

1 SUMMARY

HSL_MA86 uses a direct method to solve **large sparse symmetric indefinite linear systems** of equations $AX = B$. This package uses **OpenMP** and is designed for **multicore architectures**. It computes the sparse factorization

$$A = PLD(PL)^*$$

where $L^* = L^T$ (real symmetric or complex symmetric) or $L^* = L^H$ (complex Hermitian, where L^H denotes the conjugate transpose of L), P is a permutation matrix, L is unit lower triangular, and D is block diagonal with blocks of size 1×1 and 2×2 .

Options are provided for the complementary forward and backward substitutions.

The efficiency of HSL_MA86 is dependent on the user-supplied elimination order. The HSL package HSL_MC68 may be used to obtain a suitable ordering.

The lower triangular part of A must be supplied in compressed sparse column format. The HSL package HSL_MC69 may be used to convert data held in other sparse matrix formats and also to check the user's matrix data for errors.

If A is known to be positive definite (so that pivoting for numerical stability is not required), we recommend using the package HSL_MA87.

ATTRIBUTES — **Version:** 1.8.0 (27 January 2025). **Interfaces:** Fortran, C, MATLAB. **Types:** Real (single, double), Complex (single, double). **Calls:** HSL_MC34 and HSL_MC78, BLAS subroutines `_copy`, `_axpy`, `_swap`, `_gemm`, `_gemv`, `_trsm`, `_trsv`. **Original date:** January 2011. **Origin:** J.D. Hogg and J.A. Scott, Rutherford Appleton Laboratory. **Language:** Fortran 2003 subset (F95 + TR15581 + C interoperability). **Parallelism:** Uses OpenMP and its runtime library. **Remark:** Development of HSL_MA86 was supported by EPSRC grant EP/E053351/1.

2 HOW TO USE THE PACKAGE

2.1 C interface to Fortran code

This package is written in Fortran and a wrapper is provided for C programmers. This wrapper may only implement a subset of the full functionality described in the Fortran user documentation.

The wrapper will automatically convert between 0-based (C) and 1-based (Fortran) array indexing, so may be used transparently from C. This conversion involves both time and memory overheads that may be avoided by supplying data that is already stored using 1-based indexing. The conversion may be disabled by setting the control parameter `control.f_arrays=1` and supplying all data using 1-based indexing. With 0-based indexing, the matrix is treated as have rows and column $0, 1, \dots, n-1$. In this document, we assume 0-based indexing.

The wrapper uses the Fortran 2003 interoperability features. **Matching C and Fortran compilers must be used**, for example gcc and gfortran, or icc and ifort. If the Fortran compiler is not used to link the user's program, then additional Fortran compiler libraries may need to be linked explicitly.

2.2 OpenMP

OpenMP is used by the HSL_MA86 package to provide parallelism for shared memory environments. To run in parallel, OpenMP must be enabled at compilation time by using the correct compiler flag (usually some variant of `-openmp`). The number of threads may be controlled at runtime by setting the environment variable `OMP_NUM_THREADS`.

Although the code may be compiled and run in serial mode, we recommend it is run in parallel on a multicore machine (other HSL solvers, notably MA57 or HSL_MA77 may be more appropriate if a serial code is required).

2.3 Calling sequences

Access to the package requires inclusion of the header file

Single precision version

```
#include "hsl_ma86s.h"
```

Double precision version

```
#include "hsl_ma86d.h"
```

Complex version

```
#include "hsl_ma86c.h"
```

Double complex version

```
#include "hsl_ma86z.h"
```

It is not possible to use more than one version at the same time.

The following subroutines are available to the user:

- (a) `ma86_default_control` sets default values for members of the `ma86_control` data type needed by other subroutines.
- (b) `ma86_analyse` analyses the sparsity pattern of the matrix and, using the user-supplied elimination order, prepares the data structures for the factorization.
- (c) `ma86_factor` uses the data structures set up by `ma86_analyse` to compute a sparse factorization. More than one call to `ma86_factor` may follow a call to `ma86_analyse`.
- (d) `ma86_factor_solve` may be called in place of `ma86_factor` to factorize A and, at the same time, solve the system $AX = B$. Multiple calls to `ma86_factor_solve` may follow a call to `ma86_analyse`.
- (e) `ma86_solve` uses the computed factors generated by `ma86_factor` or `ma86_factor_solve` to solve systems $AX = B$ for one or more right-hand sides B . Multiple calls to `ma86_solve` may follow a call to `ma86_factor` or `ma86_factor_solve`. An option is available to perform a partial solution.
- (f) `ma86_finalise` should be called after all other calls are complete for a problem (including after an error return). It frees memory allocated by the package.

