +                 PDSUM,PDNRM)
```

```
* BOX CONTAINING THE EIGENVALUES IS RETURNED IN
* DPAR(3), DPAR(4), DPAR(5), DPAR(6), THE FIRST TWO ARE THE INTERVAL
* ALONG THE REAL AXIS, THE LAST TWO ARE THE INTERVAL ALONG THE IMAGINARY
* AXIS.

* MODIFY REAL INTERVAL TO REFLECT EIGENVALUES OF I-Q1*A
      MU1=DPAR(3)
      MUN=DPAR(4)
      DPAR(3)=1.0D0-MUN
      DPAR(4)=1.0D0-MU1
* SET MAXIMUM ITERATIONS FOR CHEBYSHEV
      IPAR(10)=N
      CALL PIMDCHEBYSHEV(COEFS,Q1,Q2,X,B,DWRK,IPAR,DPAR,MATVEC,
     +                   PRECON,PRECON,PDSUM,PDNRM2)
      STOP
      END
```

**Inner-products, vector norms and global accumulation** When running PIM routines on multiprocessor architectures, the inner-product and vector norm routines require accumulation and broadcast operations. On vector processors these operations are handled directly by the hardware while on distributed-memory architectures these operations involve the exchange of messages among the processors.

When a PIM iterative routine needs to compute an inner-product, it calls the BLAS routine DDOT to compute the partial sums. The user-supplied routine PDSUM is then used to generate the global sum of those partial sums. The following code shows the routines to compute the global sum and the vector 2-norm $||\,u\,||_2 = \sqrt{u^T u}$ using DDOT and the global operations provided by the TCGMSG system

```
      SUBROUTINE PDSUM(ISIZE,X)
      INTEGER ISIZE, MSGTYPE
      DOUBLE PRECISION X(*)
      EXTERNAL DGOP
* CALL TCGMSG "DGOP" ROUTINE TO PERFORM GLOBAL ACCUMULATION (A
* SUM AS INDICATED BY THE '+' IN THE CALL TO DGOP) AND BROADCAST
* OVER THE "X" ARRAY STORED IN EACH PROCESSOR.
      MSGTYPE=100
      CALL DGOP(MSGTYPE,X,ISIZE,'+')
      RETURN
      END

