STRUMPACK Users' Guide

Pieter Ghysels*
Xiaoye S. Li*
Christopher Gorman†
Gustavo Chávez*
François-Henry Rouet‡

Version **2.0.0**, September 2017

[1]Lawrence Berkeley National Laboratory, Computational Research Division, MS 50F-1650, One Cyclotron Road, Berkeley CA94720. `{pghysels,xsli,gichavez}@lbl.gov`
[2]UC Santa Barbara. `gorman@math.ucsb.edu`
[3]Livermore Software Technology Corporation. `fhrouet@lstc.com`

Contents

1 STRUMPACK Overview

STRUMPACK – STRUctured Matrix PACKage – is a C++ library for computations with dense and sparse matrices. It uses so-called *structured matrices*, i.e., matrices that exhibit some kind of low-rank property, in this case with Hierarchically Semi-Separable matrices (HSS), to speed up linear algebra operations. This version of STRUMPACK unifies two main components that were separate in previous versions: a package for dense matrix computations (**STRUMPACK-dense**) and a package (**STRUMPACK-sparse**) for sparse linear systems. The algorithms for solving dense linear systems are described in [6] while the algorithms for solving sparse linear systems are described in [4, 3]. STRUMPACK can be used as a general algebraic sparse direct solver (based on the multifrontal factorization method), or as an efficient preconditioner for sparse matrices obtained by discretization of partial differential equations. Included in STRUMPACK are also the GMRES and BiCGStab iterative Krylov solvers, that use the approximate, HSS-accelerated, sparse solver as a preconditioner for the efficient solution of sparse linear systems.

The STRUMPACK project started at the Lawrence Berkeley National Laboratory in 2014 and is supported by the FASTMath SciDAC Institute funded by the Department of Energy and by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration.
Check the STRUMPACK website for more information and for the latest code:

http://portal.nersc.gov/project/sparse/strumpack/

2 Installation and Requirements

The STRUMPACK package uses the CMake build system (CMake version >= 2.8). The recommended way of building the STRUMPACK library is as follows:

```
> tar -xvzf strumpack-x.y.z.tar.gz
> cd strumpack-x.y.x
> mkdir build
> cd build
> cmake ../ -DCMAKE_BUILD_TYPE=Release \
    -DCMAKE_INSTALL_PREFIX=/path/to/install \
    -DCMAKE_CXX_COMPILER=<C++ compiler> \                          # this and below are optional,
    -DCMAKE_C_COMPILER=<C compiler> \                              # CMake will try to autodetect
    -DCMAKE_Fortran_COMPILER=<Fortran77 compiler> \
    -DSCALAPACK_LIBRARIES="/path/to/scalapack/libscalapack.a;/path/to/blacs/libblacs.a" \
    -DMETIS_INCLUDES=/path/to/metis/incluce \
    -DMETIS_LIBRARIES=/path/to/metis/libmetis.a \
    -DPARMETIS_INCLUDES=/path/to/parmetis/include \
    -DPARMETIS_LIBRARIES=/path/to/parmetis/libparmetis.a \
    -DSCOTCH_INCLUDES=/path/to/scotch/include \
    -DSCOTCH_LIBRARIES="/path/to/ptscotch/libscotch.a;...libscotcherr.a;...libptscotch.a;...libptscotcherr.a"
> make
> make test   # optional, takes a while
> make install
```

The above will only work if you have the following dependencies, and CMake can find them:

- **C++11**, **C** and **FORTRAN77** compilers. CMake looks for these compilers in the standard locations, if they are installed elsewhere, you can specify them as follows:

```
> cmake ../ -DCMAKE_BUILD_TYPE=Release     \
       -DCMAKE_CXX_COMPILER=g++            \
       -DCMAKE_C_COMPILER=gcc              \
       -DCMAKE_Fortran_COMPILER=gfortran
```

- **MPI** (Message Passing Interface) library. You should not need to manually specify the MPI compiler wrappers. CMake will look for MPI options and libraries and set the appropriate compiler and linker flags.
- **OpenMP v3.1** support is required in the C++ compiler to use the shared-memory parallelism in the code. OpenMP v3.1 introduces task parallelism, which is used extensively throughout the code. CMake will check whether your compiler supports OpenMP and sets the appropriate compiler and linker flags.
- **BLAS, LAPACK and ScaLAPACK** libraries. For performance it is crucial to use optimized BLAS/LAPACK libraries like for instance Intel® MKL, AMD® ACML, Cray® LibSci or OpenBLAS. The default versions of the Intel® MKL and Cray® LibSci BLAS libraries will use multithreaded kernels, unless when they are called from within an OpenMP parallel region, in which case they run sequentially. This is the behavior STRUMPACK relies upon to achieve good performance when running in MPI+OpenMP hybrid mode. ScaLAPACK depends on the BLACS communication library and on PBLAS (parallel BLAS), both of which are typically included with the ScaLAPACK installation (from ScaLAPACK 2.0.2, the blacs library is included in the ScaLAPACK library file). If CMake cannot locate these libraries, you can specify their path by setting the environment variable `$SCALAPACKDIR` or by specifying the libraries manually:

  ```
  > cmake ../ -DCMAKE_BUILD_TYPE=Release \
     -DSCALAPACK_LIBRARIES="/path/to/scalapack/libscalapack.a;/path/to/blacs/libblacs.a"
  ```

