# CLIPS

# CLRC LIBRARY OF PARALLEL SUBROUTINES

# USER MANUAL AND SPECIFICATIONS *

R.J. Allan, Y.F. Hu, I.J. Bush and A.G. Sunderland

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`

*Edition 1: April 1999

# Contents

INTRODUCTION . . . 2

ACKNOWLEDGEMENTS . . . 7

CONDITIONS OF USE . . . 8

SIGNAL PROCESSING . . . 9

PARALLEL FFT . . . 10

LINEAR ALGEBRA . . . 23

PARALLEL STABILISED CONJUGATE GRADIENT . . . 24

ONE-SIDED BLOCK FACTORED JACOBI EIGENSOLVER . . . 31

NON-LINEAR OPTIMISATION . . . 39

TIMING AND MPI PROFILING ROUTINES . . . 40

ERROR HANDLING . . . 41

MPI PROFILING . . . 43

SUPPORT FUNCTIONS . . . 48

ERROR HANDLING . . . 49

RANDOM NUMBER GENERATOR . . . 51

# INTRODUCTION

## CONVENTIONS

### Routine Naming Conventions

The routines are named `CLIPS_NAME` where "NAME" is the name of the routine appearing in this documentation.

Because there are no automatic constructurs or destructors in Fortran 90, and because a number of routines require setup and shutdown phases, utility routines are provided with names like `CLIPS_NAME_INITIALIZE, CLIPS_NAME_FINALIZE` and `CLIPS_NAME_SUMMARIZE`. These are not present for every routine and the relevant chapter should be consulted.

### Argument Lists

In this release ordering in argument lists is not completely standardised. Most main routines will however have "context", "communicator" and "error" arguments in addition to relevant numerical quantities.

### Data Precision

A module is included in the library which declares basic precision kind-type parameters as follows:

```
      MODULE clips_precision
! definition of basic precisions
      INTEGER, PARAMETER :: INT4=KIND(1_4)
      INTEGER, PARAMETER :: INT8=KIND(1_8)
      INTEGER, PARAMETER :: REAL4=KIND(1.0_4)
      INTEGER, PARAMETER :: REAL8=KIND(1.0_8)
      INTEGER, PARAMETER :: COMPLEX8=KIND(CMPLX(1.0_4,1.0_4))
      INTEGER, PARAMETER :: COMPLEX16=KIND(CMPLS(1.0_8,1.0_8))

      END MODULE clips_precision
```

# LIBRARY STRUCTURE

## Directories in the Distribution

The library distribution contains a subset of the following directories. Usually the full source and machine-specific files will not be included.

`CLIPS/doc` – complete specifications document which includes chapters from every set of routines via relative path `../*/doc`;
`CLIPS/NAME` – program directory for routine from chapter "NAME", e.g. CGS, BFG, FFT;
`CLIPS/bin` – user-accessible library archive files and module interfaces;
`CLIPS/include` – user-accessible library module interfaces;
`CLIPS/examples` – example codes;
`CLIPS/arch` – set of machine-specific include files, not all may be present in the distribution;
`CLIPS/OTHERS` – collection of utility routines such as error handling, timers and random number generators.

The "NAME" subroutine directories, e.g. `FFT` or `CGS` contain sub-directories as follows:

`bin` – binaries and module interfaces for this program;
`doc` – specifications chapter for this program;
`matrices` – example matrix input data;
`others` – various test routines which will not form part of the distribution;
`src` – source code for library routine;
`test` – test suite for the library routine;
`examples` – examples of using the library routine.

Not all these will be present in the library distribution, but in addition a master `makefile` and `README` will be present.

# BUILDING THE DISTRIBUTION

THe CLIPS library directory contains a master makefile to build the distribution and test suite. It copies the machine-dependent definitions from `arch` to the required directories `src` and `test` and invokes make for each set of subroutines required. Assuming we are using C-shell this would typically be as follows:

```
setenv TARGET AIX4 make others
```

THe portion of the makefile which builds contribution from the `OTHERS` directory is as follows:

```
others:
        @echo 'Making $(TARGET) distribution...'
        cp $(INCLDIR)/$(TARGET).make OTHERS/INCLUDE.make
        cp $(INCLDIR)/mpi.$(TARGET) OTHERS/mpi.f
        (cd OTHERS/src; make)
        cp OTHERS/src/$(MODS) include/.
        $(AR) lib/libclips.a OTHERS/src/*.o
```

A similar procedure applies to each directory and simply typing `make` will build the whole distribution.

## Makefiles

Each source and test directory contains its own makefile which will appear for instance as `CLIPS/NAME/src makefile`. The makefiles include a machine-specific portion `INCLUDE.make` which contains definitions of compilation flags, libraries, options and shows what modules and binaries will be copied to `CLIPS/include`, `CLIPS/lib` and `CLIPS/bin` which will form the basis of the distribution.

The makefile contains a list of dependencies for building the object code of a particular set of subroutines. This will assure that modules are compiled in the correct order.

Note that the routines in directory `OTHERS` must be built first. This is ensured by the master makefile.

## Preprocessing

Some files in the library contain CPP preprocessing directives. An example is the timing routine shown below. These files have an extension ".F". Generally CPP is used to pre-process these to convert them to ".f" which are then passed to the compiler with machine-dependent clauses included. In some cases changes may have been included in the makefile to permit this to happen, for instance on the Cray T3E the `f90` preprocessor is used which creates files with a ".i" extension. These are simply copied to ".f" files before compilation.

### Flags

Defined pre-processing flags are:

`TIME` – use timing internal routines;
`MPI` – enable MPI for parallel execution (default serial if meaningful);
`MPITIME` – use `mpi_wtime` for timing purposes;
`F90TIME` – use `system_clock` for timing;
`DTIME` – use `dtime` for timing;
`RTC` – use `rtc` for timing;
`CRAY` – use Cray architecture specific clauses.

## Fortran 90 Modules

Compilation procedures involving Fortran 90 modules are non-standard. All required user-accessible object files will be distributed in `CLIPS/lib` as a single archive file `libclips.a` and module interfaces in `CLIPS/include`.

The list of modules currently accessible, which are described in the following chapters, is:

| | | |
|---|---|---|
| ac_mod | clips_precision | mpi_mod |
| basics | clips_profiler | packed_matrix_mod |
| cg_control_mod | clips_ran_mod | parallel_fft |
| cg_finalize_mod | clips_timer | precon_mod |
| cg_initialize_mod | cpu_mod | qsort_mod |
| cg_mod | givetake_mod | reallocation |
| cg_solve_mod | halo_mod | struct_mod |
| cgs_mod | ilu_mod | time_mod |
| clips_err_mod | int2str_mod | type_matrix_mod |
| clips_error | mdotv_mod | type_precon_mod |
| clips_parfft | mpi | vsend_mod |

## The Library Archive File

Object modules built using the above procedure will be archived in a random-access library file `CIPS/lib/libclips.a`.

## Architectures

Specific details of the procedure for each architecture are now described.

### IBM RS/6000 with AIX

This architecture is specified using `setenv TARGET AIX4` or `setenv TARGET AIX4MPI`.

IBM compilers read ".f" ".F" or ".f90" files and produce ".o" and ".mod" files. The ".mod" files in the current directory or a directory referenced via the "-I" flag are parsed when a `USE` statement is met. A ".mod" file is therefore required for each user-accessible library module which must contain at least a list of explicit public interfaces. ".o" files are archived into `libclips.a`.

A typical user program compilation would be

xlf90 -o myprog.out myprog.f -I CLIPS/incude -L CLIPS/lib -lclips

### Cray T series

### SGI

### Compaq/Digital

### HP

### SUN

### Hitachi

The Hitachi Fortran 90 compiler takes files with a ".f90" extension. Only one module must be present in the file which must have the same name as the module. The source file is parsed when a `USE` statement is found. Only ".o" files are produced. For user-accessible modules a ".f90" file must be available in the distribution directory with at least a list of explicit public interfaces. ".o" files are archived into `libclips.a`.

### NEC

### Fujitsu

# SUBPROGRAMS REQUIRED

## Linear-Algebra Subprograms

Where necessary the CLIPS library routines will call the following subprograms to perform libear-algebra operations:

BLAS
LAPACK
BLACS
PBLAS
ScaLAPACK

Many computer systems have highly optimised proprietary versions of libraries to carry out linear-algebra operations and these should be used wherever possible. They are indicated in the appropriate makefiles.

## Message Passing Subprograms

The current library distribution uses MPI-1 (Message Passing Interface) which is the *de facto* standard. Future releases may contain calls to MPI-2.

We use a module rather than an include statement to access the MPI declarations. The module is however system dependent and not allways available. In such cases it is necessary to replace the line `USE mpi` by `include ''mpif.h''` in some library source files.

