CLRC

Technical Report
DL-TR-99-003.

CLIPS: The CLRC Library of Parallel Subroutines

RJ Allan and YF Hu

July 1999

COUNCIL FOR THE CENTRAL LABORATORY OF THE RESEARCH COUNCILS

ISSN 1362-041X

Parallel Application Software on High Performance Computers. CLIPS: The CLRC Library of Parallel Subroutines. [*]

R.J. Allan and Y.F. Hu
Computational Science and Engineering Department,
CLRC Daresbury Laboratory,
Daresbury, Warrington WA4 4AD, UK

Email: r.j.allan@dl.ac.uk or y.f.hu@dl.ac.uk
This report is available from http://www.cse.clrc.ac.uk/Activity/CLIPS

July 21, 1999

Abstract

Over the last four years a number of parallel algorithms have been written at CLRC Daresbury Laboratory in order to optimise parallel applications in various areas of computational science and engineering. With a view to making these more widely available and to promote better software re-usability and modularity a parallel library is being created named CLIPS. This brief report summarises the Library structure and functionality.

The Library is written mainly in Fortran 90 (with a little C) and uses MPI for communication. It should therefore be portable to most, if not all, contemporary serial and parallel computing platforms including clusters of PCs and workstations.

Routines will be added to the Library as required. Support for SMP systems will be added in the future. There is so far no attempt to address any particular area exhaustively. Documentation, example programs and a test suite are however part of the Library distribution.

Keywords: parallel computing, numerical algorithms, subroutine library, Fortran 90, MPI, linear algebra, sparse matrices, Fourier transforms.

[*]Edition 1: July 1999

Contents

1 Introduction

Over the last four years a number of parallel algorithms have been written at CLRC Daresbury Laboratory in order to optimise parallel applications in a number of areas of computational science and engineering. With a view to making these more widely available and to promote better software re-usability and modularity a parallel library is being created named CLIPS.

The Library is written mainly in Fortran 90 (with a little C) and uses MPI for communication. It should be portable to most, if not all, contemporary serial and parallel computers including clusters of PCs and workstations.

Routines will be added to the Library as required. SMP support using OpenMP directives will be added in the future. There is so far no attempt to address any particular area exhausively. Documentation, example programs and a test suite are however provided as part of the Library distribution.

In the rest of this report we follow the format of the report by Dongarra and Waśniewski on LAPACK 90 [6] with supplementary comments on parallel aspects.

2 Other Parallel Numerical Libraries

Subroutines included in the CLIPS library have been developed only where a clear need was identified. Several surveys were carried out, of numerical software already available [1, 2, 3]. They address a number of areas of relevance to computational science and engineering on high-performance computers. These reports are separately available from the authors or via the Web page at URL
`http://www.cse.clrc.ac.uk/Activity/HPCI` and should be consulted for further information.

3 Fortran 90 and MPI

The current standard for Fortran is ISO/IEC 1539-1991 (in the USA, ANSI X3.198-1992), the so-called Fortran 90 standard. This has a number of significant advantages over previous dialects of Fortran and is particularly useful for providing modular software. We recommend the book by Metcalf and Reid [9] for more information. Some of the important features introduced in this standard include:

- array operations;
- pointers;
- improved facilities for numerical computations including a set of numerical inquiry functions;
- parameterisation of the intrinsic types to permit processors to support short integers, very large character sets, more than two precisions for real and complex and packed logicals;
- user-defined derived data types composed of arbitrary data structures and operations upon those data structures;

- facilities for defining collections called "modules", useful for global data definitions and for procedure libraries. These support a safe method of encapsulating derived data types;
- requirements on a compiler to detect the use of constructs that do not conform to syntax of the language or are obsolescent;
- a new source form, more appropriate to use at a terminal;
- new control constructs such as the SELECT CASE construct and a new form of the DO construct;
- the ability to write internal procedures and recursive procedures and to call procedures with optional and keyword arguments;
- dynamic storage (automatic arrays, allocatable arrays and pointers);
- improvements to the input-output facilities, including handling partial records and a standardised NAMELIST facility;
- many new intrinsic procedures, including those for machine constants.