  Or one can also directly modify the linker flags to add the ScaLAPACK and BLACS libraries:

  ```
  > cmake ../ -DCMAKE_BUILD_TYPE=Release \
     -DCMAKE_EXE_LINKER_FLAGS="-L/usr/lib64/mpich/lib/ -lscalapack -lmpiblacs"
  ```

- **METIS** ($\geq$ 5.1.0) for the nested dissection matrix reordering. Metis can be obtained from:

  http://glaros.dtc.umn.edu/gkhome/metis/metis/download.

  CMake looks for the Metis inlude files the library in the default locations as well as in `$METISDIR/include` and `$METISDIR/lib`. Using the Bash shell, the METISDIR environment variable can be set as `export METISDIR=/usr/local/metis/`. Alternatively, you can specify the location of the header and library as follows:

  ```
  > cmake ../ -DCMAKE_BUILD_TYPE=Release \
    -DMETIS_INCLUDES=/usr/local/metis/include \
    -DMETIS_LIBRARIES=/usr/local/metis/lib/libmetis.a
  ```

- **PARMETIS** for parallel nested dissection. ParMetis can be download from

  http://glaros.dtc.umn.edu/gkhome/metis/parmetis/download

  The steps to make sure CMake can find ParMetis are similar as for Metis. The variables are `$PARMETISDIR` or `PARMETIS_INCLUDES` and `PARMETIS_LIBRARIES`.
- **SCOTCH** and **PT-SCOTCH** ($\geq$ 5.1.12) for matrix reordering. Scotch can be downloaded from:

  http://www.labri.fr/perso/pelegrin/scotch/

  Configuring CMake to find (PT-)Scotch is similar to Metis. For (PT-)Scotch the variables are `$SCOTCHDIR` or `SCOTCH_INCLUDES` and `SCOTCH_LIBRARIES`. Make sure to specify all libraries: `libscotch`, `libscotcherr`, `libptscotch` and `libptscotcherr`.
- **getopt_long:** This is a GNU extension to the POSIX getopt() C library function.
- **TCMalloc**, **TBB Malloc** or **jemalloc**: This is **optional**, but **recommended**, as it can lead to dramatic performance improvements for multithreaded code that performs frequent memory allocations. Link with the one of these libraries, e.g.:

```
  -DCMAKE_EXE_LINKER_FLAGS="-ltcmalloc"
```

to replace the default memory allocator (C++ `new`) with a more scalable implementation. See also Section 9.

The code was tested on GNU/Linux with the GNU and Intel® compilers and the OpenBLAS, Intel® MKL® and Cray® LibSci® numerical libraries. If you encounter issues on other platforms or with other BLAS/LAPACK implementations, please let us know. Successful compilation will create a `libstrumpack.a` file.

3 Algorithm

The algorithm used in STRUMPACK is described in detail in [4], and is based on the work by Jianlin Xia [8]. Here we summarize the main algorithm features. Section 5 has more information on the low-rank compression strategy and how to tune this to get a good preconditioner for your specific problem. There are three main steps in the algorithm: matrix reordering, factorization and solve.

Matrix reordering: There are three distinct matrix reordering steps: one for stability, one to limit fill-in and one to reduce HSS-ranks. First, the matrix is reordered and possibly scaled for numerical stability by the MC64 code [2]. For many matrices, this reordering can safely be disabled. By default, MC64 is used to maximize the product of the diagonal values of the matrix, and to scale the rows and columns of the matrix. Alternatively, MC64 can be used to maximize the smallest diagonal value or to maximize the sum of the diagonals. Next, a nested dissection reordering is applied to limit fill-in. Both (Par)Metis and (PT-)Scotch are supported. We expose one user tunable parameter which controls the size of the smallest separators. Finally, when HSS compression is used, there is an extra reordering step to reduce the HSS-ranks. This reordering uses Metis and does not require user tuning.

Factorization: Before the actual numerical factorization, there is a symbolic factorization step to construct the elimination tree. After that, the multifrontal factorization procedure traverses this elimination tree from bottom (smallest separators) to top (root separator). With each node of the elimination tree a dense matrix is associated, referred to as a frontal matrix, or simply front. These fronts can possibly be compressed as Hierarchically Semi-Separable (HSS) matrices. This compression will only pay off for fronts that are large enough, which are typically the frontal matrices at the nodes in the elimination tree close to the root. Without any HSS compression, the solver acts as a standard multifrontal direct solver. HSS approximations are constructed using a randomized sampling algorithm.

Solve: Once the matrix is factorized, the factors can be used to efficiently solve a linear system of equations by doing a forward and a backward solve sweeps. When no HSS compression is used, this is a direct solver. The multifrontal solve procedure is then used within an iterative refinement loop, with typically only 1 or very few iterations. However, when the factors are compressed using HSS, a single multifrontal solve is only approximate and the solve is by default used as a preconditioner for GMRES(30). The required number of GMRES iterations will depend strongly on the quality of the HSS approximation.

