NLAFET
Parallel Numerical Linear Algebra for Future Extreme Scale Systems

H2020–FETHPC–2014: GA 671633

# D3.3

# Software for Symmetrically Structured Factorizations

January 2019

## Document information

| | |
|---|---|
| Scheduled delivery | 2019-01-31 |
| Actual delivery | 2019-01-31 |
| Version | 2.0 |
| Responsible partner | STFC |

## Dissemination level

PU — Public

## Revision history

| Date | Editor | Status | Ver. | Changes |
|---|---|---|---|---|
| 2018-10-12 | Iain Duff | Draft | 0.1 | Skeleton draft |
| 2019-01-16 | Iain Duff and Florent Lopez | Internal review | 1.0 | Major update |
| 2019-01-29 | Iain Duff and Florent Lopez | Submitted version | 2.0 | Minor update |

## Author(s)

Iain Duff, RAL
Florent Lopez, RAL

## Internal reviewers

Jan Papez, Inria
Srikara Pranesh, UMAN

## 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 3.3:
"*Implementation and adaption of methods from D3.2 on top of the common task framework. Extension from symmetric to unsymmetric (but symmetrically structured) case. Includes extensive testing, documentation and benchmarking.*"

This deliverable is the final one related to the work performed in WP3, Task 3.2 (Direct methods for (near-)symmetric systems). Our work has focused on the design and implementation of codes using matrix factorizations to solve large sparse linear systems of equations $Ax = b$, where $A$ is a symmetrically structured matrix. We discussed the design for our suite of routines in Deliverable D3.2 submitted at M24 (October 2017). Since then we have developed new routines for the solution of indefinite symmetric and unsymmetric but structurally symmetric systems. All our routines are task based. They use the runtime system StarPU for managing the task graph and are focused on single node performance although the node can comprise multiple NUMA sections and GPUs.

The earlier work on Task 3.2 that was described in Deliverable D3.2 included an in-depth discussion of the basis for our algorithms that we do not repeat here. In D3.2 we also discussed the SpLLT code[1] for symmetric positive definite systems and SSIDS, a Sparse Symmetric Indefinite Solver, that is included in the SPRAL Library. Some details of SSIDS are included in a more recent technical report that has been submitted for publication [8]. Since we submitted Deliverable 3.2, we have worked more on improving SpLLT and have compared it more with other state-of-the-art codes and show some of these comparisons in Section 4.1. We have made other significant improvements to the code that we mention in Section 2.1 and discuss in Section 4.3. We have also developed a version of SpLLT that works on nodes with GPUs and include a short discussion of this in Section 4.2. Although our new codes, SpLDLT and SpLU, in what we call the S$y$LVER package, represent significant further developments they share a number of kernels with SSIDS, and we show these as dependencies in Section 2.2 by requiring the pre-installation of SPRAL.

We introduce the S$y$LVER package in Section 2.1 and discuss the external libraries that need to be linked with S$y$LVER in Section 2.2. We describe the installation procedure in Section 2.3, the input parameters in Section 3.1 and provide simple test programs in Section 3.2. We present some results in Section 4.

# 2 Using the NLAFET S$y$LVER package

## 2.1 Overview

The S$y$LVER package is included in the library developed at RAL as part of the NLAFET project. It is available from `https://github.com/NLAFET/`. The S$y$LVER package provides an efficient implementation of codes for two different types of input matrix:

- SpLDLT for symmetric positive definite or indefinite matrices.
- SpLU for symmetrically structured unsymmetric matrices.

[1]`https://github.com/NLAFET/SpLLT`

The basic algorithms and design are the same for both codes.

The S$y$LVER package is similar to SpLLT in that it provides a tree-based implementation of a sparse direct method that accomplishes the solution of sparse equations in three steps:

1. an *analysis* that determines an ordering based on the structure of the matrix and computes data structures used in the factorization including the assembly tree,
2. a numerical factorization, *factorize*, that computes the matrix factors using output from the analysis and the numerical values of the matrix. In the case of SpLLT, the factorization will follow exactly the computations defined by the tree but for SpLDLT and SpLU it will perform additional numerical pivoting to ensure stability of the factorization,
3. a *solve* routine that uses the factors produced in the previous step to solve the system with one or more right-hand sides.

In addition to the numerical pivoting, another difference between SpLLT and S$y$LVER is that SpLLT processes the tree using a supernodal approach whereas S$y$LVER is a multifrontal code. Our experience is that the multifrontal approach is both easier to implement and more suited for a GPU implementation but requires more memory. For background information on this approach to the direct solution of sparse equations, we refer the reader to the book [6].

We described the earlier SpLLT code [7] for symmetric positive definite systems in Deliverable 3.2. Since then a significant and important addition has been a blocked and parallel routine for the *solve* phase [4]. We illustrate the performance of this enhanced parallel solve in Section 4.3.

A detailed description of the parallel pivoting strategy used in both codes is given in [8]. SpLDLT has a similar functionality to previous HSL parallel codes [12, 13] but SpLU addresses a different class of matrices, specifically matrices that are unsymmetric but structurally symmetric. The HSL code MA41 [2] solved systems with such coefficient matrices and the later development of MUMPS [1] also includes an entry for these matrices. Our code is different inasmuch as its primary target is for shared-memory multicore machines.