2.4 The derived data types

For each problem, the user must employ the structures defined in the header file to declare scalars of the types `struct ma86_info`, `struct ma86_control`, and a `void *` pointer `keep`. The following pseudocode illustrates this.

```
#include "hsl_ma86d.h"
...
struct ma86_control control;
struct ma86_info info;
void * keep;
...
```

The members of `ma86_control` and `ma86_info` are explained in Sections 2.5.8 and 2.5.9. The `void *` pointer is used to pass data between subroutines of the package.

2.5 Argument lists and calling sequences

2.5.1 Package types

The complex versions require C99 support for the double complex and float complex types. The real versions do not require C99 support.

We use the following type definitions in the different versions of the package:

Single precision version

```
typedef float pkgtype
```

Double precision version

```
typedef double pkgtype
```

Complex version

```
typedef float complex pkgtype
```

Double complex version

```
typedef double complex pkgtype
```

Elsewhere, for *single* and *single complex* versions replace double with float.

2.5.2 Input of the matrix A

The user must input the **lower** triangular part of the matrix A using the arguments `ptr` and `row` as described in Section 2.5.4.

We recommend that standard HSL format is used to ensure compatibility with other HSL routines. This is a compressed sparse column format with the entries within each column ordered by increasing row index. There is no requirement that zero entries on the diagonal are explicitly included. **No checks** are made on the user's data. It is important to note that any out-of-range entries or duplicates may cause HSL_MA86 to fail in an unpredictable way. Before using HSL_MA86, the HSL package HSL_MC69 may be used to check for errors and to handle duplicates (HSL_MC69 sums them) and out-of-range entries (HSL_MC69 removes them).

If the user's data is held using another standard sparse matrix format (such as coordinate format or sparse compressed row format), we recommend using a conversion routine from HSL_MC69 to put the data into standard HSL format. The user may additionally call `mc69_set_values` before each call to `ma86_factor` or `ma86_factor_solve` to change the values of the entries of A (without altering the sparsity pattern). The input of A and of new values is illustrated in Section 5.

2.5.3 The default setting subroutine

Default values for members of the `ma86_control` structure may be set by a call to `ma86_default_control`.

```
void ma86_default_control(struct ma86_control *control)
```

`control` has its members set to their default values, as described in Section 2.5.8.

2.5.4 To analyse the sparsity pattern and prepare for the factorization

```
void ma86_analyse(int n, const int ptr[], const int row[], int order[],
                 void **keep, const struct ma86_control *control, struct ma86_info *info)
```

`n` must hold the order of A .

`ptr` is a rank-one array of size `n+1`. `ptr[j]` must be set so that `ptr[j]` is the position in row of the first entry in column `j` and `ptr[n]` must be set to the total number of entries in the lower triangular part of A .

`row` is a rank-one array of size `ptr[n]`. It must hold the row indices of the entries of the lower triangular part of A , with the row indices for the entries in column 0 preceding those for column 1, and so on.

`order` is a rank-one array of size `n`. It must specify the elimination order, that is, `order[i]` must hold the position of variable i in the pivot sequence. On exit, `order` contains the elimination order that `ma86_factor` (or `ma86_factor_solve`) will be given; this order may give slightly more fill-in than the user-supplied order.

`keep` will be set to point at an area of memory allocated using a Fortran `allocate` statement that will be used to hold data about the problem being solved. It must be passed unchanged to the other subroutines. To avoid a memory leak the subroutine `ma86_finalise` must be used to clean up and deallocate this memory after the factorization is no longer required.

`control` is used to control the actions of the package, see Section 2.5.8.

`info` is used to return information about the execution of the package, as explained in Section 2.5.9.

2.5.5 To factorize the matrix and optionally solve $AX = B$

To factorize the matrix, a call to `ma86_factor` may be made after the call to `ma86_analyse`.

The real case:

```
void ma86_factor(int n, const int ptr[], const int row[], const pkgtype val[],
                const int order[], void **keep, const struct ma86_control *control,
                struct ma86_info *info, const double scale[])
```

The complex case:

```
void ma86_factor(int matrix_type, int n, const int ptr[], const int row[],
                const pkgtype val[], const int order[], void **keep,
                const struct ma86_control *control, struct ma86_info *info,
                const double scale[])
```

If the user wishes to solve $AX = B$ at the same time as factorizing the matrix, he or she should instead make a call to `ma86_factor_solve`.

The real case:

```
void ma86_factor_solve(int n, const int ptr[], const int row[],
                      const pkgtype val[], const int order[], void **keep,
                      const struct ma86_control *control, struct ma86_info *info,
                      const int nrhs, const int ldx, pkgtype x[], const double scale[])
```

The complex case:

```
void ma86_factor_solve(int matrix_type, int n, const int ptr[],
                      const int row[], const pkgtype val[], const int order[], void **keep,
                      const struct ma86_control *control, struct ma86_info *info,
                      const int nrhs, const int ldx, pkgtype x[], const double scale[])
```

`matrix_type` indicates the type of the matrix A in the complex case. It must be set to `-4` if A is Hermitian and to `-5` if A is complex symmetric. **Restriction:** `matrix_type = -4` or `-5`.