We have found Fortran 90 particularly useful for our purposes. Nevertheless there are still difficulties, especially in the area of support for irregular structures and sparse matrices and in binding to other libraries which were designed for older languages (e.g. C). Some particular difficulties in using makefiles and devising a sensible build procedure are addressed in the Library specifications [5]. Some problems encountered with the use of asynchronous message-passing or threads calls are mentioned here and some comments on optimisation are also noted.

3.1 Asynchronous Subroutines called from Fortran 90

If there is no explicit interface available for a routine that is called from a Fortran 90 program the compiler may make local copies of the variables passed to the routine. The reason is that, especially with array arguments, the program may be passing assumed-shape arrays (e.g. array sections with elements in non-contiguous memory storage) to a "foreign-language" routine, such as FORTRAN77 or C, which expects contiguous storage. This will depend on how the arrays were declared in the calling routine and may also vary from computer to computer. The difficulties we have encountered arise when an array copy is made and is referred to as the "copy-in/copy-out" problem.

If the copy occurs the following situation may arise:
1. we call an asynchronous library routine, e.g. `MPI_IRECV(...)`;
2. the compiler makes a local copy of the array into which we wish to receive data;
3. the library routine posts a non-blocking receive for the MPI subsystem to handle and returns control to the calling program;
4. the compiler copies back the local array into the real array *before* the MPI subsystem has received any data;
5. the MPI subsystem signals that the data is ready in an `MPI_WAIT(...)` call, but there is no mechanism for getting the data back into the real array.

In fact in the worst situation the MPI subsystem may write data into some area of memory which has by then been used for another purpose, and unpredictable errors can occur.

This is not only a problem with MPI, but with any asynchronous subroutine, including calls to threads libraries. Fortran 2000 will contain asynchronous i/o subsystem so the problem may be addressed more generally by then. Note that IBM systems already accommodate asynchronous i/o, but this is intrinsic to the compiler whereas library calls are extrinsic. They also provide a VOLATILE attribute which may be used on variable declarations to tell the compiler that there may be some *unknown* side effects when this variable is used.

3.2 Some Comments on Optimisation

A further effect of the copy-in/copy-out problem is that it may lead to inefficient code. If an extra copy occurs at subroutine boundary, more time will be taken and if it is a call to an optimised numerical library, e.g. the Basic Linear Algebra Subprograms (BLAS), there will be little benefit from using them. To avoid this problem explicit Fortran 90 interfaces must be made available by the library writers, as was done for LAPACK 90 [6] and in CLIPS.

It is very disappointing that this has not been done for MPI, even in the new MPI-2. There are a number of basic and complex issues, outwith the remit of this report, which have not been addressed. In the meantime there are comments by John Reid in the MPI-2 documentation [12].

In these cases it is necessary to move the copy-in/copy-out problem as far out of the centre of the code as possible. This can be achieved for practical purposes by declaring the offending arrays with fixed-size dimensions in the lower-level routines. The principle is illustrated by the following sketch:

```
----------------------------------------------------------------------
      CALL sub1(n/2,m/3,a(1:n:2,1:m:3))
      ...
----------------------------------------------------------------------
----------------------------------------------------------------------
      SUBROUTINE sub1(nn,mm,x)
      REAL(KIND8), DIMENSION(nn:mm), INTENT(INOUT) :: x
      ...
      CALL blas(nn*mm,x)
      ...
      END SUBROUTINE sub1
----------------------------------------------------------------------
```

Whilst this does not guarantee that there is no copy, indeed even FORTRAN77 compilers were permitted to make one, it is likely to move the copy to the boundary of `sub1` rather than the `blas` routine.

4 Interface Blocks for CLIPS

The CLIPS library is generally provided in compiled form via a randomised library file `libclips.a` which contains the code objects for a particular architecture. However the full source code and build procedures are available by arrangement, in particular for collaboration to extend the collection.