Linking to MPI is done via the appropriate makefile. This must be changed if MPI profiling is required. See the chapter on profiling.

## Utility Subprograms

### Timing Routine

Included through the `MODULE clips_timer` which is referenced from other library modules.

A variety of methods to measure elapsed time are available. The timing routine used in the library `clips_timer.F` is described in a separate section.

### Error Handling

Included through the `MODULE clips_error` which is referenced from other library modules.

See chapter on utility routines.

### Random Number Generator

Included through the `MODULE clips_random`.

See chapter on utility routines.

## DATA STORAGE

Each routine currently has its own data storage specification. However a number of well-known distributions which will be used are described here.

## Block-Cyclic Storage

## CLRC Technical Reports

A number of technical reports produced by CLRC contain material relevant to parallel numerical algorithm development. THey may provide background information interest to users of this library.

## REFERENCES

1.

2.

3.

4. R.J. Allan, Y.F. Hu and P. Lockey *A survey of parallel software packages of potential use in science and engineering applications* (Daresbury Laboratory, Edition 2, April 1999)

5. R.J. Allan, M.C. Goodman and R.R. Ward *Survey of Software Visualisation Packages for Parallel Data and Parallel Code Debugging* (Daresbury Laboratory HPCI Centre Report)

6.

7.

## ACKNOWLEDGEMENTS

The subroutines described in this library specification were developed by staff at the CLRC Daresbury Laboratory with other collaborators where noted under "Origins" in the summary of each subroutine. Funding was available through the CLRC HPCI Centre grant GR/K82635 (1/4/1994-30/10/1999) from EPSRC and a Service Level Agreement between EPSRC and the CLRC.

## CONDITIONS OF USE

# SIGNAL PROCESSING

# PARALLEL FFT

## SUMMARY

A Fortran 90 module implements a highly efficient 3-dimensional parallel multi-radix FFT routine for block-cyclically distributed data. Subroutine calls permit to initialise the package, print timing statistics and information on use and carry out normalised forward and backward 3D FFTs.

The module is written in standard conforming Fortran and MPI. The user can supply a single routine to interface to an optimised FFT routine, but there is a default driver for Temperton's GPF [?].

Though as yet not industrial strength, the code is fairly flexible in the data distribution allowing the user to specify the shape of the processor grid and the blocking factors. FFTs on subgroups of processors are possible. It has extensive, but not yet quite complete, error checking. It has been tested on IBM SP2, Cray T3D and T3E, and SUN Enterprise.

## ATTRIBUTES

**Version:** 1.0
**Public modules:** `MODULE clips_fft`
**Public calls:** `clips_fft_initialize`, `clips_fft_summarize`, `clips_pfft`
**Other modules required:** `mpi`, `clips_timer`, `clips_err_mod`
**Date:** 1998
**Origin:** I.J. Bush, CLRC Daresbury Laboratory
**Language:** Fortran 90
**Conditions on external use:** Standard, see separate chapter.

## HOW TO USE THE MODULE

This package is used through `MODULE clips_pfft`. The package is initialised for a new data distribution using `clips_fft_initialize`. The forward or backward FFT for a particular data distribution is invoked using `clips_pfft`. At any stage information may be printed using `clips_fft_summarize`.

The module uses the `MODULE clips_error` to handle errors. This is described in a separate chapter. It uses the `MODULE clips_timer` for internal timing purposes.

## SPECIFICATION OF CLIPS_FFT_INITIALIZE

Subroutine `clips_fft_initialize` initializes things like the communication pattern and data for the FFT.

```
  Subroutine clips_fft_initialize(n_dims, lengths, proc_grid, block, &
      communicator, context, error)
    Integer                     , Intent(In   ) :: n_dims
    Integer, Dimension(1:n_dims), Intent(In   ) :: lengths
    Integer, Dimension(1:n_dims), Intent(In   ) :: proc_grid
    Integer, Dimension(1:n_dims), Intent(In   ) :: block
    Integer                     , Intent(In   ) :: communicator
    Integer                     , Intent(  Out) :: context
    Integer, Optional           , Intent(  Out) :: error
```

### Argument List

`Integer, Intent(In) :: n_dims`
On entry: dimensionality of data on which FFT is to be performed. Currently must be 3D.

`Integer, Dimension(1:n_dims), Intent(In) :: lengths`
On entry: length of FFT in each dimension

`Integer, Dimension(1:n_dims), Intent(In) :: proc_grid`
On entry: number of processors in each dimension of processor grid

`Integer, Dimension(1:n_dims), Intent(In) :: block`
On entry: size of block in each dimension

`Integer, Intent(In) :: communicator`
On entry: MPI communicator that the processor grid will use. This is duplicated on entry to this routine and the duplicate communicator used for all internal communications.

`Integer, Intent(Out) :: context`
On exit: A new context is assigned to each initialisation call and is used to record data distribution and communication parameters set up in the call. Currently up to 10 contexts may be active. These can be used if several different data distributions are present in the application.

`Integer, Optional, Intent(Out) :: error`
On exit: error return value from `mpi_barrier` on the communicator. This reports if communications have been initialised correctly.

Processors are assigned a number of sections of data to work on such that in each dimension the following relation is satisfied:
$my_sections = lengths/(block * proc_grid)$.
my_sections and proc_grid must be powers of two in each dimension for every processor.

### Error Returns

status=-1
Called from CLIPS_FFT_INITIALIZE, one of the dimensions of the processor grid is not a power of two.

status=-2

Called from CLIPS_FFT_INITIALIZE, each processor must have a power of two number of blocks.

syayus=-3

Called from CLIPS_FFT_INITIALIZE, the blocks are not evenly divided amongst the processors.

status=-4

Called from CLIPS_FFT_INITIALIZE, there are insufficient processors to make up the processor cuboid status=alloc_stat

Called from CLIPS_FFT_INITIALIZE, failed to allocate memory for MY_GRID_POS when calling ALLOCATE.

status=alloc_stat

Called from CLIPS_FFT_INITIALIZE, failed to allocate memory for SET_UP_FFTS( CONTEXT )%DIMS when calling ALLOCATE

status=alloc_stat

Called from CLIPS_FFT_INITIALIZE, failed to allocate memory for SET_UP_FFTS( CONTEXT )%DIMS%MY_SEC_STARTS when calling ALLOCATE

status=alloc_stat

Called from CLIPS_FFT_INITIALIZE, failed to allocate memory for SET_UP_FFTS( CONTEXT )%DIMS%TRIGS when calling ALLOCATE

status=alloc_stat

Called from CLIPS_FFT_INITIALIZE, failed to allocate memory for SET_UP_FFTS( CONTEXT )%DIMS%TRIGS_CONJG when calling ALLOCATE

status=alloc_stat

Called from CLIPS_FFT_INITIALIZE, failed to allocate memory for SET_UP_FFTS( CONTEXT )%DIMS%TRIGS_SHORT chen calling ALLOCATE

status=alloc_stat

Called from CLIPS_FFT_INITIALIZE, failed to deallocate MY_GRID_POS chen calling DEALLOCATE

status=alloc_stat

Called from SET_COMMS, failed to allocate memory for COMMUNICATIONS%EXCHANGE when calling ALLOCATE

status=alloc_stat

Called from SET_COMMS, failed to allocate memory for COMMUNICATIONS%FIRST_HALF when calling ALLOCATE

status=alloc_stat

Called from SET_COMMS, failed to allocate memory for COMMUNICATIONS%TRIGS_OFFSET when calling ALLOCATE

## SPECIFICATION OF FFT_SUMMARIZE

Subroutine `fft_summarize` produces information relevant to a given processor for FFTs to be carried out within a given context. This information includes number of nodes, dimensionality of FFT, length, block,sections, local steps and times for each dimension and information about communications and total times for the FFTs carried out since initialisation.

```
Subroutine fft_summarize(processor, context)
  Integer, Intent(In) :: processor
```

```
    Integer, Intent(In) :: context
```

**Argument List**

`Integer, Intent(In) ::  processor`
On entry: processor about which information is required

`Integer, Intent(In) ::  context`
On entry: context about which information is required

**Information Returned to the User**

This to be added

## SPECIFICATION OF CLIPS_PFFT

Subroutine `clips_pfft` carries out the parallel 3D FFT computation within the given context and in a defined direction (forward or backward).

```
  Subroutine clips_pfft(a, work, context, direction)
    Integer, Parameter :: float = Selected_real_kind(6, 70)
    Integer, Parameter :: imag  = Kind((1.0_float, 1.0_float))
    Complex(), Dimension(0:, 0:, 0:), Intent(InOut) :: a
    Complex(), Dimension(0:, 0:, 0:), Intent(InOut) :: work
    Integer                           , Intent(In   ) :: context
    Integer                           , Intent(In   ) :: direction
```

**Argument List**

`Complex(imag), Dimension(0:, 0:, 0:), Intent(InOut) ::  a`
On entry: local contiguous sections of input data which this processor owns following the data distribution specified in `context`.
On exit: result of the requested operation in the same distribution.