## 2.2 Dependencies with external libraries

S$y$LVER depends on a few external libraries that need to be installed and linked with S$y$LVER in order to use it:

- BLAS and LAPACK [3]: BLAS is a standard library for performing basic vector and matrix operations. LAPACK is a standard software package for numerical linear algebra. Although any library providing BLAS and LAPACK can be used, we recommend MKL [17] (`https://software.intel.com/en-us/mkl`).
- CUBLAS from Nvidia CUDA (`https://developer.nvidia.com/cuda-zone`) is required if GPUs are to be used.
- MeTiS [14] is a sequential graph partitioning tool. S$y$LVER was tested with MeTiS 5.1.0. This partitioning tool can be downloaded from `http://glaros.dtc.umn.edu/gkhome/metis/metis/overview`.

- SPRAL (`http://www.numerical.rl.ac.uk/spral/`) is used for scaling and analysis routines and for kernels from SSIDS.
- StarPU (`http://starpu.gforge.inria.fr/`) is a runtime system that we use to exploit the parallel architecture. It is necessary to install this for compiling the parallel code.

## 2.3 Installation

We now describe the complete installation of S*y*LVER.

### 2.3.1 Install the dependencies

The installation of the first three packages in Section 2.2 is well described in their respective documentation therefore we discuss here the installation of SPRAL and StarPU. The compilation process for both installations is handled by the GNU Autotools package[2].

**Installing SPRAL** The latest development version of the SPRAL library can be found on the GitHub repository `https://github.com/ralna/spral`. Instructions for compiling SPRAL are shown in Listing 1.

```
# Get latest development version from github and run dev scripts
git clone --depth=1 https://github.com/ralna/spral.git
cd spral
./autogen.sh

# Build and install library
mkdir build
cd build
../configure --with-metis='-L/path/to/metis -lmetis'
make
sudo make install # Optional
```

Listing 1: Compiling the SPRAL library.

**Installing StarPU** The latest development version of the StarPU runtime system can be downloaded via git. Instructions for downloading and compiling StarPU are shown in Listing 2.

```
# Get latest development version via git
git clone https://scm.gforge.inria.fr/anonscm/git/starpu/starpu.git
cd starpu
./autogen.sh

# Build and install library
mkdir build
cd build
../configure

make
sudo make install # Optional
```

Listing 2: Compiling the StarPU library.

[2]`https://www.lrde.epita.fr/~adl/autotools.html`

### 2.3.2 Get and install S$y$LVER

The code is held in the GitHub repository `https://github.com/NLAFET/Sylver` and the compilation is handled by CMake[3] tools. Here are the steps to follow to get and install the package.

1. Get the latest version of S$y$LVER from the GitHub repository.

```
git clone git@github.com:NLAFET/sylver.git sylver
```

2. Use the `cmake` command to configure the compilation and generate the Makefile in the build directory.

```
cd sylver
mkdir build
cd build
cmake ..
```

   The choice of the runtime system can be made using the `-DRUNTIME` option. In order to use the StarPU runtime system set `-DRUNTIME=StarPU` as following:

```
cmake .. -DRUNTIME=StarPU # Use the StarPU runtime system
```

   And for generating the serial code:

```
cmake .. -DRUNTIME=Serial # produces a serial code
```

3. Type `make` to compile S$y$LVER and create the library `libsylver.a` and the example programs.

```
make
```

4. The subroutines from the S$y$LVER library can now be linked to a program as follows:

```
cd <path-to-your-code>
gfortran -o myprog myobj.o -lsylver -lmetis -lblas \
  -lstarpu-1.3
```

   The directory `drivers` provides a few stand-alone programs to test the library.

## 3 Codes in the S$y$LVER package

After the package is installed as indicated in the previous section, detailed documentation on how to use it can be found in the doc directory. However, we feel that it is useful to give an overview of the codes and their parameters to indicate what software is available in this deliverable. There are many more control parameters and more detailed information returned by the codes that are described in full in the online documentation.

There are two codes in the S$y$lver package, SpLDLT and SpLU as mentioned in Section 2.1. Each code has three callable routines performing the operations as mentioned in

[3]`https://cmake.org/`

Section 2.1. These are called `spldlt_analyse`, `spldlt_factorize`, and `spldlt_solve` and similar names for the SpLU codes.

The codes are written in Fortran 2003 but there are bindings for both C and C++.

## 3.1 Subroutine parameters

We describe the parameters to each of the three entries for SpLDLT. The SpLU routines have the same parameters but they may hold different data. For example, `akeep` when called by spldlt_analyse is of type spldlt_akeep but of type splu_akeep when called by splu_analyse.

1. `spldlt_analyse(akeep,n,colPtr,rowInd,options,inform,[order,val,ncpu])`
2. `spldlt_factorize(akeep,fkeep,pos_def,val,options,inform[,scaling])`
3. `spldlt_solve(akeep,fkeep,nrhs,x,ldx,inform)`

S*y*LVER requires input matrices to be in compressed sparse column format (CSC). This format stores the nonzero entries of the matrix by columns in twin real and integer arrays with a pointer to the first entry in each column, viz:

- `n` [integer]: the size of the matrix.
- `colPtr` [integer(long)]: the beginning position of each column within the arrays `rowInd` and `val`.
- `rowInd` [integer]: the row indices of the nonzero entries of the matrix.
- `val` [real]: the corresponding values of the nonzero entries of the matrix.

We show a matrix being input in this format in the example program in Section 3.2. There is also a wrapper so that the user can input the matrix in coordinate form.

