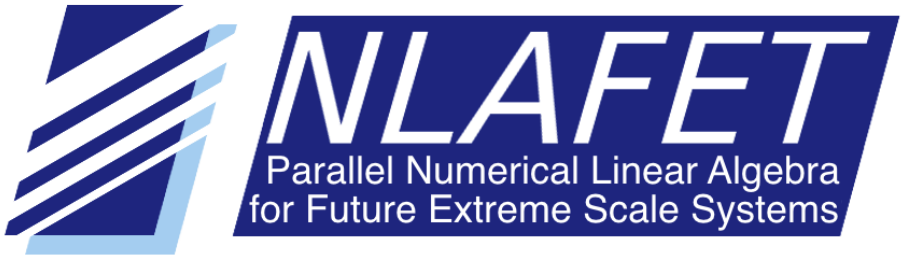


H2020–FETHPC–2014: GA 671633

D3.5

Software for highly unsymmetric factorizations

April 2018

Document information

Scheduled delivery	2018-04-31
Actual delivery	2018-04-27
Version	2.0
Responsible partner	STFC

Dissemination level

PU — Public

Revision history

Date	Editor	Status	Ver.	Changes
2018-01-24	Iain Duff	Draft	0.1	Draft to try and fix contents for deliverable
2018-04-06	Iain Duff and Stojce Nakov	Draft	1.0	Draft for internal review
2018-04-26	Iain Duff and Stojce Nakov	Draft	2.0	Draft for submission of deliverable

Author(s)

Iain Duff, STFC
Stojce Nakov, STFC

Internal reviewers

Alan Ayala, INRIA
Carl Christian Kjelgaard Mikkelsen, UMU
Sébastien Cayrols, STFC

Contributors

Tim Davis, Texas A & M University, USA

Copyright

Acknowledgements

This project has received funding from the *European Union's Horizon 2020 research and innovation programme* under the grant agreement number 671633.

Table of Contents

List of Figures

List of Tables

1 Introduction

The *Description of Action* document states for Deliverable D3.5:

> "*D3.5 Software for highly unsymmetric factorizations*
>
> Implementation of proposed methods from D3.4 on top of common task framework. Includes extensive testing, documentation and benchmarking."

This deliverable is in the context of Task 3.3 (Direct methods for highly unsymmetric systems).

This deliverable discusses software for the factorization of highly unsymmetric matrices and reports on work in Task 3.3 of Workpackage 3. We have developed a parallel algorithm for the implementation of a Markowitz/threshold strategy. We define "highly unsymmetric" matrices in Section 2 and mention other attempts to design algorithms and software for this case. We repeat our discussion from Deliverable D3.4 of what we mean by Markowitz/threshold in Section 3 before describing our parallel implementation in detail in Section 4. We do this to make this deliverable more self-contained and to define terms used in later sections. We compare our present code with that used for the Deliverable D3.4 in Section 6.1 and show the improvements over this earlier code. We present results from running our prototype code in Section 6.2 including some test comparisons with state-of-the-art codes.

To fix our notation, we will assume that we are solving the system

$$Ax = b, \tag{1}$$

where A is a sparse matrix of dimensions $n \times n$. For the matrix A we only store coefficients that can be nonzero and call these entries. It is possible that some entries might have the numerical value zero either because of operations on them or because we are studying a set of matrices where an entry is sometimes nonzero but sometimes zero. This might happen, for example, if the matrix is the Jacobian of a nonlinear problem. An entry in row i and column j is designated by a_{ij}. The right-hand side vector b and the solution vector x are of length n. In this deliverable, we consider the vectors x and b as dense. In our experiments for this deliverable, we only consider matrices with real entries although by the end of the project we will provide a version of the code for matrices and vectors with complex entries. The methods that we use for solving equation (1) are direct methods. That is we form an LDU factorization of a permutation of the matrix A, where L is a sparse lower triangular matrix, D is a diagonal matrix, and U is a sparse upper triangular matrix. The permutation is chosen to maintain sparsity in the matrices L and U while also producing a numerically stable factorization.

Our new algorithms and code are designed for execution on shared memory multi-core nodes. To obtain an efficient implementation on such architectures is already a challenge because of the low arithmetic intensity and the complexity of the underlying algorithms and data management necessitating the design of our own memory allocation routines. If such systems are to be solved on distributed memory machines, we would first perform a block partitioning of the matrix to a form of the kind shown in Figure 1. The blocks on the diagonal can be distributed while the elimination operations within each block would be performed on a single node using the algorithms and code that we now describe.

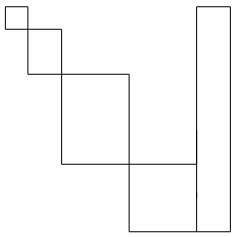


Figure 1: A singly bordered block diagonal form.

2 Highly unsymmetric matrices

We define a highly unsymmetric matrix as a matrix whose structure is not well approximated by the structure of $|A| + |A|^T$. Various authors have defined a measure of the asymmetry of a matrix and here we use that defined in [5] which is the proportion of off-diagonal entries for which there is a corresponding entry in the transpose, viz.

$$si(A) = \frac{number_{i \neq j}\{a_{ij} * a_{ji} \neq 0\}}{nz\{A\}},$$

where si is called the symmetry index and $nz\{A\}$ is the number of off-diagonal entries in the matrix A. A symmetric matrix will thus have a symmetry index of 1.0. We define 0/0 to have the value 1.0 so that a diagonal matrix will be symmetric. A triangular matrix will have symmetry index zero. Our experiments suggest that matrices with symmetry indices of less than 0.9 can be considered highly unsymmetric and these are the main target of our current work.

There are many applications that give rise to such matrices, for example, econometric modelling, chemical engineering, and linear programming although the latter has developed a large cohort of software where special techniques are used to update the factors for sequences of very related matrices.

In contrast to the case of nearly symmetric matrices, there is very little software for this class of matrices and almost no work on parallel algorithms. Available codes for this case include MA48 and `HSL_MA48` [4], UMFPACK [2], `LUSOL` [7], and `KLU` [3].