`n`, `ptr`, `row`, `order`: must be unchanged since the call to `ma86_analyse`.

`val` is a rank-one array of size `ptr[n]`. The entries must be set so that `val[k]` holds the value of the entry in position `k` of `row[k]`.

`keep`, `control`, `info`: see Section 2.5.4.

`nrhs` holds the number of right-hand sides. **Restriction:** `nrhs` ≥ 1 .

`ldx` holds the length of the leading dimension of `x` (only the first `n` locations are accessed). Note that this is the leading dimension in memory, not in C notation. **Restriction:** `ldx` $\geq n$.

`x` is a rank-2 array with size `x[nrhs][ldx]`. On entry, `x[j][i]` must hold the `i`th component of the `j`th right-hand side; on exit, it holds the corresponding solution.

`scale` may be NULL. If it is not NULL it must be a rank-1 array of size `scale[n]`. If `control.scaling==0` and `scale==NULL` no scaling is used, otherwise `scale` specifies an array of user-supplied scaling factors to be applied symmetrically to the matrix such that the matrix factored is SAS where $S = \text{diag}(\text{scale})$. If `control.scaling!=0` then a scaling is applied and `scale` specifies an array where the scaling factors used will be stored; if `scale==NULL` then the scaling is still applied but the scaling factors are not returned to the user.

2.5.6 To solve linear systems using the computed factors

After the call to `ma86_factor` (or `ma86_factor_solve`), one or more calls to `ma86_solve` may be made to solve $AX = B$. Partial solutions may be performed by appropriately setting the optional parameter `job`.

```
void ma86_solve(int job, int nrhs, int ldx, pkgtype x[], const int order[],
               void **keep, const struct ma86_control *control,
               struct ma86_info *info, const pkgtype *scale)
```

`job` If `job = 0`, $AX = B$ is solved. A partial solution may be computed by setting `job` to have one of the following values:

- 1 for solving $PLX = B$
- 2 for solving $DX = B$
- 3 for solving $(PL)^*X = B$
- 4 for solving $D(PL)^*X = B$.

Restriction: `job = 0, 1, 2, 3, 4`.

`nrhs`, `ldx`, `x`, `order`: see Section 2.5.5.

`keep`, `control`, `info`: see Section 2.5.4.

`scale` should be NULL. This argument is ignored and considered deprecated. It is provided only for backwards compatibility and may be removed in later versions of the code.

2.5.7 The finalisation subroutine

Once all other calls to HSL_MA86 routines are complete for a problem or after an error return, a call to `ma86_finalise` should be made to free memory and resources associated with `keep`.

```
void ma86_finalise(void **keep, const struct ma86_control *control)
```

`keep` must be passed unchanged. On exit, all memory associated with `keep` will have been freed and `keep` will be set to null.

`control` will be used to control printing. Only the members that control printing are accessed (see Section 2.5.8).

2.5.8 The data type for holding control parameters

The data type `struct ma86_control` is used to hold controlling data. The members, and default values that are set by a call to `ma86_default_control`, are:

C only controls

`int f_arrays` indicates whether to use C or Fortran array indexing. If `f_arrays!=0` (i.e. evaluates to true) then 1-based indexing of the arrays `ptr`, `row` and `order` is assumed. Otherwise, if `f_arrays=0` (i.e. evaluates to false), then these arrays are copied and converted to 1-based indexing in the wrapper function. All descriptions in this documentation assume `f_arrays=0`. The default is `f_arrays=0` (false).

Printing controls

`int diagnostics_level` used to control the level of diagnostic printing. The different levels are:

- < 0 No printing.
- = 0 Error and warning messages only.
- = 1 As 0, plus basic diagnostic printing.
- = 2 As 1, plus some additional diagnostic printing.
- = 3 As 2, plus all entries of user-supplied arrays.

The default is `diagnostics_level=0`.

`int unit_diagnostics` holds the Fortran unit number for diagnostic printing. Printing of diagnostics is suppressed if `unit_diagnostics<0`. The default is `unit_diagnostics=6` (stdout).

`int unit_error` holds the Fortran unit number for error messages. Printing of error messages is suppressed if `unit_error<0`. The default is `unit_error=6` (stdout).

`int unit_warning` holds the Fortran unit number for warning messages. Printing of warning messages is suppressed if `unit_warning<0`. The default is `unit_warning=6` (stdout).

Controls used by `ma86_analyse`

`int nemin` controls node amalgamation. A child node is merged with its parent node in the assembly tree if they both involve fewer than `nemin` eliminations. The default is `nemin=32`. The default is used if `nemin<1`.