      FUNCTION PDNRM2(LOCLEN,U)
      DOUBLE PRECISION PDNRM2
      INTEGER LOCLEN,MSGTYPE
      DOUBLE PRECISION U(*),PSUM,DDOT
      EXTERNAL DDOT,DGOP
      PSUM=DDOT(LOCLEN,U,1,U,1)
      MSGTYPE=200
      CALL DGOP(MSGTYPE,PSUM,1,'+')
      PDNRM2=DSQRT(PSUM)
      RETURN
      END
```

It should be noted that PDSUM is actually a wrapper to the global sum routine available on a particular machine. Also, when executing PIM on a sequential or shared-memory parallel computer, these routines are stubs i.e., the contents of the array `X` *must not* be altered since its elements already contain the inner-product values.

Note that when using the `COMPLEX/DOUBLE COMPLEX` versions of the PIM routines, it may be necessary to perform explicit type conversions before and after calling a proprietary global sum routine if the latter does not handle complex data.

The parameter lists for these routines were decided upon after inspecting the format of the global operations available from existing systems, including p4, TCGMSG and the Intel Paragon NX library.

## 4 Case study – solving a partial differential equation

In this section we look in detail at how to write efficient matrix-vector product and preconditioning routines for a specific problem, the solution of a partial differential equation (PDE) using a five-point finite-difference approximation.

Suppose that the square region on which we wish to approximate the PDE is subdivided into $l+1$ rows and columns giving a grid containing $l^2$ internal points, each point being numbered $i+(j-1)l$, $i,j=1,2,\ldots,l$ (see Figure 1). At each point we assign five different values corresponding to the *center*, *north*, *south*, *east* and *west* points of the stencil ($\alpha_{i,j}$, $\beta_{i,j}$, $\gamma_{i,j}$, $\delta_{i,j}$, $\varepsilon_{i,j}$ respectively) which are derived from the PDE and the boundary conditions. The approximate solution is then obtained by solving a linear system of order $n=l^2$.

The matrix-vector product $v=Au$ is obtained by computing

$$v_{i,j}=\alpha_{i,j}u_{i,j}+\beta_{i,j}u_{i+1,j}+\gamma_{i,j}u_{i-1,j}+\delta_{i,j}u_{i,j+1}+\varepsilon_{i,j}u_{i,j-1} \tag{2}$$

where some of the $\alpha$, $\beta$, $\gamma$, $\delta$ and $\varepsilon$ may be zero according to the position of the point relative to the grid. Note that only the neighbouring points in the vertical and horizontal directions are needed to compute $v_{i,j}$.

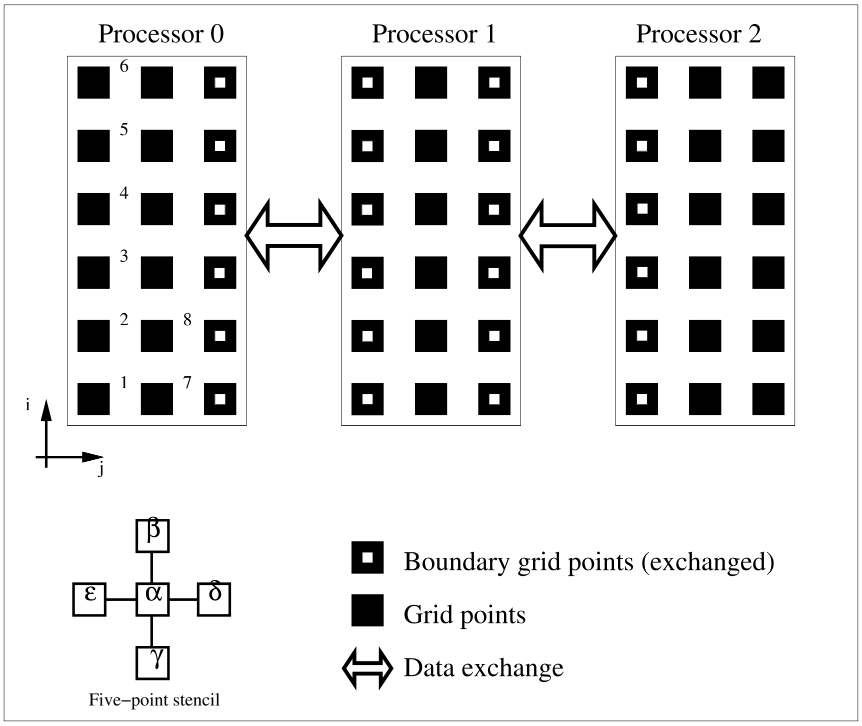
A parallel computation of (2) may be organised as follows. The grid points are partitioned by vertical panels among the processors as shown in Figure 1. A processor holds at most $\lceil l/p \rceil$ columns of $l$ grid points. To compute the matrix-vector product, each processor exchanges with its neighbours the grid points in the "interfaces" between the processors (the points marked with white squares in Figure 1). Equation (2) is then applied independently by each processor on its local grid points, except at the local interfacing points. After the interfacing grid points from the neighbouring processors have arrived at a processor, (2) is applied using the local interfacing points and those from the neighbouring processors.

This parallel computation offers the possibility of overlapping communication with the computation. If the number of local grid points is large enough, one may expect that while (2) is being applied to those points, the interfacing grid points of the neighbouring processors will have been transferred and be available for use. This means that there may be very little overhead in the transfer of data (note such an overlap is only possible if *asynchronous* transfer of messages is available). The example below is taken from the matrix-vector product routine using the Intel Paragon NX library

```
      SUBROUTINE PDMVPDE(NPROCS,MYID,MYP,LDC,L,MYL,COEFS,U,V,UEAST,UWEST)
* Declarations ...

* Send border U values to (myid+1)-st processor
      MSGTYPE = 1000
```

Figure 1: Matrix-vector product, PDE storage format.