`Complex(imag), Dimension(0:, 0:, 0:), Intent(InOut) ::  work`
On entry: local work array same size as local input array
On exit:

`Integer, Intent(In) ::  context`
On entry: the context specifying data distribution and communicator for this FFT as set up by the routine `clips_fft_initialize`

`Integer, Intent(In) ::  direction`
On entry: the direction of the FFT. Forward=1, backward/=1.

The result of the FFT is produced in the set of local arrays `a` over-writing the input data. The results are normalised by $1.0/\sqrt{(N)}$ where $N$ is the total number of data elements.

**Control parameters**

The direction of the transform is specified by the parameter `direction`.

## DATA DISTRIBUTION

The datatype for the complex arrays is defined as

```
Integer, Parameter ::  float = Selected_real_kind(6, 70)
Integer, Parameter ::  imag = Kind((1.0_float, 1.0_float)).
```

## GENERAL INFORMATION

**Workspace:**
**Use of common:** none
**Internal routines called directly:** In `clips_fft_initialize`: `set_trigs`, `set_comms`, `fft_error`. In `pfft`: `forward_3d_fft_x`, `forward_3d_fft_y`, `forward_3d_fft_z`, `back_3d_fft_x`, `back_3d_fft_y`, `back_3d_fft_z`. These are all contained routines with the PRIVATE attribute and are therefore inaccessible outside the fft module.
**Input/output:** messages are written to standard output on Fortran UNIT=6.
**Restrictions:** Number of processors must be a power of 2. Each processor must get a power of 2 sections to work on and an equal number of blocks of data.

### Future Work

There's quite a few things that still need to be done, when time permits.

- Most important: GET OFF MY FAT BUTT AND GET SOMEONE USING IT. As far as I know it is the only portable parallel 3D FFT routine aimed at MPPs ( before someone mentions FFTW look at the scaling, mines not great but ... )
- At present the data coming out of the transfom is in 'bit-twiddled' ( i.e. non-natural order ). this is not as serious as it sounds, and the back transform undoes it, but it should be adddressed.
- Implement the 1 and 2 dimensional cases. All the machinery is there so that would not be too difficult.
- Complete the error checking
- Remove as many as possible of the restrictions that the code currently places on the data and processor distributions.
- Optimisation: There are a large number of games that could be played. Serially there is still some work to do on the z, and to a lesser extent, the y transforms. That said it is probably more important to try to address the log(P) term in the communication. There are two possibilities:
  1. Given that there are so few latencies there it would be possible to pay a few extra and use non-blocking communications, allowing some overlap of computation and communication. However this won't benefit all machines, some are very poor at asynchronous comms.
  2. At the European Cray MPP Users workshop in Munich this year Oswald Haan presented an interesting talk on parallel 1D FFTs. It would be interesting to see how his method performs when used in the multidimensional case.

# METHOD

## Algorithmic detail

The underlying 1D FFT algorithm depends on which library routine is called by the `fft_module` driver routine. This is specified through the wrapper routines in file `rappers.F`. This enables user-defined wrapper routines to be added. General information about serial and parallel FFTs is available in a technical repor [1].

## Parallelism detail

### The Fast Fourier Transform

At first glance the Discrete Fourier Transform for a function 0 at n point
$f(k_p) = \Sigma_q f(r_q) e^{ik_p r_q}$
Takes $O(N^2)$ operations ( N for the sum, and there are N Values of $k_p$)
In fact an algorithm most often attributed to Cooley and Tuke [?] reduces the op count to $O(Nlog(N))$, though there is evidence that Gauss knew of it!
There are a large number of variants of the basic algorithm, some suiting certain computer architectures better than others. All (?) HPC manufacturers provide highly optimised serial FFTs, often for multiple 1D transforms and 2 and 3 dimensional transforms as well.

### The Difficulties in Implementing Scalable Parallel FFTs

- The FFT algorithm is 'global' - i.e. there is little data locality, and as a result one processor will have to 'see' data from most of the other processors.
- The FFT has such a low op count ! The ratio of the amount of the number of times a given piece of data is used to the number of times it has to be communicated is small.

We will concentrate from now on on FFTs in multiple dimensions (especially 2D for ease of illustration).

### The Traditional Transpose-based Parallel FFT

Well - we know how to do 1D serial transforms very efficiently, why not use that technology ? Consider a square 2D FFT on P processors, the data distributed such that for a given $y_i$ one single processor owns all the values of x. The method is then:

1. For each y value I hold FFT in the x direction using the best 1d library routine I can find.
2. Transpose the grid keeping the data distribution the same - i.e. If at step one I held all x for a given $y_i$, I now hold all y values for $x_i$
3. FFT along y ( same as step 1 )
4. Transpose again to regain the original distribution

In practice the last transpose is often superfluous, and so is not performed for efficiency reasons.

### The Cost of the Transposed Based Parallel FFT

There are two operations required:

1. The 1D FFTs. Assuming a square grid of order N for the above 2D case each processor has to perform $2 \times N/P$ FFTs of order N. Thus for the D dimensional case the cost is $\tau_{fft} = c_{fft} D N^D log(N)/P$ which simplifies to $\tau_{fft} = c_{fft} V log(V)/P$

2. The transpose of the distributed data. This is a little subtle ....

**Transposes in Parallel FFTs**

There are two steps:

1. Obviously the data has to be sent, it's and all to all operation with each processor sending 1/P of it's data to each of the other processors. In D Dimensions: $\tau_{comms} = (D-1)P(\alpha + V\beta/P^2)$

2. Message passing systems expect data that is stored linearly in memory, the data to be sent in step 1 is not so stored ! Hence he also have to pack some message buffers: $\tau_{pack} = c_{pack} V/P$

**Comments of the Cost of Transpose Based FFTs**

$\tau_{trans} = c_{fft} V log(V)/P + (D-1)(P(\alpha + V\beta/P^2) + c_{pack} V/P)$

- The 1D FFTs are done efficiently and perfectly in parallel - GOOD.

- The comms for large P requires a lot of short messages, e.g. consider a $256^3$ 3D FFT on 256 processors, each message is 256 complex numbers long ( 4 kbytes ). This is below $n_{1/2}$ on most machines, so for large P the method rapidly becomes latency bound - BAD.

- The message buffer packing can require memory access patterns that are horribly non-local which, as we all know, slows our 1.2 GFlop Risc chip down to a few 10s of Mflops, and in fact the packing time can be comparable with the time our wonderfully efficient 1d serial FFTs take ! At least it's perfectly parallel, but basically BAD.

**An Alternative Approach**

So these transposes kill us at high processor counts, so why do them ? Why not do it more like a standard serial 2d FFT, but in parallel: Do in PARALLEL all the FFTs in the x direction, then in PARALLEL do all the FFTs in the y direction.

First consider doing a forward 1D parallel FFT of length N on P processors. The data will be distributed such that each processor has N/P contiguous elements of the data. There are Log(N) steps to the FFT, each costing O(N) operations:

- For the first log(P) steps as well as the O(N) ops a given pair of processors have to exchange all their data. Hence the cost is $\tau_{comms} = log(P)(\alpha + N\beta/P + c_{comms} N/P)$

- For the remaining Log(N)-Log(P) steps NO comms are required. In fact careful examination of the algorithm shows that these steps are simply a 1d FFT of length N/P ! This means that the highly optimized serial library routine can be used again, and that the cost is $\tau_{fft} = c_{fft}(N/P)log(N/P)$

### A Blocked Multidimensional FFT

So for a one dimensional case

$\tau_{blocked} = log(P)(\alpha + N\beta/P + c_{comms}N/P) + c_{fft}(N/P)log(N/P)$

From the one dimensional case the way to move towards the multidimensional case is to arrange our processors into an nD grid. Assuming the FFT to be the same length in each dimension, and equal numbers of processors along each dimension of the grid, the above formula generalizes to

$\tau_{blocked} = log(P)(\alpha + V\beta/P + c_{comms}V/P) + c_{fft}(V/P)log(V/P)$

### Comparing the Transpose and Blocked Methods

Let's compare corresponding terms in the timing equations for the two methods:

$\tau_{blocked} = log(P)(\alpha + V\beta/P + c_{comms}V/P) + c_{fft}(V/P)log(V/P)$

$\tau_{trans} = (D-1)(P(\alpha + V\beta/P^2) + c_{pack}V/P) + c_{fft}V\,log(V)/P$

- Communications:
  - Latencies: The blocked method sends a lot fewer messages than the transpose method.
  - Bandwidth: The blocked method sends fewer, larger messages than the transpose method, but the total amount of data sent is larger. However the factor is only proportional to the log of the number of processors, and should not be important for large processor counts.
- The call to the FFT library
  - The calls to the FFT library appear very similar, but see later ...