4 Using STRUMPACK Sparse

This section gives an overview on the basic usage of the sparse solvers in STRUMPACK. Many STRUMPACK options can be set from the command line. Running with `--help` or `-h`, will give you a list of supported run-time options.

An example `Makefile` is available in the `examples/` directory. This `Makefile` is generated by the **`cmake`** command, see Section 2.

The STRUMPACK package is written in C++, and offers a simple C++ interface. See Section 8 if you prefer a C interface. STRUMPACK-sparse has three different solver classes, all interaction happens through objects of these classes:

- **`StrumpackSparseSolver<scalar,integer=int>`**
  This class represents the sparse solver for a single computational node, optionally using OpenMP parallelism. Use this if you are running the code sequentially, on a (multicore) laptop or desktop or on a single node of a larger cluster. This class is defined in `StrumpackSparseSolver.hpp`, so include this header if you intend to use it.

- **`StrumpackSparseSolverMPI<scalar,integer=int>`**
  This solver has (mostly) the same interface as `StrumpackSparseSolver<scalar,integer>` but the numerical factorization and multifrontal solve phases run in parallel using MPI and ScaLAPACK. However, the inputs (sparse matrix, right-hand side vector) need to be available completely on every MPI process. The reordering phase uses Metis or Scotch (not ParMetis or PTScotch) and the symbolic factorization is threaded, but not distributed. The (multifrontal) solve is done in parallel, but the right-hand side vectors need to be available completely on every processor. Make sure to call `MPI_Init[_thread]` before instantiating an object of this class and include the header file `StrumpackSparseSolverMPI.hpp`. We do not recommend this solver, instead, use `StrumpackSparseSolverMPIDist` whenever possible.

- **`StrumpackSparseSolverMPIDist<scalar,integer=int>`**
  This solver is fully distributed. The numerical factorization and solve as well as the symbolic factorization are distributed. The input is now a block-row distributed sparse matrix and a correspondingly distributed right-hand side. For matrix reordering, ParMetis or PT-Scotch are used. Include the header file `StrumpackSparseSolverMPIDist.hpp` and call `MPI_Init[_thread]`. Unfortunately, there is no distributed version of the MC64 reordering code, so if this reordering (and scaling) step is enabled, the code will gather the distributed sparse matrix on a single node and then apply MC64 sequentially.

The three solver classes `StrumpackSparseSolver`, `StrumpackSparseSolverMPI` and `StrumpackSparseSolverMPIDist` depend on two template parameters `<scalar,integer>`: the type of a scalar and an integer type. The scalar type can be `float`, `double`, `std::complex<float>` or `std::complex<double>`. It is recommended to first try to simply use the default `integer=int` type, unless you run into 32 bit integer overflow problems. In that case one can switch to for instance `int64_t` (a signed integer type).

4.1 StrumpackSparseSolver Example

The following shows the typical way to use a (sequential or multithreaded) STRUMPACK sparse solver:

```
#include "StrumpackSparseSolver.hpp"
using namespace strumpack;  // all strumpack code is in the strumpack namespace,

int main(int argc, char* argv[]) {
  int N = ...;                 // construct an NxN CSR matrix with nnz nonzeros
  int* row_ptr = ...;          // N+1 integers
  int* col_ind = ...;          // nnz integers
  double* val = ...;           // nnz scalars
  double* x = new double[N];   // will hold the solution vector
  double* b = ...;             // set a right-hand side b

  StrumpackSparseSolver<double> sp;               // create solver object
  sp.options().set_rel_tol(1e-10);                // set options
  sp.options().set_gmres_restart(10);             // ...
  sp.options().enable_HSS();                      // enable HSS compression, see section 5
  sp.options().set_from_command_line(argc, argv); // parse command line options
  sp.set_csr_matrix(N, row_ptr, col_ind, val);    // set the matrix (copy)
  sp.reorder();                                   // reorder matrix
  sp.factor();                                    // numerical factorization
  sp.solve(b, x);                                 // solve Ax=b
  ... // check residual/error and cleanup
```

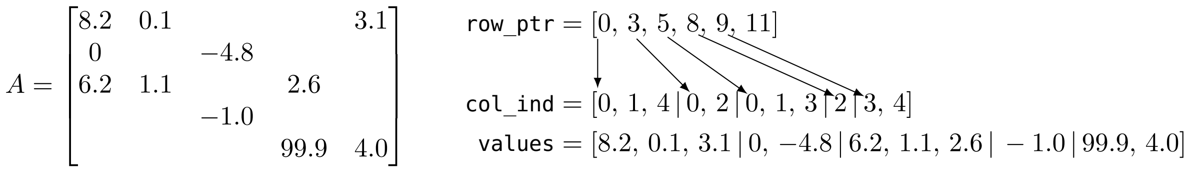
$A = \begin{bmatrix} 8.2 & 0.1 & & & 3.1 \\ 0 & & -4.8 & & \\ 6.2 & 1.1 & & 2.6 & \\ & & -1.0 & & \\ & & & 99.9 & 4.0 \end{bmatrix}$
row_ptr = [0, 3, 5, 8, 9, 11]
col_ind = [0, 1, 4 | 0, 2 | 0, 1, 3 | 2 | 3, 4]
values = [8.2, 0.1, 3.1 | 0, −4.8 | 6.2, 1.1, 2.6 | − 1.0 | 99.9, 4.0]

