DISTRIBUTED ITERATIVE LINEAR SYSTEM SOLVERS USING PVM*
USER MANUAL
VERSION 0.5, JULY 1994

VICTOR EIJKHOUT, EIJKHOUT@CS.UTK.EDU

* This work was supported by ARPA under contract number DAAL03-91-C-0047

0.1 Introduction

This manual describes a library of iterative solutions methods for linear systems of equations, on distributed processors connected through the PVM communication library[1]. The software is distributed with a stand-alone demo program, controllable through an X-windows console.

Disclaimer This software is still under development. If you have any questions, bug reports, suggestions, feel free to contact me. Especially suggestions about extensions that you think are particuarly useful are welcome.

1. A very similar package based on PICL for hypercube processors exists; write to me if you are interested.

Contents

Chapter 1

Library routines

This library contains code for a number of conjugate gradient type iterative methods and preconditioners for them. Specification of the method and preconditioner is slightly more complicated than you may have seen in other libraries from the distributed nature of these routines. Thus, a minimal program needs to call the following library routines:

```
      call init_pvm_instances(buffer,buf_len)
```

to initialize the pvm structure of the executables;

```
      call set_problem_format(buffer,buf_len)
```

to declare the storage mode of the matrix; some call to specify local variables, for instance

```
      call cprs_set_my_vars(vec_inf,leng_vec_inf,
     >     long_buffer,nitems)
```

for row/column compressed storage;

```
      call process_matrix(matrix,
     >     demo_var2proc,
     >     vec_inf,leng_vec_inf,
     >     mat_ptr,leng_mat_ptr, mat_idx, mat_con,leng_mat_con)
```

to preprocess the matrix for establishing the communication structure; a call for creating the preconditioner (even needed for unpreconditioned methods)

```
      call creat_prec_NAME(
     >     preconditioner,prec_inf,leng_prec_inf,
     >     matrix,mat_ptr,mat_idx,
     >     mat_con,leng_mat_con,
     >     vec_inf,leng_vec_inf)
```

with `NAME` replaced by a valid preconditioner name; and finally a call

```
      call METHOD_NAME
     >     (matrix,
     >     mat_ptr,leng_mat_ptr, mat_idx,leng_mat_idx,
     >     mat_con,leng_mat_con,
     >     preconditioner,prec_inf,
     >     sol,rhs, vec_inf,leng_vec_inf,
     >     success,precision,grad_hist,lda_gradhist,nhist,
     >     max_it,stop_test,
     >     tot_time,tot_flop[,EXTRA PARMS])
```

to a specific iterative method.

1.1 Program description; initialization part

In the example main program `node.f` you'll find various initializations. Some of these are only necessary for the demo code (described in a later chapter of this manual), others are required for any valid program. The initializations are described in the following sections.

1.1.1 Configuration setup: Input/output (optional)

The library may perform some input/output. For instance, in every iteration the error norm may be printed, or, in the demo version various parameters or matrices may be read from file. Also, a log file is kept, and runtime errors are written to error files.

Most input and output is only for the demo code, and for the rest it uses the standard unit numbers. Hence the subroutine call to change them is optional, and in general not necessary.

The FORTRAN channels used are in sequence, and with default value:

input (5) for input from file.
output (6) for standard out.
error (1) this is where errors get reported.
dump (7) for trace dumps.
bulk (11) for dumping/restoring stuff (not used currently).
solution (8) for dumping the final solution to file.
log (9) this writes the file 'log.dat'.
temp (10) for temporary business.

They can be set as follows:

```
      call set_file_channels(5,6,1,7,11,8,9,10)
```

1.1.2 Configuration setup: PVM information (required)

The information how many processes there are, and what their pvm instance numbers are, has to be passed.

```
call init_pvm_instances(buffer,buf_len)
```

The buffer contains the instance numbers of all processes participating[1], and their number is passed as the length parameter.

The demo code uses an additional call

```
call demo_init_pvm(host,reader)
```

to declare the instance numbers of the host code and a reader node, if any (negative number if there is no reader).

1.1.3 Configuration setup: processor connectivity (optional)

For grid-based problems it can be advantageous if the processors are themselves ordered in a grid. Even if they are not, having them communicate that way may speed up execution. The dimensions of this processor grid (or parallelepiped) can be indicated with an optional call to

```
call init_processor_grid(buffer,buf_len)
```

where the buffer contains

1. the number of space dimensions of the processor structure; at the moment only a two-dimensional grid is supported.

2, ... the number of processors along each space dimension.

The parameter `buf_len` is the length of the buffer; it has to be at least `1+buffer(1)`.

This initialization call is optional: a default structure of all processors in a linear array is automatically created.

1.1.4 Trace modes setup (optional)

The program has a number of trace modes to display various types of information. The default behaviour is to generate no trace output. Most debugging options are only for debugging. If you want to know more about them than is described here, contact me.

If you switch on tracing options, the output can go to either screen or file. If you decide to send it to file, every process makes its own `dump1234` file in the local directory (section 4.2.1) where the number is the logical process number, not the pvm process id. The files therefore start at `dump0000`.

The following sequence sets the default values

1. Not counting the host and reader programs of the demo code.

```
call default_trace_modes(buffer,buf_len)
call init_trace_modes(buffer,buf_len)
```

but calling it is optional.

These are the contents of the buffer; for most options zero means no, positive values have various meanings of yes:

1 Generate trace data (only implemented for the Picl version of the routines).

2 Trace all messages sent and received: their types, sender, receiver, and the values of items sent.

3 Trace iteration parameters:
   1 display the gradient norm
   2 display various scalars generated during the iteration proceses
   3 output all vectors generated during the iteration process; this should only be done while output goes to a dump file.