It is also necessary to provide a set of interface blocks in order for the Fortran 90 compiler to validate subroutine calls and optimise the passing of data structures on the subroutine boundaries. There is no cross-architecture standard for the way this is to be done. Platforms may expect files with extensions `.mod`, `.kmo` etc. (produced by compiling a prior module in the dependancy tree) or may simply parse the relevant source files again when a USE statement is encountered. This is discussed further in the CLIPS library specification and full text of the interfaces for each public routine is also given [5].

4.1 Example Calling Program

```
      PROGRAM test
      USE clips_cgs
      USE clips_precision
      USE read_arg_mod
! the matrix and right-hand-side.
      INTEGER(int4) :: nz
      INTEGER(int4) ::  n
      INTEGER(int4), pointer :: irn(:)
      INTEGER(int4), pointer :: jcn(:)
      REAL(real8), pointer :: val(:)
      REAL(real8), pointer :: rhs(:)
! initial guess and the solution
      REAL (real8), pointer, dimension (:) :: x
! input file name
      CHARACTER (len=60):: infile
! input unit and whether it is a single file or
! in nproc files
      INTEGER(int4) :: input_unit
      LOGICAL single_file
! working int
      INTEGER (int4) :: i
! error flag from cg
      INTEGER (int4) :: iflag
! whether the inpout is in binary or ascii
      LOGICAL binary
! mpi related
      INTEGER NUM_PES,me
      binary=.false.
! roll in MPI
      CALL MPI_INIT(ierr)
      CALL MPI_COMM_RANK(MPI_COMM_WORLD,ME,ierr)
      CALL MPI_COMM_SIZE(MPI_COMM_WORLD,NUM_PES,ierr)
! read in the argument (single or multiple files)
      CALL read_arg(input_unit,single_file,binary)
! read in the matrix and right hand side
      IF (binary)THEN
        READ(input_unit) n,nz
        ALLOCATE(irn(nz),jcn(nz),val(nz),rhs(n))
        READ(input_unit) irn
        READ(input_unit) jcn
        READ(input_unit) val
        READ(input_unit) rhs
      ELSE
        READ(input_unit,*) n,nz
        ALLOCATE(irn(nz),jcn(nz),val(nz),rhs(n))
        DO i=1, nz
          READ(input_unit,*) irn(i),jcn(i),val(i)
        END do
        DO i=1,n
          READ(input_unit,*) rhs(i)
        END do
      END if
! initial guess
      ALLOCATE(x(size(rhs)))
      CALL clips_cg_intialize(n,nz,irn,jcn,val,rhs,single_file)
! solve the system 10 times!
      DO  i=1, 1
        CALL random_number(x)
        CALL clips_cg_solve(x,iflag)
        IF (me==0) then
          WRITE(*,*) "loop=====",i
          WRITE(*,*) "x(1)=",x(1)," x(n)=",x(n)
        END IF
      END DO
! clean up
      CALL clips_cg_finalize()
! print timing info.
      CALL clips_cg_summarize()
! exit MPI
      CALL MPI_BARRIER(MPI_COMM_WORLD,ierr)
      CALL MPI_FINALIZE(ierr)
      END program test
```

4.2 List of Available Modules

CLIPS_BFG.mod – 1-sided block-factored Jacobi eigensolver;
CLIPS_CGS.mod – stabilised conjugate-gradient solver with ILU preconditioning;
CLIPS_FFT.mod – fast Fourier transforms on 3D data in a block-cyclic distribution;
CLIPS_PCRS.mod – controlled random search optimisation;
CLIPS_ERR.mod – internal error-handling routines;
CLIPS_PRECISION.mod – internal definition of precisions;
CLIPS_RAN.mod – portable random number generator;
CLIPS_TIMER.mod – portable elapsed-time routine;
CLIPS_PROFILER.mod – simple MPI profiling routines;
MPI.mod – Fortran 90 interface to platform-specific MPI.