Figure 1: Illustration of a small 5×5 sparse matrix with 11 nonzeros and its Compressed Sparse Row (CSR) or Yale format representation. We always use 0-based indexing! Let $N = 5$ denote the number of rows. The `row_ptr` array has $N+1$ elements, with element i denoting the start of row i in the `col_ind` and `values` arrays. Element `row_ptr[N] = nnz`, i.e., the total number of nonzero elements in the matrix. The `values` array holds the actual matrix values, ordered by row. The corresponding elements in `col_ind` give the column indices for each nonzero. There can be explicit zero elements in the matrix. The nonzero values and corresponding column indices need not be sorted by column (within a row).

```
}
```

The main steps are: create solver object, set options (parse options from the command line), set matrix, reorder, factor and finally solve. The matrix should be in the Compressed Sparse Row (CSR) format, also called Yale format, with 0 based indices. Figure 1 illustrates the CSR format. In the basic scenario, it is not necessary to explicitly call `reorder` and `factor`, since trying to solve with a `StrumpackSparseSolver` object that is not factored yet, will internally call the `factor` routine, which will call `reorder` if necessary.

The above code should be linked with `-lstrumpack` and with the Metis, ParMetis, Scotch, PT-Scotch, BLAS, LAPACK, and ScaLAPACK libraries.

4.2 StrumpackSparseSolverMPI Example

Usage of the `StrumpackSparseSolverMPI<scalar,integer=int>` solver is very similar:

```
#include "StrumpackSparseSolverMPI.hpp"
using namespace strumpack;

int main(int argc, char* argv[]) {
  int thread_level, rank;
  // StrumpackSparseSolverMPI uses OpenMP so we should ask for MPI_THREAD_FUNNELED at least
  MPI_Init_thread(&argc, &argv, MPI_THREAD_FUNNELED, &thread_level);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  if (thread_level != MPI_THREAD_FUNNELED && rank == 0)
    std::cout << "MPI␣implementation␣does␣not␣support␣MPI_THREAD_FUNNELED" << std::endl;

  {
    // define the same CSR matrix as for StrumpackSparseSolver
    int N = ...;                // construct an NxN CSR matrix with nnz nonzeros
    int* row_ptr = ...;         // N+1 integers
    int* col_ind = ...;         // nnz integers
    double* val = ...;          // nnz scalars
    // allocate entire solution and right-hand side vectors on each MPI process
    double* x = new double[N];  // will hold the solution vector
    double* b = ...;            // set a right-hand side b

    // construct solver and specify the MPI communicator
    StrumpackSparseSolverMPI<double> sp(MPI_COMM_WORLD);
    sp.options().set_mc64job(0);
    sp.options().set_from_command_line(argc, argv);
    sp.set_csr_matrix(N, row_ptr, col_ind, val);
    sp.solve(b, x);
    ... // check residual/error, cleanup
  }
  Cblacs_exit(1);
  MPI_Finalize();
}
```

The only difference here is the use of `StrumpackSparseSolverMPI` instead of `StrumpackSparseSolver` and the calls to `MPI_Init_thread`, `Cblacs_exit` and `MPI_Finalize`.

4.3 StrumpackSparseSolverMPIDist Example

Finally, we illustrate the usage of `StrumpackSparseSolverMPIDist<scalar,integer=int>` solver. This interface takes a block-row distributed compressed sparse row matrix as input. This matrix format is illustrated in Figure 2.

```
#include "StrumpackSparseSolverMPI.hpp"
using namespace strumpack;

int main(int argc, char* argv[]) {
  int thread_level, rank, P;
  MPI_Init_thread(&argc, &argv, MPI_THREAD_FUNNELED, &thread_level);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &P);
  {
    // define a block-row distributed CSR matrix
    int* dist     = new int[P];
    // set dist such that processor p owns rows [dist[p], dist[p+1]) of the sparse matrix
    for (int p=0; p<P; p++) dist[p] = .. ;
    // local_n is the number of rows of the input matrix assigned to me
    int local_n   = dist[rank+1] - dist[rank];
    int* row_ptr  = new int[local_n+1];
    .. // set the sparse matrix row pointers in row_ptr
    int local_nnz = row_ptr[local_n+1] - row_ptr[0];
    int* col_ind  = new int[local_nnz];
    .. // set the sparse matrix column indices in col_ind
    double* val   = new double[local_nnz];
    .. // set the matrix nonzero value in val
    double* x     = new double[local_n];        // local part of solution
    double* b     = new double[local_n];        // local part of rhs
    for (int i=0; i<local_n; i++) b[i] = ..;    // set the rhs

    StrumpackSparseSolverMPIDist<double> sp(MPI_COMM_WORLD);
    sp.options().set_reordering_method(ReorderingStrategy::PARMETIS);
    sp.options().set_from_command_line(argc, argv);
    sp.set_distributed_csr_matrix(local_n, row_ptr, col_ind, val, dist);
    sp.solve(b, x);
    ... // check residual/error, cleanup
  }
  Cblacs_exit(1);
  MPI_Finalize();
}
```