`int nb` controls the target number of rows used in the block data structure used to hold the factor L (see Section 4.1). The target number of rows in each block is `nb`. The default is `nb=256`. The default is used if `nb<1`.

Controls used by `ma86_factor` and `ma86_factor_solve`

`int action` controls behaviour when a matrix is singular. If A is found to be singular (that is, to have rank less than n), the computation continues after issuing a warning if `action` evaluates to true or terminates (see error -11) if it evaluates to false. The default is `action=1` (true).

`int nbi` holds the inner block size used in the `factorize_column` tasks (see Section 4). Using a value of `nbi` that is smaller than `nb` increases the amount of computation performed using Level 3 BLAS. The default is `nbi=16`. The default is used if `nbi<1`.

`int pool_size` holds the initial size of the arrays that store the task pool (see Section 4). Whenever the size of these arrays is found to be too small, their size is doubled. The default is `pool_size=25000`. The default is used if `pool_size<1`.

`double small_` controls the definition of small pivots. Any pivot whose modulus is less than `small_` is treated as zero. The default is `small_=10-20`.

`int scaling` is used to control scaling. The available options are:

- `≤ 0` No scaling (`scale` argument is NULL), or user-supplied scaling (`scale` argument is not NULL).
- `= 1` Generate a scaling using a weighted bipartite matching using the package MC64.
- `≥ 2` Generate a scaling by applying the iterative method of the package MC77 for one iteration in the infinity norm and three iterations in the one norm.

The default is `scaling=0`.

`double static_` is used to control static pivoting. If `static_>0.0` and if, at any stage of the computation, fewer than the expected number of pivots can be found with relative pivot tolerance greater than `umin`, diagonal entries are accepted as pivots. If a candidate diagonal entry has absolute value at least `static_`, it is selected as a pivot; otherwise, the pivot is given the value that has the same sign but absolute value `static_`. Further details are given in Section 4.3. The default value is 0.0. **Restriction:** Either `static_=0.0` or `static_≥small_`.

`double u` holds the initial value of the relative pivot tolerance u used. The default is `u=0.01` in the double precision version and `u=0.1` in the single precision. Values outside the range $[0, 1.0]$ are treated as the default.

`double umin` holds the minimum value of the relative pivot tolerance. If, at any stage of the computation, fewer than the expected number of stable pivots have been found using the current tolerance u and the candidate pivot with greatest relative pivot tolerance has tolerance $v ≥ u_{min}$, this is accepted as a pivot and the tolerance u is set to v . The default is `umin=0.01`. Values of `umin` greater than `u` are treated as `u` and values less than 0 are treated as 0.

2.5.9 The data type for holding information

The data type `struct ma86_info` is used to hold parameters that give information about the progress and needs of the algorithm. The members of `ma86_info` (in alphabetical order) are:

`double detlog` holds, on exit from `ma86_factor` or `ma86_factor_solve`, the logarithm of the absolute value of the determinant of A or zero if the determinant is zero.

`double detarg` is only present in the complex case. On exit from `ma86_factor` or `ma86_factor_solve` in the complex symmetric case, it holds the determinant of A divided by its absolute value or one if the determinant is zero.

`int detsign` holds, on exit from `ma86_factor` or `ma86_factor_solve` in the real or complex Hermitian case, the sign of the determinant of A or zero if the determinant of A is zero.

`int flag` gives the exit status of the algorithm (details in Section 2.6).

`int matrix_rank` holds, on exit from `ma86_factor` and `ma86_factor_solve`, the rank of the factorized matrix.

`int maxdepth` holds, on exit from `ma86_analyse`, the maximum depth of the assembly tree.

`int num_delay` holds, on exit from `ma86_factor` and `ma86_factor_solve`, the number of eliminations that were delayed, that is, the total number of pivot candidates that were passed to the parent node in the assembly because of stability considerations. If a variable is passed further up the assembly tree, it will be counted again. A large value of `num_delay` indicates that substantial modifications were made to the pivot sequence to ensure stability (and the number of entries `num_factor` in L and number of flops `num_flops` used to compute L will be significantly more than predicted by `ma86_analyse`).

`long num_factor` holds, on exit from `ma86_analyse`, the number of entries that will be in the L factor, assuming the pivot sequence can be used without modification. On exit from `ma86_factor` and `ma86_factor_solve`, it holds the actual number of entries in the L factor. Note that, $2n$ entries of D^{-1} are also held.

`long num_flops` holds, on exit from `ma86_analyse`, the number of floating-point operations that will be needed to perform the factorization, assuming the pivot sequence can be used without modification. On exit from `ma86_factor` and `ma86_factor_solve`, it holds the number of floating-point operations performed.