So in conclusion one would expect the blocked method to be better at large processor counts, but at smaller P the log(P) term in the comms may be growing fast enough to favour the transpose method. But how large is large ?

So how do they compare ? Results are taken from a typical example with a 3D grid of size $25 \times 120 \times 240$ run on a 160 MHz IBM SP2-SC.

| Processors | transpose | Block |
|---|---|---|
| 2 | 55991.9 | 43083.1 |
| 4 | 28007.5 | 25209.6 |
| 8 | 14026.8 | 14441.5 |
| 16 | 7059.6 | 8142.3 |
| 32 | 3622.2 | 4535.7 |
| 64 | 1942.5 | 2504.4 |
| 128 | 1312.2 | 1375.3 |
| 256 | 1395.2 | 754.5 |
| 512 | 2175.3 | 416.6 |

An interesting tale! Very low P favours the blocked method, probably because no message packing is required, at moderate P the log(P) term is growing fast enough to favour the transpose method, but as expected blocking finally wins out at very high P.

### And it all seemed so simple ....

To achieve the above performance another step must be taken, and it's those supposedly optimised library routines that are the problem.

In the transpose method the FFTs are always stride one. For the blocked multidimensional case this is no longer true, except for the $x$ direction. In particular for a 3D case the transform in the $z$ direction can have very large strides, and the “optimized” FFT routines (IBM's in ess [?], Cray's in LibSc [?], Temperton's GPF [?], Schwartztrauber's FFTpac [?]) can't cope with this, the performance often drops by over an order of magnitude.

So what to do ?

**Borrowing an idea from linear algebra**

Something similar happens in implementing a matrix multiply. Once the amount of data exceeds the size of the cache the performance can drop markedly. The way around this is to do the matrix multiply in cache sized “chunks”, a process known as strip mining.

We can play the same game here by moving from a blocked distribution, which gives the processors just one big chunk of the FFT grid, to block cyclic distribution. This gives each processor a number of smaller parts of the grid, the size of which are controlled by the “blocking factor”. Thus a careful choice of the blocking factor can be used to improve the performance of the routine. To people in the know, the resulting data distribution corresponds to the generalization to n dimensions of that used by ScaLAPACK.

## EXAMPLE

### Example text

```
      PROGRAM test_parallel
      USE mpi
      USE clips_pfft
      USE clips_timer
      IMPLICIT NONE
      INTEGER, PARAMETER :: nrep=1
      COMPLEX(8), DIMENSION(:,:,:), ALLOCATABLE :: a, work, orig
      COMPLEX(8), DIMENSION(:), ALLOCATABLE :: tmp,tmp2
      INTEGER nnode, inode,np,context, ier
      INTEGER i, j, k, sec, m, n,n_dims, jump, start
      INTEGER, DIMENSION(1:3) :: sections,len
      INTEGER, DIMENSION(1:13) :: ifax
      INTEGER, Allocatable, Dimension(:) :: lengths, block, proc_grid, &
     &         kx, ky, kz, bit_index, bit_work
      INTEGER :: n_groups, length,offset, cut, beta,len_sec, n_secs, base_sec
      REAL(8) arg, elapsed1, elapsed2,v, pi,ops,av
      INTEGER(8) begin, finish,begin2, finish2
      pi=4.0_8*atan(1.0_8)
! initialise MPI
      CALL mpi_init(ier)
      CALL mpi_comm_size(mpi_comm_world, nnode, ier)
      CALL mpi_comm_rank(mpi_comm_world, inode, ier)
! read problem data and broadcast
      IF(inode == 0)THEN
        OPEN(5,file='daft.con',status='old')
        WRITE(6,*) 'dims'
```

```
    READ(5, *) n_dims
  ELSE
    OPEN(6,file='node'//char(ichar('0')+inode))
  END IF
  CALL mpi_bcast(n_dims, 1, mpi_integer, 0, mpi_comm_world, &
 &     ier)
  ALLOCATE(lengths(1:n_dims),STAT=ier)
  ALLOCATE(block(1:n_dims),STAT=ier)
  ALLOCATE(proc_grid(1:n_dims),STAT=ier)
  DO WHILE(.TRUE.)
  IF(inode == 0)THEN
    READ(5,*,END=999) proc_grid
    WRITE(6,*) 'grid'
    WRITE(6,*) proc_grid
    READ(5,*) lengths
    WRITE(6,*) 'length'
    WRITE(6,*) lengths
    READ(5,*) block
    WRITE(6,*) 'block'
    WRITE(6,*) block
  END IF
  CALL mpi_bcast(lengths, n_dims, mpi_integer, 0, mpi_comm_world, &
 &     ier)
  CALL mpi_bcast(block, n_dims, mpi_integer, 0, mpi_comm_world, &
 &     ier)
  CALL mpi_bcast(proc_grid, n_dims, &
 &                mpi_integer, 0, mpi_comm_world, ier)
  np=PRODUCT(proc_grid)
  WRITE(6,'('' np, nnode = '',2i4)')np,nnode
  IF(np/=nnode)CALL clips_error(1,'test','wrong number of procs')
  ALLOCATE(tmp (1:lengths(1)),STAT=ier)
  ALLOCATE(tmp2(1:lengths(1)),STAT=ier)
  len=lengths/proc_grid
  ALLOCATE(kx(1:len(1)),STAT=ier)
  ALLOCATE(ky(1:len(2)),STAT=ier)
  ALLOCATE(kz(1:len(3)),STAT=ier)
  ALLOCATE(bit_index(0:len(1)-1),STAT=ier)
  ALLOCATE(bit_work (0:len(1)-1),STAT=ier)
  ALLOCATE(a(1:len(1), 1:len(2), 1:len(3)),STAT=ier)
  CALL clips_error(ier,'test','insufficient memory')
  ALLOCATE(orig(1:len(1), 1:len(2), 1:len(3)),STAT=ier)
  CALL clips_error(ier,'test','insufficient memory')
  ALLOCATE(work(1:len(1), 1:len(2), 1:len(3)),STAT=ier)
  CALL clips_error(ier,'test','insufficient memory')
  CALL fft_initialize(n_dims, lengths, proc_grid, block,  &
 &     mpi_comm_world, context, ier)
  sections=lengths/(block*proc_grid)
  jump=block(1)*proc_grid(1)
  start=inode*block(1)
  k=1
  DO sec=1, sections(1)
    DO j=start, start+block(1)-1
      kx(k)=j
```

```
          k=k+1
        END DO
        start=start+jump
      END DO
      DO i=0, len(1)-1
        bit_index(i)=i
      END DO
      beta=INT(LOG(REAL(sections(1), 8)+0.1_8)/LOG(2.0_8))
      length=lengths(1); len_sec=length
      n_secs =1
!$$$      Do cut=1, beta
!$$$         offset=len_sec/2
!$$$         Do i=0, n_secs-1
!$$$            base_sec=i*len_sec
!$$$            Do j=0, len_sec-2, 2
!$$$               bit_work(base_sec+j/2)=bit_index(base_sec+j)
!$$$            End Do
!$$$            Do j=1, len_sec-1, 2
!$$$               bit_work(base_sec+j/2+offset)= &
!$$$    &             bit_index(base_sec+j)
!$$$            End Do
!$$$         End Do
!$$$         bit_index=bit_work
!$$$         len_sec=len_sec/2
!$$$         n_secs=n_secs*2
!$$$      End Do
      jump=block(2)*proc_grid(2)
      start=inode*block(2)
      k=1
      DO sec=1, sections(2)
        DO j=start, start+block(2)-1
          ky(k)=j
          k=k+1
        END DO
        start=start+jump
      END DO
      jump=block(3)*proc_grid(3)
      start=inode*block(3)
      k=1
      DO sec=1, sections(3)
        DO j=start, start+block(3)-1
          kz(k)=j
          k=k+1
        END DO
        start=start+jump
      END DO
      a=0.0d0
! set up some data
      DO i=1, len(3)
        DO j=1, len(2)
          DO k=1, len(1)
!               arg=(2.0*kx(k)+3.0*ky(j)+kz(i))
            arg=SIN(3.0_8*2.0_8*pi*kx(k)/lengths(1))
```

```
!               arg=arg*SIN(2.0_8*pi*ky(j)/lengths(2))
!               arg=arg*SIN(2.0_8*pi*kz(i)/lengths(3))
             a(k, j, i)=arg
           END DO
         END DO
       END DO
! save it
       orig=a
!       tmp=(0.0_8, 0.0_8)
!       Do i=1, len(1)
!          Write(6,*) i, len(1), kx(i)
!          tmp(kx(i)+1)=a(i, 1, 1)
!       End Do
       CALL mpi_allreduce(tmp, tmp2, 2*lengths(1), &
      &     mpi_real8, mpi_sum, mpi_comm_world, ier)
!       CALL mpi_barrier(mpi_comm_world, ier)
! repeat FFT nrep times and save elapsed time
       elapsed1=0.0d0; elapsed2=0.0d0
       Do k=1, nrep
         begin=clips_time()
         CALL pfft(a, work, context, 1)
         elapsed1=elapsed1+clips_time()-begin
!          CALL mpi_barrier(mpi_comm_world, ier)
         begin=clips_time()
         CALL pfft(a, work, context, -1)
         elapsed2=elapsed2+clips_time()-begin
!       CALL mpi_barrier(mpi_comm_world, ier)
       END DO
       DO i=0, nnode-1
         CALL fft_summarize(i, context)
       END DO
       WRITE(6,'('' max error = '',g12.5)') MAXVAL(ABS(a-orig))
       elapsed1=elapsed1/REAL(nrep,8)
       elapsed2=elapsed2/REAL(nrep,8)
       av=0.5d0*(elapsed1+elapsed2)
       WRITE(6,'('' times forward, reverse, av = '',3g12.5)') &
      &     elapsed1, elapsed2,av
       v=PRODUCT(lengths)
       ops=5.0d-6*v*LOG(v)/log(2.0)
       WRITE(6,'('' Mflop/s forward, reverse, av = '',3g12.5)') &
      &     ops/elapsed1,ops/elapsed2,ops/av
!       CALL mpi_barrier(mpi_comm_world, ier)
       DEALLOCATE(tmp,tmp2,a,orig,work,kx,ky,kz,bit_index,bit_work)
       END DO
 999   WRITE(6,*) 'end of data reached'
       DEALLOCATE(lengths,block,proc_grid)
       CALL mpi_finalize(ier)
       END PROGRAM test_parallel
```