5 Code of CLIPS Routines

The layout of code in the CLIPS library routines, both public and private, follows the layout given for the LAPACK 90 library [6]. They are divided into the following parts:

- heading of the routine
  - SUBROUTINE or FUNCTION statement
  - USE statements
    - CLIPS_PRECISION module
    - modules from CLIPS_OTHERS if needed, e.g. timers, CLIPS_ERROR
    - specific CLIPS modules for operations required
  - IMPLICIT NONE statement
  - argument specifications
- argument descriptions (comments)
- local variable declarations
- executable statements
  - local variable initialisations
  - testing the arguments
  - work space allocation if needed
  - write warning messages if needed (by invoking CLIPS_ERROR handler). Note this may abort the code and return an error flag.
  - duplicate MPI communicator ro initialise a new context
  - code text and calling private routines and other public CLIPS routines
  - work space de-allocation if needed
- routine END statement

Note that in the current version of the Library some, or all of these steps may be carried out in one routine. There is however provision for dividing the procedure into three steps as follows:

1. CLIPS_name_INITIALIZE
2. CLIPS_name or CLIPS_name_SOLVE
3. CLIPS_name_FINALIZE

This adopts a similar procedure to that used in the MPI standard. It enables `CLIPS_name` to be called multiple times for each initialisation therefore reducing overheads. In some cases a supplementary `CLIPS_name_SUMMARIZE` is provided to return run-time information to the user. These routines will generally be provided in a module `MODULE CLIPS_name_mod`. The use of these was illustrated in the above example.

6 CLIPS Documentation

Full documentation of all the CLIPS public library routines and module interfaces is available [5]. This will be updated as new routines are added. Each set of routines comprising a module is described in a stand-alone chapter. This reflects the structure of the Library in which only a subset of routines need be included for a specific purpose. Further information and a copy of the Library specifications is available from the authors.

7 CLIPS Test Examples and Programs

In addition to the Library modules and routines we have written a test suite and example programs for CLIPS. The test suite is used to test the functionality of the routines as described in the documentation and is only available to developers. The example programs however are intended to illustrate how to use individual routines and provide a spot check that things are working. The code shown in 4.1 is taken from the example code suite to illustrate the CGS routines.

8 CLIPS User-callable Routines

Appendix C provides a short description of routines currently in the CLIPS library. The call of the routine and a brief statement of purpose are given. For example the call to `clips_cg_solve`:

```
SUBROUTINE clips_cg_solve(x,iflag)
REAL(REAL8), DIMENSION(:), TARGET, INTENT(INOUT) :: x
INTEGER(INT4), INTENT(OUT) :: iflag
```

Subroutine `clips_cg_solve` carries out the parallel BiCGSTAB algorithm with ILU preconditioner. The related routine `clips_cg_initialize` must have been called to set up essential information about the sparse matrix to be solved.

An initial guess of the solution may be provided in `x`. On exit `x` contains the solution of the linear system to the required precision.

The error flag `iflag` returned from `clips_cg_solve` is 0 for a successful solve or 1 if we have exceeded the maximum number of iterations.

9 Acknowledgements

The preparation of this report, and the work reported, was funded by EPSRC partly through a grant GR/K82635 to the CLRC HPCI Centre at Daresbury Laboratory and partly through its Service Level Agreement with the CSE Department.

References

[1] R.J. Allan and I.J. Bush *Parallel Diagonalisation Routines* Edition 1 (CLRC Daresbury Laboratory, 1996)

[2] R.J. Allan and I.J. Bush *Serial and Parallel FFT Routines* Edition 1 (CLRC Daresbury Laboratory, 1996)

[3] R.J. Allan, Y.F. Hu and P. Lockey *Survey of Parallel Numerical Analysis Software* Edition 2, Technical Report DLT-99-01 (CLRC Daresbury Laboratory, April 1999)

[4] R.J. Allan, J. Heggarty, M.C. Goodman and R.R. Ward *Survey of Parallel Performance Tools and Debuggers* (CLRC Daresbury Laboratory, 1999)