3 Markowitz/threshold pivoting

The sequential algorithms for Gaussian elimination choose one pivot at a time and update the trailing matrix (called the active matrix) before choosing the next pivot. In our discussions and notation in this section and the next one, we describe the situation at the beginning when the active matrix is the original matrix but the pivot selection is performed similarly for the successive active matrices.

We define by fill-in an entry that is zero in A but is nonzero in the corresponding entry of the factors. We note that we want our algorithm to reduce the fill-in since more fill-in has a direct adverse impact on storage and consequent extra cost in system solution. Clearly, for each pivot in Gaussian elimination the maximum fill-in that can be caused by using this pivot is the product of the number of other entries in the pivot row with the number of other entries in the pivot column. Thus if there are c_j entries in column j and r_i entries in row i, then we define the Markowitz count for a potential pivot in row i,

column j as

$$Mark_{ij} \;=\; (r_i - 1) \times (c_j - 1). \tag{2}$$

We choose candidate entries with low or minimum Markowitz count to reduce the amount of fill-in. Of course such a candidate would be unacceptable if its numerical value was zero or very small relative to other entries. We therefore introduce a threshold of acceptability for a pivot and only consider entries a_{ij} that satisfy

$$|a_{ij}| \geq u \cdot \max_k |a_{kj}|, \quad k = 1, \ldots, n \tag{3}$$

where u is a threshold parameter $0 < u \leq 1.0$. That is to say we only consider entries that are at least u times as large as the largest entry in modulus of all entries in the column. We call such entries eligible entries. If u were equal to 1.0 then we would be using partial pivoting that is the most common algorithm for dense matrices.

To continue with the factorization we must first update the trailing matrix using the outer product of the pivot row and column, updating the numerical entries and normally introducing fill-in. This is clearly a right-looking algorithm. For selecting the next pivot we then perform the Markowitz/threshold algorithm on this trailing or active matrix of order one less than the previous one, and we continue in this way until all n pivots have been chosen. The algorithm is simple but the data structures to implement it efficiently, even in serial mode, are not. We consider the details of the data structures that we use in the next section.

4 Parallel implementation of Markowitz/threshold pivoting

4.1 Introduction

For our parallel implementation, we use essentially the same pivoting strategy, that is a threshold Markowitz/threshold algorithm using the same terminology as the previous section. As is common in the design of a parallel code, we will obtain much of our parallelism using blocking. We find a block of pivots at each step rather than a single pivot as described in the previous section.

In our implementation, we find a set of independent pivots that can be used in parallel. We illustrate this in Figure 2 where the independent pivots have been permuted to the top left-hand corner of the matrix. We then use these as a block pivot to update in parallel the active matrix. We repeat these two steps on the updated Schur complement (that is the updated active matrix) and continue doing this until either the Schur complement becomes denser than a preset value or the number of pivots found is less than a preset value. In fact, when we conducted the experiments that we describe in Section 6, we found that the number of pivots chosen at each stage was not, as could be expected, monotonically decreasing but the selection of a low number of pivots might be followed by a much larger number of independent pivots. We thus monitor the number at each stage and do not switch unless the last few steps (the actual number is a parameter) have yielded only a very few pivots (another parameter). We then switch to using a dense factorization routine on the remaining Schur complement. In our present implementation on multi-core machines we use `GETRF` from `PLASMA`[1].

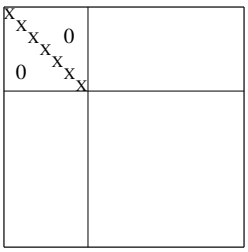


Figure 2: Block of independent pivots.

In sequential codes like MA48 [4], we select the eligible pivot which has the minimum Markowitz count, as defined in equation (2). Because we want to get large blocks of independent pivots, we relax this by accepting eligible pivots within a factor of the minimum, that is an entry (i, j) can be chosen as a pivot if its Markowitz count satisfies the condition

$$Mark_{ij} \leq \alpha_{Mark} \times Best_{Mark} \tag{4}$$

where the Markowitz factor α_{Mark} is greater than or equal to one and $Best_{Mark}$ is the lowest Markowitz count among all eligible entries.

We described algorithms for doing a parallel right-looking factorization in Deliverable D3.4. While we still follow the general framework of these earlier algorithms, we have developed a completely new pivot selection algorithm based on Luby's algorithm [6] for obtaining a maximal independent set of nodes in undirected graphs. One of our main contributions is to extend this algorithm to deal with directed graphs corresponding to unsymmetric matrices. We show, in Section 6.1, that this new algorithm outperforms the previous algorithm both in execution time and in parallel scalability.

As in the case of our Deliverable D3.4 algorithm, it is very important to first remove singletons. A *singleton* is an entry a_{ij} having no other entries in either its row i or column j. An entry a_{ij} is a row singleton if it is the only entry in its row and is a column singleton if it is the only entry in its column. Clearly, choosing a singleton as pivot will incur no fill-in.

There are two reasons for identifying and choosing singleton pivots before continuing with the main pivot selection phase. If these pivots are not handled properly, they can cause the unsymmetric-Luby search phase to find a very small pivot set. That is to say an entry a_{ij} can be a column singleton, meaning that entries a_{kj} are all zero for $k \neq i$ but there can be many entries a_{ik} that are nonzero. This is fine as a pivot choice as there would be no fill-in but the presence of the dense row will greatly reduce the number of pivots that are independent and could be chosen at this stage. For example, if the row of the column singleton were completely dense then it would not be possible to choose any independent pivots after the choice of the column singleton. The other issue is that a singleton would have zero Markowitz count. Thus the condition (4) would mean that we can only choose singletons in this pass of the algorithm.

Singletons can occur both in the original input matrix and also in the active submatrix as the factorization progresses. We thus precede the unsymmetric-Luby pivot search algorithm (described in the next section) with a phase for selecting singletons. This phase finds all the singleton pivots and eliminates them. No update of the Schur complement is required for singleton pivots, except for the removal of entries. In this case, the set of independent pivots may not form a diagonal submatrix, but they would still be independent since singletons have no effect on the Schur complement. We improve the

performance by allowing singletons and Luby-selected pivots to be used in the same pass in the update of the active matrix.