`int num_neg` holds, on exit from `ma86_factor` or `ma86_factor_solve` in the real or complex Hermitian case, the number of negative eigenvalues of the matrix D and is set to zero otherwise.

`int num_nodes` holds, on exit from `ma86_analyse`, the number of nodes in the assembly tree.

`int num_nothresh` holds, on successful exit from `ma86_factor` and `ma86_factor_solve`, the number of pivots which did not satisfy the threshold criteria based on the value of `control.u`.

`int num_perturbed` holds, on successful exit from `ma86_factor` and `ma86_factor_solve`, the number of pivots that were replaced by `control.static_`.

`int num_sup` holds, on exit from the final call to `ma86_analyse`, the number of supervariables in the problem.

`int num_two` holds, on exit from `ma86_factor` and `ma86_factor_solve`, the number of 2×2 blocks in D .

`int pool_size` holds, on exit from `ma86_factor` and `ma86_factor_solve`, the maximum number of tasks that are in the task pool during the factorization. Note that on repeated runs using the same matrix data, this may vary.

`int stat` holds the Fortran `stat` parameter.

`double usmall` holds, on successful exit from `ma86_factor` and `ma86_factor_solve`, the following:

If `num_perturbed=0`, `usmall` holds the threshold parameter that was used; otherwise `usmall` is set to zero.

2.6 Warning and error messages

A successful return from a subroutine in the package is indicated by `info.flag` having the value zero. A negative value is associated with an error message that by default will be output on unit `control.unit_error`. Possible negative values are:

- 1 Allocation error. The `stat` parameter is returned in `info.stat`.
- 2 Returned by `ma86_analyse` if an error is found in the user-supplied elimination order (held in `order`).
- 3 Returned by `ma86_factor` and `ma86_factor_solve` if `control.action` evaluates to false and the matrix is found to be singular.
- 4 Returned by `ma86_factor_solve` and `ma86_solve` if there is an error in the size of array x (that is, `ldx < n` or `nrhs < 1`).

- 5 Returned by `ma86_factor` and `ma86_factor_solve` if IEEE infinities found in the factorization, probably caused by `control.small_` or `control.u` having too small a value.
- 6 Returned by `ma86_solve` if `job` is out of range.
- 7 Immediate return from `ma86_factor` and `ma86_factor_solve` if `control.static_ < abs(control.small_)` and `control.static_ ≠ 0.0`.
- 8 Returned by `ma86_factor` and `ma86_factor_solve` in the complex case if `matrix_type` is invalid.
- 9 Returned by `ma86_solve` if the argument `scale` either (a) was NULL on the call to `ma86_factor` or `ma86_factor_solve` and is not NULL on the call to `ma86_solve`; or (b) was not NULL on the call to `ma86_factor` or `ma86_factor_solve` and is NULL on the call to `ma86_solve`.

A positive value for `info.flag` is used for warnings. Possible values are:

- +1 Returned by `ma86_factor` and `ma86_factor_solve` if `control.pool_size` found to be too small. The size of the task pool used is returned in `info.pool_size`.
- +2 Returned by `ma86_factor` and `ma86_factor_solve` if `control.action` evaluates to true and the matrix is found to be singular.
- +3 Returned by `ma86_factor` and `ma86_factor_solve` if both of the above two warnings are issued (that is, the task pool is found to be too small and the matrix is found to be singular).

3 GENERAL INFORMATION

Workspace: HSL_MA86 handles its own memory allocations.

Other routines called directly: HSL packages HSL_MC34 and HSL_MC78, BLAS routines `_copy`, `_axpy`, `_swap`, `_gemm`, `_gemv`, `_trsm`, `_trsv`.

Input/output: Output is provided under the control of `control.diagnostics_level`, which allows error, warning and diagnostics messages to be printed on units `control.unit_error`, `control.unit_warning` and `control.unit_diagnostics`, respectively.

Restrictions: $nrhs \geq 1$, $lx \geq n$, `job = 0, 1, 2, 3, 4`. Complex case `matrix_type = -4` or `-5`.

Portability: Fortran 2003 subset (F95 + TR15581 + C interoperability).

4 METHOD

HSL_MA86 divides the sparse factorization into tasks, each of which alters a single block or a block column. The three different types of tasks are referred to as the `factorize_column` task, the `update_internal` task, and the `update_between` task; they are discussed in detail in [1]. The tasks are partially ordered; for example, the updating of a block from a block column of L has to wait until all the rows that it needs from the block column have been calculated. As soon as all the data that a task needs are available, the task is placed in a pool of tasks for execution by any processor. The whole factorization is thus represented by a directed acyclic graph (DAG) with vertices representing tasks and edges representing dependencies.