## Example data

Initial array set to (1.0,0.0) in test program. This can be changed. Size and shape of arrays is read from a file "daft.con" on Fortran UNIT=5.

### Example results

Array transformed forward and backward should be same as input array. Elements are sampled in the test program. The time taken to perform the transforms and an estimate of Mflop/s performance is printed on UNIT=6.

## REFERENCES


1. R.J.Allan, I.J.Bush, D.S.Henty, T.Bush, K.Takeda and O.Haan *Serial and Parallel Fast Fourier Transforms* (Daresbury Laboratory, 1999).
2. R.C. Agarwal, F.G. Gustavson and M. Zubair *A High Performance Parallel Algorithm for 1D FFT* Proc. Supercomputing (1994) 34-40
3. E.O. Brigham "The Fast Fourier Transform" (Prentice Hall, 1974)
4. P.N. Swarztrauber, *Vectorizing the FFTs, in Parallel Computations*, ed. G. Rodrigue (Academic Press, 1982) 51-83.
5. P.N. Swarztrauber, *Multiprocessor FFTs* Parallel Computing 5 (1987) 197-210
6. C. Temperton, *Self sorting mixed-radix fast Fourier transforms*, J. Comp. Phys. 52 (1983) 1-23.
7. O. Haan *A Parallel One-dimensional FFT for Cray T3E* Proceedings of the 4th European Cray MPP Workshop, Garching, 1998. See URL *http://www.rzg.mpg.de/mpp-workshop/papers/ipp-report.html*
8. C.Temperton, *A Generalised Prime Factor FFT Algorithm for any N = (2**P)(3**Q)(5**R)*, SIAM J. Sci. Stat. Comp. (May 1992).
9. J.W. Cooley and J.W. Tuckey *An algorithm for the machine calculation of complex Fourier series* Math. Comp. 19 (1965) 297-301

## LINEAR ALGEBRA

# PARALLEL STABILISED CONJUGATE GRADIENT

## SUMMARY

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

## 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.

## HOW TO USE THE PACKAGE

This package is used through `MODULE clips_cgs`.

More to follow

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

## 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 factorization of fill-in zero by default (can be turned off);
d) set up control parameters for `clips_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_initialize(comm,nn,nz,irn,jcn,val,rhs,single_file, &
     my_nstart,my_level,my_nprecon,my_tol,my_maxit)
  integer :: comm
  integer (kind=myint) :: nn,nz
  integer (kind=myint), pointer :: irn(:)
  integer (kind=myint), pointer :: jcn(:)
  real (kind = myreal), pointer :: val(:)
  real (kind = myreal), 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
  double precision, intent(in), optional :: my_tol
```

### Argument List

`integer, intent (in) :: comm`
On entry: the communicator for MPI.

`integer (kind=myint), 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 (kind=myint), pointer :: irn(:)`
On entry: the array of non-zero row indices.

`integer (kind=myint), pointer :: jcn(:)`
On entry: the array of non-zero column indices.

`real (kind = myreal), 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 (kind = myreal), 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.

- If nstart=0, cold start, scheduling and ILU factorization (when nprecon=1) will be performed;
- If nstart=1, warm start, that assume that the sparse structures are unchanged, scheduling will not be calculated again, but ILU factorization will be recalculated;
- If nstart>=1, hot start, then LU factorization and scheduling is assumed to be known and will not be recalculated

`integer, intent(in), optional :: my_level` On enry: 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

`double precision, 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

## 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 (kind = myreal), target :: x(:)
    integer, intent (out) :: iflag
```

### Argument List

`real (kind = myreal), 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`
error flag from cg: 0 for successful solve, 1 for exceeding maximum iterations.

### Errors and Warnings

`clips_cg_solve` returns with an error flag `iflag`, see the section on Arguments.

## SPECIFICATION OF CLIPS_CG_SUMMARIZE

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

```
  subroutine clips_cg_summarize()
```

### Argument List

There are no arguments.

## SPECIFICATION OF CLIPS_CG_FINALIZE

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

```
  subroutine clips_cg_finalize()
```

### Argument List

There are no arguments.

## GENERAL INFORMATION

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

## METHOD

to be supplied later

### Algorithmic detail

### Parallelism detail

## EXAMPLE

### 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, as follows:

```
program test
  use clips_cg_mod
  use read_arg_mod

  ! the matrix and right-hand-side.
  integer (kind=myint) :: nz
  integer (kind=myint) ::  n
  integer (kind=myint), pointer :: irn(:)
  integer (kind=myint), pointer :: jcn(:)
  real (kind = myreal), pointer :: val(:)
  real (kind = myreal), pointer :: rhs(:)

  ! initial guess and the solution
  real (kind = myreal), pointer, dimension (:) :: x

  ! input file name
  character (len=60):: infile

  ! input unit and whether it is a single file or
  ! in nproc files
  integer (kind=myint) :: input_unit
  logical single_file
```

```
! working int
integer (kind=myint) :: i

! error flag from cg
integer (kind=myint) :: 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_initialize(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
```

### Example Data

As an illustration, consider solving a $4 \times 4$ 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 4 processors can be done as follows:

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

Alternatively you can distribute the matrix into 4 horizantally slices 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
```

### 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
```

## REFERENCES

# ONE-SIDED BLOCK FACTORED JACOBI EIGENSOLVER

## SUMMARY

An implmentation of the one-sided block Jacobi iterative eigen-solver algorithm as described by Maschhoff and Littlefield [4]. Some improvements to both algorithms were made in order to test performance and those improvements are described.

## ATTRIBUTES

**Version:** 2.0
**Public calls:** `clips_jacobi`
**Public modules:** `clips_bfg`
**Other modules required:**
**Date:** 1999
**Origin:** I.J. Bush and A.G. Sunderland, CLRC Daresbury Laboratory
**Language:** Fortran 90
**Conditions on external use:** Standard, see separate chapter

The original version of the BFG parallel Jacobi algorithm was written by I.J. Bush in Feburary 1995.

## HOW TO USE THE PACKAGE

## SPECIFICATION OF CLIPS_JACOBI

This routine solves the eigenvalue problem

**GV = VE**

for real symmetric **G**.

The matrices are distributed by blocks of columns. The only restriction on the "width" of the blocks is that each processor *must* have at least two columns.

