ParPre: A Parallel Preconditioners Package reference manual for version 2.0.21, revision 1

Victor Eijkhout and Tony Chan*

April 1998

This is the reference manual for the ParPre library of parallel preconditioners. The design philosophy and mathematical background are covered in a companion paper [3].

Contents

*This research was supported by the ARO under contract DAAL-03-9-C-0047 (Univ. of Tenn. subcontract ORA 4466.04, Amendment 1 and 2).

1 Introduction

This is the manual and users' guide of ParPre, a package of parallel preconditioners implemented using MPI communication routines, and matrix/vector primitives from the Petsc library.

1.1 Preconditioners

In many scientific applications, one needs to solve linear systems

$$Ax = b$$

with A square and non-singular, often symmetric, or positive definite, or an M-matrix. For a variety of reasons, one can choose to solve this system by an iterative method, rather than by some form of Gaussian elimination.

Iterative methods feature a preconditioning step, that is, the exact solution of a linear system

$$Cx = b$$

where C in some sense approximates A. In general, a more accurate choice of C will yield a more speedy solution of the iterative process, but such a choice will also be more costly to construct and to apply.

For general background information about iterative methods and preconditioners, see the 'Templates' book [2], and books by Axelsson [1], Hackbusch [7], and Saad [10]. More specifically for domain decomposition methods and Schwarz methods, see, e.g., [4, 11].

1.2 Parallel preconditioners in ParPre

The ParPre package contains preconditioners that are specifically designed for execution on parallel computers. For this, we have taken two approaches in the design of the package:

1. Certain preconditioners are based on subdomain partitioning of the physical problem. In the context of the matrix-vector equation to be solved, this corresponds to a partitioning of the problem variables. On each subdomain, a sub-system is then solved by a classical, uni-processor preconditioner or solver.

   This class of preconditioners comprises Schwarz methods, Schur complement domain decomposition, and block factorisation methods. This last category can reduce to classical methods such as $ILU(0)$ for certain parameter settings.

2. Other preconditioners are based on multi-colourings of the problem variables. Given a multi-colouring and a simple partitioning of the problem variables, one can reasonably expect each colour to be present on each processor in roughly equal measure.

This class of preconditioners comprises multi-colour $SSOR$ and ILU factorisations, and various types of algebraic multigrid.

1.3 Library design

1.3.1 Design philosophy

We have made the ParPre library to a large extent a 'black-box' library in order to make it as easy to use as possible. This is apparent in the following:

- The user never has direct access to, the preconditioner data structure.
- Any method parameters are set by function calls. As a result, parameter lists are always short, and no unnecessary parameters are passed.
- Parameters have sensible default settings, so that, in the absence of user calls setting them, the method will still perform satisfactorily.

The design philosophy and mathematical background are covered in detail in a companion paper [3].

1.3.2 MPI and Petsc

The ParPre library uses MPI for the message passing protocol, and Petsc for intermediate level primitives for distributed matrix manipulation. These packages have to be installed before you can use ParPre.

A program using ParPre preconditioners has to perform some conversion of data structures to Petsc format; see section 3. After the initial setup, the cost of this is essentially zero. See section B for more information on Petsc and the relationship of ParPre to Petsc.

1.3.3 Language

The ParPre library is written, like the Petsc library, in C. A Fortran interface will be added in a next version.

1.3.4 Availability

The ParPre library can be found on Netlib:

`http://www.netlib.org/`

and on the ftp site of one of the authors:

`ftp://math.ucla.edu/pub/eijkhout/software/parpre-2.0.17.tar.Z`

This manual can be read online at

`http://www.math.ucla.edu/~eijkhout/reports/parpre_manual/manual.html`

Part I
Setup and context

This part of the document describes all practical issues in using the ParPre package.

2 Program structure

2.1 Example program structure

The following is an example of the global structure of a program using ParPre preconditioners in an iterative method. Many details have been omitted here.