4.1 Data structures

A node of the assembly tree represents a set of contiguous columns of L with the same (or nearly the same) sparsity structure below a dense (or nearly dense) triangular submatrix. Each nodal matrix is held as a dense trapezoidal matrix. We store this matrix using a row hybrid blocked structure and use “full” storage for the blocks on the diagonal (which allows us to exploit efficient BLAS and LAPACK routines). If the number of columns in the nodal matrix is large, we use the block size nb specified through the control parameter `control.nb` and the blocks will be of size near $nb \times nb$. For example, if the block size was 3, a node with 5 columns and 8 rows would be stored as

1				
4	5			
7	8	9		
10	11	12	25	
13	14	15	27	28
16	17	18	29	30
19	20	21	31	32
22	23	24	33	34

4.2 The analyse phase

The analyse phase uses only the sparsity pattern of A . It requires the user to input the lower triangular part of the matrix in compressed sparse column format; no checks are made on the matrix data. The user must also input an elimination order (which may be computed using, for example, `HSL_MC68`). For the given elimination order, the analyse phase computes the assembly tree using `HSL_MC78`. A child node is merged with its parent node if they both involve fewer than `control.nemin` eliminations. The block structure of L is computed and the task DAG is established and to track which tasks are ready, a dependency count for each block is computed. If the block is on the diagonal, the count is the number of updates that will be applied to it. If the block is not on the diagonal, the count is one more than the number of updates that will be applied to it.

4.3 The factorize phase

The factorize phase uses the data structures prepared in the analyse phase and takes a copy of the dependency counts computed by the analyse phase. We also hold a dependency count for each block column. This equal to the sum of the dependency counts of the blocks within the block column. Each block of L is set to zero the first time that it is accessed and the entries of A are added into the blocks within a block column at the start of a `factorize_column` task.

During factorize, the block count (and corresponding block column count) is decremented by one after the completion of each update for it. When the block column count reaches zero, a `factorize_column` task is added to the task pool.

When a `factorize_column` task completes, its count is decremented to flag this event with a negative value. A column lock is set and each update task that depends on the completion of this task and does not depend on a task that has not yet completed is added to the task pool. Once this has been done, the lock is released.

An optimisation of this approach used for machines with separate caches is to give each cache its own task stack, which overflows into the task pool. If the local stack and task pool are empty, workstealing is used to obtain tasks — if another cache has spare tasks in its stack, half of these are moved to the task pool.

The `factorize_column` task incorporates threshold partial pivoting. The relative pivot tolerance u is initially set to `control.u`. If a pivot candidate does not satisfy the threshold pivot criteria, the action taken depends on the control parameters `control.umin` and `control.static_`. If static pivoting is not requested (`control.static_=0.0`) and `control.umin=control.u`, the pivot is delayed (this is the default case). In our experience, if a large number

of pivots are delayed (see `info.num_delay`), the performance of HSL_MA86 can be badly effected and so we have included options that can help limit the number of delayed pivots (possibly at the cost of a less stable factorization). If `control.umin < control.u` and the relative pivot tolerance for the pivot candidate is $v \geq \text{control.umin}$, the candidate is accepted and u is set to v . The factorization continues using this new value. If $v < \text{control.umin}$, the pivot is delayed unless static pivoting is being used. In this case, if the candidate has absolute value at least `control.static_`, it is selected and `info.num_nothresh` is incremented by one; otherwise, the pivot is given the value that has the same sign but absolute value `control.static_` and `info.num_perturbed` is incremented by one. Note that if a small relative pivot tolerance is used and/or static pivoting is used, the factorization is likely to be inaccurate and an iterative procedure (such as iterative refinement) may be needed once the factorization is complete to try and restore accuracy. Our experience is that the accuracy can be very sensitive to the choice of `control.static_`; in our tests in double precision, a value of $\|A\| * 10^{-6}$ was an appropriate choice.

If the user wishes to solve $AX = B$ at the same time as factorizing the matrix, the call to `ma86_factor` should be replaced by a call to `ma86_factor_solve`. The user must pass right-hand vectors to `ma86_factor_solve` using the argument `x1` (single right-hand side) or `x` (multiple right-hand sides). The forward substitutions are performed as the factor entries are generated. Once the factorization is complete, `ma86_factor_solve` performs the back substitutions by calling `ma86_solve` with `job = 4`. Using `ma86_factor_solve` is more efficient than calling `ma86_factor` followed by `ma86_solve`.

4.4 The solve phase

The solve phase uses the data structures prepared by the factorize phase to perform a full or partial solution of the equation

$$AX = B$$

The matrix factor L must be accessed once for the forward substitution and once for the back substitution. This is independent of the number of right-hand sides so that solving for several right-hand sides at once is significantly faster than repeatedly solving for a single right-hand side.

References:

[1] J.D. Hogg and J.A. Scott. (2010). An indefinite sparse direct solver for multicore machines Technical Report TR-RAL-2010-011.