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

```
INTEGER          ldg
DOUBLE PRECISION G(1:ldg, 1:ncols)
INTEGER          ldv
DOUBLE PRECISION V(1:ldv, 1:ncols)
LOGICAL          initialize
DOUBLE PRECISION tolerance
INTEGER          nprocs
INTEGER          map(1:nprocs, 1:3)
LOGICAL          rank
INTEGER          rank_array(1:ncols)
EXTERNAL         global_sum
INTEGER          iterations
```

**Argument List**

`Integer, Intent(In) ::  status`

**INTEGER N**
On entry: the order of G

**INTEGER NCOLS**
on entry: the largest number of columns held on any processor+1;
on exit: how many evals returned on this processor

**INTEGER LDG**
on entry: must be equal to N in this version

**DOUBLE PRECISION G(1:ldg,1:ncols)**
on entry: the portion of the matrix held locally;
on exit: the G(i,i) elements, 1 <= i <= NCOLS hold evals, the rest is destroyed

**INTEGER LDV**
on entry: must be equal to N in this version

**DOUBLE PRECISION V(1:ldv,1:ncols)**
on entry: if INITIALIZE is TRUE holds a guess at the evecs. If INITIALIZE is FALSE not required to be set;
on exit: the i'th column holds the evec coressponding to eval G(i,i)

**LOGICAL INITIALIZE**
see V

**DOUBLE PRECISION TOLERANCE**
on entry: the tolerance on the off diagonal elements for rotations to be skipped. If negative on input set to 1.0d-8

**INTEGER NPROCS**
on entry: the number of processors to be used

**INTEGER MAP(1:nprocs,1:3)**
on entry: describes the mapping of the columns of G onto the processors:
MAP(i,1): The PE number of processor i
MAP(i,2): The first column held by processor i
MAP(i,3): The number of columns held by processor i 1 ¡= i ¡= nprocs

**LOGICAL RANK**
on entry: if TRUE on output RANK_ARRAY(i) will contain the rank of eval G(i,i) globally. A value of 1 corresponds to the smallest eval.

**INTEGER RANK_ARRAY(1:ncols)**

see RANK

**EXTERNAL GLOBAL_SUM**
Performs a sum of the elements of ARRAY across the processors held in MAP. An external user-supplied routine wor reverse calling, with syntax:

```
CALL GLOBAL_SUM(array,size_array,map)
      REAL    array(1:size_array)
      INTEGER size_array
```

**INTEGER ITERATIONS**

**External routines called**

DOUBLE PRECISION ddot EXTERNAL ddot
EXTERNAL dcopy
EXTERNAL dgemv
EXTERNAL esbuffers_init
EXTERNAL esinfo_send, esinfo_recv
EXTERNAL esbuffer_send, esbuffer_recv
EXTERNAL esmat_send, esmat_recv

## GENERAL INFORMATION

**Workspace:**
**Use of common:** none
**Other routines called directly:**
**Input/output:**
**Restrictions:**
**Notes:**

## METHOD

There are now a number of parallel implementations of eigensolvers for dense symetric matrices. Some of these were evaluated in separte reports [1, 2] and information is also available from the World Wide Web[5]. In this paper we concsntrate on the new one-sided Jacobi solver BFG, referring only to other packages for comparison of performance.

The iterative Jacobi method works by repeated application of orthogonal Jacobi rotations, which, depending on the order of application, bring the matrix more or less quickly to diagonal form. The original method used by Jacobi in 1846 [6] applied the rotations to zero the largest non-zero off-diagonal element, but on a computer a pre-designed pattern is easier.

It is also well known that iterative eigensolvers converge faster for more diagonally matrices, quite simply there are less iterations to achieve the required accuracy. This makes them particularly suitable to applications which are of a 'perturbative' nature; that is, a solution is already known to a closely similar problem. This is the case in certain classes of optimisation problem where a solution is followed in time or in configuration space. It may be possible to apply the already know similarity transform to bring the matrix to near-diagonal form and then continue with a few more Jacobi rotations.

### The serial two-sided Jacobi

On serial machines the two-sided algorithm is simply:

$$\mathbf{A}_{k+1} = \mathbf{J}_k^T \mathbf{A}_k \mathbf{J}_k \tag{1}$$
$$= \mathbf{V}_{k+1}^T \mathbf{A} \mathbf{V}_{k+1} \tag{2}$$

The eigenvector matrix is just the product of all the Jacobi matrices. This is true when a two-sided similarity transform is used but a one-sided variant is known to be more efficient in a parallel implementation, see [4]. A one-sided tridiagonalisation, rather than the usual Householder reflection, was described by Hegland [3].

Direct implementation of the two-sided algorithm on a parallel machine can cause problems, e.g.
Pre-multiplication by the rotation matrix updates the *rows* of **A**, but
post-multiplication by the rotation matrix updates the *columns* of **A**.

So there is no natural data distribution which fits both the multiplications. This can result in a lot of communications. A one-sided method is likely to be better for this reason.

In the sequential implementation rotations are typically skipped if the value to which they would be applied is less than a threshold. This is however difficult to schedule in parallel.

### A parallel two-sided Jacobi code – the Pittsburgh PJAC routine

The PJAC two-sided parallel Jacobi algorithm is available from the University of Pittsburgh [7]. It is currently able to handle complex symmetric or Hermitian matrices as well as real symmetric ones. This diagonaliser routine was developed from a project code written for the 1994 Institute on Parallel Computing. The algorithm employed is essentially a hybrid of Jacobi methods. Problem sizes ranging from n=2 to 8192 have been solved.

The PJAC library provides a scalable eigensolver for real/complex, symmetric/Hermitian matrices. File handling routines for creating input systems and viewing output systems are bundled with the solver.

The algorithm employed is again a block-factored Jacobi, but this time the traditional two-sided variety. The data distribution is by square blocks, each of which must be the same size. This requires the order of the matrix to be an exact multiple of the number of nodes used to diagonalise it, which could be inconvenient.

The parallel Jacobi method described by Brent and Luk assumes the existence of a two-dimensional grid of $(n/2)^2$ processors to resolve an n$\times$n matrix; i.e. the input system is mapped onto a collection of 2$\times$2 elements. A variation of this method featuring virtualised elements was implemented first.

Performance and scaling of the virtualised Brent-Luk method did not work well for larger problems on the Cray T3D. Essentially, the grain size associated with the algorithm was found to be too small. Time spent fetching data points could not be offset by cache reuse.

This led the authors to work on a generalisation of the Brent-Luk work in which the grain size was made a function of n and the processor count. As the processor count ranges from 1 to n/2, their macro-element Jacobi method varies from the serial Jacobi method to the Brent-Luk parallel model. Good performance and scaling was then observed for a fairly wide range of element dimensions.

Complete documentation and examples are available on the World Wide Web server at Pittsburgh Supercomputing Center
`http://www.psc.edu/general/software/packages/pjac/pjac.html` .
See also [7].

## A new one-sided Jacobi code

A new implementation of a one-sided Jacobi algorithm is now described.
Following Littlefield and Maschhoff [4] we can factor $\mathbf{A}_k$ as

$$\mathbf{A}_k = \mathbf{G}_k \mathbf{V}_k \tag{3}$$

where

$$\mathbf{G}_k = \mathbf{V}_k^T \mathbf{A} \tag{4}$$

Then at each step, given a rotation matrix $\mathbf{J}_k$, we can update the two factors by

$$\mathbf{G}_{k+1} = \mathbf{J}_k^T \mathbf{G}_k \tag{5}$$

$$\mathbf{V}_{k+1} = \mathbf{V}_k \mathbf{J}_k \tag{6}$$

This does not involve the explicit formation of $\mathbf{V^T A}$ from $\mathbf{A}$ at each step but only a rotation of rows of $\mathbf{G}$. Remaining off-diagonal elements of $\mathbf{A}_k + 1$ must be eliminated as we go along.

To calculate $\mathbf{J}$ to eliminate say $A_{pq}$ we calculate

$$A_{pq} = \mathbf{G}_{p\bullet} \cdot \mathbf{V}_{\bullet q} \tag{7}$$

and then use the serial technique to get the rotation angle.

## Advantages of the method

The updates for eliminating $A_{pq}$

$$\mathbf{G}_{k+1} = \mathbf{J}_k^T \mathbf{G}_k \tag{8}$$

$$\mathbf{V}_{k+1} = \mathbf{V}_k \mathbf{J}_k \tag{9}$$

act *only* on
*rows* p and q of $\mathbf{G}_k$ and
*columns* p and q of $\mathbf{V}_k$.

Threrefore steps taken are:

1. distribute $\mathbf{G}$ by rows and $\mathbf{V}$ by columns – all local eliminations require no communications;

2. each processor can then work simultaneously on eliminating different elements as the rotations don't interfere with each other.

The only other required component for the implementation is an efficient shuffling scheme for **G** and **V** between the processors such that each element of **A** is eliminated once per sweep.

The one-sided Jacobi method requires less communications than the traditional two-sided Jacobi or Householder reduction methods enabling it to scale better. It however requires of the order $n^2$ extra dot-products over and above the requirements for the two-sided Jacobi.