The `akeep` and `fkeep` derived data types are really not of concern to the user but they are present so that information can be shared between the three subroutines. That is, *factorize* needs the ordering and tree information held in akeep that is generated by *analyse*. *solve* needs information in akeep that is generated by *analyse* and also in fkeep that is generated by *factorize*.

The parameter `options` is used to control the code and to set various options. We list the components of most interest below. There are several other components that could be of interest to specialist users that are presented in the online documentation.

- `print_level [integer, default=0]`: the level of printing. Negative for no printing, zero for errors and warnings only, one for some diagnostics, and higher values for additional diagnostic printing.
- `unit_error [integer, default=6]`: Fortran unit number for printing of error messages. Printing is suppressed if $< 0$.
- `unit_warning [integer, default=6]`: Fortran unit number for printing of warning messages. Printing is suppressed if $< 0$.
- `unit_diagnostics [integer, default=6]`: Fortran unit number for diagnostics printing. Printing is suppressed if $< 0$.

- `use_gpu [logical, default=.true.]`: If .true., the code will use an Nvidia GPU.
- `scaling [integer, default=0]`: scaling algorithm used. Options are:

  `0` : for no scaling or input scaling,

  `1` : weighted matching using code based on the MC64 algorithm [9],

  `2` : weighted matching using an auction algorithm,

  `3` : this option is discussed in online documentation,

  `4` : norm-equilibration algorithm [15, 16].
- `small [real, default=1d-20]`: threshold below which an entry is treated as equivalent to 0.0.
- `u [real, default=0.01]`: relative pivot threshold used in numerical pivoting. Negative values are treated as 0 and values greater than 0.5 as 0.5 for the $LDL^T$ factorization. For the LU factorization, values greater than 1.0 are treated as 1.0.
- `unsymm [integer, default=1]`: SpLU only. `0`: matrix is symmetrically structured, `1`: matrix is unsymmetric also in pattern.

The parameter `inform` provides information on the running of the routine. We list the components of most interest below. There are several other components that could be of interest to specialist users and that are documented in the online documentation.

- `flag [integer]`: exit status of the algorithm. Zero if successful, positive if a fatal error, and negative if a warning.
- `num_flops [integer (long)]`: number of floating-point operations required to factorize the matrix.
- `num_factor [integer (long)]`: number of entries in the factors, $L$ (without numerical pivoting after analyse phase, and including numerical pivoting after factorize phase) in the Cholesky and $LDL^T$ factorization, and $L$ and $U$ in the case of the $LU$ factorization.
- `matrix_rank [integer]`: (estimated) rank (structural after analyse phase, numerical after factorize phase).
- `stat [integer]`: Fortran allocation status parameter in event of allocation error (0 otherwise).

The other parameters in the call to the *analyse* entry are optional. `order` is an integer array and can be used to define a user supplied ordering for the analysis. If it is omitted, the default ordering is to use a nested dissection ordering from METIS. Other possibilities are discussed in the online documentation. For one of these orderings, it is necessary to supply the matrix values in the array `val`. The user can set `ncpu` to the number of cores to use. It is an optional parameter and, if omitted, the routine will use *hwloc*[4] to obtain information on the hardware.

The other parameters in the call to the *factorize* entry are:

[4]https://www.open-mpi.org/projects/hwloc/

- `pos_def [logical]`: that the user must set to indicate whether the matrix is positive definite (`pos_def =.true.`) or not (`pos_def = .false.`)

and an optional parameter

- `scaling [real]`: that can be set to a user input scaling vector in the case when the `options%scaling` component is set to `zero`.

The other parameters in the call to the *solve* entry are

- `nrhs [integer]`: number of right-hand sides.
- `x [real]`: right-hand side(s) on input and solution(s) on output.
- `ldx [integer]`: if there is more than one right-hand side (nrhs $\geq$ 2) then ldx is the leading dimension of the matrix x.

There are three other user-callable routines.

- `sylver_init(ncpu,ngpu)` to initialize the package and StarPU.
- `sylver_finalize()` to release memory and clean up data structures at the end of the run.
- `spldlt_free(akeep,fkeep)` to deallocate the arrays `akeep` and `fkeep`.

The parameters to `sylver_init` can be used to set the number of cores (`ncpu`) and the number of GPUs (`ngpu`). The value of `ngpu` will be ignored if `options%use_gpu` is set to `.false.`.