```
      TO = MYID + 1
      SID0 = ISEND(MSGTYPE,U(EI0),MSGLEN,TO,MYP)

* Post to receive border U values from (myid+1)-st processor
      MSGTYPE = 1001
      RID0 = IRECV(MSGTYPE,UEAST,MSGLEN)

* Send border U values to (myid-1)-st processor
      MSGTYPE = 1001
      TO = MYID - 1
      SID1 = ISEND(MSGTYPE,U(WI0),MSGLEN,TO,MYP)

* Post to receive border U values from (myid-1)-st processor
      MSGTYPE = 1000
      RID1 = IRECV(MSGTYPE,UWEST,MSGLEN)

* Compute with local grid points ...

* Needs "east" data, wait for completion of receive
      CALL MSGWAIT(RID0)

* Compute with local interfacing grid points in the "east" ...

* Needs "west" data, wait for completion of receive
      CALL MSGWAIT(RID1)

* Compute with local interfacing grid points in the "west" ...
```

```
* Release message ID from isend
      CALL MSGWAIT(SID0)
* Release message ID from isend
      CALL MSGWAIT(SID1)

      RETURN
      END
```

The computation of the transpose matrix-vector product for the PDE case may be implemented in a similar fashion. Before the computation starts, each processor exchanges with its left and right neighbouring processors the *east* and *west* coefficients corresponding to the interfacing grid points. The computation performed is then similar to the matrix-vector product described above except that for each interfacing grid point we apply

$$v_{i,j} = \alpha_{i,j}u_{i,j} + \gamma_{i+1,j}u_{i+1,j} + \beta_{i-1,j}u_{i-1,j} + \varepsilon_{i,j+1}u_{i,j+1} + \delta_{i,j-1}u_{i,j-1} \tag{3}$$

Comparing (3) to (2) we see that the coefficients are swapped in the north-south and east-west directions. Note that due to the partitioning imposed we do not need to exchange the north and south coefficients.

## 4.1 A matrix-vector product for parallel vector architectures

For parallel vector architectures like the Cray Y-MP2E, the routines outlined above are not efficient, because of the small vector lengths involved. A routine that entails the use of long vectors is obtained if one uses a *diagonal-wise* matrix-vector product for the 5-point stencil, which can be written as a sequence of _AXPYs. The use of _AXPYs will also bring a better performance because these operations are usually very efficient on such machines.

Consider the same storage scheme described before i.e., five coefficients ($\alpha$, $\beta$, $\gamma$, $\delta$ and $\varepsilon$) are stored per grid point, and numbered sequentially as $i + (j-1)l$, $i, j = 1, 2, \ldots, l$. If the coefficients are stored in five separate arrays of size $n = l^2$, then the matrix-vector product $v = Au$ can be obtained by the following sequence of operations

$$\begin{aligned}
v_k &= \alpha_k u_k, \quad k = 1, 2, \ldots, n \\
v_k &= v_k + \beta_k u_{k+1}, \quad k = 1, 2, \ldots, n-1 \\
v_k &= v_k + \gamma_k u_{k-1}, \quad k = 2, 3, \ldots, n \\
v_k &= v_k + \delta_k u_{k+l}, \quad k = 1, 2, \ldots, n-l \\
v_k &= v_k + \varepsilon_k u_{k-l}, \quad k = l+1, l+2, \ldots, n
\end{aligned}$$

and the transpose matrix-vector product, $v = A^T u$, is obtained similarly,

$$\begin{aligned}
v_k &= \alpha_k u_k, \quad k = 1, 2, \ldots, n \\
v_{k+1} &= v_{k+1} + \beta_k u_k, \quad k = 1, 2, \ldots, n-1 \\
v_{k-1} &= v_{k-1} + \gamma_k u_k, \quad k = 2, 3, \ldots, n \\
v_{k+l} &= v_{k+l} + \delta_k u_k, \quad k = 1, 2, \ldots, n-l \\
v_{k-l} &= v_{k-l} + \varepsilon_k u_k, \quad k = l+1, l+2, \ldots, n
\end{aligned}$$

Experiments on the Cray Y-MP2E/232 showed that these routines brought more than a three-fold increase in performance, from 40MFLOPS to 140MFLOPS.