[5] R.J. Allan, Y.F. Hu, I.J. Bush and A.G. Sunderland *CLIPS: CLRC Library of Parallel Subroutines. User Manual and Specifications* (CLRC Daresbury Laboratory, 1999)

[6] J. Dongarra and J. Waśniewski *High Performance Linear Algebra Package - LAPACK90* UNI-C Report UNIC-98-01 (Danish Computing Centre for Research and Education, Technical University of Denmark, 1998)

[7] R.J. Littlefield and K.J. Maschhoff *Investigating the Performance of Parallel Eigensolvers for large Processor Counts* Theor. Chim. Acta 84 (1993) 457-73

[8] G. Marsaglia, A. Zaman and W.W. Tsang *A Universal Random Number Generator* Statistics and Probability Letters 8 (1990) 35-39

[9] M. Metcalf and J. Reid *Fortran 90 Explained* (Oxford University Press, 1990)

[10] Fortran 90 standard ISO/IEC 1539-1991 and ANSI X3.198-1992

[11] *MPI: A message-passing Interface Standard* MPI Forum, (June 1995)

A. Skjellum, N.E. Doss and P. V. Bangalore *Writing libraries in MPI* in "Proceedings of the Scalable Parallel Libraries conference" A. Skjellum and D.S. Reese (eds.) (IEEE Computer Society Press, 1993). Available by anonymous ftp from
`ftp://aurora.cs.msstate.edu/pub/reports/SPLC93`

[12] *MPI-2: Extensions to the Message-Passing Interface* MPI Forum (July, 1997)

E. Minty *MPI-2: Extending the Message-Passing Interface* v1.0 (EPCC, 1998). Available from URL `http://www.epcc.ed.ac.uk/epcc-tec/documents`

[13] *NetLib* On-line repository of numerical algorithms and other high-performance computing software at URL `http://www.netlib.org`

A Module Precision

```
      MODULE clips_precision
! definition of basic precisions
! INTEGER-Kinds
      INTEGER, PARAMETER :: INT1 = SELECTED_INT_KIND(2)
      INTEGER, PARAMETER :: INT2 = SELECTED_INT_KIND(4)
      INTEGER, PARAMETER :: INT4 = SELECTED_INT_KIND(9)
      INTEGER, PARAMETER :: INT8 = SELECTED_INT_KIND(18)
! logical kinds
      INTEGER, PARAMETER :: BYTE = INT2
      INTEGER, PARAMETER :: WORD = KIND(.FALSE.)
! real kinds
      INTEGER, PARAMETER :: SINGLE = SELECTED_REAL_KIND(p=4)
      INTEGER, PARAMETER :: REAL4 = SELECTED_REAL_KIND(p=4)
      INTEGER, PARAMETER :: DOUBLE = SELECTED_REAL_KIND(p=14)
      INTEGER, PARAMETER :: REAL8 = SELECTED_REAL_KIND(6,70)
      INTEGER, PARAMETER :: QUAD = SELECTED_REAL_KIND(p=18)
      INTEGER, PARAMETER :: float = SELECTED_REAL_KIND(6, 70)
! complex kinds
      INTEGER, PARAMETER :: CMPLX16 = SINGLE
      INTEGER, PARAMETER :: COMPLEX8 = REAL4
      INTEGER, PARAMETER :: CMPLX32 = DOUBLE
      INTEGER, PARAMETER :: COMPLEX16 = REAL8
      INTEGER, PARAMETER :: CMPLX64 = QUAD
      INTEGER, PARAMETER :: imag = KIND((1.0_float, 1.0_float))
      END MODULE clips_precision

      MODULE clips_precision
```

B Example of Documentation – Parallel CGS

B.1 Summary

A Fortran 90 module which implements a sparse ILU preconditioner used with BiCGSTAB for solving non-symmetric linear systems in parallel.