We discuss the implementation of our algorithm in the following two subsections where effectively only Section 4.2 differs from our previous work although we have also improved the other phases. We give details for the unsymmetric Luby-style pivot search in Section 4.2 and a detailed description of how we update the active submatrix via a Schur complement computation on both the column-form (pattern and values) and row-form (just the pattern) in Section 4.3. The two steps in 4.2 and 4.3 repeat until the matrix is factorized, or until the pivot sets become too small, or until the density of the active submatrix becomes too high. If the matrix is not factorized we complete the factorization using the dense factorization code `GETRF` from `PLASMA` [1].

We use the acronym ParSHUM for our code that stands for **Par**allel **S**olver for **H**ighly **U**nsymmetric **M**atrices. Details on the code are given in Section 5.

4.2 Unsymmetric Luby-style pivot search algorithm

Our new algorithm for finding a set of independent pivots uses an extension of Luby's algorithm [6] for finding a maximal independent set of nodes in an undirected graph. In common with this earlier algorithm we assign a random number to the potential pivots and use this to enable us to develop a parallel search algorithm as described in the following text.

We define an *eligible* pivot as an entry a_{ij} that is both numerically acceptable (in the sense of inequality (3)) and that has a Markowitz count that is no higher than a given multiple of the minimum Markowitz count (see condition (4)).

Our unsymmetric-Luby pivot search algorithm finds a set of eligible pivots that are structurally independent. Thus the pivots form a diagonal submatrix when they are permuted to the diagonal. After this pivot search algorithm completes, a parallel Schur complement update to the active matrix is performed (described in the next subsection), and the algorithm is used again to find another set on the active matrix. The entire process repeats until the matrix is factorized, or until too few pivots are selected or the matrix reaches a prescribed density, at which point a dense matrix factorization algorithm is used.

The pivot search algorithm can be divided into two phases: the initialisation phase and the search phase. The initialisation phase is done just once for each successive active matrix, while the search phase can be repeated until no further pivots are found.

Initialisation During this phase, a first pass by columns on the matrix is done in parallel and the numerically ineligible entries are discarded. Discarded entries are kept in the Schur complement but simply excluded as pivot candidates. This is done by partitioning each column into two sets: entries that would be numerically acceptable as pivots, and ineligible entries that are too small. At the same time, the minimum Markowitz count over all currently eligible entries is calculated. For working on subsequent Schur complements, we only have to perform these operations on columns that have been changed by the previous update, and we can identify these at the end of the search phase. Finally, a random *Luby score* is assigned to each column containing eligible pivots.

Search phase The search phase is divided into the following six steps. The method is fully parallel, with no synchronization required except for the barrier synchronizations

between each of the six steps. A critical section is needed for allocating memory space if fill-in cannot be accommodated in the current available space. No atomic operations and no other critical sections are used.

- Step 1: The columns are split into subsets and each thread searches the columns within a subset. In each column with an eligible pivot, a single *potential* pivot is found. The *potential* pivots are a superset of the final *chosen* pivots, but they may form an incompatible set. By incompatible we mean that there are potential pivots that are not independent. In fact they could even have the same row index. Steps 2 and 3 of this search phase prune this set to find a set of valid chosen pivots.

- Step 2: Each thread examines each of its potential pivots and discards those that are incompatible with other potential pivots. A thread may discard both its own potential pivots and those of another thread. Let a_{i_1,j_1} be a potential pivot. The thread that owns this pivot examines the column indices j of all nonzero entries in row i_1. If $a_{i_1,j}$ is nonzero and column j contains a potential pivot (this is recognized because column j will have a Luby score assigned to it, then we have two pivots that are incompatible: one in column j (call it a_{ij}, but the row index i is not needed) and one in column j_1 (namely, the pivot a_{i_1,j_1} owned by this thread). Let l_j be the Luby score of column j. If $l_j > l_{j_1}$, then the potential pivot in column j takes precedence over the one in column j_1, and column j_1 is flagged, by negating the appropriate entry of the inverse permutation array. If $l_j < l_{j_1}$, then the potential pivot in column j_1 takes precedence over the one in column j, and j is flagged. The potential pivot in a flagged column will not become a chosen pivot. The potential pivot in column j may be owned by another thread. To avoid race conditions, even if a potential pivot a_{i_1,j_1} is flagged during this step, it continues in this step to flag other potential pivots.

- Step 3: All threads examine their own potential pivots. If the column j_1 of a potential pivot a_{i_1,j_1} has not been flagged in step 2 above, then it becomes a chosen pivot. Each thread makes a local list of its chosen pivots. We assume that we have p threads and that thread t ($1 \le t \le p$) has found k_t chosen pivots.

- Step 4: We construct a global list of the rows and columns of the chosen pivots, by first, in Step 4, computing the cumulative sum of $k_1, k_2, \ldots, k_p$. If p is large, this can be done in parallel in $O(\log p)$ time. If p is small then a single thread can compute the cumulative sum in $O(p)$ time. This provides each thread with its positions in the global list of pivots for step 5.

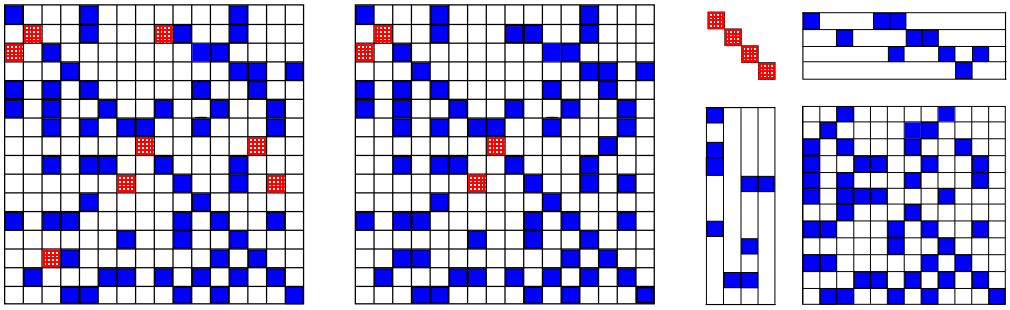
- Step 5: Each thread copies its set of chosen pivots into the global list of chosen pivots. When permuted to the top left of the active submatrix, this set of chosen pivots forms a diagonal matrix of independent pivots.