4 post mortem statistics: a value of 2 pops up an xgraph plot of the gradient history, 3 dumps this as a matlab file. This only happens in the demo code, since it relies on some script files.

5 progress: a nonzero value will let the code give short messages reporting what stage it is in.

6 trace to screen: $0 =$ everyone, $1 =$ master node, > 1 means all to file

7 keep a log file, reporting on input parameters and results.

8 dump solution; only for the demo code

9 dump matrices

10 trace allocation of temporary arrays in the iterative method.

11 trace the opening and closing of input and output files.

12 trace the processing of local matrices to determine traffic patterns between the processors.

13 test the dynamically allocated memory for integrity:
   0 don't test
   1 test once every iteration
   2 test after everything that can possibly corrupt memory

The buffer length has to be at least 13.

Displaying dumped vectors

Although the output of some trace modes may be a bit esoteric, one mode is interesting enough for the general user, namely dumping vectors generated during the iteration process. Pushing the `to Dump File` and `Trace level: vectors` buttons in the interface, or giving settings as described above, causes the program to dump vectors generated in the iteration process to the `dump9999` files. You can assemble these distributed vectors by going into the local directory and entering the command `make vectors`. This will extract the vectors and put them in separate files you can then inspect. The file names consist of a few characters describing the vector, followed by the iteration number in which they were generated.

The extracting is done by two utilities (`asmbl_grid` and `asmbl_cprs`) in the utilities directory, that are made on demand.

1.1.5 Problem format setup (required)

The storage method for the coefficient matrix, and similar facts, have to be declared:

```
call set_problem_format(buffer,buf_len)
```

The following locations in the integer array `buffer` contain parameters:

1. the matrix format; this can be
   1. if the matrix is grid-based, that is, using stencil storage (section 2.1).
   2. if the matrix is stored by diagonals. In this case every processor is supposed to own a consecutive stretch of the problem's variables (section 1.1.6).
   3. if the matrix is stored in compressed row or column format.
2. a further specification of the format;
   - if the format is grid-based, this is the dimension of the grid; currently only 2 is supported.
   - if the format is compressed, this can be
     1. for compressed by rows
     2. for compressed by columns
     3. for symmetric compressed by rows and columns; the matrix storage can be interpreted as either an upper triangle stored by rows, or a lower triangle by columns.
   - if the storage format is by diagonals, this buffer location is ignored.
3. this determines whether the sparsity pattern of the matrix (not the matrix itself!) is symmetric; 0 for symmetric, 1 for non-symmetric. This option is only for the row/column compressed case.
4. this determines the type of partitioning of the variables over the processors (for compressed diagonal and compressed row/column format only):
   1. every processor has as consecutive stretch of owned variables, and every processor knows what variables every other processor owns.
   2. (only for compressed row/column storage) there is an integer function of one variable that, given a global variable number, returns the processor that owns that variable; this function will be passed to a library routine lateron (see section 1.2.1).
   3. the mapping of variables to processors is unknown, and each processor knows only the variables it owns. In the matrix preprocessing

stage (section 1.2.1) the processors will then communicate to exchange the desired information.

The following parameters are only for the demo. By specifying a buffer length of 4 you will have them ignored. However, you can use them if you want to adapt the demo code to your own purposes.

5 the right hand side: 1=user-supplied, 2=identically one.
6 matrix source: 1=internally generated, 2=read from file.

If the matrix is read from file, its name is declared by

```
call set_demo_matrix_file(cbuffer)
```

where `cbuffer` is declared as

```
character*3 cbuffer
```

If its value is `'xyz'`, the demo code will read any matrix from the file `xyz.mat` in the input directory (section 4.2.1).

The demo code reads parameters by a call to

```
call demo_read_problem_format(buffer,buf_len,cbuffer)
```

1.1.6 The variable map (required)

In order for the processors to be able to establish their communication patterns, they have to have some knowledge of what variables reside on what other processors.

For grid-based matrices, this is explained in section 2.1.

For the compressed diagonal and compressed row/column storage cases there are various options. Minimally, each processor has to know what variables are owned. From this knowledge the processors can (by all-to-all communication) figure out their interconnection patterns.

However, if more information is available, processors can establish communications patterns more efficiently.

Consecutive owned variables

If every processor owns a consecutive stretch of variables in the global numbering (this is specified in the call to `set_problem_format`; see section 1.1.5), the variables of a processor can be described by the number of the first and the length of the stretch. Minimally, if the partitioning is otherwise unknown, a processor needs to declare its own variables:

```
      call diag_set_owned_stretch
     >     (vec_inf,leng_vec_inf,first_var_of_proc,n_vars_of_proc)
```

If the variables of the other processors are known, a loop has to be made over all processors:

```
      do 10 iproc=0,no_processors()-1
         call set_consec_of_proc
     >        (first_var_of_proc,n_vars_of_proc,iproc,
     >        vec_inf,leng_vec_inf)
 10   continue
```

The integer function `no_processors` is available to the user.

Variable to processor function

If the user is able to write a function that will specify the variable to processor mapping, this can be passed to the matrix preprocessing routine; see below. However, we still require each processor to declare its own variables, consecutively in the compressed diagonal case (above), or enumerated in the compressed row/column case (below).

Enumeration of owned variables

In the compressed row/column case, the owned variables can have completely arbitrary numbers, so if the partitioning has not been declared as consecutive, each processor has to declare an explicit enumeration of its variables:

```
      call cprs_set_my_vars(vec_inf,leng_vec_inf,
     >     long_buffer,nitems)
```

where the number of owned variables is `nitems`, and their (global) numbers are stored in `long_buffer`.

1.2 Main routines

This section describes the routines for preprocessing the matrix, creating the preconditioner, and invoking various iterative methods.