## 3.2 Example programs

The following program listings illustrate how S*y*LVER can be used for solving a sparse linear system. We provide a simple Fortran program using SpLDLT in Listing 3 and a C program using SpLU in Listing 4.

```fortran
program spldlt_example
  use spldlt_mod
  implicit none

  ! Derived types
  type (spldlt_akeep)   :: akeep
  type (spldlt_fkeep)   :: fkeep
  type (sylver_options) :: options
  type (sylver_inform)  :: inform

  ! Parameters
  integer, parameter :: long = selected_int_kind(16)
  integer, parameter :: wp = kind(0.0d0)

  ! Matrix data
  logical :: posdef
  integer :: n, row(9)
  integer(long) :: ptr(6)
  real(wp) :: val(9)

  ! Other variables
  integer :: ncpu, ngpu
  integer :: nrhs
  real(wp) :: x(5)

  ! Data for matrix:
  ! ( 2  1          )
  ! ( 1  4  1     1 )
  ! (    1  3  2    )
  ! (       2 -1    )
  ! (    1        2 )
  posdef = .false.
  n = 5
  ptr(1:n+1)        = (/ 1,        3,              6,        8,    9,  10
    /)
  row(1:ptr(n+1)-1) = (/ 1,   2,   2,   3,   5,   3,   4,   4,    5   /)
  val(1:ptr(n+1)-1) = (/ 2.0, 1.0, 4.0, 1.0, 1.0, 3.0, 2.0, -1.0, 2.0 /)

  ! The right-hand side with solution (1.0, 2.0, 3.0, 4.0, 5.0)
  nrhs = 1
  x(1:n) = (/ 4.0, 17.0, 19.0, 2.0, 12.0 /)

  ncpu = 1
  ngpu = 0

  call sylver_init(ncpu, ngpu)

  ! Perform analyse and factorize
  call spldlt_analyse(akeep, n, ptr, row, options, inform)
  if(inform%flag<0) go to 100

  call spldlt_factorize(akeep, fkeep, pos_def, val, options, inform)
  if(inform%flag<0) go to 100

  call spldlt_solve(akeep, fkeep, nrhs, x, n, inform)
  if(inform%flag<0) go to 100
  write(*,'(a,/,(3es18.10))') ' The computed solution is:', x(1:n)

100 continue
  call spldlt_free(akeep, fkeep)

  call sylver_finalize()

end program spldlt_example
```

Listing 3: Simple example program in Fortran using SpLDLT.

```c
#include "splu.h"

int main(void) {

   void *akeep, *fkeep;
   splu_inform_t info;
   splu_options_t options;

   /* Data for matrix:
    * ( 2 -1          )
    * ( 1  4 -1    -1 )
    * (     1   3  -2    )
    * (         2  -1    )
    * (     1         2 ) */

   int n = 5;
   long ptr[]   = { 1,        3,                    7,               10,       12,        14 };
   int row[]    = { 1,   2,   1,    2,   3,   5,   2,    3,   4,   3,   4,    2,    5   };
   double val[] = { 2.0, 1.0, -1.0, 4.0, 1.0, 1.0, -1.0, 3.0, 2.0, -2.0, -1.0, -1.0, 2.0 };

   /* The right-hand side with solution (1.0, 2.0, 3.0, 4.0, 5.0) */
   double x[] = { 0.0, 1.0, 19.0, 2.0, 12.0 };

   /* Machine Topology */
   ncpu  = 1;
   ngpu  = 0;

   /* Initialize SyLVER */
   sylver_init(ncpu, ngpu);

   /* Perfom analyse */
   splu_analyse(n, ptr, row, val, &akeep, &options, &info);

   /* Factorize matrix */
   splu_factorize(val, akeep, &fkeep, &options, &info);

   /* Solve system */
   int job = 0; // Forward and backward substitution
   splu_solve(job, nrhs, x, n, akeep, fkeep, &info);

   printf("The computed solution is:\n");
   for(int i=0; i<n; i++) printf(" %18.10e", x[i]);
   printf("\n");

   splu_free(akeep, fkeep);

   /* Shutdown SyLVER */
   splu_finalize();

   return 0;
}
```

Listing 4: Simple example program in C using SpLU.

## 4 Numerical experiments

In this section, we present some results from runs of SpLLT and from routines in the S$y$LVER package. We use test matrices from the SuiteSparse set of sparse matrices [5]. In Table 1 we list the characteristics of the positive-definite matrices that we use for our experiments with SpLLT presented in Figures 1 and 4 and Tables 3 and 4. In Table 2 we list the characteristics of indefinite matrices that we use for our experiments with S$y$LVER presented in Figures 2 and 3.

| # | Problem | $n$ | $nz(A)$ | $nz(L)$ | $flops$ |
|---|---|---|---|---|---|
| | | $\times 10^3$ | $\times 10^6$ | $\times 10^6$ | $\times 10^9$ |
| 1 | Janna/Flan_1565 | 1565 | 59.5 | 1477.9 | 3859.8 |
| 2 | Oberwolfach/bone010 | 987 | 36.3 | 1076.4 | 3876.2 |
| 3 | Janna/StocF-1465 | 1465 | 11.2 | 1126.1 | 4386.6 |
| 4 | GHS_psdef/audikw_1 | 944 | 39.3 | 1242.3 | 5804.1 |
| 5 | Janna/Fault_639 | 639 | 14.6 | 1144.7 | 8283.9 |
| 6 | Janna/Hook_1498 | 1498 | 31.2 | 1532.9 | 8891.3 |
| 7 | Janna/Emilia_923 | 923 | 21.0 | 1729.9 | 13661.1 |
| 8 | Janna/Geo_1438 | 1438 | 32.3 | 2467.4 | 18058.1 |
| 9 | Janna/Serena | 1391 | 33.0 | 2761.7 | 30048.9 |

Table 1: Positive-definite test matrices and their characteristics without node amalgamation. $n$ is the matrix order, $nz(A)$ represents the number of entries in the matrix $A$, $nz(L)$ represents the number of entries in the factor $L$, and *flops* corresponds to the operation count for the matrix factorization.