- Step 6: At the synchronization barrier between Steps 5 and 6, we can redistribute work among the threads to give an equal amount of work to each thread. We now compute the logical sum of all column indices in the pivotal rows. This will determine what columns are active in the Schur update. We do this by using an auxiliary array of length n. We scan the pivotal rows in parallel and flag the position in this auxiliary vector of each column that we encounter. There is no problem with multiple writing or race conditions. When this is completed (we need

a barrier synchronization), we scan the auxiliary vector to identify and use part of our column permutation vector to store the columns that will be updated in the Schur complement update. We perform the same operations on the pivot columns to identify rows to update in the Schur complement.

The search phase can be repeated to augment the set of chosen pivots, until no more independent pivots are found or until the number found is below a preset threshold. However, in our initial experiments we have found that a single pass provides the best overall performance.

We illustrate the selection of potential pivots and the reduction to a set of chosen independent pivots in Figure 3. The potential pivots are shown as dotted squares in Figure 3(a) and the subset of chosen pivots are shown similarly in Figure 3(b). These are permuted to the top left-hand corner in Figure 3(c).



(a) (b) (c)

Figure 3: Illustration of selection of independent pivots.

4.3 Schur complement update

The active submatrix is held in two forms: both as a set of sparse column vectors, and as set of sparse row vectors. The column-form holds the numerical values, while only the pattern is held by rows. The unsymmetric-Luby search relies on both the column and row-form. The Schur complement update is divided into two phases that are completely independent of each other. In a future version they could be done at the same time, with some threads updating the column-form while other threads update the row-form.

Updating the column-form All pivotal columns are removed from the column-form of the active submatrix and placed in L, which is held by columns.

The update of each column in the list constructed in Step 6 of the Luby pivot search is independent of any other columns, and no locks are required unless memory reallocation is required. We discuss the operations needed in column j in this list. First, column j is scanned and any pivotal rows (identified by an O(1) test on the inverse permutation array) are removed and placed in the jth column of U, which is held by columns. Next, the updates from each of the pivots corresponding to these rows in U are found and are applied to column j. We make this update more efficient and avoid potential multiple memory reallocations by using three temporary vectors of length n per thread (one with values, one with row indices, and the other with pointers into these arrays) to accumulate the updated column vector so that the resultant vector is only updated once with possible memory reallocation because of fill-in, if the updates cause column j to exceed its current

allocated space. Allocating this extra memory is done inside a critical section. Every row and column is given extra space at the beginning of the factorization to accommodate some fill-in before needing to be reallocated.

Updating the row-form The Schur complement update of the row-form is similar. Entries in pivotal columns in any given row i are placed at the front of the row. The updates from these pivots are applied one at a time, but with their pattern only and no numerical values are computed. Next, the pivotal columns are removed from row i. As in the column-form update, if row i exceeds its space, new memory is allocated to hold the row.

5 Using the ParSHUM library

We have developed a parallel library called ParSHUM that implements the algorithm described in the previous section. The configuration and installation process for the library is explained next, followed by a description of how to use the driver that is provided by the library and the interface of the library.

5.1 Installing

Getting sources. ParSHUM is developed at STFC in the context of the NLAFET project, and its source codes are available on GitHub. The source codes can be obtained with the following git command:

```
$ git clone https://github.com/NLAFET/Highly-Unsymmetric.git
```

Configuration. This library relies on cmake for the building of the library. The ParSHUM library depends on two external libraries: the PLASMA library for dense factorization and the SPRAL library for the Rutherford-Boeing matrix reader. The PLASMA library should be provided through the pkg-config tool, which helps to insert the correct compiler options. The PKG_CONFIG_PATH environment variable should be set to the directory where the libplasma.pc file is located. The root directory for the SPRAL install should be provided by using the "-DSPRAL_DIR" option when calling cmake. The verbosity of the library can be controlled with the ParSHUM_VERBOSITY variable. By default this option is activated. It can be deactivated by passing the option "-DParSHUM_VERBOSITY=0" to the cmake command. By deactivating this option, during the preprocessing phase all outputs and functions that generate additional data (timings, sizes etc) are suppressed. This should slightly increase the performance of the library.

Installing. Once the library has been downloaded into a directory, issue the following commands in that directory:

```
$ mkdir build && cd build
$ cmake -DSPRAL_DIR=<spral installation directory>
```

To build the library use:

```
$ make
```

In order to install the library use the following command:

```
$ make install
```

By default, ParSHUM will be installed in /usr/local/bin, /usr/local/lib, etc. The installation directory can be changed with the "--prefix" option of the cmake command.

5.2 Using the test driver

Once the library has been built a test driver called ParSHUM_test is created. This driver takes a large set of parameters:

- *--matrix (string)*
  specifies the path to the matrix file. Our driver is able to read Matrix Market files (suffixed by "*.ml" or "*.mm"), classic IJV files (suffixed by "*.ijv" or "*.mtl") and Harwell-Boeing matrices (suffixed by "*.hb" or "*.rb"). If a matrix file is not specified, the driver generates a random matrix of size 1000.

- *--RHS_file (string)*
  specifies the file for the RHS. It is able to use all data files as in the *--matrix* option. If an RHS is not provided, a random vector is generated.

- *--marko_threshold (real)*
  Markowitz threshold of acceptability for a pivot. This threshold should be larger than one. The default value is 4.

- *--threshold_value (real)*
  threshold value of acceptability for a pivot. This threshold should be between zero and one. The default value is 10^{-4}.

- *--schur_density_tolerance (real)*
  maximum density allowed of the Schur complement. Once the Schur complement reaches this density, we switch to dense factorization. This parameter should be between zero and one. The default value is 0.2.