1.2.1 Preprocessing the matrix

In a parallel program, processors have to know how to communicate. This information is arrived at by inspection of the matrix structure, in a call

```
      call process_matrix(matrix,
     >     demo_var2proc,
     >     vec_inf,leng_vec_inf,
     >     mat_ptr,leng_mat_ptr, mat_idx, mat_con,leng_mat_con)
```

which also allocates the buffer space for communication. This has to be done prior to creating the preconditioner or calling the iterative method routine. The function argument can be any dummy identifier; it is only used if the variable-to-processor mapping is specified as known by such a function in the call to `set_problem_format`; see section 1.1.5.

In processing the matrix, the following arrays may be affected:

`vec_inf` The separation between owned and non-owned local variables (see section 2.5.1) is made, and the mapping of local numbers to global numbers is written into this array.

`matrix` This array contains the matrix elements. If the matrix is in compressed row/column format, on exit the elements per row or column are sorted according to increasing local variable numbers; see the next point and section 2.5.2.

`mat_idx` For row or column compressed storage, this array contains row or column indexes on entry, expressed in terms of global variable numbers. On exit, these numbers have changed to local variables numbers (see section 2.5.2).
For diagonal storage this array is empty on entry, on exit it contains information relating to the diagonal offsets; see section 2.3.

`mat_con` This array contains on exit information about how processors have to communicate during the matrix-vector product and possibly during the preconditioner solve.

For a description of the matrix storage, see section 2.

If a number of systems are solved with different matrices but of the same sparsity structure, this preprocessing need not be repeated, and the integer arrays can simply be reused.

1.2.2 Preconditioner creation

The different available preconditioners all need one real and one integer array. These arrays are filled in by a call

```
      call creat_prec_NAME(
     >     preconditioner,prec_inf,leng_prec_inf,
     >     matrix,mat_ptr,mat_idx,
     >     mat_con,leng_mat_con,
     >     vec_inf,leng_vec_inf)
```

where `NAME` is a name of one of the available preconditioners (see below). After such a call, the arrays `preconditioner` and `prec_inf` contain all necessary information, and they can be passed to the iterative method routine (section 1.2.3) without any further indication what the preconditioner is.

The available preconditioners are

none No preconditioner is used. Note that the call to `create_preconditioner_none` has to be made nonetheless.

full_jacobi A simple Jacobi preconditioner is used.

ssor_jacobi A block Jacobi preconditioner to let all processors operate fully in parallel, but the processors locally apply an SSOR preconditioner to their local variables.

ilu_jacobi A block Jacobi preconditioner to let all processors operate fully in parallel, but the processors locally apply an ILU preconditioner to their local variables.

full_ssor A sequential sweep over all domain variables is made, performing a true SSOR preconditioner. This is sequential in nature.

full_ssor A sequential sweep over all domain variables is made, performing a true ILU preconditioner. This is sequential in nature.

par_ssor A multi-colour (see below) preconditioner is applied, where all processors locally compute (in parallel) an SSOR preconditioner on their local variables.

par_ilu A multi-colour (see below) preconditioner is applied, where all processors locally compute (in parallel) an ILU preconditioner on their local variables.

All factorization type preconditioners are of the $ILU(0)$ type, that is, at present all fill-in is discarded[2]. We plan to add more elaborate preconditioners later.

The parallel methods have not been implemented for the symmetric row/column storage formats (section 2.4).

The matrix is supposed to be already in the array `matrix`, with its description in `mat_inf`, `mat_idx`, `mat_con`, that is, the preconditioner can only be created after a call to `process_matrix`.

The following parameters are affected:

preconditioner A real array of length the number of variables. On exit it contains either the inverses of the matrix diagonal elements (for Jacobi or SSOR preconditioners), or the inverses of pivots from an incomplete factorization.

prec_inf An integer array, containing only a few information items. Its length has to be at least 10.

vec_inf An integer array. This array has already been partly written by `process_cprs_matrix`. If the preconditioner involves a multi-colour ordering, this array records the colours of the variables. Its length has to be enough to accomodate these data for the local, that is, owned and bordering, variables.

The other parameters are:

2. More precisely, the preconditioner is of the *D-ILU* type (see [4]) where not only is fill discarded, but also nonzero off-diagonal elements are never altered.

`leng_prec_inf` An integer indicating the length of the `prec_inf` array. It has to be at least 10. If it is insufficient for storing the preconditioner under construction, the program terminates.

`leng_vec_inf` An integer indicating the length of the `vec_inf` array. (See the point about `vec_inf` above.) If this is insufficient for storing the multi-colour information, the program terminates.

`matrix` A real array storing the matrix.

`mat_ptr` An integer array storing pointers into the `mat_idx` and `matrix` array.

`mat_idx` An integer array storing row or colums indexes of the elements of the `matrix` array. These indexes are supposed to have been localized by a call to `process_matrix`.

`mat_con` An integer array storing connectivity information of the matrix, as output by a call to `process_cprs_matrix`.

1.2.3 Iterative methods routines

The routines for iterative methods conform to the following scheme:

```
 call METHOD_NAME
>     (matrix,
>     mat_ptr,leng_mat_ptr, mat_idx,leng_mat_idx,
>     mat_con,leng_mat_con,
>     preconditioner,prec_inf,
>     sol,rhs, vec_inf,leng_vec_inf,
>     success,precision,grad_hist,lda_gradhist,nhist,
>     max_it,stop_test,
>     tot_time,tot_flop[,EXTRA PARMS])
```

where `METHOD_NAME` is `cg` for Conjugate Gradients, `cgne` for Conjugate Gradients on the Normal Equation, `nscg` for Non-Symmetric Conjugate Gradients (equivalent to Orthomin), `gmres` for Generalized Minimal Residual, `bicg` for BiConjugate Gradients, `cgs` for Conjugate Gradients Squared, or `bcgs` for BiConjugate Gradients Stabilized.