$$A = \left[\begin{array}{ccccc}
8.2 & 0.1 & & & 3.1 \\ \hline
0 & & -4.8 & & \\
6.2 & 1.1 & & 2.6 & \\ \hline
 & & -1.0 & & \\
 & & & 99.9 & 4.0
\end{array}\right]
\begin{array}{l}
P_0 \\
\\
P_1 \\
\\
P_2
\end{array}$$

$$\texttt{dist} = [0,\ 1,\ 3,\ 5]$$

$$\begin{array}{lll}
P_0 & P_1 & P_2 \\
\texttt{row\_ptr} = [0,\ 3] & \texttt{row\_ptr} = [0,\ 2,\ 5] & \texttt{row\_ptr} = [0,\ 1,\ 3] \\
\texttt{col\_ind} = [0,\ 1,\ 4] & \texttt{col\_ind} = [0,\ 2 \,|\, 0,\ 1,\ 3] & \texttt{col\_ind} = [2 \,|\, 3,\ 4] \\
\texttt{values} = [8.2,\ 0.1,\ 3.1] & \texttt{values} = [0,\ -4.8 \,|\, 6.2,\ 1.1,\ 2.6] & \texttt{values} = [-1.0 \,|\, 99.9,\ 4.0]
\end{array}$$

Figure 2: Illustration of a small 5×5 sparse matrix with 11 nonzeros and its block-row distributed compressed sparse row representation. We always use 0-based indexing! Process P_0 owns row 0, process P_1 has rows 1 and 2 and process P_2 has rows 3 and 4. This distribution of rows over the processes is represented by the `dist` array. Process p owns rows `[dist[p],dist[p+1])`. If $N = 5$ is the number of rows in the entire matrix and P is the total number of processes, then `dist[P]=N`. The (same) `dist` array is stored on every process. Each process holds a CSR representation of only its local rows of the matrix, see Figure 1.

4.4 Initializing the Solver Object

Let

```
typedef strumpack::StrumpackSparseSolver<scalar,integer> Sp;
typedef strumpack::StrumpackSparseSolverMPI<scalar,integer> SpMPI;
typedef strumpack::StrumpackSparseSolverMPIDist<scalar,integer> SpMPIDist;
```

Each of the solver classes has two constructors:

```
Sp::StrumpackSparseSolver(bool verbose=true, bool root=true);
Sp::StrumpackSparseSolver(int argc, char* argv[], bool verbose=true, bool root=true);
```

```
SpMPI::StrumpackSparseSolverMPIDist(MPI_Comm comm, bool verbose=true);
SpMPI::StrumpackSparseSolverMPIDist(MPI_Comm comm, int argc, char* argv[], bool verbose=true);
```

```
SpMPIDist::StrumpackSparseSolverMPIDist(MPI_Comm comm, bool verbose=true);
SpMPIDist::StrumpackSparseSolverMPIDist(MPI_Comm comm, int argc, char* argv[], bool verbose=true);
```

where `argc` and `argv` contain the command line options and the `verbose` option can be set to `false` to suppress output of the solver. Note that since `SpMPIDist` is a subclass of `SpMPI`, which is a subclass of `Sp`, all public members of Sp are also members of `SpMPI` and `SpMPIDist`. The public interface to the `SpMPI` class is exactly the same as that for the `Sp` class.

4.5 Sparse Matrix Format

The sparse matrix should be specified in compressed sparse row format [7]:

```
void Sp::set_csr_matrix(int N, int* row_ptr, int* col_ind, scalar* values,
                        bool symmetric_pattern=false);
```

Internally, the matrix is copied, so it will not be modified. Previous versions of STRUMPACK also supported the CSC format, but this is now deprecated. If the sparsity pattern of the matrix is symmetric (the values do not have to be symmetric), then you can set `symmetric_pattern=true`. This saves some work in the setup phase of the solver.

For the `SpMPIDist` solver the input is a block-row distributed compressed sparse row matrix (as illustrated in the example above):

```
void SpMPIDist::set_distributed_csr_matrix
       (integer local_rows, integer* row_ptr, integer* col_ind,
        scalar* values, integer* dist, bool symmetric_pattern=false);
```

Alternatively, you can also specify a sequential CSR matrix to the `SpMPIDist` solver:

```
void SpMPIDist::set_csr_matrix
       (integer N, integer* row_ptr, integer* col_ind,
        scalar* values, bool symmetric_pattern=false);
```

For this routine, the matrix only needs to be specified completely on the root process. Other processes can pass `NULL` for the arrays.

4.6 Setting and Parsing Options

The solver class has an object of type `SPOptions<scalar>`, which can be accessed through:

```
SPOptions<scalar>& Sp::options();
```

The `SPOptions<scalar>` class is defined in `SPOptions.hpp`. The complete public interface for the `SPOptions<scalar>` class is given in Section 4.10.1. The following subsections describe some of the options available from `SPOptions<scalar>` in more detail.