Available from <http://www.numerical.rl.ac.uk/reports/reports.shtml>

5 EXAMPLE OF USE

5.1 Example 1

We wish to solve the indefinite system

$$\begin{pmatrix} -3. & 1. & & & \\ 1. & 4. & 1. & & 1. \\ & & 1. & 3. & 2. \\ & & & 2. & 4. \\ & 1. & & & 2. \end{pmatrix} X = \begin{pmatrix} -1. \\ 12. \\ 10. \\ 8. \\ 4. \end{pmatrix}.$$

The following code may be used.

```
/* hsl_ma86ds.c */
#include <stdio.h>
```

```
#include <stdlib.h>
#include "hsl_mc69d.h"
#include "hsl_ma86d.h"

/* Simple code to illustrate use of hsl_ma86 */
int main(void) {

    /* Derived types */
    void *keep;
    struct ma86_control control;
    struct ma86_info info;

    int i, n, flag, more;
    int *ptr, *row, *order;
    double *val, *x;

    /* Read the lower triangle of the matrix */
    scanf("%d\n", &n);
    ptr = (int *) malloc(sizeof(int)*(n+1));
    for(int i=0; i<n+1; i++) scanf("%d", &ptr[i]);
    row = (int *) malloc(sizeof(int)*(ptr[n]));
    for(int i=0; i<ptr[n]; i++) scanf("%d", &row[i]);
    val = (double *) malloc(sizeof(double)*(ptr[n]));
    for(int i=0; i<ptr[n]; i++) scanf("%lf", &val[i]);
    /* Read the right hand side */
    x = (double *) malloc(sizeof(double)*n);
    for(int i=0; i<n; i++) scanf("%lf", &x[i]);

    /* Use the input order */
    order = (int *) malloc(sizeof(int)*n);
    for(int i=0; i<n; i++) order[i] = i;

    /* Uncomment the following lines to enable checking (performance overhead) */
    /*
    flag = mc69_verify(6, HSL_MATRIX_REAL_SYM_INDEF, 0, n, n, ptr, row, NULL,
        &more);
    if(flag != 0) {
        printf("Matrix not in HSL standard format. flag, more = %i, %i\n",
            flag, more);
        return 1;
    }
    */

    /* Set up control */
    ma86_default_control(&control);

    /* Analyse */
    ma86_analyse(n, ptr, row, order, &keep, &control, &info);
    if(info.flag < 0) {
```

```
    printf("Failure during analyse with info.flag = %i\n", info.flag);
    return 1;
}

/* Factor */
ma86_factor(n, ptr, row, val, order, &keep, &control, &info);
if(info.flag < 0) {
    printf("Failure during factor with info.flag = %i\n", info.flag);
    return 1;
}

/* Solve */
ma86_solve(0, 1, n, x, order, &keep, &control, &info);
if(info.flag < 0) {
    printf("Failure during solve with info.flag = %i\n", info.flag);
    return 1;
}

printf(" Computed solution:\n");
for(int i=0; i<n; i++) printf("%10.3lf ", x[i]);
printf("\n");

/* Clean up */
ma86_finalise(&keep, &control);
free(ptr); free(row); free(val);
free(order);
free(x);

return 0;
}
```

The following input, supplied on stdin,

```
5
0 2 5 7 8 9
0 1 1 2 4 2 3 3 4
-3. 1. 4. 1. 1. 3. 2. 4. 2.
-1. 12. 10. 8. 4.
```

produces the following output.

```
Computed solution:
    1.000    2.000    2.000    1.000    1.000
```

5.2 Example 2

We wish to solve the similar indefinite systems

$$\begin{pmatrix} -3. & 1. & & & \\ 1. & 4. & 1. & & 1. \\ & 1. & 3. & 2. & \\ & & 2. & 4. & \\ 1. & & & & 2. \end{pmatrix} X = \begin{pmatrix} -1. \\ 12. \\ 10. \\ 8. \\ 4. \end{pmatrix}, \quad \text{and} \quad \begin{pmatrix} -5. & 2. & & & \\ 2. & 9. & 3. & & -2. \\ & 3. & 6. & 1. & \\ & & 1. & -5. & \\ -2. & & & & 6. \end{pmatrix} X = \begin{pmatrix} -1. & -11. \\ 19. & 21. \\ 28. & 14. \\ -17. & -9. \\ 26. & 14. \end{pmatrix},$$