First of all, the main program declares a coefficient matrix and a preconditioner (see section 2.2):

```
#include "petsc.h"
#include "parpre.h"
int main(int Argc,char **Argv)
{
  Mat       A;
  PC        B;
```

Since the ParPre library is based on Petsc, we have to initialise Petsc, and create the data structures for the matrix and the preconditioner:

```
  PetscInitialize(&Argc,&Args,PETSC_NULL,PETSC_NULL);
  ierr = MatCreateMPIAIJ
         (MPI_COMM_WORLD,PETSC_DECIDE,PETSC_DECIDE,
          xdim*ydim,xdim*ydim,0,0,0,0,&A);
  ierr = PCCreate(MPI_COMM_WORLD,&the_pc);
```

The three main user routines are now to construct the matrix and preconditioner, and to use them in an iterative method:

```
  ierr = make_mat(A, [other arguments] );
  ierr = prec_setup(A,&B);
  ierr = cg(A,B, [other arguments] );
```

At the end of the program, the data structures have to be released again:

```
  MatDestroy(A);
  PCDestroy(B);
  PetscFinalize();
}
```

The matrix construction routine calculates or retrieves the elements of the coefficient matrix, and stores them in a Petsc version of the matrix. This doubling of the matrix storage is unfortunately unavoidable, but see section 3.3 for details.

```
int make_mat(Mat A, [other arguments] )
{
  for ( i=0; i<m; i++ ) {
    for ( j=jlo; j<jhi; j++ ) {
      /* compute or retrieve element value */
      ierr = MatSetValues(A,1,&I,1,&I,&val,INSERT_VALUES);
    }
  }
```

In the conjugate gradient method, vectors are allocated normally as `double*`; the Petsc vectors subsequently created from them are basically wrappers for these pointers; see section 3.2.

```
int cg(Mat A, PC B, [other arguments] )
{
  double *z,*r; /* allocatable vectors */
  Vec r_vec,z_vec; /* Petsc equivalents */

  ierr = VecCreateMPIWithArray(MPI_COMM_WORLD,lsize,r,&r_vec);
  ierr = VecCreateMPIWithArray(MPI_COMM_WORLD,lsize,z,&z_vec);
```

Application of the preconditioner is a simple call in the iterative loop (see section 2.3):

```
  for (it=1; rho/rho_0>tolerance; it++) {
    /* solve Mz = r */
    ierr = PCApply(B,r_vec,z_vec);
    /* other calls in the iterative method */
  }
}
```

Creating the preconditioner involves several standard calls, and possibly a number of method-specific auxiliaries; see section 2.2. The following is an example setup for the domain decomposition method:

```
int prec_setup(
  ierr = PCSetType(the_pc,PCDomainDecomp);
  {
    PC intface_pc;
    ierr = PCDomainDecompGetInterfacePC(the_pc,&intface_pc);
    ierr = PCSetType(intface_pc,PCJACOBI);
  }
  /* Declare local/global aspects */
  {
    PC local_pc;
    ierr = PCParallelGetLocalPC(the_pc,&local_pc);
    ierr = PCSetType(local_pc,PCLU);
  }
```

```
  ierr = ParPreSetup(MPI_COMM_WORLD,A,the_pc);
  return 0;
}
```

After this example, we will give detailed information on how to declare and construct preconditioners in the rest of this report.

2.2 Common Setup

The setup for all preconditioners in the ParPre package has some common elements.

Assume that the following variables have been declared:

```
#include "petsc.h"
#include "parpre.h"
#include "mat.h"
#include "pc.h"

  MPI_Comm comm;
  Mat A;
  PC the_pc;
  int ierr;
```

(The `Mat` type is a Petsc datastructure; the conversion of user structures to such a type is explained in section 3.)

The setup then takes the following steps[1].

In order to create the preconditioner, first the data structure has to be created:

```
  ierr = PCCreate(comm,&the_pc);
```

Next, the name of the method needs to be declared. This can be done by a direct call in the program, eg,

```
  PCSetType(the_pc,PCNONE);
```

or the type can be set by a runtime option[2], eg,

```
 mpirun -pctype none [other parameters and options]
```

In addition to the preconditioner type declaration some auxiliary setup calls may be required. The declaration ends with:

```
  ierr = ParPreSetup(comm,A,the_pc);
```

The available preconditioners and the applicable auxiliary functions are listed and explained in part II.