B.2 Attributes

Version: 1.0
Public calls: `clips_cg_initialize`, `clips_cg_solve`, `clips_cg_finalize`, `clips_cg_summarize`
Public modules: `clips_cgs`
Other modules required: `mpi`, `clips_timer`
Date: 1998
Origin: Y.F. Hu, CLRC Daresbury Laboratory
Language: Fortran 90 and C
Conditions on external use: Standard, see separate chapter.

B.3 How to use the Package

This package is used through `MODULE clips_cgs`.

See specifications for further details.

The module uses the `MODULE clips_timer` for internal timing purposes. This is described in a separate chapter.

B.4 Specification of CLIPS_CG_INITIALIZE

There are a number of control parameters which control the use of the preconditioner, the tolerance and maximum number of iterations. See the section on Arguments.

Subroutine `clips_cg_initialize`:
a) copies and converts the input sparse matrix-related things into internal format;
b) work out the scheduling and halos;
c) carries out an incomplete factorisation of fill-in zero by default (can be turned off);
d) set up control parameters for `cg_solve`.

This routine needs only be called once if the user has multiple right hand side to solve with the same matrix. The routine allocates working storage, therefore when the matrix is no longer needed this routine should be followed by a call to `clips_cg_finalize`.

```
      SUBROUTINE clips_cg_intialize(comm,nn,nz,irn,jcn,val,rhs, &
     &   single_file,my_nstart,my_level,my_nprecon,my_tol,my_maxit)
      INTEGER :: comm
      INTEGER(int4) :: nn,nz
      INTEGER(int4), pointer :: irn(:)
      INTEGER(int4), pointer :: jcn(:)
      REAL(real8), pointer :: val(:)
      REAL(real8), pointer :: rhs(:)
      LOGICAL :: single_file
! optional control parameters
      INTEGER, INTENT(IN), OPTIONAL :: my_nstart
      INTEGER, INTENT(IN), OPTIONAL :: my_level
      INTEGER, INTENT(IN), OPTIONAL :: my_nprecon
      INTEGER, INTENT(IN), OPTIONAL :: my_maxit
      REAL(real8), INTENT(IN), OPTIONAL :: my_tol
```

B.4.1 Argument List

`INTEGER, INTENT(IN) ::  comm`
On entry: the communicator for MPI.

`INTEGER(int4), INTENT(IN) ::  nn,nz`
On entry: number of non-zeros and size of matrix (if single_file = .false. this is the local number of

non-zeros and size, otherwise it is the global matrix).

`INTEGER(int4), POINTER :: irn(:)`
On entry: the array of non-zero row indices.

`INTEGER(int4), POINTER :: jcn(:)`
On entry: the array of non-zero column indices.

`REAL(real8), POINTER :: val(:)`
On entry: the array of non-zero matrix entries. Val(i), together with `irn(i), jcn(i)`, gives the i-th non-zero matrix element.

`REAL(real8), POINTER :: rhs(:)`
`rhs(j)` is the j-th element of the right-hand-side vector of the linear system.

`LOGICAL, INTENT(IN) :: single_file`
On entry: whether the inputing matrix is a single input matrix or distributed matrices

`INTEGER, INTENT(IN), OPTIONAL :: my_nstart`
On entry: starting option. By default my_nstart = 0.
o If nstart=0, cold start, scheduling and ILU factorisation (when nprecon=1) will be performed;
o If nstart=1, warm start, that assume that the sparse structures are unchanged, scheduling will not be calculated again, but ILU factorisation will be recalculated;
o If nstart>=1, hot start, then LU factorisation and scheduling is assumed to be known and will not be recalculated

`INTEGER, INTENT(IN), optional :: my_level`
On entry: print level: should be 0 or 1, default 1

`INTEGER, INTENT(IN), OPTIONAL :: my_nprecon`
On entry: whether ILU preconditioner should be used. 0 for not using ILU and 1 for use ILU preconditioner. Default is 1.

