Science and Technology Facilities Council

HSL_MP54

PACKAGE SPECIFICATION HSL

1 SUMMARY

For a matrix that is full, symmetric and positive definite, this package performs **parallel partial and complete factorisations and solutions of corresponding sets of equations, using OpenMP**.

We consider the factorization

$$A = \begin{pmatrix} A_{11} & A_{21}^T \\ A_{21} & A_{22} \end{pmatrix} = \begin{pmatrix} L_{11} & \\ L_{21} & I \end{pmatrix} \begin{pmatrix} I & \\ & S_{22} \end{pmatrix} \begin{pmatrix} L_{11}^T & L_{21}^T \\ & I \end{pmatrix} = LSL^T$$

where A is order n, L_{11} is lower triangular and both A_{11} and L_{11} have order $p \leq n$.

Subroutines are also provided for the complementary partial forward and backward substitutions, that is, solving

$$LX = B \quad \text{and} \quad L^T X = B.$$

ATTRIBUTES — Version: 1.2.0 (15 June 2022). **Types:** Real (single, double). **Calls:** _AXPY, _COPY, _GEMM, _SYR, _SYRK, _TPSV, _TRSM **Original date:** August 2008. **Origin:** J. D. Hogg, Rutherford Appleton Laboratory. **Language:** Fortran 95, plus allocatable components of derived types. **Parallelism:** Uses OpenMP and its runtime library. **Remark:** Development of HSL_MP54 was supported by EPSRC grant EP/F006535/1.

2 HOW TO USE THE PACKAGE

2.1 OpenMP

OpenMP is used by the HSL_MP54 package to provide parallelism for shared memory environments. The number of threads may be controlled at runtime by setting the environment variable OMP_NUM_THREADS.

If OpenMP is not available you may be better using the HSL_MA54 package.

2.2 Calling sequences

Access to the package requires a USE statement
Single precision version

```
USE HSL_MP54_single
```

Double precision version

```
USE HSL_MP54_double
```

If it is required to use more than one module at the same time then the derived types (Section 2.3) must be renamed in one of the use statements.

All procedures can be called from serial code, in which case MP54_work will generate a number of OpenMP threads and perform its tasks in parallel.

It is also possible to call MP54_work from within an OpenMP parallel region if the optional argument parallel is used (see Section 2.4.7). In this case any number of threads may call MP54_work, and their return is controlled by the optional parameter terminate. For ease of programming, we recommend that the user picks a master thread to add all the jobs to the queue and control execution, as demonstrated by the second example program (see Section 5.2).

HSL_MP54 uses a job queue paradigm. A job is created by a call to MP54_factor, MP54_forward or MP54_back on a single thread, and the associated work is performed by one or more threads executing MP54_work. Each job is given a unique job identifier (henceforth the job id) that increases monotonically and indicates queue position. The user may

specify that MP54_work returns after a particular job has completed. The user is guaranteed that jobs are executed in order of increasing job id and that the previous job must have completely finished before the subsequent job is begun. The following procedures are available to the user:

- MP54_get_nb is a function that returns a recommended block size nb for given n, p, and number of threads. The block size is a parameter to other calls and a good choice is critical to achieving good performance, in particular nb should be a multiple of the number of REALs that fit in a cache line (on the Intel Core architecture this means a multiple of 8).
- MP54_init initialises the keep variable. It must be executed on exactly one thread, and needs to return before any subroutines utilising this variable are called.
- MP54_factor places the partial factorization of A in the job queue. It must be executed on exactly one thread for each factorization.
- MP54_forward and MP54_back place a partial forward or backward substitution in the job queue. They must be executed on exactly one thread for each substitution. Normally one would expect to execute the MP54_factor, MP54_forward and MP54_back jobs in that order.
- MP54_work performs jobs initiated by MP54_factor, MP54_forward and MP54_back. If MP54_work is called from within an otherwise serial code the user need not use any OpenMP directives themselves. If the call is from within an OpenMP parallel region, each thread calling MP54_work will contribute to the completion of jobs in the queue.
- MP54_all_terminate will either add a job to the queue that causes threads to return from MP54_work, or, if the abort optional argument is used, to terminate immediately. It must be executed by exactly one thread.
- MP54_reset allows a new set of jobs to begin after a call to MP54_all_terminate. If an abort was issued then it clears the job and task queues. The user must ensure that there is a synchronisation between a call to MP54_all_terminate and MP54_reset, for example using an OpenMP barrier.
- MP54_finalise frees memory and OpenMP resources allocated by MP54_init.