The matrix and preconditioner parameters have been described above, the other parameters are:

`sol` On entry, the starting guess; on exit the last computed iterate. This vector contains only values for the variables owned by this processor, that is, the starting vector has already been distributed on entry, and the final solution has not been collected on exit.

`rhs` The right-hand side vector. This parameter is not affected by the routine. Like `sol`, it is a distributed vector.

`success` On exit: 0 for successful convergence, 1 for no convergence to the specified precision within the specified number of iterations.

`precision` On entry, the required reduction in the residual norm (see section 1.2.4 below); on exit, the norm of the final residual.

`max_it` On entry, the maximum allowed number of iterations; on exit, the number of iterations actually performed.

`stop_test` The type of stopping test to be used: 1=absolute size of residual, 2=reduction of residual, 3=reduction with respect to norm of the right hand size, 4=attempt to reduce true error (this is criterion 1 in [1, pg. 54]).

`time` The elapsed time over the iteration loop, measured by a system call to `getrusage`.

`flops` Number of flops performed.

`grad_hist` A real array of dimension

```
dimension grad_hist(lda_grad_hist,nhist)
```

in which the norms of the successive gradients will be reported if `nhist.gt.0`. It is required that `lda_grad_hist.geq.max_it` on entry. If `nhist.eq.2` and the method is GMRES or QMR, the norms of gradients of the underlying method (Orthomin and BiCG respectively) will be reported in `grad_hist(*,2)`.

The `EXTRA_PARMS` is empty, except for `nscg` and `gmres`, when it is `trunc`, the truncation length.

The matrix (see section 2) for a description of matrix storage) is assumed to have been preprocessed for parallelism as described in section 1.2.1.

1.2.4 Stopping test

The currently implemented stopping tests test the norm of the current residual; depending on the `stop_test` parameter to the iterative methods call this norm is measured against

1 an absolute size:

$$||r|| \leq \epsilon;$$

2 the initial residual:

$$||r|| \leq \epsilon ||r_0||;$$

3 the right hand side:

$$||r|| \leq \epsilon ||b||;$$

4 a dynamic quantity

$$||r|| \leq \epsilon(||A||\,||x|| + ||b||),$$

where the norm of the matrix A is estimated at the start of the iterative method, and the norm of the iterate x is computed every iteration.

See [1, sec. 4.2] for a discussion of these stopping tests.

Chapter 2

Storage formats

This section describes the matrix storage schemes supported in the library. All storage schemes are strictly distributed, that is, there is a non-overlapping division of variables over the processors, and each processor never has any part of the matrix but that corrresponding to its own variables.

2.1 Grid storage

There is support for matrices that can be described as a stencil on a 2-dimensional grid (extensions to higher dimensions will be implemented later). Such matrices are descibed by the following parameters:

- the size of the domain,
- the number of legs of the stencil, and their offsets wrt to the central node,
- the part of the domain owned by each processor.

These quantities are declared as follows. First of all, the fact that the matrix is of the 2D grid/stencil type has already been declared; see section 1.1.5. The sizes of the domain are set by

```
      call set_global_domain(vec_inf,
     >      buffer,nitems,leng_vec_inf)
```

where `nitems` is the number of space dimensions of the grid (only 2 is supported at the moment), and the buffer contains in its first `nitems` locations the sizes of the grid along the subsequent space dimensions.

After declaration of the global grid, the following call declares how big a slice each processor has.

```
      call set_domain_slice(vec_inf,slice)
```

This writes into `vec_inf`, and the array `slice` is declared

```
      dimension slice(2,*)
```

It contains in the (1,i) and (2,i) positions the highest and lowest coordinates in the i-th coordinate direction respectively.

Declaration of the legs of the matrix stencil is as follows:

```
      call set_grid_offsets(mat_ptr,grid_offsets,
     >        ndim,ndiags,vec_inf,need)
```

The array mat_ptr is written with the offsets contained in the array

```
      dimension offsets(ndim,ndiags)
```

where ndim is the number of space dimensions, and ndiags is the number of legs of the stencil. For example, in two space dimensions, if $(2,-1)$ is offsets(*,3) then the third component of the stencil is the coefficient by which $u_{i+2,j-1}$ is multiplied if the stencil is applied at to a vector u at coordinate (i,j).

2.2 Row or column compressed storage

Compressed row or column matrix storage comprises one real and three integer arrays. This is the compressed row/column storage (Harwell/Boeing) format as described in [2, Ch. 2], with some extensions for more efficient parallel processing.

matrix A real array, containing the rows or columns of the matrix, depending on the storage scheme used. Its length is equal to the number of nonzeros in the part of the matrix owned by this processor.

mat_idx An integer array, containing the column or row indexes for compressed row or column storage respectively. Its length is equal to the number of nonzeros in the part of the matrix owned by this processor.
The indexes in this array initially refer to the global numbering; the call to process_matrix changes this to refer to the local numbering (see section 1.2.1).

mat_ptr An integer array, containing first of all the pointers to where the rows or colums start in both matrix and mat_idx in compressed column and row storage respectively. Secondly, this array stores pointers needed for parallel processing of the matrix. Its length has to be at least $4N+1$ where N is the number of variables owned by this processor.

mat_con An integer array, containing information about how processors are to exchange information for the matrix vector product and the preconditioner solve. Its size is unfortunately not easily predictable.
This array contains lists of all processors that will be sending to this one, and that will be receiving from it. For each processors it has the (local) numbers of the variables that are involved: for a receiving processors it has the numbers of the variables that have to be sent; for a sending processor the variables that will be sent. An estimate for the storage required would

be three times the number of border variables of this processor, but this may not be enough.