[1] The `CHKERRQ` macro is defined in Petsc; it catches error conditions and exits the surrounding routine with a nonzero exit code. After this example we will ignore the return code; a careful programmer should catch and check the return code, at least during the testing phase.

[2] Runtime options are explained in section A.

2.3 Preconditioner Application

The setup may differ per preconditioner, but all methods are applied in the same manner:

```
PC P;          /* created as described above */
Vec rhs,sol; /* declaration of vectors */
[... lots of code ...]
PCApply(P,rhs,sol);
```

(The `Vec` type is a Petsc datastructure; the conversion of user structures to such a type is explained in section 3.)

3 Interface to user data structures

3.1 Interface to Petsc data structures

The coefficient matrix used in the preconditioner setup (section 2.2), and the vectors involved in the preconditioner solve need to be in a suitable form. The vector can most likely be transformed in a single (essentially costless) instruction; for the matrix it takes some work, and in fact an extra copy of the matrix will be built[3].

3.2 Vector conversion

Under the assumption that the user application supplies a vector, declared on each processor as

```
int local_size; Scalar *vec_values;
```

then this vector can be made acceptable to the preconditioner by the following call:

```
Vec b; /* the resulting parallel vector in Petsc format */
MPI_Comm comm; /* the MPI context */
  ierr = VecCreateMPIWithArray
           (comm,local_size,vec_values,&b);
```

The user needs to specify the local `local_size`.

Since this operation is essentially a pointer redirection, its cost is negligible.

The vector structure needs to be de-allocated by

```
VecDestroy(b);
```

This does not de-allocate the array of values.

3.3 Matrix conversion

The Petsc structure for distributed matrices precludes such an easy solution for matrices as was possible for vectors. The only way to form a Petsc matrix out of a user-supplied matrix is to go through the following steps on every processor.

- Declare the matrix structure:

```
Mat mat; /* The Petsc matrix structure */
MatCreateMPIAIJ( <mpi context> ,
    n_local_rows, n_local_columns,
    n_global_rows,n_global_cols, 0,0,0,0, &mat);
```

[3]Since the Parpre library is built on top of Petsc primitives, a matrix already in Petsc format need not be transformed, of course.

The local sizes need to be specified, the global sizes can be left unspecified by substituting `PETSC_DECIDE`; see the Petsc manual for the meaning of the zero parameters.

- Declare the matrix elements. This is done through repeated calls to

  ```
  int m,n;
  int *row_idxs, *col_idxs;
  Scalar *v;
  MatSetValues(mat, m,row_idxs, n,col_idxs, v,INSERT_VALUES);
  ```

  This inserts an $m \times n$ dense block of values into the matrix structure. Choosing the values $m = n = 1$ will insert a single value.

  Matrix values can be declared by arbitrary processors. Eg, it is allowed to have one processor declare all values while other processors declare none. This has no relevance to the final distribution of the matrix, which had already been declared with the `Create` call above.

- Assembling the matrix.

  ```
  MatAssemblyBegin(mat,FINAL_ASSEMBLY);
  MatAssemblyEnd(mat,FINAL_ASSEMBLY);
  ```

  These calls will properly distribute values over the processors and set up communication structures for distributed matrix operations.

Part II
The Preconditioners

Available ParPre preconditioner types are, with their type and option listed:

PCAdditiveSchwarz with option `addschwarz`; section 4. This method and the next are based on overlapping domain decomposition.

PCMultiplicativeSchwarz with option `mulschwarz`; section 5.

PCGenBlockSSOR with option `genblockssor`; section 6. A generalisation of SSOR and ILU methods.

PCDomainDecomp with option `domdecomp`; section 7. A Schur complement method based on domain decompositioning with an interface system.