`INTEGER, INTENT(IN), OPTIONAL :: my_maxit`
On entry: maximum number of iterations allowed. Default 10000

`REAL(real8), INTENT(IN), OPTIONAL :: my_tol`
On entry: tolerance to be achieved. Defined to be level that the relative preconditioned residual has to go down to. Default 1.0d-10

B.5 Specification of CLIPS_CG_SOLVE

Subroutine `clips_cg_solve` carries out the parallel BiCGSTAB algorithm with ILU preconditioner.

```
    SUBROUTINE clips_cg_solve(x,iflag)
    REAL(real8), TARGET :: x(:)
    INTEGER, INTENT(OUT) :: iflag
```

B.6 Argument List

`REAL(real8), TARGET ::  x(:)`
On entry: initial guess of the solution. On exit: The solution of the linear system. The size of this vector is the size of the whole matrix if the input is the whole matrix, otherwise

`INTEGER, INTENT(OUT) ::  iflag`
On exit: error flag from cg: 0 for successful solve, 1 for exceeding maximum iterations.

B.6.1 Errors and Warnings

`cg_solve` returns with an error flag `iflag`, see the section on arguments.

B.7 Specification of CLIPS_CG_SUMMARIZE

Subroutine `clips_cg_summarize` prints out timing informations to the screen.

```
    SUBROUTINE clips_cg_summarize()
```

B.7.1 Argument List

There are no arguments.

B.8 Specification of CLIPS_CG_FINALIZE

Subroutine `clips_cg_finalize` deallocates spaces allocated for preconditioner and scheduling of communication.

```
    SUBROUTINE clips_cg_finalize()
```

B.8.1 Argument List

There are no arguments.

B.9 General Information

Workspace:
Use of common:
Other routines called directly:
Notes:

B.10 Method

See specifications.

B.11 Example

B.11.1 Example text

The program comes with test matrices: matrix_ascii and matrix_ascii_1, matrix_ascii_2, matrix_ascii_3, matrix_ascii_4. They can be used to test the subroutines in single-file mode and in distributed file mode. The example code text was given in 4.1 above.

B.11.2 Example Data

As an illustration, consider solving a 4×4 linear system

$$\begin{pmatrix} 2 & -1 & 2 & 0 \\ 0 & 4 & 5 & 1 \\ 0 & 1 & 8 & 0 \\ 0 & 1 & 0 & 8 \end{pmatrix} x = \begin{pmatrix} 4 \\ 10 \\ 9 \\ 9 \end{pmatrix}$$

The matrix is stored in a single file "matrix_simple_ascii" (see directory "matrices/matrix_simple_ascii") as

```
4 10
1 1 2.0
1 2 -1.0
1 3 3.0
2 2 4.0
2 3 5.0
2 4 1.0
3 2 1.0
3 3 8.0
4 2 1.0
4 4 8.0
4.0
10.0
9.0
9.0
```

The first row means that the system is of order 4, with 10 non zeros. The next 10 rows gives the individual elements of the matrix. The last four rows give the right hand side.

Solving this system on four processors can be done as follows:

```
mpirun -np 4 <name_of_test_program> <the_matrix_file> single .
```

Alternatively you can distribute the matrix into four horizontally-sliced matrices and do

```
mpirun -np 4 <name_of_test_program> <the_matrix_file> multiple .
```

For example the slice of matrix on processor 2 will be the second row of the matrix together with the second right-hand-side of the matrix, thus the matrix file is:

```
1 3
 1 2 4.0
 1 3 5.0
 1 4 1.0
 10.0
```

B.11.3 Example results

The system is solved in four iterations and the solution is returned in x. The output to the screen is:

```
res0 =  3.575     relat. res0 =  1.000
ir =        1 res = 0.3170     relat. res = 0.8867E-01
ir =        2 res = 0.1964     relat. res = 0.5495E-01
ir =        3 res = 0.4647E-16 relat. res = 0.1300E-16
 final residual is   4.647407769350522E-017
 x(1) =  1.00000000000000       x(n) =  1.00000000000000
```