2.3 Compressed diagonal storage

The compressed diagonal storage format implemented here is based on a block-row decomposition of the global matrix, where every processor owns a consecutive stretch of variables. Thus the owned variables of processor are fully described by the first variable and the number of variables. The matrix as passed to the `process_matrix` routine resides in a single array defined as

```
dimension matrix(nvars,ndiag)
```

where `nvars` is the number of variables owned by this particular processor, and `ndiag` is the number of diagonals in the matrix. The location of the diagonals in the matrix is stored in an integer array

```
dimension mat_ptr(1+ndiag)
```

where `mat_ptr(1)` = `ndiag`, and `mat_ptr(1+dia)` contains the offset of diagonal `dia` with respect to the main diagonal. Thus, `matrix(i,j)` contains matrix element $(i, i+$`mat_ptr`$(1+j))$ of the local matrix, that is, $(i+$`first`$, i+$`first`$+$`mat_ptr`$(1+j))$ of the global matrix, where `first` is the number of the first owned variable of this processor.

The call to `process_matrix` writes information in the first `ndiag` elements of `mat_idx`.

2.4 Symmetric compressed storage formats

If a matrix is symmetric, it is sufficient to store half of it. Such symmetric storage formats has been implemented for compressed row/column storage. The stored elements of the matrix should take the form of an upper triangular matrix stored by rows, that is, the column index array contains values greater than or equal to the row number.

At present, the multicolour, parallel, preconditioners (section 1.2.2) have not been implemented for symmetric storage.

2.5 Concepts for distributed compressed storage

Making a row compressed matrix available to distributed processors takes a few steps. The main difficulty is that processors have to determine what information others will be wanting from them during the iterative solution process.

The next few sections explain the basic concepts of how knowledge of the problem variables is made available to processors.

2.5.1 Owned, local, and bordering variables

Every processor has a set of 'local' variables, divided into 'owned' and 'bordering' variables. The owned variables are the basis: the user determines what variables are taken care of by what processors. The sets of owned variables of the processors don't overlap.

The bordering variables are determined by the sparsity structure of the matrix. Suppose a processor has a set I of owned variables, then j is a bordering variable for that processor if there is a matrix element $a_{ij} \neq 0$ or $a_{ji} \neq 0$ with $i \in I$.

For matrices in compressed row storage, each processor has the full matrix rows of the variables it owns, and similarly for compressed column storage.

2.5.2 Local versus global numbering

It is assumed that the problem variables have an overall numbering, and the current implementation is limited to letting each processor own variables the numbers of which are contiguous. Locally, each processor works with variable numbers starting at one, that is, a renumbering is applied to the global numbers.

In order to perform a matrix-vector product, every processor computes the values in its owned variables, and for that they need copies of the values in the bordering variables. Thus a knowledge of bordering variables is needed to determine the data exchange prior to a product calculation: a processor must both know what owned variables to send off to other processors, and what bordering variables to receive from other processors.

For the regular matrix-vector product each processor needs to know the full matrix row for every variable it owns. The part of the matrix that a processor posesses this way will be called a block row, since for contiguous variables it is a horizontal slice of the full matrix.

At the moment, two compressed storage formats have been implemented: row and column storage. In row compressed storage, each processor stores the full matrix rows of the variables it owns, in column compressed storage it stores whole columns[1].

These two storage schemes require a different arrangement of computation and communication in the matrix vector product.

---

1. Conceivably, a mixed form, where a processor has whole rows, but stored in column compressed format, is possible. If anyone thinks this is desirable, I'll implement it. Just let me know.

Consider the case of row compressed storage. A processor posessing row i of the matrix A can compute

$$y_i = \sum_j A_{ij} x_j.$$

That is, if the processor has the whole of the input vector x (or at least the components x_j for which $A_{ij} \neq 0$) it can compute one component of the the result vector y.

This implies an arrangement where first communication takes place to let each processor have local copies of non-owned components of the input vector, and subsequently all communication is performed. A future implementation will improve efficiency by split the computation in parts that can be performed without and with communication. It then becomes possible to send all necessary components of the input vector, perform the computations involving them, receive the non-owned components of the input vector, and perform the remaining computations.

In the case of column compressed storage, a processor in the posession of column j can form the vector

$$y^{(j)} = (A_{1j} x_j, \ldots, A_{nj} x_j)^t.$$

For this, only the j-th component of the input vector is needed, so the whole computation can be performed without any communication. Then, those components $y_i^{(j)}$ for which j is not an owned variable need to be sent off, so that each processor can form

$$y_i = \sum_j y_i^{(j)}.$$

For the case of a symmetrically stored compressed matrix we follow a strategy that combines the above two schemes. After multiplying by the lower half of the matrix in compressed column format we perform all communication. Then the upper half of the matrix is used in a row compressed multiplication.

Chapter 3

Implementation details

The whole code is based on double precision real numbers and four-byte integers. In PVM messages, therefore, the types are indicated as `REAL8` and `INTEGER4`. Write to me if you need a single precision version of the library.

3.1 Unix library files

The easiest way to link the library to your application code is to make the demo, and use the resulting Unix library file `pvmcg.a`, located in the architecture subdirectory of `pvm3/examples/cg`; see section 4.4. Alternatively, you can take the sources in `pvm3/examples/cg/src/library`, compile them and link them explicitly.

3.2 Storage allocation

In various stages of the iterative method, specifying the amount of workspace needed in advance is very hard. Therefore the code uses a combination of C and FORTRAN routines, where the subprograms in C perform a `malloc` in order to allocate workspace.