Furthermore serial Jacobi diagonalisers are normally around three times slower than those using Householder techniques. Thus the serial speed of a one-sided Jacobi is known to be poor unless the matrix is nearly diagonal at the start.

The performance of this diagonaliser is thus a trade off between better scaling and poor single-node speed. To be practical it must scale much better than the other methods. This is further discussed below

**The algorithm**

Initialize $\mathbf{G}_{p\bullet}$ and $\mathbf{V}_{\bullet p}$ for all p assigned to this processor
Split the set of p's owned by this processor into Group I and Group II
Loop to convergence
  – First do the Eliminations
  For all pairs of p and q selected from Group I eliminate $A_{pq}$
  For all pairs of p and q selected from Group II eliminate $A_{pq}$
  For round = 1, ... $2{\times}p$ - 1
    If Group I > Group II
      Loop over all p's in Group I
        Loop over all q's in Group II
          Eliminate $A_{pq}$
        End Loop over Group II
      End Loop over Group I
    Else If Group II >= Group I
      Perform above loops switching Groups I and II
    Endif
    – Now do the Shuffling
    If round is even and processor number = round / 2
      Send/receive Group II i.e. Send Group II to the
      next processor and recieve a new Group II from the
      previous one
    Else
      Send/receive Group I
    Endif
  End For round
  Check Convergence
End Loop to Convergence

**The elimination step** This is fairly standard apart from the dot products

– Reconstruct Required Elements of **A**

$A_{pq} = \mathbf{G}_{p\bullet} \cdot \mathbf{V}_{\bullet q}$
$A_{pp} = \mathbf{G}_{p\bullet} \cdot \mathbf{V}_{\bullet p}$
$A_{qq} = \mathbf{G}_{q\bullet} \cdot \mathbf{V}_{\bullet q}$
– Compute Rotation Angle
$\theta = (a_{qq} - a_{pp})/(2a_{pq})$
$t = sgn(\theta)/(\mid \theta \mid + \sqrt{(\theta^2 + 1)})$
$c = 1/\sqrt{(t^2 + 1)}$
$s = tc$
– Update G and V
For i = 1, ... order of Matrix
  $T = G_{pi}$
  $G_{pi} = cT - sG_{qi}$
  $G_{qi} = sT + cG_{qi}$
  $T = V_{ip}$
  $V_{ip} = cT - sV_{iq}$
  $V_{iq} = sT + cV_{iq}$
End For i

**Optimisations** Several optimisations are possible:

1. rather than store $\mathbf{G}$, store $\mathbf{G}^T$ – unit strides are then possible in the dot products and update loop;

2. combining the two elimination steps outside the `round` loop with the `round = 1` iteration gives a standard Jacobi on the local data. Since this corresponds to a square block on the diagonal of the matrix the symmetry of the matrix can then be exploited;

3. the dot products to generate the diagonal elements of $\mathbf{A}$ can be done outside the elimination loops and these elements can be updated as the process proceeds. This saves two dot products in the elimination step;

4. suppose the outer loop in the elimination stage is over p. Then once all the updates on $\mathbf{G}_{p\bullet}$ and $\mathbf{V}_{\bullet p}$ have been done they can be sent onto the next processor. Even better this can be done asynchronously while step p + 1 is being performed. On a low latency machine (e.g. the T3D) this was found to improve scalability.

All these improvements have been included in the current version of the code.

The data is distributed by blocks of columns. A mapping array allows different processors to hold different numbers of columns, but the routine assumes that the block allocated to a processor is at least two elements wide. The solver will also optionally rank the eigenvalues, and the user may provide an initial guess at the eigenvectors (useful if a solution to a quite similar matrix is known).

The algorithm as it stands requires that at least two columns of the matrix are stored on each processor. This may cause problems with small matrices on large MPP machines. For instance on a 256 processor machine the smallest matrix you can diagonalise by this method is order 512.

It was however found in the original pape [4] that the BFJ method is the most suitable for solving problems where the number of processors $p$ is comparable with the order of matrix n.

The way to implement it is also described in the paper. A secondary layer of distribution is required such that the columns of the matrices are themselves distributed over a number of processors. About the only extra complication is that the dot products have to be distributed. This extra stage has not, as yet, been implemented.

## EXAMPLE

## REFERENCES

1. R.J. Allan and I.J. Bush *Parallel Application Software on High Performance Computers: Parallel Diagonalisation Routines* (CLRC, Daresbury Laboratory, 1996). Available from URL `http://www.cse.clrc.ac.uk/Activity/HPCI`

2. J. Demmel and K. Stanley *The Performance of finding eigenvalues and eigenvectors of dense symmetric matrices on distributed-memory computers* Preprint (University of California, Berkeley, 1994)

3. M.Hegland *A distribution independent algorithm for the reduction to tridiagonal form using one-sided rotations* in Proc. 1st IEEE Int. Conf. on Algorithms and Architectures for Parallel Processing (IEEE, Computer Society Press, 1995) pp286-9

4. R.J. Littlefield and K.J. Maschhoff *Investigating the performance of parallel eigensolvers for large processor counts* Theor. Chim. Acta 84 (1993) 457-73

5. K. Stanley *Parallel Symmetric Eigensolver Web Page* `http://www.cs.berkeley.edu/\~stanley/symeig/index.html`

6. Jacobi 1846

7. The PJAC library was written by D.C. O'Neal and R. Reddy, PSC Parallel Applications Group. A paper describing this work was presented at the Fall 1994 Cray User Group Conference in Tours, France. Copies may be obtained by contacting the authors at `oneal@psc.edu` or `rreddy@psc.edu`. See `http://www.psc.edu/general/softwaren/packages/pjac/pjac.html`

8. G. Fann and R.J. Littlefield *Parallel inverse iteration with reorthogonalisation* in Proc. 6th SIAM conference on Parallel Processing for Scientific Computing (SIAM, 1993) pp409-13

9. D. Elwood, G. Fann and R. Littlefield *Parallel Eigensolver System* User Manual available from `ftp://pnl.gov`

G. Fann and ??? PeIGS v3.0 paper from Web page...

10. L.S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker and R.C. Whaley *SCALAPACK Users' Guide* (SIAM, Philadelphia, 1997)

Scalapack Users' Guide is available in html form from WWW URL `http://netlib.org/scalapack`

11. C. Moler and R. Chamberlain's EISCUBE package released by Intel. Based on parallel versions of EISPACK routines with no explicit orthogonalisation

## EXAMPLE

## NON-LINEAR OPTIMISATION

**TIMING AND MPI PROFILING ROUTINES**

# ERROR HANDLING

## SUMMARY

We have included one timing routine `clips_time` which returns the value of the elapsed time clock in double precision seconds. The operation of this routine on different archutectures is controlled by pre-processing flage in the appropriate makefile.

## ATTRIBUTES

**Version:** 1.0
**Public calls:** `clips_time`
**Public modules:** `clips_timer`
**Other modules required:**
**Date:** 1998
**Origin:** R.J. Allan, CLRC Daresbury Laboratory
**Language:** Fortran 90
**Conditions on external use:** Standard, see separate chapter

## HOW TO USE THE PACKAGE

This package is used through `MODULE clips_timer`.

## SPECIFICATION OF CLIPS_ERROR

```
FUNCTION clips_time()
REAL(KIND=8) :: clips_time
```

### Argument List

There are no arguments.

### Information returned to the User

THe function returnes the number of seconds since epoch. Calling it twice and subtracting the return values therefore yields the real time which has elapsed between the calls.

### Preprocessing Flags

Defined pre-processing flags are:

`CRAY` – use Cray architecture specific clauses
`MPITIME` – use `mpi_wtime` for timing purposes
`F90TIME` – use `system_clock` for timing
`DTIME` – use `dtime` for timing
`RTC` – use `rtc` for timing

## GENERAL INFORMATION

**Workspace:**
**Use of common:** none
**Other routines called directly:**
**Input/output:**
**Restrictions:**
**Notes:**

## EXAMPLE

```
USE clips_timer
REAL(REAL8) :: start, finish, time
start=clips_time()
...
finish=clips_time()
time-finish-start
WRITE(6,'('' time taken = '',g12.5)')time
```

## MPI PROFILING

### SUMMARY

The strategy for profiling MPI functionality is defined in chapter 8 of the MPI standard documen [2]. This states that MPI functions that are to be profiled should have a wrapper function written for each of them. The wrapper function performs the required profiling tasks, and the real MPI function is called inside the wrapper. The wrapper function is given the same name as the MPI function and therefore the real MPI function that is called insied it has to be given a different name. The standard suggests that the new name should be the original MPI name prefixed by "P", e.g. `PMPI_SEND` replaces `MPI_SEND`.