2.3 The derived data types

For each problem, the user must employ the derived types defined by the module to declare scalars of the types MP54_keep and MP54_control. The following pseudocode illustrates this.

```
use HSL_MP54_double
...
type(MP54_control) :: control
type(MP54_keep) :: keep
```

The components of MP54_control are explained in Section 2.4.11. The components of MP54_keep are private and are used for communication between threads.

2.4 Argument lists and calling sequences

2.4.1 Optional arguments

We use square brackets [ ] to indicate OPTIONAL arguments, which always follow the argument info. Since we reserve the right to add additional optional arguments in future releases of the code, **we strongly recommend that all optional arguments be called by keyword, not by position**.

2.4.2 Integer and real kinds

INTEGER(short) denotes default INTEGER and INTEGER(long) denotes INTEGER(kind=selected_int_kind(18)). REAL denotes default real in the single precision version and double precision real in the double precision version.

2.4.3 The block size function

To obtain an appropriate block size for given n, p and number of threads, the user is advised to use the following function.

```
integer(short) :: nb
nb = MP54_get_nb(n, p, nthread, control)
```

n is a scalar INTENT(IN) argument of type INTEGER(short) that holds the order of the matrix A. **Restriction:** 1≤n.

p is a scalar INTENT(IN) argument of type INTEGER(short) that holds the order of A_{11}. **Restriction:** 1≤p≤n.

nthread is a scalar INTENT(IN) argument of type INTEGER(short) that holds the number of threads being used. **Restriction:** 1≤nthread.

control is a scalar INTENT(IN) argument of type MP54_control (see Section 2.4.11).

On success, MP54_get_nb returns a value greater than 0; otherwise it has a negative value (see Section 2.5).

2.4.4 The initialisation subroutine

The following subroutine initialises the shared variable keep and must be called by exactly one thread.

```
call MP54_init(maxnrblk, keep, control, info)
```

maxnrblk is a scalar INTENT(IN) argument of type INTEGER(short) and must be set by the user to the maximum value of (n-1)/nb + 1 to be encountered. **Restriction:** 1≤maxnrblk.

keep is a scalar INTENT(OUT) argument of type MP54_keep. It is used for communication between threads and subroutines, and on output it will have been initialised ready for use. If the call is from within an OpenMP region it must have the OpenMP attribute SHARED.

control is a scalar INTENT(IN) argument of type MP54_control (see Section 2.4.11).

info is a scalar INTENT(OUT) argument of type INTEGER(short). On a successful return it has value 0; for other values see Section 2.5.

2.4.5 The factorization initiation subroutine

The following call places a partial factorization job in the queue, and must be called exactly once for each desired factorization. The job will be executed by a thread calling MP54_work.

```
call MP54_factor(n, p, nb, aoffset, keep, control, info[, jid])
```

n is a scalar INTENT(IN) argument of type INTEGER(short) that holds the order of the matrix A. **Restriction:** 1≤n.

p is a scalar INTENT(IN) argument of type INTEGER(short) that holds the order of A_{11}. **Restriction:** 1≤p≤n.

nb is a scalar INTENT(IN) argument type INTEGER(short) that holds the block size to be used. It is recommended that this block size is chosen by calling the function MP54_get_nb. **Restriction:** 1≤nb.