3.3 PVM message ids

The present implementation calculates message ids dynamically without ever reusing them. You can shift the lowest message id generated by a call to

```
call set_msgid_origin( nnn )
```

where the argument is the lowest number allowed. A default value of zero is set if you omit this call.

3.4 System dependencies

The only system dependent part of the library I am aware of is a timer routine `clock00`. This is based on the Unix system call `getrusage`. (See section 4.4 for the case of a Hewlett-Packard `HPPA` architecture.) To disable or replace this timer, link your own definition instead of the one in file `clock00.c`.

Chapter 4

Demo code

This part of the manual describes the executables and directories into which the demo solution code is organized.

4.1 Installation of the demo

The tar file containing the library software has to be put in the home directory, and it will install a directory `pvm3/examples/cg` with subdirectories. The various components of the library software are made by going to this directory and typing

`make` to make the machine dependent executables for the demo code. You have to do this once for every architecture that you will enroll in the PVM configuration. If you are using the demo code, this call has to be done last for the architecture from which the processes will be spawned, since it remakes the host program `cghost`.

`make interface` to make a `Wcl`-based X window interface program for easy operation of the demo program; see section 4.7. This interface is obsolescent: you are encouraged to use the `Tcl/Tk`-based interface (see section 4.6), which doesn't need making.

4.2 General installation

If you've installed the demo program, you'll see that there are machine dependent subdirectories. There is nothing against taking them and moving their contents elsewhere if you are going to use the library in your own application. It is only the demo program that rigidly relies on certain directory paths.

4.2.1 File and directory structure

The following paths are used by the pvm iterative methods code, relative to the home directory.

main directory `pvm3/examples/cg`. From this directory, various `make` calls can be made, and the `xcg` and `tcg` interfaces have to be started from here.

source directory `pvm3/examples/cg/src`. This directory contains all sources for the iterative methods library. The sources that are written in `C` require an architecture dependent `.h` file, to be found in the architecture subdirectories of the main directory.

data directory `pvm3/examples/cg/input.data`. This contains data and script files for the interfaces (sections 4.7 and 4.6).

utilities directory `pvm3/examples/cg/util`. This contains some utility programs for the demo.

Wcl interface directory `pvm3/examples/cg/xint`. This contains sources for a `Wcl`-based X interface to the demo code (section 4.7). If at all possible, do not use this interface, but rather the one described in section 4.6.

local directory `pvm3/examples/cg/local.data`. This directory contains distributed input/output data and dumpfiles.

4.3 Program structure

The demo code comes in three executables.

`cghost` This is the host program that spawns the others. In a real application this will not be needed. The host program is remade every time the `aimk` command is used (see section 4.1), so you have to do this last on the host machine.

`cgnode` This is the main node program; it contains both initialization and solution code. To incorporate the solution code in an application, the user has to supply an alternate main program for this executable.

`cgreader` For the case of nodes running on disjoint file systems, this pogram performs read operations from file (if any), and broadcasts data to the `cgnode` programs. It is only necessary for the demo code. After all reads have been performed this program terminates.

4.3.1 Host program

The demo program comes with a host program. In practice its functions will be performed by the main application code. For the demo, the host code communicates to the node codes how many spawned processes there are, and what their instance numbers are. See section 1.1.2 for how this information appears in the node codes.

4.3.2 Reader program

The reader code of the demo program will read in parameter files and matrices and distribute them. In a true application such information will most likely be generated in a distributed manner, so a reader program is not necessary then. For demo purposes, the reader code is strictly speaking only necessary when the pvm configuration spans more than one file system, so that not all node codes are able to read the data files themselves.

The source files for the reader program are recognizable by the `r-` prefix of their names.

4.3.3 Node program

The node code of the demo program is a minimal 'application' main program using the iterative methods library. It contains initialization calls, some of them only necessary for the demo, and the calls to a preconditioner creation routine and an iterative method routine. All of this is described in sections 1.1 and 1.2.

4.3.4 Input

File input is performed if parameters or matrices are read from file instead of being generated by the main program. This is mostly the case in the demo program, as an application program will typically generate all data internally. The demo program reads all of its data from `pvm3/examples/cg/input.data/`.

4.3.5 Output

The iterative routines and the demo main program may generate some output, for instance iteration numbers and the error norm at that iteration. This output is written to standard out, which is in the case of PVM winds up in a file `/tmp/pvml.123` in your directory, where the `123` is your userid.

The demo host program and utilities will also generate output to standard out; since these programs themselves do not run under pvm, this will wind up in your window where the host and utilities have been started.

The CG library can maintain a log file with the parameter settings and the result of the iteration process. Additionally, trace data and errors can be written to file. The log data is written in `pvm3/examples/cg/log.dat`, errors are reported in `pvm3/examples/cg/local.data/err9999` and trace data goes to `pvm3/examples/cg/local.data/dump9999` (where `9999` is replaced by the logical number of the process).

With the exception of the data in the log file, every line of output is preceded by the logical process number, so that data can be traced back to a specific processor if it is displayed on the screen.

4.4 Use of the library without the demo

There are two ways of attaching the library routines to your code: using the `.a` library file generated by the demo, or taking the actual source files.

4.4.1 Library files generated by the demo code

After making the demo on a specific architecture, the directory for that architecture will contain library files `pvmcg.a`, `pvmcg_demo.a` and `pvmcg_reader.a`. You can simply link the first of these to your program.

4.4.2 Library sources

The sources for the library code are in `pvm3/examples/cg/src`. However, in one respect these sources need to be adapted to specific architectures: the interface between Fortran and C routines can differ. This is solved by a separate file `cg_headers.h` in the architecture-dependent directories. Take this file for your specific architecture.