4.7 Reordering

There are three types of matrix reordering: for numerical stability, to reduce fill-in and to reduce the HSS-ranks. These reorderings are all performed when calling

```
ReturnCode Sp::reorder();
```

The return value is of type `ReturnCode` (defined in `strumpack_parameters.hpp`) and can be

```
enum class ReturnCode {
  SUCCESS,          /*!< Operation completed successfully. */
  MATRIX_NOT_SET,   /*!< The input matrix was not set.     */
  REORDERING_ERROR  /*!< The matrix reordering failed.     */
};
```

4.7.1 Reordering for numerical stability

The reordering for numerical stability is performed using the MC64 code. For many matrices, this reordering is not necessary and can safely be disabled! MC64 supports 5 different modes

0: no reordering for stability, this disables MC64

1: currently not supported

2: maximize the smallest diagonal value

3: maximize the smallest diagonal value, different strategy

4: maximize sum of diagonal values

5: maximize product of diagonal values and apply row and column scaling

which can be selected via

```
void SPOptions::set_mc64job(int job);
void SPOptions::set_mc64job(MC64Job job);
int SPOptions::mc64job() const;
```

where `mc64()` queries the currently selected strategy (the default is **5:** maximum product of diagonal values plus row and column scaling). Instead of entering a number, you can specify one of the enumerations in `MC64Job`:

```
enum class MC64Job {
  NONE,                         /*!< Don't do anything                                    */
  MAX_CARDINALITY,              /*!< Maximum cardinality                                  */
  MAX_SMALLEST_DIAGONAL,        /*!< Maximum smallest diagonal value                      */
  MAX_SMALLEST_DIAGONAL_2,      /*!< Same as MAX_SMALLEST_DIAGONAL, different algorithm   */
  MAX_DIAGONAL_SUM,             /*!< Maximum sum of diagonal values                       */
  MAX_DIAGONAL_PRODUCT_SCALING  /*!< Maximum product of diagonal values and row and column scaling  */
};
```

The command line option

```
--sp_mc64job [0-5]
```

can also be used.

4.7.2 Nested dissection reordering

The STRUMPACK sparse solver supports both (Par)Metis and (PT-)Scotch for the matrix reordering. The following functions can set the preferred method or check the currently selected method:

```
void SPOptions::set_reordering_method(ReorderingStrategy m);
ReorderingStrategy SPOptions::reordering_method() const;
```

The options for `MatrixReorderingStrategy` are

```
enum class ReorderingStrategy {
  NATURAL,    /*!< Do not reorder the system                    */
  METIS,      /*!< Use Metis nested-dissection reordering       */
  PARMETIS,   /*!< Use ParMetis nested-dissection reordering    */
  SCOTCH,     /*!< Use Scotch nested-dissection reordering      */
  PTSCOTCH,   /*!< Use PT-Scotch nested-dissection reordering   */
  RCM,        /*!< Use RCM reordering                           */
  GEOMETRIC   /*!< A simple geometric nested dissection code that
                only works for regular meshes. (see Sp::reorder)  */
};
```

When the solver is an object of `Sp`, `PARMETIS` or `PTSCOTCH` are not supported. When the solver is parallel, either an `SpMPI` or `SpMPIDist` object, and `METIS`, `SCOTCH` or `RCM` are chosen, then the graph of the complete matrix will be gathered onto the root process and the root process will call the (sequential) Metis, Scotch or RCM reordering routine. For large graphs this might fail due to insufficient memory.

The `GEOMETRIC` option is only allowed for regular grids. In this case, the dimensions of the grid should be specified in the function

```
ReturnCode Sp::reorder(int nx=1, int ny=1, int nz=1);
```

For instance for a regular 2d 2000×4000 grid, you can call this as `sp.reorder(2000, 4000)`. In the general algebraic case, the grid dimensions don't have to be provided. The reordering method can also be specified via the command line option

```
--sp_reordering_method [metis|parmetis|scotch|ptscotch|geometric|rcm]
```

4.8 Factorization

Compute the factorization by calling

```
ReturnCode Sp::factor();
```

where the possible return values are the same as for `Sp::reorder()`. If `Sp::reorder()` was not called already, it is called automatically. When HSS compression is not enabled, this will compute an exact LU factorization of the (permuted) sparse input matrix. If HSS compression is enabled (with `SPOptions::enable_HSS()` or `--sp_enable_HSS`, see Section 5), the factorization is only approximate.

4.9 Solve

Solve the linear system $Ax = b$ by calling

```
ReturnCode Sp::solve(scalar* b, scalar* x, bool use_initial_guess=false);
```

By default (`bool use_initial_guess=false`) the input in `x` is ignored. If `bool use_initial_guess=true`, `x` is used as initial guess for the iterative solver (if an iterative solver is used, for instance iterative refinement or GMRES). If the `Sp::factor()` was not called, it is called automatically. The return values are the same as for `Sp::reorder()`.