aoffset is a scalar INTENT(IN) argument of type INTEGER(long) that specifies the position of the first element of the matrix to be factorised within the array a that is passed to the MP54_work subroutine. The values in the array a that are involved in a factorization must be set before any corresponding call to MP54_work reaches this job. In practice the best way to guarantee this is to ensure they are set before calling MP54_factor. **Restriction:** 1≤aoffset.

keep is a scalar INTENT(INOUT) argument of type MP54_keep. It is used for communication between threads and subroutines, and must have been initialised by a call to MP54_init before use. If the call is from within an OpenMP region it must have the OpenMP attribute SHARED.

control is a scalar INTENT(IN) argument of type type(MP54_control) (see Section 2.4.11).

info is a scalar INTENT(OUT) argument of type INTEGER(short). On successful return it has value 0. Otherwise it has a negative value described in Section 2.5.

jid is an optional scalar INTENT(OUT) argument of type INTEGER(short). If present, on exit it will contain the job id for this factorization.

2.4.6 The solve initiation subroutines

The following call places a partial forward or back solve in the queue, and must be called exactly once for each desired solve. The job will be executed by a thread calling MP54_work.

```
call MP54_forward(n, p, nb, loffset, nrhs, rhsoffset, ldr, keep, control, info[, jid])
call MP54_back(n, p, nb, loffset, nrhs, rhsoffset, ldr, keep, control, info[, jid])
```

n,p,nb are of INTENT(IN) and must be unchanged since the corresponding call to MP54_factor.

loffset is a scalar INTENT(IN) argument of type INTEGER(long) that specifies the first element of the matrix factor within the array a passed to the MP54_work subroutine. The factor stored in the first p columns starting at loffset must be the same as that generated by the corresponding factorization job, however loffset may be different to aoffset if the user has relocated the factor data within the array a, for example when using an out-of-core approach. **Restriction:** 1≤loffset.

nrhs is a scalar INTENT(IN) argument of type INTEGER(short) that specifies the number of right-hand sides. **Restriction:** 1≤nrhs.

rhsoffset is a scalar INTENT(IN) argument of type INTEGER(short) that specifies the first element within the array rhs passed to MP54_work containing the right-hand sides. The ith right-hand side occupies the elements rhs(rhsoffset+(i-1)*ldr:rhsoffset+(i-1)*ldr+n-1) of the array passed to MP54_work (ie are stored in column major order with a leading extent of ldr). The values in this array that are involved in a solve must be set before any corresponding call to MP54_work reaches this job. In practice, the best way to guarantee this is to ensure they are set before calling MP54_forward or MP54_back. **Restriction:** 1≤rhsoffset.

ldr is a scalar INTENT(IN) argument of type INTEGER(short) that specifies the leading extent of the right-hand side matrix, such that the *i*th right-hand side starts at element rhs(rhsoffset+(i-1)*ldr) of the array passed to MP54_work. **Restriction:** n≤ldr.

keep,control,info see Section 2.4.5.

jid is an optional scalar INTENT(OUT) argument of type INTEGER(short). If present, on exit it will contain the job id for this solve.

2.4.7 The work subroutine

MP54_work processes jobs on the queue until termination conditions (if any) are met. MP54_work may be called from:

Serial code where the user has not created their own team of tasks through an OpenMP parallel construct. In this case the optional parallel argument need not be used, and MP54_work will use an internal OpenMP parallel section to generate a team of threads to perform the factorization in parallel.

Parallel code where the call is from within an OpenMP parallel region. In this case the user must supply the optional parallel argument with the value .true., and each thread should call MP54_work.

```
call MP54_work(la, a, keep, control, info[, parallel, lrhs, rhs, terminate])
```

la is a scalar INTENT(IN) argument of type INTEGER(long), that holds the length of the array a. **Restriction:** 1≤la.