- *--nb_previous_steps (integer)*
  number of previous steps for which we keep track of how many pivots have been found. This parameter should be at least one. The default value is 5.

- *--min_pivot_per_steps (integer)*
  minimum number of pivots in the last nb_previous_steps steps. If the number of pivots is less than this value, the solver switches to a dense factorization. This parameter should be at least nb_previous_steps. The default value is ten times nb_previous_steps.

- *--max_dense_schur (real)*
  the maximum allowed size for the dense factorization. If the Schur complement is larger than this value, then the dense factorization is not performed and an error is returned by the solver. The default value is 20,000.

- *--nb_threads (integer)*
  the number of threads used. This parameter should be at least 1. The default value is 1.

- *--extra_space (real)*
  the factor of the memory initially allocated for the solver in comparison with the size of the input matrix. The default value is 3.

- *--trace*
  activates a creation of a trace of the execution of the algorithm in Pajé format. By default this option is deactivated.

5.3 Using the ParSHUM interface

In this subsection we will explain how to use the interface of the ParSHUM library. The source code for the test driver presented in Section 5.2 is given in Listing 1.

Listing 1: The source code for the test driver

```
int
main(int argc, char **argv)
{
  ParSHUM_solver solver;
  ParSHUM_vector X, B;
  /* Create the solver */
  solver = ParSHUM_solver_create();

  /*  Parse the arguments */
  ParSHUM_solver_parse_args(solver, argc, argv);
  /*  Read the matrix */
  ParSHUM_solver_read_matrix(solver);

  /*  Initialize the vectors */
  X = ParSHUM_vector_create(solver->A->n);
  B = ParSHUM_vector_create(solver->A->n);
  ParSHUM_vector_read_file(solver, X);

  /* copy the vector B in X */
  ParSHUM_vector_copy(B, X);
  /*  Initialize the solver */
  ParSHUM_solver_init(solver);

  /*  Perform the factorization */
  ParSHUM_solver_factorize(solver);

  /*  Perform the solve operation */
  ParSHUM_solver_solve(solver, B);

  /*  Compute the norms */
  ParSHUM_solver_compute_norms(solver, X, B);

  /*  Finalize the solver */
  ParSHUM_solver_finalize(solver);

  /*  Free all the data */
  ParSHUM_vector_destroy(X);
  ParSHUM_vector_destroy(B);
  ParSHUM_solver_destroy(solver);

  return 0;
}
```

First the solver structure is created by calling ParSHUM_solver_create in line 7. This will only allocate the basic data for the solver. Then on line 10, the arguments are processed and the solver's parameters are set to the values provided. The matrix is then read from the file followed by the initialisation of the two vectors B and X. Once everything is in place, the function ParSHUM_solver_init is called which will allocate all the internal data that is needed for the factorization. The factorization is performed by the function ParSHUM_solver_factorize on line 25, followed by the solve operation on line 28 and the computation of the norms on line 31. The ParSHUM_solver_finalize needs to be called before the data is freed. This function finalises the PLASMA library, creating the trace file if requested and prints the output of the solver. Finally, all the data that has been used is freed (lines 37-39). Alternatively, is possible for the user to control the input matrix and parameters for the solver directly in the code and we now explain how it can be done.

In order to change the input matrix and parameters, we use the ParSHUM_solver structure, a short version of which is presented in Listing 2. The only fields that should be modified by the user are the input matrix A (see Listing 3) and the exe_parms field shown in Listing 4. We will now show how this could be done.

Listing 2: The ParSHUM_solver structure

```
1  typedef struct _ParSHUM_solver {
2    ParSHUM_matrix A;  /* The input matrix */
3
4    /* The factors */
5    ParSHUM_L_matrix L;
6    ParSHUM_matrix D;
7    ParSHUM_U_matrix U;
8
9    /* The Schur complement */
10   ParSHUM_schur_matrix S;
11
12   ParSHUM_exe_parms exe_parms;
13   ParSHUM_verbose verbose;
14   .....
15 } * ParSHUM_solver;
```

In the example in Listing 1, the input matrix was loaded on line 12 by calling the function ParSHUM_solver_read_matrix. The other option is to set it as the input matrix in the ParSHUM_solver structure. The structure of the ParSHUM_matrix type is given in Listing 3 and an example of how to assign a CSR matrix as the input matrix is shown in that listing.

In Listing 1 where the code for the driver is presented, the solver is parametrized by calling the function ParSHUM_solver_parse_args in line 10. The other option is to modify directly the exe_parms field of the solver structure. The exe_parms field is of type ParSHUM_exe_parms and an example of how to use this structure to change the threshold value and Markowitz threshold is given in Listing 4. All the other parameters can be changed in the same way.

Listing 3: An example of a user-provided matrix

```
typedef enum _ParSHUM_matrix_type
{
  ParSHUM_CSC_matrix,
  ParSHUM_CSR_matrix,
  ParSHUM_Diag_matrix,
  ParSHUM_Rutherford_matrix
} ParSHUM_matrix_type;

typedef struct _ParSHUM_matrix {
  ParSHUM_matrix_type type;
  int n;

  long allocated;
  long nz;

  int *row;
  long *col_ptr;
  double *val;

  int *col;
  long *row_ptr;

  /* This is used by the SPRAL matrix driver since it is a Fortran code (see
      http://www.test-numerical.rl.ac.uk/spral/doc/sphinx/C/rutherford_boeing.html) */
  void *handle;
} * ParSHUM_matrix;

   .....
   /* An example for assigning a CSR matrix as an input matrix.
   The following code should replace the line 12 in the previous example. */

   ParSHUM_matrix matrix = calloc(1, sizeof(struct _ParSHUM_matrix);
   matrix->type = ParSHUM_CSR_matrix;
   matrix->n = n;
   matrix->allocated = nz;
   matrix->nz = nz;
   /* If the input matrix is a CSC matrix, the field type should be assigned with
       ParSHUM_CSC_matrix and instead of assigning arrays to the col and row_ptr
       fields, the row and col_ptr fields should be assigned. */
   matrix->col = col;
   matrix->row_ptr = row_ptr;
   solver->A = matrix;
   .....
```

Once the matrix and the parameters are set, the solver should be initialised by calling the ParSHUM_solver_init. From this point the ParSHUM_solver structure should not be accessed or modified directly by the user.