| # | Problem | $n$ | $nz(A)$ | $nz(L)$ | $flops$ |
|---|---|---|---|---|---|
| | | $\times 10^3$ | $\times 10^6$ | $\times 10^6$ | $\times 10^9$ |
| 1 | Boeing/pct20stif | 52.33 | 1.38 | 12.60 | 5.63 |
| 2 | GHS_indef/copter2 | 55.48 | 0.41 | 12.70 | 6.10 |
| 3 | GHS_indef/helm2d03 | 392.26 | 1.57 | 33.00 | 6.16 |
| 4 | Boeing/crystk03 | 24.70 | 0.89 | 10.90 | 6.26 |
| 5 | Oberwolfach/filter3D | 106.44 | 1.41 | 23.80 | 8.71 |
| 6 | Koutsovasilis/F2 | 71.50 | 2.68 | 23.70 | 11.30 |
| 7 | McRae/ecology1 | 1000.00 | 3.00 | 72.30 | 18.20 |
| 8 | Cunningham/qa8fk | 66.13 | 0.86 | 26.70 | 22.10 |
| 9 | Oberwolfach/gas_sensor | 66.92 | 0.89 | 27.00 | 22.10 |
| 10 | Oberwolfach/t3dh | 79.17 | 2.22 | 50.60 | 70.10 |
| 11 | Lin/Lin | 256.00 | 1.01 | 126.00 | 285.00 |
| 12 | PARSEC/H2O | 67.02 | 2.22 | 234.00 | 1290.00 |
| 13 | GHS_indef/sparsine | 50.00 | 0.80 | 207.00 | 1390.00 |
| 14 | PARSEC/Ge99H100 | 112.98 | 4.28 | 669.00 | 7070.00 |
| 15 | PARSEC/Ga10As10H30 | 113.08 | 3.11 | 690.00 | 7280.00 |
| 16 | PARSEC/Ga19As19H42 | 133.12 | 4.51 | 823.00 | 9100.00 |

Table 2: Indefinite test matrices and their characteristics without node amalgamation. $n$ is the matrix order, $nz(A)$ represents the number of entries in the matrix $A$, $nz(L)$ represents the number of entries in the factor $L$, and *flops* corresponds to the operation count for the matrix factorization.

## 4.1 Some comparisons with other codes

At the end of the day, we expect our project to be judged in part by the performance of the NLAFET software compared to state-of-the-art codes from other libraries. We have published and submitted several papers[4, 7, 8, 10, 11] that compare our codes to others and are working on more, but we just give a flavour of some of our comparisons in this section.

In Figure 1 we show both factor and solve times for our SpLLT code, PARDISO, and PaStiX. The runs were performed using the 28 CPU cores (2× Intel Broadwell CPU equipped with 14 cores clocked at 2.6 GHz) of a compute node which is part of the Kebnekaise machine[5] hosted by the High Performance Computing Center North (HPC2N). The experiments show that our code is significantly faster than the others both for factorize and solve phases.

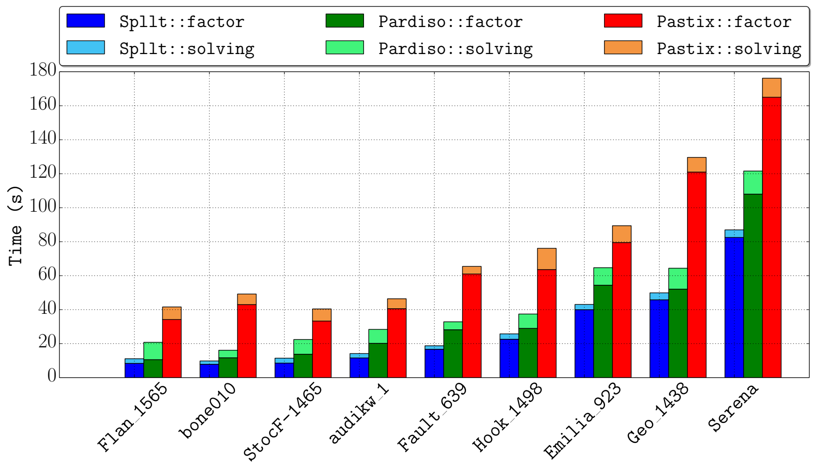


Figure 1: Timing of our SpLLT code compared to PARDISO and PaStiX on positive-definite systems.

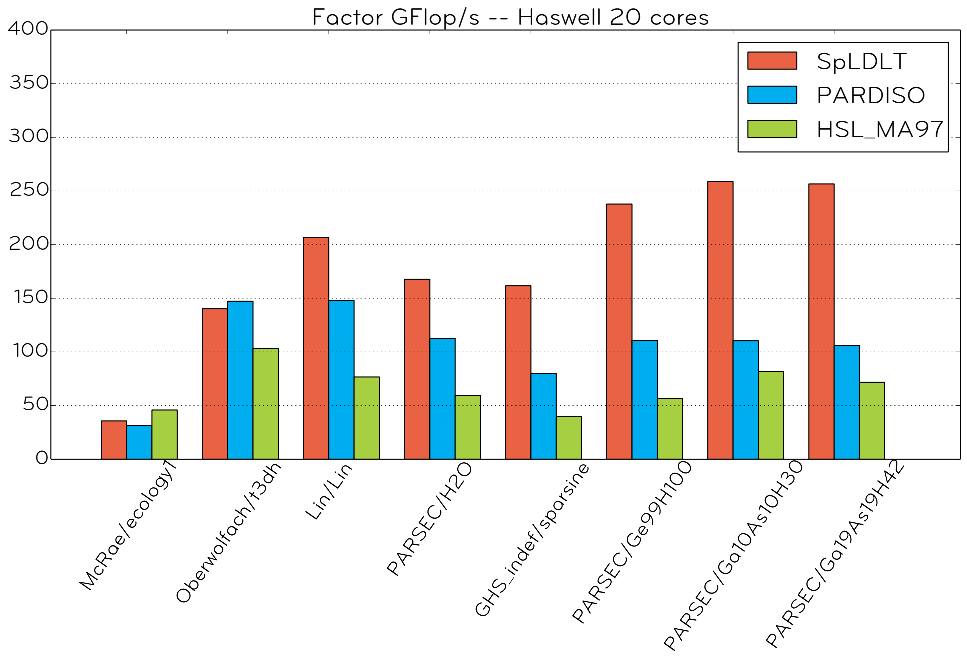


Figure 2: Gflop rates of SpLDLT compared with PARDISO and HSL_MA97 on indefinite matrices.