a is an array INTENT(INOUT) argument of type REAL and dimension la. This array is used to communicate the values of the matrix A to factorise, or the values of the factor L to be used in a solve. When a factorization job is encountered the values of A must be held in lower packed form in the entries a(aoffset:aoffset+n*(n+1)/2)-1. When the factorization job completes these values will have been replaced by the factor L. When a solve job is encountered the entries a(loffset:loffset+n*(n+1)/2)-(n-p)*(n-p+1)/2-1 must hold the values of the first p columns of L. If the call is from within an OpenMP parallel region, it must have the OpenMP attribute SHARED.

keep,control,info see Section 2.4.5.

parallel is an optional INTENT(IN) argument of type LOGICAL. If present with the value .true., it indicates that MP54_work is being called from within an OpenMP parallel region. Otherwise, MP54_work will generate a new team of threads with its own OpenMP parallel region.

lrhs is an optional INTENT(IN) argument of type INTEGER(long). It should hold the size of the optional array rhs, which must also be present if lrhs is. If the call is from within an OpenMP region it must have the OpenMP attribute SHARED. **Restriction:** 1≤lrhs.

rhs is an optional assumed-size array INTENT(INOUT) argument of type REAL and size lrhs. If rhs is present, lrhs must also be present. If the call is from within an OpenMP region it must have the OpenMP attribute SHARED.

terminate is an optional scalar INTENT(INOUT) argument of type INTEGER(short). If present, it specifies the job id after which this call to MP54_work will terminate, and on return holds the job id of the last job executed (either completed or aborted). If terminate is not present, or is present with the value huge(terminate), the action taken depends on whether the parallel argument is present with the value .true.. If it is, MP54_work will not terminate until MP54_all_terminate is called by another thread. Otherwise MP54_work will terminate when all jobs in the queue have completed. **Restriction:** 1≤terminate.

2.4.8 The termination subroutine

To cause threads executing MP54_work to return, the following subroutine must be used. Normally a job is added to the queue which causes each thread that encounters it to return, however the abort option allows immediate termination.

```
call MP54_all_terminate(keep, control, info[, abort])
```

keep,control,info see Section 2.4.5.

abort is an optional scalar INTENT(IN) argument of type LOGICAL. Following a call to this subroutine, when a thread executing MP54_work finishes its current task, it will exit immediately if abort is present with the value .true.; otherwise, it will exit only when it encounters the termination job in the queue.

2.4.9 The reset subroutine

If a job is aborted (either through MP54_all_terminate or due to a fatal error) the keep parameter will need to be reset before it can be used again, deleting all outstanding tasks and jobs, The following subroutine will do this. If a termination job has been added to the queue, the only way to move past it is to use the following subroutine.

```
call MP54_reset(keep, control, info)
```

keep,control,info see Section 2.4.5.

2.4.10 The finalise subroutine

HSL_MP54 allocates its own memory and initialises its own OpenMP lock variables internally. The following subroutine will release these resources.

```
call MP54_finalise(keep, control, info)
```

keep,control,info see Section 2.4.5.

2.4.11 The derived type for holding control parameters

The derived type MP54_control is used to hold controlling data. The components, which are automatically given default values in the definition of the type, are as follows:

diag_unit is a scalar of type INTEGER(short). If diag_unit>0, diagnostic messages are output on this unit. The default value is diag_unit=-1.

error_unit is a scalar of type INTEGER(short). If error_unit>0, error messages are output on this unit. The default value is error_unit=6.

min_nb is a scalar of type INTEGER(short), and specifies the minimum block size to return from MP54_get_nb. This prevents parallel communication overheads from overwhelming the numerical work. The default value is min_nb = 100. **Restriction:** 1≤min_nb.

nbi is a scalar of type INTEGER(short), and specifies the inner block size to use in the block factorization kernel. The default value is nbi = 32. **Restriction:** 1≤nbi.

stack_size is a scalar of type INTEGER(short), and specifies the maximum size of the stack used to hold tasks. If this number is too small a stack overflow will occur (info=-23). The default value is stack_size=32000. This control parameter is only accessed on a call to MP54_init, though it will affect later calls. **Restriction:** 1≤stack_size.

upd_stack_size is a scalar of type INTEGER(short), and specifies the maximum number of updates which may be stacked. If this number is too small performance may be adversely effected. The default value is upd_stack_size=100. This control parameter is only accessed on a call to MP54_init, though it will affect later calls. If this parameter has a value of less than or equal to zero, stacking of updates will be disabled. For a description of what this parameter does, please see the technical report [1].