PCMultiLevel with option `multilevel`; section 8. A collection of multilevel incomplete factorisations and algebraic multigrid methods.

The use of these identifiers and options is explained in section 2.2.

4 Additive Schwarz

Type identifier: `PCAdditiveSchwarz`
usage: `PCSetType(the_pc,PCAdditiveSchwarz);`
Option: `addschwarz`
usage: `-pctype addschwarz`

See section 2.2 for more details on declaring the preconditioner by the above identifier or option.

4.1 General auxiliaries

- Local method. See section 9.1.

4.2 Specific auxiliaries

- Domain Overlap. The Schwarz methods, being based on overlapping subdomains, need an auxiliary routine for setting the overlap size.

  Command: `int PCSchwarzSetHaloSize(PC pc,int size)`
  Option: `-pc_halo_size <int>`

  For example:

  ```
  ierr = PCSchwarzSetHaloSize(the_pc,1);
  ```

  The overlap size has to be zero or more; for zero it reduces the additive method to block Jacobi and the multiplicative method to generalised block SSOR methods.

4.3 Example of usage of the Additive Schwarz method

The following is one example of the setup calls for the Additive Schwarz preconditioner. It is not exhaustive, and other possibilities do exist.

```
ierr = PCCreate(MPI_COMM_WORLD,&the_pc);
ierr = PCSetType(the_pc,PCAdditiveSchwarz);
/* parametrised setting of overlap */
ierr = PCSchwarzSetHaloSize(the_pc,1);
/* Declare local solve */
{
  PC local_pc;
  ierr = PCParallelGetLocalPC(the_pc,&local_pc);
```

```
    /* set the local solve to some LU solve */
    ierr = PCSetType(local_pc,PCLU);
}
ierr = ParPreSetup(MPI_COMM_WORLD,coeff_matrix,the_pc);
```

5 Multiplicative Schwarz

Type identifier: `PCMultiplicativeSchwarz`
usage: `PCSetType(the_pc,PCMultiplicativeSchwarz);`
Option: `mulschwarz`
usage: `-pctype mulschwarz`

See section 2.2 for more details on declaring the preconditioner by the above identifier or option.

5.1 General auxiliaries

- Local method. See section 9.1.
- Domain sequence. See section 9.2.

5.2 Specific auxiliaries

- Domain Overlap. See under section 4.

5.3 Example of usage of the Multiplicative Schwarz method

The following is one example of the setup calls for the Multiplicative Schwarz preconditioner. It is not exhaustive, and other possibilities do exist.

```
ierr = PCCreate(MPI_COMM_WORLD,&the_pc);
ierr = PCSetType(the_pc,PCMultiplicativeSchwarz);
/* parametrised setting over overlap */
ierr = PCSchwarzSetHaloSize(the_pc,1);
/* Declare global communication scheme */
ierr = PCParallelSubdomainPipelineSetType
(the_pc,PIPELINE_REDBLACK);
/* Declare local solve */
{
   PC local_pc;
   ierr = PCParallelGetLocalPC(the_pc,&local_pc);
   /* set the local solve to SSOR, 10 iterations */
   PCSetType(local_pc,PCSOR);
   PCSORSetSymmetric(local_pc, SOR_SYMMETRIC_SWEEP);
   PCSORSetIterations(local_pc,10);
}
ierr = ParPreSetup(MPI_COMM_WORLD,coeff_matrix,the_pc);
```

6 Generalised Block SSOR

Type identifier: `PCGenBlockSSOR`
usage: `PCSetType(the_pc,PCGenBlockSSOR);`
Option: `genblockssor`
usage: `-pctype genblockssor`

See section 2.2 for more details on declaring the preconditioner by the above identifier or option.

6.1 General auxiliaries

- Local method. See section 9.1.
- Domain sequence. See section 9.2.

6.2 Specific auxiliaries

- Global factorisation.