## 4.2 Preconditioners

We present two different preconditioners for this problem, the *IDLU*(0) (a variant of the usual *ILU*(0) preconditioner) and polynomial preconditioners.

**The *IDLU*(0) preconditioner** This is a modification of the *ILU*(0) preconditioner to allow the computation of the preconditioning step without any communication being performed. To achieve this, note that the matrices arising from the five-point finite-difference discretisation have the following structure

$$A = \begin{bmatrix} B & E & & \\ F & B & \ddots & \\ & \ddots & \ddots & E \\ & & F & B \end{bmatrix}, \quad B = \begin{bmatrix} \alpha & \beta & & \\ \gamma & \alpha & \ddots & \\ & \ddots & \ddots & \beta \\ & & \gamma & \alpha \end{bmatrix}$$

where $E$ and $F$ are diagonal matrices and $\alpha$, $\beta$ and $\gamma$ are the *central*, *north* and *south* coefficients derived from the discretisation (the subscripts are dropped for clarity). Each matrix $B$ approximates the unknowns in a single vertical line of the grid in Figure 1.

To compute a preconditioner, $Q = LU$, we modify the *ILU*(0) algorithm in the sense that the blocks $E$ and $F$ are discarded (because only the diagonal blocks are considered we refer to this factorisation as *IDLU*(0)). The resulting $L$ and $U$ factors have the following structure

$$\begin{aligned} L &= \begin{bmatrix} X & & & \\ & X & & \\ & & \ddots & \\ & & & X \end{bmatrix}, \quad X = \begin{bmatrix} 1 & & & \\ \check{\gamma} & 1 & & \\ & \ddots & \ddots & \\ & & \check{\gamma} & 1 \end{bmatrix}, \\ U &= \begin{bmatrix} Y & & & \\ & Y & & \\ & & \ddots & \\ & & & Y \end{bmatrix}, \quad Y = \begin{bmatrix} \check{\alpha} & \check{\beta} & & \\ & \check{\alpha} & \ddots & \\ & & \ddots & \check{\beta} \\ & & & \check{\alpha} \end{bmatrix} \end{aligned}$$

where $\check{\alpha}$, $\check{\beta}$ and $\check{\gamma}$ are the modified coefficients arising from the *ILU*(0) algorithm. From the structure of $L$ and $U$ it may be clearly seen that the preconditioning step reduces to the solution of small (order $l$), independent, triangular systems. Each of these systems corresponds to a vertical line in the grid (the grid was partitioned in vertical panels) and these systems may be solved independently in each processor.

**Polynomial preconditioners** These are preconditioners that can be expressed by

$$\left( \sum_{i=0}^{m} \gamma_{m,i} \left( I - (diag(A))^{-1} A \right)^{i} \right) (diag(A))^{-1}$$

which can easily be computed as a sequence of vector updates and matrix-vector products using Horner's algorithm. Thus, this preconditioner is available for use as soon as an efficient matrix-vector product has been developed.

Note that the $\gamma_{m,i}$ coefficients define the kind of polynomial preconditioner being used. The Neumann preconditioner is obtained with $\gamma_{m,i} = 1, \forall i$; the weighted and unweighted least-squares polynomial preconditioners are those reported in [16].

### 4.3 Results

We present some results for the solution of a system derived from the five-point finite-difference discretisation of the convection-diffusion equation

$$-\epsilon\left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}\right) + \cos(\alpha)\frac{\partial u}{\partial x} + \sin(\alpha)\frac{\partial u}{\partial y} = 0 \tag{4}$$