In Figure 2 we show Gflop rates for our SpLDLT code from S*y*LVER on symmetric indefinite systems compared with PARDISO and HSL_MA97. These rates are for the

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

factorize phase only and again they show that our code is very competitive with other state-of-the art codes. The runs for this figure were performed on the Alembert computer in the Innovative Computing Laboratory (ICL) at the University of Tennessee.

## 4.2 Results using GPUs

One reason why we use the StarPU runtime system is so we can more easily port our code to systems that have GPUs in addition to CPUs. We present some results in Figure 3 where see that we always benefit from adding a GPU sometimes significantly so. These experiments were also run on Alembert, a heterogeneous CPU-GPU machine with two NUMA nodes equipped with a 10 core Intel Haswell CPU clocked at 2.3 GHz, plus one NVIDIA Pascal GPU device P100.

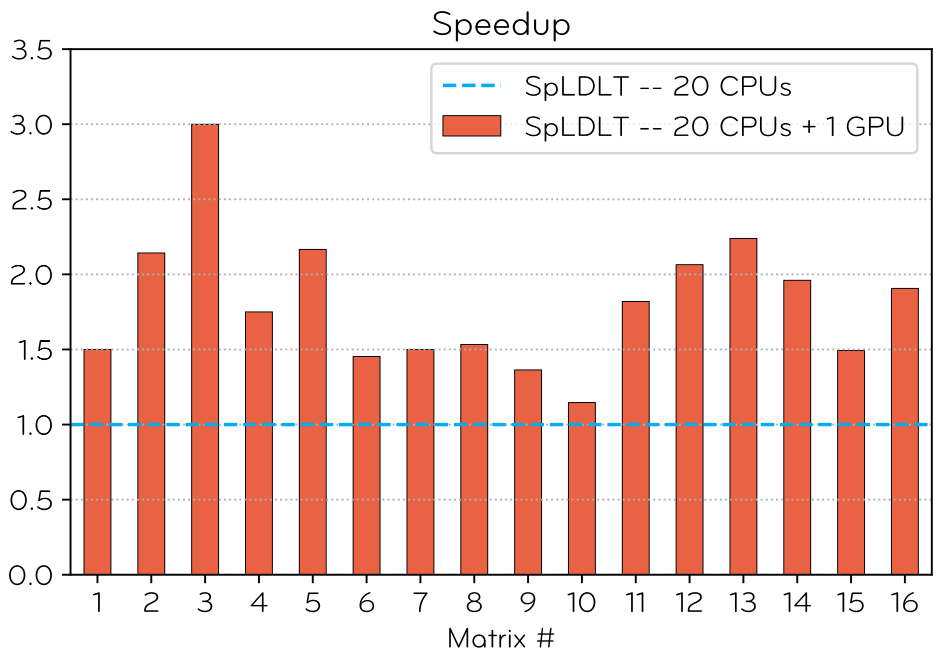


Figure 3: Results comparing runs of SpLDLT on CPUs only and on CPUs with one GPU on indefinite matrices.

## 4.3 Scalability of solve routines

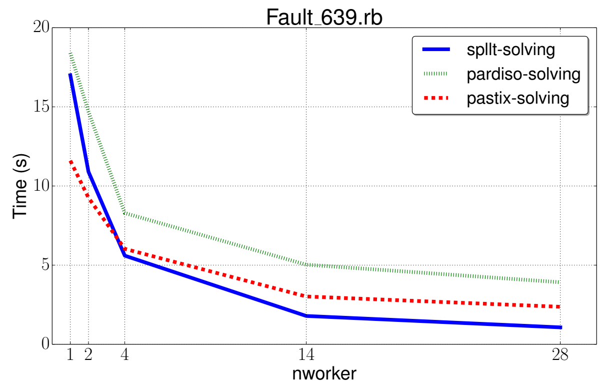
We saw in Figure 1 that the time for the *solve* routine was much less than for the *factorize* phase and indeed that is the case in general for sparse direct solvers. For this reason, the efficient implementation of the *solve* phase has been paid very little attention in the development of sparse solvers. However, as we show later, there are cases when a single call to the *factorize* phase is followed by many calls to the *solve* phase so that the time for this phase can strongly affect the overall computation time. We have therefore invested quite some effort on improving the *solve* routines so that they scale better both with respect to the number of right-hand sides and also with the number of cores.

We first compare the performance of our new *solve* code with PARDISO, PaStiX, and the HSL code HSL_MA87 on 28 cores when solving for blocks of 1, 2, 16, and 128 right-hand sides. The results in Table 3 from experiments on the Kebnekaise machine show clearly that our efforts in improving the solve phase were worthwhile.