where input is in coordinate form. The following code may be used.

```
/* hsl_ma86ds1.c */
#include <stdio.h>
#include <stdlib.h>
#include "hsl_mc68i.h"
#include "hsl_mc69d.h"
#include "hsl_ma86d.h"

void stop_on_bad_flag(const char *context, const int flag);

/* Code to illustrate use of more complex hsl_ma86 features */
int main(void) {
    typedef double pkgtype;

    struct mc68_control control68;
    struct mc68_info info68;

    void *keep;
    struct ma86_control control;
    struct ma86_info info;

    int i, j, n, ne, nrhs, lmap, lrow, noor, ndup, flag;
    int *crow, *ccol, *ptr, *row, *order, *map;
    pkgtype *cval, *val, *x, *x2;

    /* Read the first matrix in coordinate format */
    scanf("%d %d\n", &n, &ne);
    ccol = (int *) malloc(sizeof(int)*ne);
    for(i=0; i<ne; i++) scanf("%d", &ccol[i]);
    crow = (int *) malloc(sizeof(int)*ne);
    for(i=0; i<ne; i++) scanf("%d", &crow[i]);
    cval = (pkgtype *) malloc(sizeof(pkgtype)*ne);
    for(i=0; i<ne; i++) scanf("%lf", &cval[i]);
    /* Read the first right hand side */
    x = (pkgtype *) malloc(sizeof(pkgtype)*n);
    for(i=0; i<n; i++) scanf("%lf", &x[i]);

    /* Convert to HSL standard format */
    ptr = (int *) malloc(sizeof(int)*(n+1));
    lrow = ne; /* maximum size of output matrix cannot exceed size of input */
```

```

lmap = 2*ne; /* large enough for worst case */
row = (int *) malloc(sizeof(int)*lrow);
val = (pkgtype *) malloc(sizeof(pkgtype)*lrow);
map = (int *) malloc(sizeof(int)*lmap);
flag = mc69_coord_convert(6, HSL_MATRIX_REAL_SYM_INDEF, 0, n, n, ne,
    crow, ccol, cval, ptr, lrow, row, val, &noor, &ndup, &lmap, map);
stop_on_bad_flag("mc69_coord_convert", flag);

/* Call mc68 to find a fill reducing ordering (1=AMD) */
order = (int *) malloc(sizeof(int)*n);
mc68_default_control(&control68);
mc68_order(1, n, ptr, row, order, &control68, &info68);
stop_on_bad_flag("mc68_order", info68.flag);

/* Initialize control parameters */
ma86_default_control(&control);

/* Analyse */
ma86_analyse(n, ptr, row, order, &keep, &control, &info);
stop_on_bad_flag("analyse", info.flag);

/* Factor */
ma86_factor(n, ptr, row, val, order, &keep, &control, &info);
stop_on_bad_flag("factor", info.flag);

/* Solve */
ma86_solve(0, 1, n, x, order, &keep, &control, &info);
stop_on_bad_flag("solve", info.flag);

printf(" Computed solution:\n");
for(i=0; i<n; i++) printf(" %lf", x[i]);

/* Read the values of the second matrix and the new right hand sides */
for(i=0; i<ne; i++) scanf("%lf", &cval[i]);
scanf("%d\n", &nrhs);
x2 = (pkgtype *) malloc(sizeof(pkgtype)*n*nrhs);
for(i=0; i<nrhs; i++) {
    for(j=0; j<n; j++) {
        scanf("%lf", &x2[i*n+j]);
    }
}
printf("\n");

/* Convert the values to HSL standard form */
mc69_set_values(HSL_MATRIX_REAL_SYM_INDEF, 0, lmap, map, cval,
    ptr[n], val);

/* Perform second factorization and solve */
ma86_factor_solve(n, ptr, row, val, order, &keep, &control, &info,

```

```

    nrhs, n, x2);
stop_on_bad_flag("factor_solve", info.flag);

printf(" Computed solutions:\n");
for(i=0; i<nrhs; i++) {
    for(j=0; j<n; j++) printf(" %lf", x2[i*n+j]);
    printf("\n");
}

/* Clean up */
ma86_finalise(&keep, &control);
free(crow); free(ccol); free(cval);
free(ptr); free(row); free(val);
free(map); free(order);
free(x); free(x2);

return 0;
}

void stop_on_bad_flag(const char *context, const int flag) {
    if(0==flag) return;
    printf("Failure during %s with flag = %d\n", context, flag);
    exit(1);
}

```

The following input, supplied on stdin,

```

5 9
0 0 1 1 1 2 2 3 4
0 1 1 2 4 2 3 3 4
-3. 1. 4. 1. 1. 3. 2. 4. 2.
-1. 12. 10. 8. 4.
-5. 2. 9. 3. -2. 6. 1. -5. 6.
2
-1. 19. 28. -17. 26.
-11. 21. 14. -9. 14.

```

produces the following output.

```

Computed solution:
1.000000 2.000000 2.000000 1.000000 1.000000
Computed solutions:
1.000000 2.000000 3.000000 4.000000 5.000000
3.000000 2.000000 1.000000 2.000000 3.000000

```