All files with names starting with `r-` are for the reader program of the demo, and you will not be likely to need them. Also, all files with names starting with `demo` are not part of the library code, as are the files `cghost.f` (host code), `readr.f` (reader routines), and `node.f` (sample main program).

There is a timer routine in file `clock00.cbs` that works on practically all Unix systems. However, for HPPA architectures there is a different timer routine: copy the file `HPtimer` in the `HPPA` subdirectory over the original `clock00.c` timer file. In the demo code this is done automatically by the `HPPA` makefile.

4.5 The X interfaces to the demo

The iterative methods code can be controlled through an Xwindow interface program. This program sets parameters, calls auxiliary programs, starts a main program. All actions are performed by calling small script files. This interface is only to facilitate use of the demo program.

There are two interface available. The one based on `Wcl` comes with all sources provided, and you have to make it explicitly by calling `make interface`. It is activated by the call `xcg`. Unfortunately its portability has proved doubtful.

The interface based on `Tcl` does not need to be made, and it is more portable, but it requires you to have `Tcl`, `Tk` and `wish` on your system (see below). It is started by calling `tcg` in the main directory.

4.6 The Tcl interface

The `Tcl`-based interface does not need making. Simply call `tcg` from the `pvm3/examples/cg` directory. The actual interface code is a small script file, `tcg`, in the input directory (`pvm3/examples/cg/input.data`. The file contains commands to be interpreted by `wish`.

You need to have `Tcl`, `Tk`, and `wish` installed on your system. If `wish` is in your search path the script file as given will work. Otherwise replace the first three lines in the script file by the single line

```
#!/your/path/to/wish -f
```

with the proper path substituted.

You can get the `Tcl` and `Tk` distribution from various ftp sites, for instance from `ftp.cs.berkeley.edu` in `/ucb/tcl` or from `ftp.ibr.fr` in `/pub/tcl`. The `wish` windowing shell is included in the `Tk` distribution.

4.6.1 Structure of the interface

The interface has several sets of buttons, to set up the problem, generate test matrices, and start the program; there are numerical values that can be specied, and a log window keeps track of the setup procedure, not of the actual program run (see section 4.3.5).

Do not forget to hit return after changing a numerical value!

These are the buttons and fields of the interface.

Processors Specify the number of processors, optionally organized as a two-dimensional grid.

Reader Specify whether there will be a dedicated reader node, or that all nodes will read data, including potentially large matrices.

Matrix format Grid-based matrices can be internally generated or on file, matrices in compressed row/column or compressed diagonal storage can be only external, so you need to press `Generate Matrix` if you specify these formats.

PDE The partial differential equation for all test problems is

$$-\Delta u + \alpha h(u_x + u_y) - \beta h^2 u = f$$

where α is the 'unsymmetry' and β the 'indefiniteness' value. Note that the first two options are domain size, so the matrix size is their product.

Iterative method The names are fairly self-explanatory.

Iteration trace The amount of information generated during the iterative process ranges from nothing ('silent' option, this should be chosen during timing runs) to dumping all vectors generated ('vectors' option, this should be combined with redirection trace output to file; see further section 1.1.4). Most of the time you will want the 'error' option, which displays the norm of the residual.

Truncate Truncation length for nonsymmetric cg and GMRES.

Reduction The numerical value for the stopping test (see also the next item); this has to be readable by Fortran.

Stopping test The quantity that the residual norm is tested against: an absolute size, relative to the initial size, relative to the size of the right hand side, or relative to an estimate of the norm of the true solution.

Preconditioner No preconditioner, a full Jacobi method, block Jacobi methods with either SSOR or ILU on the processors, sequential SSOR and ILU methods, and parallel, multi-colour SSOR and ILU methods.

Trace modes Since these are mostly for my debugging convenience, I will not go into most of them, except for the following.

- **to Screen / to File** If you generate large amounts of trace data, this will reroute it to `dump9999` files in the local directory.
- **Memory tests** Test the dynamically allocated memory for integrity. If you set this to 'full', such tests are performed after every single operation that may cause memory overwrites. Good for tracking down bugs.

Log window This displays only the output of the setup, not the output of the iterative method or the utility programs; see section 4.3.5.

Master controls Start the iterative process, or generate a test matrix.

4.7 The Wcl interface to the demo

The iterative methods code can be controlled through an X interface program, written using `Wcl`. This program sets parameters, calls auxiliary programs, starts a main program. All actions are performed by calling small script files. This interface is only to facilitate use of the demo program. It is documented below how all parameters can be set in the FORTRAN program.

Important Note Compiling this interface has proved to be a not quite portable operation. Hence, if at all possible, use the `Tcl`-based interface described in section 4.6. The Wcl interface will no longer be supported.

4.8 Making the interface

A single call `make interface` will build the interface program. You can make the interface on several architectures, since the `xint` subdirectory has architecture dependent subdirectories. You start up the interface by calling the `xcg` script file in the main directory.

The interface has some system dependencies. To resolve those, there are several makefiles, and the one for the current architecture is automatically taken. Correct makefiles are currently in place for `ALPHA`, `HPPA`, `RS6K`, `SUN4`. Problems have been detected with `CM5` and `PMAX`.

If there is no makefile for your architecture, go into the `xint` directory and call `xmkmf`. This – hopefully – transforms the `IMakefile` into a correct `Makefile`. Rename this to `Makefile.[YourArch]`.

4.9 Structure of the interface

The `xcg` interface has a number of buttons controlling the iterative method, the preconditioner, the trace mode, and various other choices.

Processors Specify two numbers; the processors pretend to be connected in a grid with sides these sizes. See section 1.1.3.