  Command: `int PCGenBlockSSORSetGlobalFactorisation(PC pc)`
  Option: `-pc_global_fac`

  Command: `int PCGenBlockSSORSetNoGlobalFactorisation(PC pc)`
  Option: `-pc_noglobal_fac`

  Determines whether on the block level the factorisation is of ILU (with global factorisation) or SSOR (without global factorisation) type. The default value is to have no global factorisation. This may considerably reduce the setup time; it also reduces the efficiency of the preconditioner.

6.3 Example of usage of the Generalised Block SSOR method

The following is one example of the setup calls for the Generalised Block SSOR preconditioner. It is not exhaustive, and other possibilities do exist.

```
ierr = PCCreate(MPI_COMM_WORLD,&the_pc);
ierr = PCSetType(the_pc,PCGenBlockSSOR);
/* Declare local solve */
{
  PC local_pc;
  ierr = PCParallelGetLocalPC(the_pc,&local_pc);
  ierr = PCSetType(local_pc,PCILU);
```

```
}
ierr = PCGenBlockSSORSetNoGlobalFactorisation(the_pc);
ierr = PCParallelSubdomainPipelineSetType
(the_pc,PIPELINE_SEQUENTIAL );
ierr = ParPreSetup(MPI_COMM_WORLD,coeff_matrix,the_pc);
```

7 Schur system Domain Decomposition

Type identifier: `PCDomainDecomp`
usage: `PCSetType(the_pc,PCDomainDecomp);`
Option: `domdecomp`
usage: `-pctype domdecomp`

See section 2.2 for more details on declaring the preconditioner by the above identifier or option.

7.1 General auxiliaries

- Local method. See section 9.1.

7.2 Specific auxiliaries

- Schur complement system. Routine for retrieving the interface preconditioner from the Domain Decomposition preconditioner:

  ```
  int PCDomainDecompGetInterfacePC(PC the_pc,PC *interface_pc)
  ```

  The interface preconditioner can be set through the options database: use any existing preconditioner option and prefix it with `interface_`.

7.3 Example of usage of the Schur system Domain Decomposition method

The following is one example of the setup calls for the Schur system Domain Decomposition preconditioner. It is not exhaustive, and other possibilities do exist.

```
ierr = PCCreate(MPI_COMM_WORLD,&the_pc);
ierr = PCSetType(the_pc,PCDomainDecomp);
{
  PC intface_pc;
  ierr = PCDomainDecompGetInterfacePC(the_pc,&intface_pc);
  ierr = PCSetType(intface_pc,PCJACOBI);
}
/* Declare local/global aspects */
{
  PC local_pc;
  ierr = PCParallelGetLocalPC(the_pc,&local_pc);
  ierr = PCSetType(local_pc,PCLU);
```

```
}
ierr = ParPreSetup(MPI_COMM_WORLD,coeff_matrix,the_pc);
```

8 Algebraic Multilevel

Type identifier: `PCMultiLevel`
usage: `PCSetType(the_pc,PCMultiLevel);`
Option: `multilevel`
usage: `-pctype multilevel`

See section 2.2 for more details on declaring the preconditioner by the above identifier or option.

8.1 General auxiliaries

- None.

8.2 Specific auxiliaries

- Fill Strategy. Routine:

  ```
  int AMLSetFillMethod(PC pc,AMLFillMethod fill_method)
  ```

  where the method parameter is one of the following:

  AMLFillNone no fill-in.

  AMLFillDiag allow fill-in on the diagonal.

  AMLFillStrong allow large fill-in elements; a drop tolerance method.

  AMLFillFull we accept the full fill matrix.

  See further section 10.1.

- Coarse Grid Choice.

  ```
  int AMLSetCoarseGridDependent(PC pc)
  int AMLSetCoarseGridIndependent(PC pc)
  ```

  See further section 10.2.

- Solution Scheme. Routine:

  ```
  int AMLSetSolutionScheme(PC pc,AMLSolveScheme scheme)
  ```

  where the scheme parameter is one of the following:

  AMLSolveILU a straightforward Gaussian factorisation solve.

AMLSolveMG same, but with pre and post smoother; see section 10.3.

AMLSolvePolynomial a generalised W-cycle solver. Not implemented yet.