on the unit square, with $\epsilon = 0.1$, $\alpha = -\pi/6$ and $u = x^2 + y^2$ on $\Omega R$. The first order terms were discretised using forward differences (this problem was taken from [20]). The system derived from this discretisation is nonsymmetric. The results are given for a square region of size $l = 128$ leading to a system of order $n = 16384$.

We used both the $IDLU(0)$ and the Neumann polynomial preconditioner of degree 1 as *left*-preconditioners to solve this problem. The stopping criterion used was $\| z_k \|_2 < 10^{-10} \| b \|_2$ where $z_k$ is the residual obtained either by recurrence equations or approximations; using this criterion a solution will be accepted if $\| z_k \|_2 < 3.802 \times 10^{-14}$. The maximum number of iterations allowed was 512 and the initial value of the solution vector was $(0, 0, \ldots, 0)^T$. For the restarted GMRES and GCR the restarting value used was 10. The results are reported for the real, double-precision versions of the routines.

Tables 1 and 2 show the results obtained with the PIM routines for the $IDLU(0)$ and Neumann preconditioners on an SGI Indy II (MIPS R4400 CPU). A status value of 0 indicates convergence, and $-1$ indicates that no convergence was obtained in the maximum number of iterations allowed. This example is characteristic of the problems facing the user of iterative method i.e., not all methods converge to the solution and some preconditioner may cause an iterative method to diverge (or converge slowly). We stress that the methods that have failed to converge in this example may converge for other systems.

In Table 3 we present the execution times obtained solving the test problem with the `PIMDRGMRES` routine and the Neumann polynomial preconditioning on the Intel Paragon XP, Kendall Square Research KSR1, SGI Challenge, Cray Y-MP2E and Cray C9016E. The results for the Cray machines were obtained with the modified matrix-vector product routines described in §4.1. The programs running on the SGI Challenge are from the set of examples available with the PIM distributed software using the PVM message-passing library. Note that for both the SGI Challenge and the Intel Paragon XP superlinear effects occur; we believe this is due to the specific memory organization of those machines (hierarchic memories and/or presence of a cache memory).

The graphs in Figure 2 show the speed-up curves for the `PIMDCG` and `PIMDRGMRES` routines running on the Paragon. We include in those graphs the curves for the same problem but with a system of order $n = 4096$ to show the effects of problem scaling. Almost linear speed-ups occur for $p \leq 8$ for both preconditioners for $n = 16384$. Note also that the polynomial preconditioner has a better speed-up for $p = 32$ than the $IDLU(0)$ preconditioner for $n = 16384$. We should point out that preliminary versions of PIM that did not include the reduction of synchronization points in the computation of several inner-products had a markedly inferior performance, even for the smaller problem size shown.

Table 1: Example with $IDLU(0)$ preconditioner.

| Method | $k^*$ | Time(s) | $\|\| r^{(k^*)} \|\|_2$ | Status |
|---|---|---|---|---|
| CG | 512 | 78.24 | 29 | $-1$ |
| CGEV | 512 | 194.65 | 29 | $-1$ |
| Bi-CG | 512 | 132.31 | $5.1 \times 10^5$ | $-1$ |
| CGS | 291 | 60.28 | $7.2 \times 10^{-15}$ | 0 |
| Bi-CGSTAB | 312 | 69.07 | $9.9 \times 10^{-15}$ | 0 |
| RGMRES | 150 | 288.35 | $1.32 \times 10^{-14}$ | 0 |
| RGMRESEV† | 150 | 289.51 | $1.3 \times 10^{-14}$ | 0 |
| RGCR | 138 | 333.88 | $1.3 \times 10^{-14}$ | 0 |
| CGNR | 512 | 132.81 | $1.2 \times 10^{-4}$ | $-1$ |