2.5 Warning and error messages

A successful return from a subroutine in the package is indicated by `info` having the value zero. A non-zero value indicates an error return. Negative values indicate a specific error detailed below, while a positive value indicates the matrix was not positive definite, with the first negative pivoted encountered at position `info`. For returns from `MP54_work`, the job id corresponding to the job that caused the error is returned in `terminate` if that argument is present. The job does not normally fail entirely and other threads may continue executing it. The exceptions to this are if a non-positive pivot is encountered or a stack overflow (`info=-23`) occurred; in these cases all threads are aborted.

The meanings of the negative return values are shown below:

`-1` Error in call sequence.

`-2` Previously aborted and `MP54_reset` has not been called.

`-3` `p` ≤ 0.

`-4` `n` $<$ `p`.

`-5` `nthread` ≤ 0.

`-6` `nb` ≤ 0.

`-7` `maxnrblk` ≤ 0.

`-8` `la` ≤ 0.

`-9` `aoffset` ≤ 0 or `loffset` ≤ 0.

`-10` `nrhs` ≤ 0.

`-11` `rhsoffset` ≤ 0.

`-12` `lrhs` ≤ 0.

`-13` `ldr` $<$ `n`.

`-14` Exactly one of `lrhs` and `rhs` is present on a call to `MP54_work`. Either neither must be present or both must be present.

`-15` `terminate` < 0.

`-16` `control%min_nb` ≤ 0.

`-17` `control%nbi` < 1.

`-19` `MP54_work` encountered a job requiring data access outside of `a(1:la)`.

`-20` `MP54_work` encountered a job involving a right-hand side, but was not called with the optional argument `rhs`.

`-21` `MP54_work` encountered a job for which the right-hand side data lies outside the supplied `rhs`.

`-22` `MP54_work` encountered a job for which the value of `(n-1)/nb+1` exceeded the value of `maxnrblk` passed to `MP54_init`.

`-23` `MP54_init` was called with `control%stack_size`<1, or `MP54_work` suffered a stack overflow. The user should try using a larger `control%stack_size` or `nb` value.

`-24` `MP54_init` multiple errors were detected by different threads.

3 GENERAL INFORMATION

Workspace: HSL_MP54 handles its own memory allocations.

Other routines called directly: _AXPY, _COPY, _GEMM, _SYR, _SYRK, _TPSV, _TRSM.

Input/output: Output is provided based on the values of control%error_unit and control%diag_unit.

Restrictions: 1≤nthread, 1≤p≤n, 1≤nb, 1≤maxnrblk, 1≤maxnnb, 1≤la, 1≤aoffset, 1≤loffset, 1≤lrhs, 1≤rhsoffset, n≤ldr, 1≤terminate, 1≤control%nbi, 1≤control%min_nb, 1≤control%stack_size

Portability: Fortran 95, plus allocatable components of derived types.

4 METHOD

HSL_MP54 uses a job queue to allow multiple factorization and solve jobs to be stacked up ready for a pool of threads to work on. To avoid dependency problems, no job is allowed to start before the previous job has finished. Any thread may add jobs to the queue, however each job should be added by exactly one thread. If multiple threads are adding jobs to the queue without synchronisation the order is defined by the job id, which is optionally returned from MP54_factor, MP54_forward and MP54_back.

Each shared object of type MP54_keep represents a job queue and the user may declare more than one if he or she has several independent collections. Any thread that has access to one of these variables and the corresponding data may perform work on this collection by calling MP54_work with the correct arguments. A thread may therefore switch its execution between different collections.