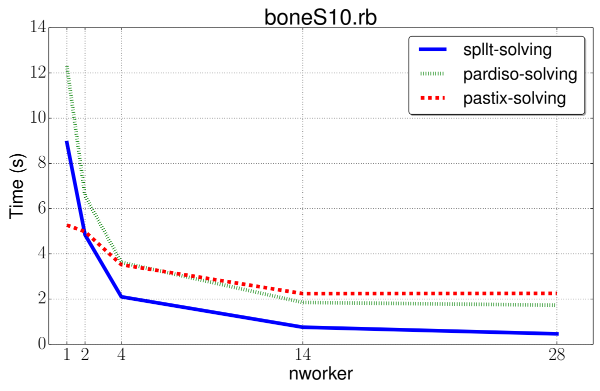
| Matrix | Fault_639 | boneS10 | Emilia_923 | Serena |
|---|---|---|---|---|
| | 1 rhs | | | |
| SpLLT | **33.10** | **10.10** | 44.70 | **75.80** |
| PARDISO | 226.00 | 61.30 | 358.00 | 485.00 |
| MA87 | 33.30 | 16.80 | 50.80 | 86.30 |
| PaStiX | 45.30 | 17.10 | **43.70** | 130.00 |
| | 2 rhs | | | |
| SpLLT | **38.10** | **12.40** | **47.70** | **80.70** |
| PARDISO | 253.00 | 105.00 | 345.00 | 630.00 |
| MA87 | 45.40 | 28.40 | 63.60 | 114.00 |
| PaStiX | 42.20 | 20.60 | 58.60 | 113.00 |
| | 16 rhs | | | |
| SpLLT | **53.20** | **18.40** | **67.70** | **117.00** |
| PARDISO | 290.00 | 115.00 | 391.00 | 654.00 |
| MA87 | 144.00 | 165.00 | 222.00 | 295.00 |
| PaStiX | 87.20 | 72.80 | 147.00 | 211.00 |
| | 128 rhs | | | |
| SpLLT | **190.00** | **86.10** | **291.00** | **443.00** |
| PARDISO | 520.00 | 197.00 | 717.00 | 1380.00 |
| MA87 | 1020.00 | 1310.00 | 1520.00 | 2130.00 |
| PaStiX | 428.00 | 415.00 | 878.00 | 1080.00 |

Table 3: Comparison of solve times for SpLLT, PARDISO, and PaStiX for 1, 2, 16 and 128 rhs. The experiments are run using 28 CPU cores and the times are shown in $10^{-2}$ seconds.

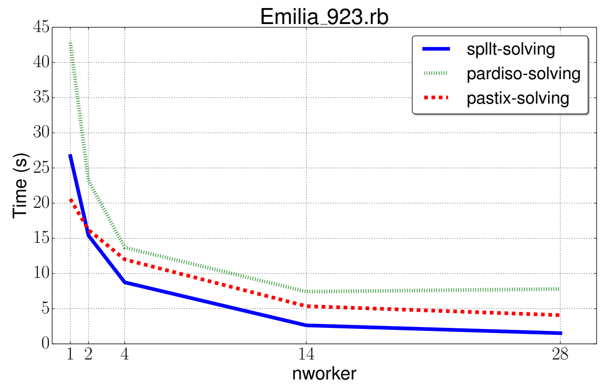
If we keep the number of right-hand sides at 64 but vary the number of cores, we obtain the results shown in Figure 4 where the scalabilty of our new code with respect to the number of cores and in comparison with the other codes is evident.



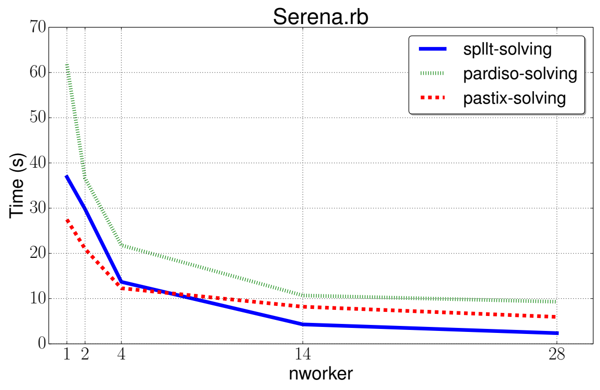
a Fault_639



b boneS10



c Emilia_923



d Serena

Figure 4: Comparison of the time to solve the system with 64 right-hand sides for SpLLT, PARDISO and PaStiX.

Finally we mentioned at the start of this section that one reason for spending so much time improving the solve phase was for cases when many solves would be performed for each factorize. Such a case has been provided by our partners in the NLAFET Project from Inria. When solving sparse equations using their Enlarged Conjugate Gradient (ECG) method, they need only factorize the matrix once but then must do one solve for each iteration of ECG. Effectively the direct solver is being used as a block Jacobi preconditioner. We show a comparison of using our code with using PARDISO as the direct solver in Table 4. We see that while the number of iterations is essentially unaltered, the time to solution is always better when using our direct solver and sometimes markedly so. In fact, when there are relatively few partitions the direct solver is working on larger systems and so the gains from using our solve routine are somewhat greater.