C List of Public Routines

Driver Routines for Stabilised Conjugate Gradient with ILU Pre-conditioning

```
      SUBROUTINE clips_cg_intialize(comm,nn,nz,irn,jcn,val,rhs, &
     &           single_file,my_nstart,my_level,my_nprecon,my_tol, &
     &           my_maxit)
```

Initialises the CG solver system as follows:
a) copies and converts the input sparse matrix-related things into internal format;
b) work out the scheduling and halos;
c) carries out an incomplete factorization of fill-in zero by default (can be turned off);
d) set up control parameters for `clips_cg_solve`.

```
      SUBROUTINE clips_cg_solve(x,iflag)
```

solves the system using the parallel BiCGSTAB algorithm.

```
      SUBROUTINE clips_cg_summarize()
```

prints out timing informations to the screen.

```
      SUBROUTINE clips_cg_finalize()
```

deallocates spaces allocated for preconditioner and scheduling of communication.

Block-cyclic multi-dimensional FFT

```
      SUBROUTINE clips_fft_initialize(n_dims, lengths, proc_grid, &
     &           block, communicator, context, error)
```

initialises things like the communication pattern and data for the FFT.

```
      SUBROUTINE clips_fft(a, work, context, direction)
```

carries out the parallel nD FFT computation within the given context and in a defined direction (forward or backward).

```
      SUBROUTINE clips_fft_summarize(processor, context)
```

produces information relevant to a given processor for FFTs to be carried out within a given context.

```
      SUBROUTINE clips_fft_finalize()
```

releases storage associated with stored FFT information.

Optimisation

```
      SUBROUTINE clips_pcrs_initialize()
```

initialises parallel Controlled Random Search optimisation module.

```
      SUBROUTINE clips_pcrs_solve(context, mypid, num_pes, my_func, &
     &           my_box, n, nratio, maxfun, tol, num_offspring, &
     &           print_level, method, restart_flag, hotstart_flag)
```

carries out parallel Controlled Random Search optimisation.

```
      SUBROUTINE clips_pcrs_summarize()
```

summarises performance of parallel Controlled Random Search optimisation module.

Dense Iterative Eigensolvers

```
      SUBROUTINE clips_jacobi(n, ncols, ldg, G, ldv, V, initialize, &
     &      tolerance, nprocs, map, rank, rank_array, global_sum, &
     &     iterations)
```

This routine solves the eigenvalue problem **GV=VE** for real symmetric G. The algorithm used is very closely based on that descibed by Littlefield et al. [7].

Error Handling Routine

```
      SUBROUTINE clips_error(status,calling_routine,message, &
     &           called_routine)
```

The package will stop when `clips_error` is called with a positive `status` value. It then calls `MPI_abort` to close down the application and tidy any outstanding parallel communications. If `status` has a negative value a warning message only is printed and the exectution allowed to continue. If `status==0` no action is taken and control returned directly to the calling program.

Random Number Generator

```
      SUBROUTINE clips_ran_initialize(i,j,k,l)
```

initialises the random number generator using four arbitrary integer seeds.

```
      FUNCTION clips_ran()
```

returns a random number in the range [0,1). See Marsaglia et al. [8].

Elapsed Time Routine

```
      FUNCTION clips_time()
```

returns a double-precision value of seconds since an arbitrary epoch.

MPI Profiler

```
      SUBROUTINE clips_profile_initialize()
```

Initialises the profiling system using the MPI Profiling Interface, see separate report [4]. Resets all internal counters.

```
      SUBROUTINE clips_profile_on()
```

switches on statistics collecting.

```
      SUBROUTNE clips_profile_off()
```

switches off statistics collecting.

```
      SUBROUTINE clips_profile_time()
```

saves a value from `clips_time()` in an internal variable.

```
      SUBROUTINE clips_profile_collect(i,count,type)
```

updates statistics for each instrumented event.

```
      SUBROUTINE clips_profile_summarize()
```

prints out average/max statistics for all events.