Listing 4: An example of parametrizing the solver

```
typedef struct ParSHUM_exe_parms {
  double value_tol;
  double marko_tol;
  double extra_space;
  double density_tolerance;

  char *matrix_file;
  char *RHS_file;

  int min_pivot_per_steps;
  int nb_threads;
  int nb_previous_pivots;
  int max_dense_schur;
  int trace;
} *ParSHUM_exe_parms;

   .....
   /* A code example for changing the parameters of the solver */
   /* The following code should replace the call the ParSHUM_solver_parse_args (line
       10). */
   solver->exe_parms->value_threshold = 0.001;
   solver->exe_parms->marko_threshold = 16;
   .....
```

6 Performance on applications

In this section we present results obtained from the ParSHUM library. All the tests performed in this section were performed on a system called Kebnekaise, which is located in the High Performance Computing Center North (HPC2N) at Umeå University[1]. Each compute node contains 28 Intel Xeon E5-2690v4 cores organized into 2 NUMA islands with 14 cores in each. The nodes are connected with a FDR Infiniband Network. Each CPU core has 32 KB L1 data cache, 32 KB L1 instruction cache and 256 KB L2 cache. Moreover, for every NUMA island there is 35 MB of shared L3 cache. The total amount of RAM per compute node is 128 GB. In our experiments, we use only one NUMA node, so all the tests presented below are executed on fourteen cores.

All the libraries used in these runs were compiled with GCC V6.4.0. PLASMA V3.0.0 is used for the dense factorization. PLASMA uses the Intel MKL 2017.3.196 Library for the BLAS operations. The matrices lung2, twotone and hvdc2 are from the SparseSuite set of test matrices and the other six are from the Power Systems application supplied by Bernd Klöss of DigSILENT GmbH (see Deliverable 5.1). The main attributes of the matrices used in this study are given in Table 1. For a given matrix A, ParSHUM factorizes the matrix as:

$$PAQ = LU,$$

where P and Q are row and column permutation matrices respectively, and L and U are lower and upper triangular matrices respectively. We define the fill-in factor as the number of entries (nz) in the L and U factors divided by the number of entries in A viz:

$$\frac{nz\{L\} + nz\{U\}}{nz\{A\}}.$$

[1] See https://www.hpc2n.umu.se/resources/hardware/kebnekaise.

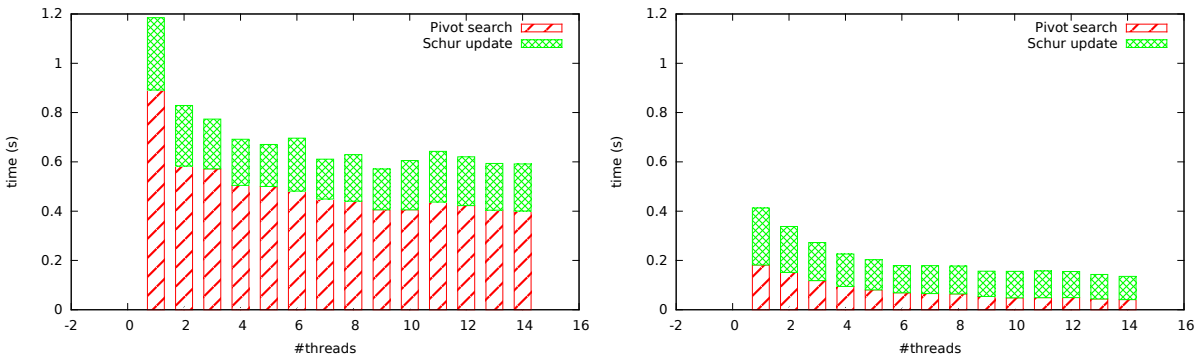
First, we present a comparison with the algorithm presented in Deliverable 3.4, followed by a numerical stability and performance assessment of our solver, and then we compare our code with two state-of-the-art solvers, MA48 V2.2.0 and UMFPACK V5.6.1.

Matrix	n	nz	si
lung2	109K	492K	0.57
twotone	120K	1.22M	0.26
hvdc2	190K	1.35M	0.99
InnerLoop1	197K	745K	0.44
InnerLoop2	197K	806K	0.46
InnerLoop3	197K	806K	0.46
InnerLoop4	197K	806K	0.46
Jacobian_unbalancedLdf	203K	2.41M	0.80
Newton_Iteration1	427K	2.38M	0.14

Table 1: Statistics for the matrices that are used in this study.

6.1 Comparison with the algorithm from Deliverable D3.4

The algorithm that was presented in Deliverable D3.4 was based on creating multiple independent sets and merging them using a tree based reduction. With this design, one potential pivot could potentially be processed as many times as the height of the reduction tree. On the other hand, with our current algorithm, each potential pivot will be treated only once.



(a) (b)

Figure 4: The execution time for the InnerLoop1 matrix for the algorithm presented in Deliverable D3.4 (left) and the current algorithm (right).

This effect is most noticeable for the InnerLoop1 matrix. The execution times for the InnerLoop1 matrix for algorithm from Deliverable 3.4 and our current implementation are presented in Figures 4a and 4b respectively. All the other matrices have similar behaviour.

6.2 Performance assessment

First we investigate the numerical stability of our algorithm by calculating the backward error as:

$$\frac{\|b - Ax\|_2}{\|b\|_2 + \|A\|_\infty \|x\|_2}.$$

In Figure 5 we present the impact on the backward error of the threshold parameter u in equation (3) for selecting pivots. When a low threshold is used, pivots with smaller values are accepted resulting in an increase of the backward error. This is more noticeable for the twotone and hvdc2 matrices (see Figure 5). However, when the threshold is increased, we obtain a backward error of 10^{-16} for all the matrices.