The iterative solver can be chosen through:

```
void SPOptions::set_Krylov_solver(KrylovSolver s);
```

where `KrylovSolver` can take the following values:

```
enum class KrylovSolver {
  AUTO,          /*!< Use iterative refinement if no HSS compression is used, otherwise PGMRES.       */
  DIRECT,        /*!< No outer iterative solver, just a single application of the multifrontal solver. */
  REFINE,        /*!< Iterative refinement.                                                            */
  PREC_GMRES,    /*!< Preconditioned GMRES. The preconditioner is the (approx) multifrontal solver.    */
  GMRES,         /*!< UN-preconditioned GMRES. (for testing mainly)                                    */
  PREC_BICGSTAB, /*!< Preconditioned BiCGStab. The preconditioner is the (approx) multifrontal solver. */
  BICGSTAB       /*!< UN-preconditioned BiCGStab. (for testing mainly)                                 */
};
```

with `KrylovSolver::AUTO` being the default value. The `KrylovSolver::AUTO` setting will use iterative refinement when HSS compression is not enabled, and preconditioned GMRES when HSS compression is enabled, see Section 5. To use the solver as a preconditioner, or a single (approximate) solve, set the solver to `KrylovSolver::DIRECT`. When calling `SpMPIDist::solve`, the right-hand side and solution vectors should only point to the local parts!

4.10 All Options for the Sparse Solver

The HSS specific options are stored in an object of type `HSSOptions<scalar>`, inside the `SPOptions` object. These options are described in Section 5.

4.10.1 `SPOptions<scalar>` Interface

The complete public interface to the options class is as follows, wher the `real` type is the real part of a scalar, i.e., `decltype(std::real(scalar(0)))`.

```
template<typename scalar> class SPOptions {
public:
  SPOptions();
  SPOptions(int argc, char* argv[]);

  /* print statistics?                                              */
  void set_verbose(bool verbose);                        bool verbose() const;
  /* maximum iterations in iterative solver                         */
  void set_maxit(int maxit);                             int maxit() const;
  /* relative residual stopping criterion for iterative solver      */
  void set_rel_tol(real rel_tol);                        real rel_tol() const;
  /* absolute residual stopping criterion for iterative solver      */
  void set_abs_tol(real abs_tol);                        real abs_tol() const;
  /* type of iterative solver to use, see section 4.9               */
  void set_Krylov_solver(KrylovSolver s);                KrylovSolver Krylov_solver() const;
  /* GMRES restart                                                  */
  void set_gmres_restart(int m);                         int gmres_restart() const;
  /* type of Gram-Schmidt used in GMRES                             */
  void set_GramSchmidt_type(GramSchmidtType t);          GramSchmidtType GramSchmidt_type() const;
  /* nested-dissection code, see section 4.7.2                      */
  void set_reordering_method(ReorderingStrategy m);      ReorderingStrategy reordering_method() const;
  /* stop nested-dissection when domains are smaller than nd_param */
  void set_nd_param(int nd_param);                       int nd_param() const;
  /* use the internal (undocumented) metis routine METIS_NodeNDP    */
  void enable_METIS_NodeNDP();                           bool use_METIS_NodeNDP() const;
  /* do not use METIS_NodeNDP, use METIS_NodeND instead             */
  void disable_METIS_NodeNDP();
  /* use METIS_NodeND instead of METIS_NodeDNP                      */
  void enable_METIS_NodeND();                            bool use_METIS_NodeND() const;
  /* do not use METIS_NodeND, use METIS_NodeNDP instead             */
  void disable_METIS_NodeND();
  /* build the supernodal tree using the MUMPS_SYMQAMD code         */
  void enable_MUMPS_SYMQAMD();                           bool use_MUMPS_SYMQAMD() const;
  /* do not use MUMPS_SYMQAMD, use fundamental supernodes           */
  void disable_MUMPS_SYMQAMD();
  /* when using MUMPS_SYMQAMS, enable aggressive amalgamation       */
  void enable_agg_amalg();                               bool use_agg_amalg() const;
  /* when using MUMPS_SYMQAMS, disable aggressive amalgamation       */
  void disable_agg_amalg();
  /* set the job to be used for static pivoting, see section 4.7.1  */
  void set_mc64job(int job);                             int mc64job() const;
  void set_mc64job(MC64Job job);
  /* not used at the moment                                          */
  void enable_assembly_tree_log();                       bool log_assembly_tree() const;
  void disable_assembly_tree_log();

  /* enable HSS compression, see section 5                           */
  void enable_HSS();                                     bool use_HSS() const;
  /* disable HSS compression                                         */
  void disable_HSS();
  /* set the minimum size of a front for HSS compression             */
  void set_HSS_min_front_size(int s);                    int HSS_min_front_size() const;
  /* set the minimum size of a separator for HSS compression         */
  void set_HSS_min_sep_size(int s);                      int HSS_min_sep_size() const;
```

```
    /* set level to 1 to enable length 2 connections in the separator
     * before computing separator reordering to reduce HSS ranks.
     * Set to to disable length 2 connections.                     */
    void set_separator_ordering_level(int l);           int separator_ordering_level() const;

    /* best not to touch this                                      */
    void enable_indirect_sampling();
    void disable_indirect_sampling();                    bool indirect_sampling() const;

    void enable_replace_tiny_pivots();                   bool replace_tiny_pivots() const;
    void disable_replace_tiny_pivots(;

    /* get the HSS specific options, see section 5.                */
    const HSS::HSSOptions<scalar>& HSS_options() const;
    HSS::HSSOptions<scalar>& HSS_options();

    /* parse the options in argc/argv set in the constructor       */
    void set_from_command_line();
    /* parse the options in argc/argv                              */
    void set_from_command_line(int argc, char* argv[]);
    /* print out message listing all command line options          */
    void describe_options() const;
  };
```