- Smoothers. See section 10.3.

8.3 Example of usage of the Algebraic Multilevel method

The following is one example of the setup calls for the Algebraic Multilevel preconditioner. It is not exhaustive, and other possibilities do exist.

```
ierr = PCCreate(MPI_COMM_WORLD,&the_pc);
ierr = PCSetType(the_pc,PCMultiLevel);
ierr = AMLSetSolutionScheme(the_pc,AMLSolveMG);
ierr = AMLSetSmootherChoice(the_pc,AMLPrePostSmooth);
ierr = AMLSetCoarseGridIndependent(the_pc);
ierr = AMLSetFillMethod(the_pc,AMLFillFull);
{
   PC local_pc;
   /* last level solver */
   ierr = AMLSetCutoffSize(the_pc,30);
   ierr = PCMultiLevelGetLastLevelSolver(the_pc,&local_pc);
   ierr = PCSetType(local_pc,PCSOR);
   ierr = PCSORSetSymmetric(local_pc,SOR_LOCAL_SYMMETRIC_SWEEP);
   ierr = PCSORSetIterations(local_pc,4);
}
ierr = ParPreSetup(MPI_COMM_WORLD,coeff_matrix,the_pc);
```

9 Auxiliary functions

9.1 Local Solves

All ParPre methods (with the exception of the multilevel methods) feature a local, per processor, solve component for the subdomains. This method is accessible by the following routine:

```
int PCParallelGetLocalPC(PC the_pc,PC *local_pc)
```

Any existing call defined for the `PC` type can then be applied to the `local_pc`. No call to `PCSetup` is needed for the local method.

Properties of the local method can be set through the options database by taking any existing option and prefixing it with `sub_`.

Example

The user can determine what method to take here by specifying it as follows:

```
PC the_pc; /* the parallel preconditioner */
{
  PC local_pc; /* a pointer to the local solver */
  ierr = PCParallelGetLocalPC(the_pc,&local_pc);
  /* example: set the local solve to full LU solve */
  ierr = PCSetType(local_pc,PCLU);
}
```

The possible choices for the second argument are now all of the non-parallel preconditioners available in the Petsc library; see below. The default is an identity preconditioner.

The local solve type could have been set by the following option:

```
-sub_pc_type lu
```

The subdomain solvers may need additional parameters, eg the type of sweep in the case of a `PCSOR` solver. This can be done using the normal functions on the `local_pc`:

```
PCSORSetSymmetric(local_pc,SOR_SYMMETRIC_SWEEP);
```

See the Petsc manual for the functions and values available.

This is the list of the local methods available:

- `PCNONE` Do a simple copy instead of a solve; equivalently, use the identity matrix as solver.
- `PCJACOBI` Use the diagonal of the matrix as solver.
- `PCSOR` Use a triangular part of the matrix; see the Petsc manual for further options.

PCILU Use an incomplete factorisation as solver.

PCEISENSTAT The Eisenstat variant of $ILU(0)$.

PCICC Use the Cholesky incomplete factorisation; this presupposes a symmetric matrix.

with corresponding option values `none`, `jacobi`, `sor`, `bjacobi`, `eisenstat`, `ilu`, `icc`.

9.2 Subdomain sequence

Some parallel preconditioners, e.g., Alternating Schwarz and Generalised Block SSOR, involve a certain amount of sequentiality in that the subdomains are handled in some order. The way the data flows through the processors is called 'pipeline'. (This is analogous to wavefronts in sparse direct methods.)

The data pipeline can be declared by the user with the following routine or option:

Command: `int PCParallelSubdomainPipelineSetType(PC pc,PipelineType pipeline_type)`
Option: `-pc_pipeline <pipeline_type`

The available choices for the `pipeline_type` are:

- Simultaneous processing:

  Argument: `PIPELINE_NONE`
  Option value: `none`

  All processor act on their subdomains simultaneously. This reduces the GBSSOR method 6 to block Jacobi, and multiplicative Schwarz 5 to additive Schwarz 4.