| CGNE | 512 | 120.89 | $1.1 \times 10^3$ | $-1$ |
| TFQMR | 512 | 310.48 | $1.0 \times 10^{-9}$ | $-1$ |

† Interval containing eigenvalues (real axis): $[0.0058, 1.9332]$

Table 2: Example with Neumann polynomial preconditioner.

| Method | $k^*$ | Time(s) | $\|\| r^{(k^*)} \|\|_2$ | Status |
|---|---|---|---|---|
| CG | 512 | 94.63 | $1.2 \times 10^2$ | $-1$ |
| CG | 512 | 175.58 | $1.2 \times 10^2$ | $-1$ |
| Bi-CG | 512 | 176.72 | 32 | $-1$ |
| CGS | 235 | 74.82 | $2.9 \times 10^{-15}$ | 0 |
| Bi-CGSTAB | 225 | 80.47 | $8.3 \times 10^{-15}$ | 0 |
| RGMRES | 70 | 153.90 | $1.3 \times 10^{-14}$ | 0 |
| RGMRESEV† | 70 | 154.76 | $1.3 \times 10^{-14}$ | 0 |
| RGCR | 65 | 204.25 | $1.3 \times 10^{-14}$ | 0 |
| CGNR | 512 | 146.88 | $3.4 \times 10^{-4}$ | $-1$ |
| CGNE | 512 | 135.01 | $2.8 \times 10^2$ | $-1$ |
| TFQMR | 512 | 432.32 | $4.5 \times 10^{-10}$ | $-1$ |

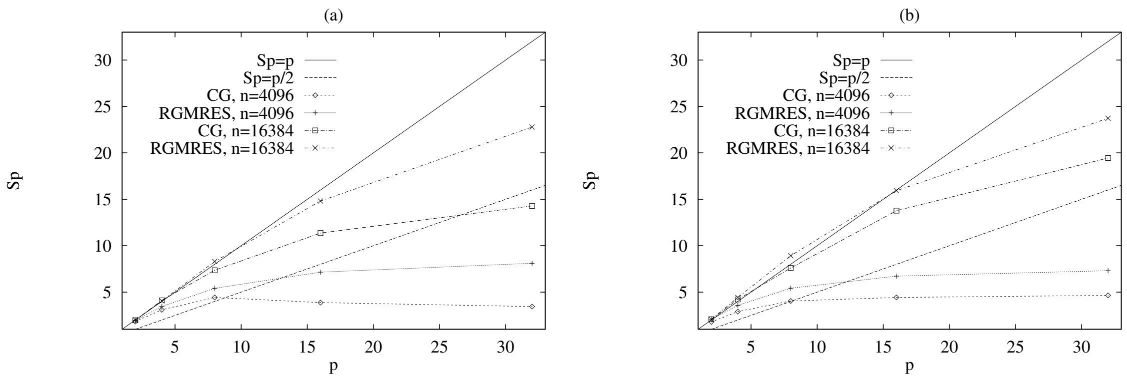
† Interval containing eigenvalues (real axis): $[0.0028, 0.9889]$

Table 3: Execution time (in seconds) for test problem solved by `PIMDRGMRES` with Neumann polynomial preconditioning.

| $p$ | Intel Paragon XP | SGI Challenge† | KSR1‡ | Cray Y-MP2E | Cray C9016 |
|---|---|---|---|---|---|
| 1 | 75.95 | 400.77 | 453.20 | | |
| 2 | 36.59 | | 297.4 | 11.64 | |
| 4 | 17.16 | 84.03 | | | 4.99 |
| 8 | 8.52 | | 166.8 | | |
| 16 | 4.77 | | | | |
| 32 | 3.20 | | | | |

† S. Thomas, CERCA/Montréal
‡ A. Pindor, U. of Toronto [18]

Figure 2: Speed-ups on the Intel Paragon XP (NX library): (a) *IDLU*(0) preconditioner, (b) Neumann polynomial preconditioner.




# 5 Concluding remarks

We have presented the design details of the PIM package together with some practical results which show the efficiency of the implementation.

Among the future improvements we intend to make available a set of matrix-vector product and preconditioning routines for some typical applications, using the PVM and MPI message-passing libraries. Another is a preliminary step towards a HPF version, initially providing a version of PIM coded in Fortran 90. We believe these improvements will reduce the effort of porting the package to other machines.

### Acknowledgements

We would like to thank Steve Thomas (Centre de Recherche en Calcul Apliqué/Montréal), Andrzej Pindor (University of Toronto) and Paulo Tibério M. de Bulhões (Cray Research) who tested PIM on the SGI Challenge, the Kendall Square Research KSR1 and the Cray C9016E respectively and the *National Supercomputing Centre, Federal University of Rio Grande do Sul (Brazil)*, the *Parallel Laboratory, University in Bergen (Norway)* and *Digital Equipment Corporation* (via the Internet Alpha Program), who kindly made their facilities available for our tests.

**Obtaining PIM** PIM can be obtained via anonymous FTP from `unix.hensa.ac.uk`, file `/misc/netlib/pim/pim.tar.Z` or from `euler.mat.ufrgs.br`, file `/pub/pim/pim.tar.Z`.

The distribution software comes with the PIM routines for `REAL`, `DOUBLE PRECISION`, `COMPLEX` and `DOUBLE COMPLEX` data types, and a set of examples, for sequential and parallel shared-memory computers and distributed-memory computers. In the latter case the examples are for message-passing systems including PVM, p4 and TCGMSG, and for the Intel Paragon using NXLIB.

# References

[1] A. Agarwal. An application of conjugate gradient methods in electromagnetic induction studies. Private communication.

[2] S.F. Ashby and M.K. Seager. A proposed standard for iterative linear solvers (version 1.0). Report UCRL-102860, Numerical Mathematics Group, Computing & Mathematics Research Division, Lawrence Livermore National Laboratory, January 1990.