Reader The choice 'one' is mostly useful for a machine configuration that does not share one single file system. In this case a separate executable will be created that reads input data and broadcasts it to the other programs. The choice 'all' lets all processes read their own data. The executable `cgreader` is not used in this case.

Matrix The demo code can generate its own matrix or read it from file, and the matrix can be based on a 2D grid problem, or be in compressed storage format.

- **2D internal** The matrix is generated internally, based on a 2D grid problem of a size and with coefficients indicated on the 'Problem' line of buttons.
- **2D external** The matrix is based on a grid problem, and resides in a file `fsh.mat` in the input directory. You can generate such a matrix with the 'External matrix: generate' button; see below.
- **Compressed internal** The matrix is of a compressed storage format, and generated by the demo code. This option has not been implemented; it is basically a hook for you to stick you own matrix generator into the demo code.
- **Compressed external** The matrix is of a compressed format, and resides in file `hbf.mat` in the input directory. You can generate such a matrix with the 'External matrix: generate' button; see below.

External matrix: generate If the matrix is declared as external, either of 2D grid or compressed format, pressing this button will generate a matrix on file of the appropriate format. The differential equation and problem size are based on the values in the 'Problem' lines of buttons; see below.

Compressed format Matrices in compressed format can be interpreted as compressed rows, compressed colums, or symmetric. In the symmetric case, the matrix structure can be interpreted either as the upper triangle of the matrix stored by rows, or the lower triangle by columns.

Sparsity pattern For matrices of row or column compressed storage, the sparsity pattern can be symmetric, even if the matrix itself is not. In this case, it is considerably easier for the processors to find out who they will be communicating with than in the case of a non-symmetric sparsity pattern. Therefore, you can make the execution of the matrix preprocessing phase (see section 1.2.1) more efficient by setting this option. There is no harm in declaring a matrix to have a non-symmetric pattern when in fact it is symmetric; see section 1.1.5.

Dump solution This controls whether the solution vector gets dumped to a file; each process generates its own file `sol9999.dat` in the local data directory, where `9999` is the logical processor number. Each line in such a file contains a vector value and the global variable number, or coordinate of that variable in the 2D grid.

Right hand side The program can generate, for testing purposes, a unit right hand side; an 'external' right hand side is supposed to reside in the file `rhs.dat`.

Problem There are routines for generating constant coefficient matrices from a partial differential equation on a rectangular grid. The 'domain size' indicates the size of the grid. The matrix is then represented by the central differences stencil for

$$-\Delta u + \beta h^{-1}(u_x + u_y) - h^{-2}\alpha = f,$$

where α is the 'indefiniteness' and β the 'unsymmetry' value.

Iterative method The methods currently available are: conjugate gradients, nonsymmetric conjugate gradients (equivalent to orthomin; see [3, sec. 6.2] for the exact method), generalized minimal residual, conjugate gradients on the normal equations, biconjugate gradients, quasi-minimal residuals, conjugate gradients squared, and biconjugate gradients stabilized.

Truncate The nonsymmetric cg and gmres methods carry old information around with them. This option controls how many old vectors are being kept. The more, the better your chances of speedy convergence.

Maximum iterations Like it says.

Reduction Give a power of ten; the method halts when the residual has been reduced by this much.

Preconditioner Choice of no preconditioner, Jacobi preconditioner, block Jacobi preconditioners with all processors executing either an incomplete factorization or ssor method in parallel, sequential incomplete factorization and ssor preconditioners, and parallel multi-colour versions of ilu/ssor.

The `xcg` program makes calls to script files in the input directory. These script files create or add to files with `.dat` extensions, which are read by the demo code.

4.10 Utility programs

There are two utility programs for generating test data: `genmat` and `gendiv`. They reside in the `util` subdirectory, and get called by the interface program. In fact, they get created on demand by the interface program.

The first utility, `gendiv`, is a program that reads the number of variables from `siz.dat` and the number of processors from `con.dat`, and generates from it a file `div.dat` that contains the division of variables over the processor. Format:

- One line with processor number / number of variables;
- the following lines contain the variable numbers.

The second utility, `genmat`, generates from the parameters in `pde.dat` and `siz.dat` a compressed row storage matrix of the appropriate size. This can then be loaded by specifying an ‘external’ matrix in the window interface program.

There are two further utilities `asmbl_cprs` and `asmbl_grid` that reside in the `util` subdirectory. They are for analysis of dumped data, and get made on demand the first time you type `make vectors` in the `local.data` subdirectory. See further section 1.1.4.

4.11 Other remarks

4.11.1 Lapack and blas

The cg library uses a few lapack and blas routines. In case you are already linking to the lapack library, don't link the cg library file `lapack_auxs.f`; if you have optimized blas, don't link to `blas_auxs.f`.

4.11.2 Limitations and known bugs

Matrices based on a 2D grid can be declared symmetric, but the preconditioner routines are not correct for this case. Unpreconditioned runs will work.

4.11.3 History

0.4 First announced release, β-release, 14th July 1994.
0.5 Sources date-stamped, Tcl interface added. 18th July 1994.
0.6 Library files described. Minor bugs. 20th July 1994.

Bibliography

[1] 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.

[2] I.S. Duff, A.M. Erisman, and J.K. Reid. *Direct Methods for Sparse Matices*. Clarendon Press, Oxford, 1986.

[3] Victor Eijkhout. Lapack working note 51: Qualitative properties of the conjugate gradient and lanczos methods in a matrix framework. Technical Report CS 92-170, Computer Science Department, University of Tennessee, 1992.

[4] Claude Pommerell. *Solution of Large Unsymmetric Systems of Linear Equations*, volume 17 of *Series in Micro-electronics, volume 17*. Hartung-Gorre Verlag, Konstanz, 1992.