Each job is broken down into a series of tasks with dependencies upon each other, which we represent in an abstract sense by use of a directed acyclic graph (DAG). We execute these tasks in a prioritised order obeying these dependencies. This approach allows better speedups on large numbers of threads than the fork-join loop based parallelism of HSL_MA54. A full description of this process along with benchmark comparisons is available in [1].

Before obtaining a new task, each thread checks if an abort flag has been set, returning to the user if this is so. This abort flag may have been set by another thread calling MP54_all_terminate with the optional abort argument, or by MP54_work finding a non-positive pivot. After all tasks for the current job have been completed, each thread checks a termination flag (set by MP54_all_terminate without the abort argument) and the termination conditions implied by the presence and value of the terminate argument of MP54_work to determine if it should return to the user. If no return is required then a new job is started if available, or the thread awaits the arrival of a new job or the setting of the termination flag by the user through a call to MP54_all_terminate without the optional abort argument. It is important to note as a consequence of this that synchronisation of affected threads is necessary between a call to MP54_all_terminate and any call to MP54_reset.

Internally, a blocked hybrid format is used. Each block of the lower triangular part of A is held columnwise, and data is reordered to this form on entry. The layout of this format for a 7×7 matrix with nb=3 would be

$$\left(\begin{array}{ccc|cccc} 1 & & & & & & \\ 2 & 4 & & & & & \\ 3 & 5 & 6 & & & & \\ \hline 7 & 10 & 13 & 19 & & & \\ 8 & 11 & 14 & 20 & 22 & & \\ 9 & 12 & 15 & 21 & 23 & 24 & \\ \hline 16 & 17 & 18 & 25 & 26 & 27 & 28 \end{array}\right)$$

On return, if p≠n, the trailing lower triangular submatrix of order $n-p$ is reordered to packed format. The leading p columns are left in the block hybrid form to be used by the solve. If a partial factorization splits a block column then

that block column is converted so that the first part is held as a hybrid matrix and latter part is held as the start of the lower packed format of the remaining `n-p` columns. For example, if `p=5`, the data from the previous example would be returned as

$$\left(\begin{array}{ccccc||cc} 1 & & & & & & \\ 2 & 4 & & & & & \\ 3 & 5 & 6 & & & & \\ \hline 7 & 10 & 13 & 19 & & & \\ 8 & 11 & 14 & 20 & 22 & & \\ 9 & 12 & 15 & 21 & 23 & 26 & \\ \hline 16 & 17 & 18 & 24 & 25 & 27 & 28 \end{array}\right)$$

It is perhaps worth commenting that although the first `p` columns of the data returned from a factorization must be unchanged for the corresponding partial solve, the position of that data within the array `a` may have changed, for example due to out-of-core or other storage considerations.

Reference:

[1] J.D.Hogg. [2008]
A DAG-based parallel Cholesky Factorization for multicore systems.
Technical Report TR-RAL-2008-029

5 EXAMPLE OF USE

5.1 Call from serial code

The following code reads a matrix in lower packed format, performs a Cholesky factorization, and solves a set of equations. It demonstrates usage from within an otherwise serial code.

```
program hsl_mp54ds
   use hsl_mp54_double
   use omp_lib
   implicit none

   integer, parameter :: wp = kind(0d0)
   integer, parameter :: long = selected_int_kind(18)

   type(mp54_keep) :: keep
   type(mp54_control) :: control
   integer :: n, nb, nrblk, info
   integer(long) :: la, lrhs
   real(wp), dimension(:), allocatable :: a
   real(wp), dimension(:), allocatable :: rhs

   ! Read the matrix order
   read(*,*) n

   ! Get a suggested block size
   nb = mp54_get_nb(n, n, omp_get_max_threads(), control)

   ! Allocate memory
   la = n*(n+1)/2
   lrhs = n
   allocate(a(la), rhs(lrhs))

   ! Initialize workspaces
   nrblk = (n-1)/nb + 1
   call mp54_init(nrblk, keep, control, info)

   ! Read the lower triangular matrix in the lower packed format
   read(*,*) a(1:la)

   ! Factorize the matrix
   call mp54_factor(n, n, nb, 1_long, keep, control, info)
   call mp54_work(la, a, keep, control, info)
   if(info.ne.0) then
      write(*,*) "Stopping after failure during factorize with info = ", info
      stop
   endif

   ! Read the right hand side
   read(*,*) rhs(1:n)
```

```
   ! Solve
   call mp54_forward(n, n, nb, 1_long, 1, 1, n, keep, control, info)
   call mp54_back(n, n, nb, 1_long, 1, 1, n, keep, control, info)
   call mp54_work(la, a, keep, control, info, lrhs=lrhs, rhs=rhs)
   if(info.ne.0) then
      write(*,*) "Stopping after failure during solve with info = ", info
      stop
   endif

   write(*,'(8f10.3)') rhs(1:n)

   ! Finalize
   call mp54_finalise(keep, control, info)
end program
```