The philosophy behind this approach is that the user does not have to change the source code of his application at all for it to be profiled. Typically the link step is changed to include libraries `-lpmpi -lmpi`. Some additional subroutine calls to switch profiling on and off may however be useful. The routine `MPI_PCONTROL` is provided for this purpose. Its argument is an integer which specifies the profiling level as follows: level==0 - profiling is disabled; level==1 - profiling is enabled at defalut level; level==2 - profiling buffers are flushed immediately after writing.

### ATTRIBUTES

**Version:** 1.0
**Public calls:** `clips_error`
**Public modules:** `clips_init`, `clips_on`, `clips_off`, `clips_collect`, `clips_summary` **Other modules required:** `clips_timer`
**Date:** November 1997
**Origin:** R.J. Allan, CLRC Daresbury Laboratory
**Language:** Fortran 90 and FORTRAN77
**Conditions on external use:** Standard, see separate chapter

### HOW TO USE THE PACKAGE

This package is used through `MODULE clips_profiler`.

Subroutine `dltime.f` provides a small suite of routine which collect elapsed time information. This information is then analysed and the max, min and average across all processors is printed. Total time, message lenght per call and number of calls are sampled.

Subroutine `fudge.f` is the set of profiling hooks for MPI as described in the MPI standard chapter 8. Note that it cannot contain a proper Fortran 90 interface because it has to copy a buffer of unknown type/size/dimension. This is therefore done in the old FORTRAN77 way by transmitting an address declared as `INTEGER buf(*)`.

Subroutine `mpi_pcontrol` is specified in the MPI standard and we have implemented it as follows:

`mpi_pcontrol(0)` – switches off profiling;
`mpi_pcontrol(1)` – switches on profiling. This is invoked automatically by `mpi_init()` as required by the standard so that the user need not change the source code;
`mpi_pcontrol(2)` – prints a summary of profiling stats and flushes the buffers. This is invoked automatically by `mpi_finalize()` as required by the standard so that the user need not change the source code.

### What routines are profiled:

The MPI standard insists that routines can be added on an incremental basis. So far we have included the following set

Control routines

mpi_pcontrol – controls profiling modes
MPI_finalize – prints a summary
MPI_init – switches profiling on

Routines which are profiled

MPI_recv
MPI_send
MPI_allreduce
MPI_alltoallv
MPI_barrier
MPI_bcast

Deliberate omissions

MPI_comm_rank
MPI_comm_size
MPI_abort

## SPECIFICATION OF CLIPS_

### Argument List

`Integer, Intent(In) :: status`
On entry: status flag controlling the routine, may be an error flag

## GENERAL INFORMATION

**Workspace:**
**Use of common:** none
**Other routines called directly:**
**Input/output:**
**Restrictions:**
**Notes:**

## METHOD

Some fragments of code illustrating this procedure are now given. Firstly a module which can be linked to the user application to control profiling and collection of profiling data:

```
      MODULE clips_profiler
      IMPLICIT NONE
      PRIVATE
      PUBLIC :: clips_init, clips_on, clips_off, clips_collect, clips_summary
      INTEGER, PARAMETER :: nstat=10, nbins=4
      INTEGER, DIMENSION(nstat,nbins) :: ncall
      REAL(8), DIMENSION(nstat,nbins) :: ctime, nbyte
      REAL(8) :: startt, cpustart, cpufinish, cpuelapsed
      LOGICAL :: onoff
      SAVE ncall, nbyte, ctime, startt, onoff, cpustart, cpufinish, cpuelapsed

      CONTAINS

      SUBROUTINE clips_init()
      ncall=0; nbyte=0.0d0; ctime=0.0d0
      cpuelapsed=0.0d0
      END SUBROUTINE clips_init

      SUBROUTINE clips_on()
      USE mpi
      onoff=.TRUE.
      cpustart=mpi_wtime()
      END SUBROUTINE clips_on

      SUBROUTINE clips_off()
      USE mpi
      onoff=.FALSE.
      cpufinish=mpi_wtime()
      cpuelapsed=cpuelapsed+(cpufinish-cpustart)
      END SUBROUTINE clips_off

      SUBROUTINE clips_collect(i,count,type)
      USE mpi
      INTEGER, INTENT(IN) :: i,type,count
      INTEGER :: bytes,ier,j
      REAL(8) :: time,nbytes
!      INCLUDE 'mpif.h'
! sample bins at 0-1000-4000-32000-above bytes
      IF(onoff)THEN
        time=MPI_WTIME()-startt
        IF(count>0)THEN
           CALL MPI_TYPE_SIZE(type,bytes,ier)
           nbytes=REAL(bytes*count)
           j=4
           IF(nbytes<32000)j=3
           IF(nbytes<4000)j=2
           IF(nbytes<1000)j=1
```

```
          ncall(i,j)=ncall(i,j)+1
          ctime(i,j)=ctime(i,j)+time
          nbyte(i,j)=nbyte(i,j)+nbytes
        ELSE
          ncall(i,1)=ncall(i,1)+1
          ctime(i,1)=ctime(i,1)+time
        END IF
      END IF
      END SUBROUTINE clips_collect

      SUBROUTINE clips_summary()
...
      END SUBROUTINE clips_summary

      END MODULE clips_profiler
```

Next an example of a set of MPI wrapper routines which collect timing data for each routine to be profiled. Note that an alternative version of `mpi_pcontrol` has been provided:

```
      SUBROUTINE mpi_pcontrol(level)
      USE clips_timer
      INTEGER level
      SELECT CASE(level)
      CASE(0)
         call clips_off()
      CASE(1)
         call clips_on()
      CASE(2)
         call clips_summary()
         call clips_init()
      CASE DEFAULT
         WRITE(6,'('' profiling level '',i4,'' not understood'')')level
      END SELECT
      END SUBROUTINE mpi_pcontrol

      SUBROUTINE mpi_init(ier)
      USE clips_timer
      INTEGER ier
      CALL PMPI_INIT(ier)
      call clips_init()
      call clips_on()
      END SUBROUTINE mpi_init

      SUBROUTINE mpi_send(big,count,type,where,tag,comm,ier)
      USE clips_timer
      INTEGER count, type, where, tag, comm, ier
      INTEGER big(*)
      call clips_time()
      CALL pmpi_send(big,count,type,where,tag,comm,ier)
      call clips_collect(1,count,type)
      END SUBROUTINE mpi_send
```

```
      SUBROUTINE MPI_reduce(inbuf,outbuf,count,type,op,root,comm,ier)
      USE clips_timer
      INTEGER inbuf(*),outbuf(*)
      INTEGER count,type,op,root,comm,ier
      call clips_time()
      CALL PMPI_reduce(inbuf,outbuf,count,type,op,root,comm,ier)
      call clips_collect(3,count,type)
      END SUBROUTINE MPI_reduce
```

...

The makefile to link to this profiling library should contain references to a number of object files and libraries as follows:

```
program: ;
ld -o program program.o wrappers.o clips_profiler.o clips_timer.o mpi.o -lmpi
```

## SUPPORT FUNCTIONS

# ERROR HANDLING

## SUMMARY

One routine `clips_error` is included as a basic error-handling method.

It prints an error return value together with optional information about the calling and called routine and an informational message. It is used to pass back information from calls to MPI routines as well as module routines.

## ATTRIBUTES

**Version:** 1.0
**Public calls:** `clips_error`
**Public modules:** `clips_err_mod`
**Other modules required:**
**Date:** 1998
**Origin:** I.J. Bush and R.J. Allan, CLRC Daresbury Laboratory
**Language:** Fortran 90
**Conditions on external use:** Standard, see separate chapter

## HOW TO USE THE PACKAGE

This package is used through `MODULE clips_err_mod`.

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.

## SPECIFICATION OF CLIPS_ERROR

### Argument List

```
Subroutine clips_error(status, calling_routine, &
                       message, called_routine)
  Integer       , Intent( In )          :: status
  Character( * ), Intent( In )          :: calling_routine
```

```
    Character( * ), Intent( In ), Optional :: message
    Character( * ), Intent( In ), Optional :: called_routine
```

`Integer, Intent(In) :: status`
On entry: status flag controlling the routine, may be an error flag returned from a called subroutine.

`Character(*), Intent(In) :: calling_routine`
On entry: name of the routine calling the error handler.

`Character(*), Intent(In), Optional :: message`
On entry: a message which will be printed to standard output Fortran UNIT=6.

`Character(*), Intent(In), Optional :: called_routine`
On entry: the name of a subroutine which was called and may have returned the error flag passed to the handler in status.

## GENERAL INFORMATION

**Workspace:**
**Use of common:** none
**Other routines called directly:**
**Input/output:**
**Restrictions:**
**Notes:**

## EXAMPLE

## RANDOM NUMBER GENERATOR