| Problem | | ECG/PARDISO | | ECG/SpLLT | |
|---|---|---|---|---|---|
| | # MPI | t (s) | # iter | t (s) | # iter |
| Flan_1565 | 16 | 57.82 | 141 | 23.18 | 141 |
| | 32 | 32.92 | 177 | 14.94 | 177 |
| | 64 | 20.15 | 216 | 9.77 | 216 |
| | 128 | 11.35 | 270 | 6.53 | 270 |
| | 256 | 6.70 | 325 | 5.50 | 325 |
| Hook_1498 | 16 | 32.10 | 87 | 12.68 | 87 |
| | 32 | 16.04 | 101 | 7.84 | 101 |
| | 64 | 9.85 | 128 | 5.53 | 128 |
| | 128 | 6.05 | 154 | 4.00 | 154 |
| | 256 | 3.46 | 183 | 2.55 | 183 |
| 3DSKY175P1 | 16 | 195.81 | 159 | 67.35 | 160 |
| | 32 | 104.75 | 179 | 42.48 | 174 |
| | 64 | 55.51 | 205 | 27.86 | 205 |
| | 128 | 32.71 | 248 | 17.78 | 248 |
| | 256 | 17.02 | 277 | 11.22 | 276 |

Table 4: Time to solution using PARDISO and SpLLT. ECG is set with a tolerance of $10^{-5}$, an enlarge factor of 12 (equal to the number of right-hand sides), and SpLLT has a block size of 256 with 14 workers.

## 5 Acknowledgements

This project is funded from the European Union's Horizon 2020 research and innovation program under the NLAFET grant agreement No 671633.

## References

[1] P. R. Amestoy, I. S. Duff, J. Koster, and J.-Y. L'Excellent. A fully asynchronous multifrontal solver using distributed dynamic scheduling. *SIAM J. Matrix Anal. Appl.*, 23(1):15–41, 2001.

[2] Patrick R. Amestoy and Iain S. Duff. Vectorization of a multiprocessor multifrontal code. *Int. J. of Supercomputer Applics.*, 3:41–59, 1989.

[3] Edward Anderson, Zhaojun Bai, Christian Bischof, Susan Blackford, Jack Dongarra, Jeremy Du Croz, Anne Greenbaum, Sven Hammarling, A. McKenney, and D. Sorensen. *LAPACK User's Guide*, volume 9. SIAM, 1999.

[4] Sébastien Cayrols, Iain S. Duff, and Florent Lopez. Parallelization of the solve phase in a task-based Cholesky solver using a sequential task flow model. Technical Report RAL-TR-2018-008, Rutherford Appleton Laboratory, Oxfordshire, England, 2018. NLAFET Working Note 20.

[5] T. A. Davis and Y. Hu. The University of Florida sparse matrix collection. *ACM Trans. Math. Softw.*, 38(1):1:1–1:25, December 2011.

[6] Iain S. Duff, Albert M. Erisman, and John K. Reid. *Direct Methods for Sparse Matrices. Second Edition.* Oxford University Press, Oxford, England, 2017.

[7] Iain S. Duff, Jonathan Hogg, and Florent Lopez. Experiments with sparse Cholesky using a sequential task-flow implementation. *Numerical Algebra, Control and Optimization*, 8:235–258, June 2018.

[8] Iain S. Duff, Jonathan Hogg, and Florent Lopez. A new sparse symmetric indefinite solver using a posteriori threshold pivoting. Technical Report RAL-TR-2018-012, Rutherford Appleton Laboratory, Oxfordshire, England, 2018. NLAFET Working Note 21.

[9] Iain S. Duff and Jacko Koster. On algorithms for permuting large entries to the diagonal of a sparse matrix. *SIAM J. Matrix Anal. Appl.*, 22(4):973–996, 2001.

[10] Iain S. Duff and Florent Lopez. Experiments with sparse Cholesky using a Parametrized Task Graph implementation. In Roman Wyrzykowski, Jack Dongarra, Ewa Deelman, and Konrad Karczewski, editors, *Parallel Processing and Applied Mathematics*, volume LNCS 10777, pages 197–206. Springer International Publishing, 2018.

[11] Iain S. Duff, Florent Lopez, and Stojce Nakov. Sparse direct solution on parallel computers. In M. Al-Baali, L. Grandinetti, and A. Purnama, editors, *Numerical Analysis and Optimization: NAOIV 2017*, volume Springer Proceedings in Mathematics & Statistics 235, pages 67–98. Springer, June 2018.

[12] J.D. Hogg, J.K. Reid, and J.A. Scott. Design of a multicore sparse Cholesky factorization using DAGs. *SIAM J. Scientific Computing*, 32(6):3627–3649, 2010.

[13] J.D. Hogg and J.A. Scott. HSL_MA97 : a bit-compatible multifrontal code for sparse symmetric systems. Technical Report RAL-TR-2011-024, Rutherford Appleton Laboratory, Oxfordshire, England, 2011.

[14] George Karypis and Vipin Kumar. MeTiS –unstructured graph partitioning and sparse matrix ordering system, version 2.0, 1995.

[15] Philip A. Knight, Daniel Ruiz, and Bora Uçar. A symmetry preserving algorithm for matrix scaling. *SIAM J. Matrix Anal. Appl.*, 35(3):931–955, 2014.

[16] Daniel Ruiz. A scaling algorithm to equilibrate both row and column norms in matrices. Technical Report RAL-TR-2001-034, Rutherford Appleton Laboratory, Oxfordshire, England, 2001.

[17] Endong Wang, Qing Zhang, Bo Shen, Guangyong Zhang, Xiaowei Lu, Qing Wu, and Yajuan Wang. Intel Math Kernel Library. In *High-Performance Computing on the Intel® Xeon Phi*, pages 167–188. Springer, 2014.