[3] R. Barrett, M. Berry, T. Chan, J. Demmel, J. Donald, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. van der Vorst. *Templates for the solution of linear systems: building blocks for iterative methods.* SIAM, Philadelphia, 1993.

[4] E.J. Craig. The N-step iteration procedures. *Journal of Mathematical Physics*, 34:64–73, 1955.

[5] R.D. da Cunha. *A Study on Iterative Methods for the Solution of Systems of Linear Equations on Transputer Networks.* PhD thesis, Computing Laboratory, University of Kent at Canterbury, July 1992.

[6] R.D. da Cunha and T.R. Hopkins. Parallel preconditioned Conjugate-Gradients methods on transputer networks. *Transputer Communications*, 1(2):111–125, 1993. Also as TR-5-93, Computing Laboratory, University of Kent at Canterbury, U.K.

[7] E.F. D'Azevedo and C.H. Romine. Reducing communication costs in the Conjugate Gradient algorithm on distributed memory multiprocessors. Research Report ORNL/TM-12192, Oak Ridge National Laboratory, 1992.

[8] J.J. Dongarra, J. Du Croz, S. Hammarling, and R.J. Hanson. An extended set of FORTRAN Basic Linear Algebra Subprograms. *ACM Transactions on Mathematical Software*, 14(1):1–17, 1988.

[9] S.C. Eisenstat. A note on the generalized Conjugate Gradient method. *SIAM Journal of Numerical Analysis*, 20:358–361, 1983.

[10] R. Fletcher. *Conjugate Gradient Methods for Indefinite Systems*, volume 506 of *Lecture Notes in Mathematics*, pages 73–89. Spring-Verlag, Heidelberg, 1976.

[11] R.W. Freund. A transpose-free quasi-minimal residual algorithm for non-Hermitian linear systems. Submitted to SIAM Journal of Scientific and Statistical Computing.

[12] G.H. Golub and C.F. Van Loan. *Matrix Computations.* Johns Hopkins University Press, Baltimore, 2nd edition, 1989.

[13] R.G. Grimes, D.R. Kincaid, and D.M. Young. ITPACK 2.0 user's guide. Report No. CNA-150, Center for Numerical Analysis, University of Texas at Austin, August 1979.

[14] L.A. Hageman and D.M. Young. *Applied Iterative Methods.* Academic Press, New York, 1981.

[15] M.R. Hestenes and E.L. Stiefel. Method of Conjugate Gradients for solving linear systems. *Journal of Research National Bureau of Standards*, 49:435–498, 1952.

[16] W.H. Holter, I.M. Navon, and T.C. Oppe. Parallelizable preconditioned Conjugate Gradient methods for the Cray Y-MP and the TMC CM-2. Technical report, Supercomputer Computations Research Institute, Florida State University, December 1991.

[17] T.C. Oppe, W.D. Joubert, and D.R. Kincaid. NSPCG user's guide – version 1.0. Report No. CNA-216, Center for Numerical Analysis, University of Texas at Austin, April 1988.

[18] A. Pindor. Experiences with implementing PIM (Parallel Iterative Methods) package on KSR1. In *Supercomputing Symposium '94*, Toronto, June 1994.

[19] Y. Saad and M.H. Schultz. GMRES: a generalized minimal residual algorithm for solving nonsymmetric linear systems. *SIAM Journal of Scientific and Statistical Computing*, 7:856–869, 1986.

[20] P. Sonneveld. CGS, a fast Lanczos-type solver for nonsymmetric linear systems. *SIAM Journal of Scientific and Statistical Computing*, 10:36–52, 1989.

[21] H.A. van der Vorst. Bi-CGSTAB: A fast and smoothly converging variant of Bi-CG for the solution of nonsymmetric linear systems. *SIAM Journal of Scientific and Statistical Computing*, 13:631–644, 1992. Also as Report No. 90-50, Mathematical Institute, University of Utrecht.