- Sequential ordering:

  Argument: `PIPELINE_SEQUENTIAL`
  Option value: `sequential`

  All processors are ordered sequentially according to their ranking in the MPI context. This does not necessarily mean that they will not work in parallel: eg, consider processors ordered in a grid structure.

- Red-black ordering:

  Argument: `PIPELINE_REDBLACK`
  Option value: `redblack`

  A red-black ordering is used, with processors coloured according to whether their number is odd or even. Note that this may not reflect the processor connectivity.

- Multi-colour ordering:

  Argument: `PIPELINE_MULTICOLOUR`
  Option value: `multicolour`

  A multi-colouring is applied to the processors such that processors of the same colour are not connected in the matrix graph.

10 Multilevel method parameters and auxiliary functions

10.1 Multilevel fill strategy

In the elimination of a level we may allow for a certain amount of fill-in. Without any fill-in or with fill-in only on the diagonal, the original sparsity structure of the matrix is preserved; with other fill strategies the matrix on subsequent levels may become more and more dense.

These are the fill strategies defined:

AMLFillNone no fill-in, this results in an SSOR type factorisation. Basically, the factorisation is only concerned with identifying the level sets.

AMLFillDiag allow fill-in on the diagonal. This gives an $ILU(0)$-type factorisation.

AMLFillStrong allow large fill-in elements. This is in effect a drop tolerance method. The tolerance is out of user control for the time being.

AMLFillFull we accept the full fill matrix.

10.2 Multilevel coarse grid selection strategy

We use the Jones-Plassman [8] strategy for finding, in parallel, an independent set in the matrix graph. We are then faced with the choice of letting this independent set function as the coarse or as the fine grid. The former is commonly encountered in Algebraic Multigrid methods [9], the latter is the preferred strategy for multi-colour methods.

For this choice, the user has the following two routines:

```
int AMLSetCoarseGridDependent(PC pc)
int AMLSetCoarseGridIndependent(PC pc)
```

It should be noted that choosing the coarse grid as the independent set gives a dramatically lower number of levels than letting it be the dependent set.

10.3 Multilevel methods: smoothers

If the multilevel solution scheme has been declared as `AMLSolveMG`, the method will use a pre and post smoother. Any parallel Petsc preconditioner can be used for this.

For technical reasons, the smoothers can only be set through the options database (see section A): take any preconditioner option and prefix it by `presmoother_` or `postsmoother_`.

If the matrix is symmetrical, it is advisable to let the pre and post smoothers be each other's transpose, though for well-conditioned matrices there is some tolerance for a degree of unsymmetry.

Part III

Appendices

A Use of the command line options database

Various settings in ParPre (and in fact in the underlying Petsc library) can be set by a function call in the program, eg `PCSetType(the_pc,PCNONE);`, or by using command line options.

Command line options can be set in any of the following ways:

- really on the command line:

```
mpirun a.out -pc_type none
```

- by setting the options, in the environment variable `PETSC_OPTIONS` or in the `.petscrc` file or any other file which can then be passed through `PetscInitialize`;
- in the code:

```
    OptionsSetValue("-pc_type","ilu");
```

Sometimes command line options are formed from the basic options, eg to set the preconditioner type for subdomain solvers we use an option `-sub_pc_type none`.

See the Petsc manual for more information about setting options.

B Relationship to Petsc

The ParPre library uses Petsc routines for basic matrix and vector manipulation. Thus you need to have the Petsc library [5, 6] installed.

B.1 Petsc installation

Some of the Petsc routines have been changed for use in ParPre. If you use Petsc as an independent package, you need not be aware of this, since the changes affect internal routines. No calling sequences of user routines were changed. It is necessary to install Petsc *before* ParPre, since the ParPre installation adds routines to some Petsc libraries.

The Petsc library home page is

`http://www.mcs.anl.gov/petsc/petsc.html`

and the latest version of the library can be downloaded from there.

C Installation

C.1 Setup for library creation

In order to install the ParPre library, the MPI and Petsc libraries have to be in place. You also have to have write permission for the Petsc library files.