Given the data

```
   3
   5 1 1    5 1   5
   7 7 7
```

this produces the output

```
     1.000     1.000     1.000
```

5.2 Calls from within an OpenMP parallel region

The following code performs the same task as previously, however it sets up its own parallel region to do this, further, it demonstrates the use of offsets other than `1`.

```
program hsl_mp54ds1
   use hsl_mp54_double
   use omp_lib
   implicit none

   integer, parameter :: wp = kind(0d0)
   integer, parameter :: long = selected_int_kind(18)

   type(mp54_keep) :: keep
   type(mp54_control) :: control
   integer :: n, nb, nrblk, rhsoffset, info, jid
   integer(long) :: aoffset, la, lrhs
   real(wp), dimension(:), allocatable :: a
   real(wp), dimension(:), allocatable :: rhs

   ! Read the matrix order
   read(*,*) n

   ! Get a suggested block size
   nb = mp54_get_nb(n, n, omp_get_max_threads(), control)
```

```
! Allocate memory
aoffset = 3_long
rhsoffset = 2
la = aoffset + n*(n+1)/2 - 1
lrhs = rhsoffset+n-1
allocate(a(la), rhs(lrhs))

! Initialize workspaces
nrblk = (n-1)/nb + 1
call mp54_init(nrblk, keep, control, info)

!$OMP PARALLEL DEFAULT(NONE) &
!$OMP PRIVATE(jid, info) &
!$OMP SHARED(n, nb, keep, control, la, a, lrhs, rhs, aoffset, rhsoffset)

if(omp_get_thread_num().eq.0) then ! Master thread
   ! Read the lower triangular matrix in the lower packed format
   read(*,*) a(aoffset:aoffset+n*(n+1)/2-1)

   ! Read the right hand side
   read(*,*) rhs(rhsoffset:rhsoffset+n-1)

   ! Setup jobs
   call mp54_factor(n, n, nb, aoffset, keep, control, info)
   call mp54_forward(n, n, nb, aoffset, 1, rhsoffset, n, keep, control, info)
   call mp54_back(n, n, nb, aoffset, 1, rhsoffset, n, keep, control, info, &
      jid=jid)

   ! Join in with the work
   call mp54_work(la, a, keep, control, info, parallel=.true., &
      lrhs=lrhs, rhs=rhs, terminate=jid)

   ! Stop other threads when they have finished
   call mp54_all_terminate(keep, control, info)
else ! Not master thread
   call mp54_work(la, a, keep, control, info, parallel=.true., lrhs=lrhs, &
      rhs=rhs)
endif
if(info.ne.0) then
   write(*, "(2(a,i4))") "Thread ", omp_get_thread_num(), &
      " terminated with info = ", info
endif

!$OMP END PARALLEL

write(*,'(8f10.3)') rhs(rhsoffset:rhsoffset+n-1)

! Finalize
```

```
   call mp54_finalise(keep, control, info)

end program hsl_mp54ds1
```

Given the data

```
   3
   5 1 1    5 1   5
   7 7 7
```

this produces the output

```
     1.000     1.000     1.000
```