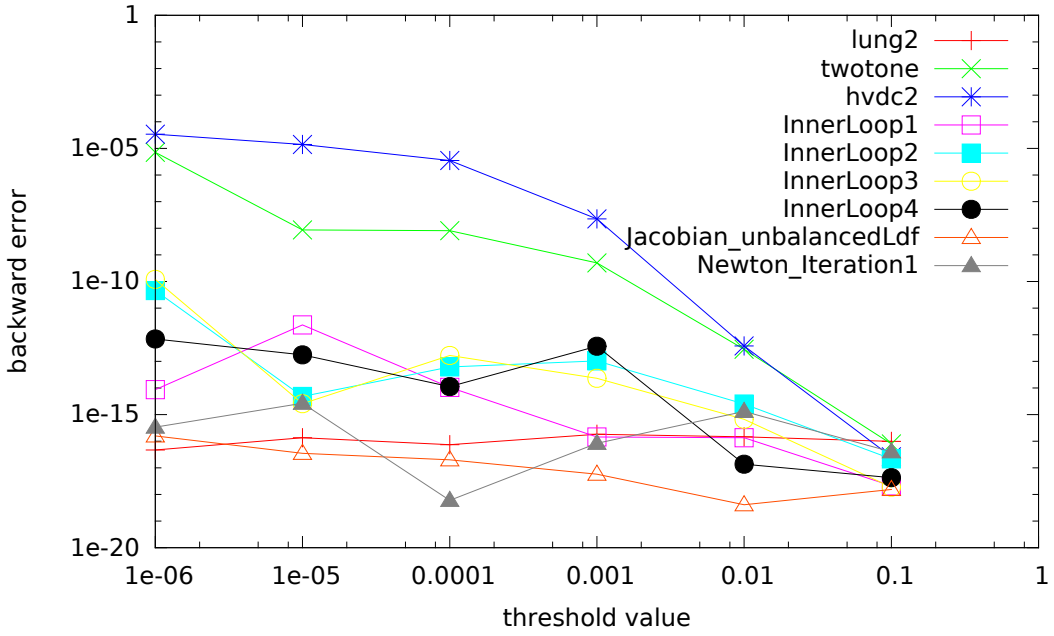


Figure 5: The impact of the threshold parameter, u, on the backward error for ParSHUM.

Another parameter in our algorithm is the Markowitz threshold. By relaxing this parameter, we allow pivots with higher Markowitz count to be chosen as pivot. These pivots could potentially increase the fill-in in the factors. We observe the effect of this parameter on the fill-in for each matrix in Figure 6. When the Markowitz threshold is increased, each matrix tends to have increased fill-in.

In order to get good performance for our algorithm, we need to find the best combination of three main parameters: a good threshold parameter so that we get a numerically correct solution; a small enough Markowitz tolerance so that the fill-in is reasonable but a relatively large number of pivots are obtained at each stage; and the Schur density for determining when to switch to the dense code. We have done extensive testing and for each matrix we have calculated the best combination of parameters in terms of execution time for all three solvers (the Markowitz threshold does not make sense for the MA48 and UMFPACK solvers, and the density switch does not make sense for the UMFPACK solver). These parameters are presented in Table 2. All the tests from now on use these values for the parameters, and we ensure that all our results have a backward error of at least 10^{-12}.

In Figure 7, we present the execution time for two matrices, Jacobian_unbalancedLdf and InnerLoop1 when the number of threads is increased. The algorithm is

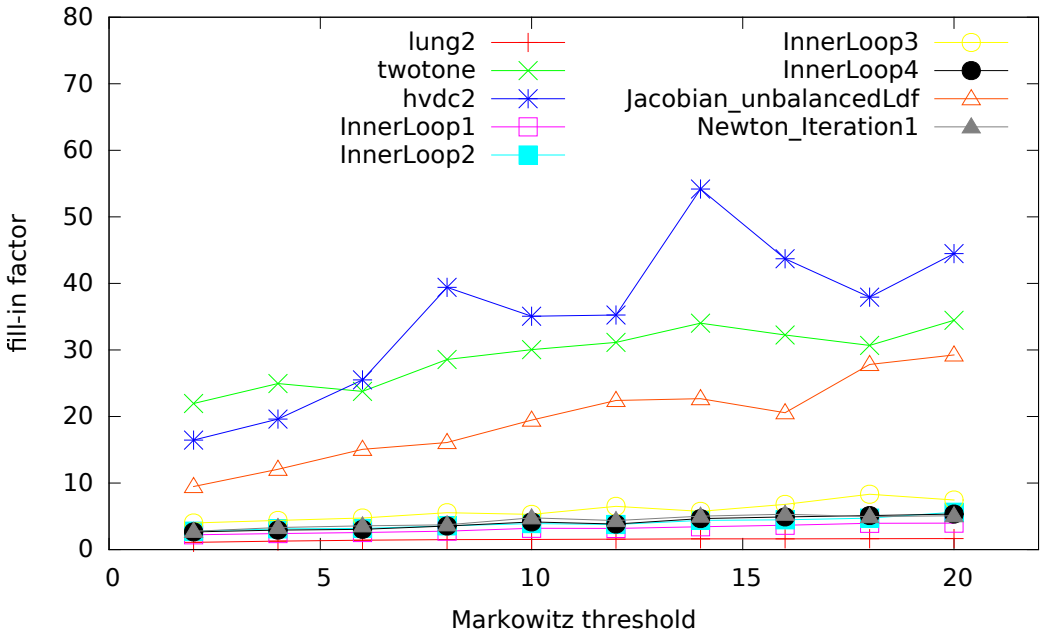


Figure 6: The impact of the Markowitz threshold on the fill-in factor for ParSHUM.