The following environment variables need to be set, either globally, or at the invocation of the makefile:

```
PETSC_DIR <location of the Petsc library>
PETSC_ARCH <machine type; check $PETSC_DIR/bmake for possibilities>
PARPRE_DIR <location of the ParPre library>
BOPT <optimisation type; eg, g for debug>
```

Further environment has to be specified by copying Petsc information to Parpre:

```
cp $PETSC_DIR/bmake/$PARPRE_ARCH/base.site \
   $PARPRE_DIR/bmake/$PARPRE_ARCH/base.site
```

Then:

```
cd $PARPRE_DIR ; make all
```

C.2 Setup for user programs

By far the easiest way to specify all flags and include paths for user programs is to have the following statement in the makefile:

```
include $(PARPRE_DIR)/bmake/$(PARPRE_ARCH)/base
```

You can then specify the flags as

```
CFLAGS        =  $(PARPRE_INCLUDE) $(PETSC_INCLUDE) $(CONF) $(COPT)
```

This presupposes that your system's setup of 'make' uses the `CFLAGS` macro: type `make -p` and you'll probably see the following lines:

```
.c.o:
        $(CC)  -c $(CFLAGS) $(BASEOPT) $*.c
```

or, in the ParPre directory:

```
.c.o:
        $(CC) -c $(CFLAGS) $(COPTFLAGS) $<
```

If not, adjust accordingly.

The linker statement has to include `PARPRE_LIB`, eg

```
$(CLINKER)  -o test prog.o $(PARPRE_LIB)
```

This macro is defined in the included base file mentioned above.

D Revision history of this manual

D.1 Version 2.0.17

First full release.

D.1.1 Revision 1

- Options corrected for Schwarz methods and multilevel; 4, 5 and 8.
- Global factorisation added for GBSSOR methods; 6.
- Subdomain sequence calls and parameters changed; 9.2.
- Options added for GBSSOR global factorisation; 6.

D.2 Version 2.0.21

Internals adapted to new Petsc version.

D.2.1 Revision 1

Fixed 'pipeline none' in GBSSOR.

References

[1] O. Axelsson. *Iterative Solution Methods.* Cambridge University Press, Cambridge, New York, Melbourne, 1994.

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

[3] Tony Chan and Victor Eijkhout. Design of a library of parallel preconditioners. Technical Report UCLA CAM report 97-58, University of California, Los Angeles, 1997.

[4] Tony F. Chan and Tarek P. Mathew. Domain decomposition algorithms. In *Acta Numerica 1994*, pages 61–143. Cambridge University Press, 1994.

[5] W. D. Gropp and B. F. Smith. Scalable, extensible, and portable numerical libraries. In *Proceedings of the Scalable Parallel Libraries Conference, IEEE 1994*, pages 87–93.

[6] W. D. Gropp and B. F. Smith. The design of data-structure-neutral libraries for the iterative solution of sparse linear systems. *Scientific Programming*, 5:329–336, 1996.

[7] Wolfgang Hackbusch. *Iterative Solution of Large Sparse Systems of Equations.* Springer Verlag, New York, 1994.

[8] M.T. Jones and P.E. Plassmann. Parallel solution of unstructed, sparse systems of linear equations. In R.F. Sincovec, D.E. Keyes, M.R. Leuze, L.R. Petzold, and D.A. Reed, editors, *Proceedings of the Sixth SIAM conference on Parallel Processing for Scientific Computing*, pages 471–475, Philadelphia. SIAM.

[9] J.W. Ruge and K. Stüben. Algebraic multigrid. In Stephen F. McCormick, editor, *Multigrid Methods.* SIAM, 1987. chapter 4.

[10] Youcef Saad. *Iterative Methods for Sparse Linear Systems.* PWS Publishing Company, Boston, 1996.

[11] Barry Smith, Petter Bjorstad, and William Gropp. *Domain Decomposition, Parallel Multilevel Methods for Elliptic Partial Differential Equations.* Cambridge University Press, 1996.