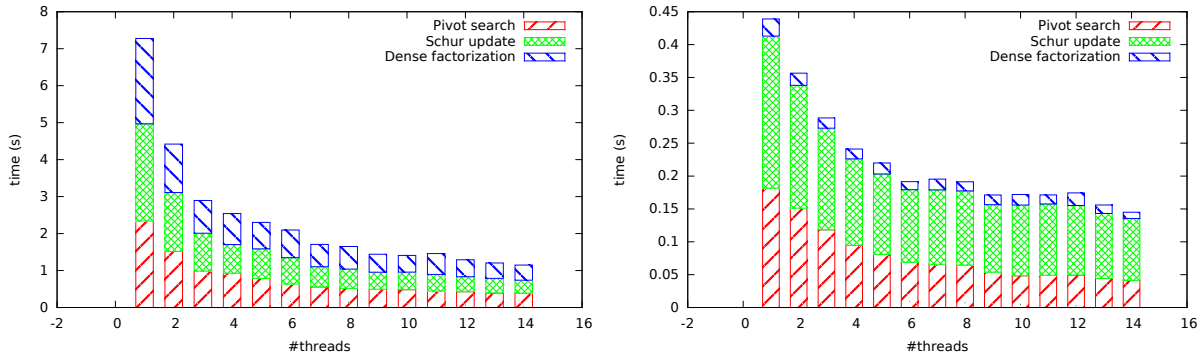


Figure 7: Multi-threaded results for our algorithm. The Jacobian_unbalancedLdf matrix is presented on the left and the InnerLoop1 matrix is presented on the right.

clearly a memory bandwidth bound algorithm. This effect is very evident for the Jacobian_unbalancedLdf matrix. When the number of threads increase, the execution time for all steps decreases for up to eight threads, at which point the bandwidth is saturated. The execution time decreases from 7.2 secs in the sequential case, down to 1.7 secs when eight threads are used. But when fourteen threads are used, the execution time is 1.2 secs. On the other hand, we do not get the same scalability for the InnerLoop1 matrix, mainly because of granularity issues.

In Table 3 the optimal execution times for MA48, UMFPACK and ParSHUM are presented. We obtain the lowest execution times with ParSHUM for all the matrices except the hvdc2 and the Jacobian_unbalancedLdf matrix for which the UMFPACK solver yields the lowest execution time. The main reason for this is the fill-in factor. For instance, for the Jacobian_unbalancedLdf matrix, the UMFPACK solver has a fill-in of 3.88 while a fill-in of 12.4 is obtained with the ParSHUM solver.

	ParSHUM			MA48		UMFPACK
Matrix	u	α_{Mark}	Φ	u	Φ	u
lung2	10^{-6}	2	0.1	1.0	0.4	10^{-4}
twotone	10^{-2}	4	0.15	10^{-2}	0.8	10^{-5}
hvdc2	10^{-2}	3	0.09	10^{-1}	0.4	10^{-5}
InnerLoop1	10^{-4}	9	0.1	10^{-3}	0.8	10^{-7}
InnerLoop2	10^{-4}	8	0.04	10^{-3}	0.6	10^{-3}
InnerLoop3	10^{-4}	9	0.02	10^{-4}	0.8	10^{-7}
InnerLoop4	10^{-4}	9	0.04	10^{-4}	0.2	10^{-3}
Jacobian_unbalancedLdf	10^{-6}	4	0.08	10^{-3}	0.8	10^{-1}
Newton_Iteration1	10^{-6}	16	0.08	10^{-2}	0.8	10^{-4}

Table 2: The parameters and their optimal values for the results in Figure 7 and Table 3 are shown in columns 2-4 (ParSHUM), 5-6 (MA48) and 7 (UMFPACK). The parameters u and α_{Mark} are defined in Sections 3 and 4.1 and Φ is the density at which the switch to full code is made.

	ParSHUM		MA48		UMFPACK	
Matrix	time	fill-in	time	fill-in	time	fill-in
lung2	**0.05**	1.10	0.37	2.55	0.10	1.44
twotone	**0.38**	9.70	4.00	18.8	0.49	5.45
hvdc2	0.47	6.92	0.42	1.82	**0.38**	2.06
InnerLoop1	**0.15**	3.00	0.22	1.99	0.30	2.48
InnerLoop2	**0.17**	3.60	0.23	1.94	0.29	2.40
InnerLoop3	**0.18**	5.64	0.23	1.91	0.29	2.37
InnerLoop4	**0.16**	3.72	0.23	1.91	0.29	2.40
Jacobian_unbalancedLdf	1.15	12.4	4.21	5.40	**0.74**	3.88
Newton_Iteration1	**0.66**	4.99	1.23	2.55	1.00	2.55

Table 3: The execution time and the fill-in factor for ParSHUM, MA48 and UMFPACK solvers. The results correspond to the best execution for each algorithm with the parameter values given in Table 2. The lowest execution time is shown in bold type.

7 Acknowledgments

This project is funded from the European Union's Horizon 2020 research and innovation programme under the NLAFET grant agreement No 671633. We thank the High Performance Computing Center North (HPC2N) at Umeå University for providing computational resources and valuable support.

References

[1] A. Buttari, J. Langou, J. Kurzak, and J. Dongarra, *A class of parallel tiled linear algebra algorithms for multicore architectures*, Parallel Computing, 35 (2009), pp. 38–53.

[2] T. A. Davis and I. S. Duff, *An unsymmetric-pattern multifrontal method for sparse LU factorization*, SIAM J. Matrix Analysis and Applications, 18 (1997), pp. 140–158.

[3] T. A. Davis and E. P. Natarajan, *Algorithm 907: KLU, A direct sparse solver for circuit simulation problems*, ACM Trans. Math. Softw., 37 (2010), p. 17 pages.

[4] I. S. Duff and J. K. Reid, *The design of MA48, a code for the direct solution of sparse unsymmetric linear systems of equations*, ACM Trans. Math. Softw., 22 (1996), pp. 187–226.

[5] A. M. Erisman, R. G. Grimes, J. G. Lewis, W. G. Poole Jr., and H. D. Simon, *Evaluation of orderings for unsymmetric sparse matrices*, SIAM Journal on Scientific and Statistical Computing, 7 (1987), pp. 600–624.

[6] M. Luby, *A simple parallel algorithm for the maximal independent set problem*, SIAM J. Computing, 15 (1986), pp. 1036–1053.

[7] M. A. Saunders, *LUSOL: Sparse LU for $Ax = b$*, 2009. `http://stanford.edu/group/SOL/lusol/`.