This uses the following (scoped) enumeration for the Gram-Schmidt type used in GMRES:

```
  enum class GramSchmidtType {
    CLASSICAL,   /*!< Classical Gram-Schmidt is faster, more scalable.   */
    MODIFIED     /*!< Modified Gram-Schmidt is slower, but stable.       */
  };
```

4.10.2 Command Line Options

To get a list of all available options, make sure to pass "`int argc, char* argv[]`" when initializing the `StrumpackSparseSolver` or when calling `SPOptions::set_from_command_line` and run the application with `--help` or `-h`. Some default values listed here are for double precision and might be different when running in single precision.

```
 STRUMPACK options:
   --sp_maxit int (default 5000)
   --sp_rel_tol real (default 1e-06)
   --sp_abs_tol real (default 1e-10)
   --sp_Krylov_solver auto|direct|refinement|pgmres|gmres|pbicgstab|bicgstab
   --sp_gmres_restart int (default 30)
   --sp_GramSchmidt_type modified|classical
   --sp_reordering_method natural|metis|scotch|parmetis|ptscotch|rcm|geometric
   --sp_nd_param int (default 8)
   --sp_enable_METIS_NodeNDP (default true)
   --sp_disable_METIS_NodeNDP (default false)
   --sp_enable_METIS_NodeND (default false)
   --sp_disable_METIS_NodeND (default true)
   --sp_enable_MUMPS_SYMQAMD (default false)
   --sp_disable_MUMPS_SYMQAMD (default true)
   --sp_enable_agg_amalg (default false)
   --sp_disable_agg_amalg (default true)
   --sp_mc64job 0-5 (default 0)
```

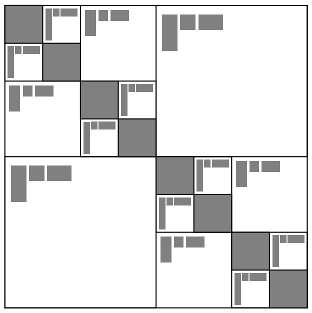

Figure 3: Illustration of a Hierarchically Semi-Separable (HSS) matrix. Gray blocks are dense matrices. Off-diagonal blocks, on different levels of the HSS hierarchy, are low-rank. The low-rank factors of off-diagonal blocks of different levels are related.

```
--sp_enable_hss (default false)
--sp_disable_hss (default true)
--sp_hss_min_front_size int (default 1000)
--sp_hss_min_sep_size int (default 256)
--sp_separator_ordering_level (default 1)
--sp_enable_indirect_sampling
--sp_disable_indirect_sampling
--sp_enable_replace_tiny_pivots
--sp_disable_replace_tiny_pivots
--sp_verbose or -v (default true)
--sp_quiet or -q (default false)
--help or -h
```

5 HSS Preconditioning

The sparse multifrontal solver can optionally use Hierarchically Semi-Separable, rank-structured matrices to compress the fill-in. In the multifrontal method, computations are performed on dense matrices called frontal matrices. A frontal matrix can be approximated as an HSS matrix, but this will only be beneficial (compared to storing the frontal as a standard dense matrix and operating on it with BLAS/LAPACK routines) if the frontal matrix is large enough.

Figure 3 illustrates the HSS matrix format. The matrix is partitioned as a 2×2 block matrix, with the partitioning recursively applied on the diagonal blocks, until diagonal blocks are smaller than a specified *leaf size*. The off-diagonal block on each level of the hierarchy are approximated by a low-rank product. This low-rank storage format asymptotically reduces memory usage and floating point operations, while introducing approximation errors. HSS compression is not used by default in the STRUMPACK sparse solver (the default is to perform exact LU factorization), but can be turned on/off via the command line:

```
--sp_enable_hss   (no argument)
--sp_disable_hss   (no argument)
```

or via the C++ API as follows

```
void SPOptions<scalar>::enable_HSS();
void SPOptions<scalar>::disable_HSS();
bool SPOptions<scalar>::use_HSS();    // check whether HSS compression is enabled
```

When HSS compression is enabled, the default STRUMPACK behavior is to use the HSS enabled approximate LU factorization as a preconditioner within GMRES. This behavior can also be changed, see Section 4.9.

However, HSS compression has a considerable overhead and only pays off for sufficiently large matrices. Therefore STRUMPACK has a tuning parameter to specify the minimum size a dense matrix needs to be to be considered a candidate for HSS compression. The minimum dense matrix size for HSS compression is set via the command line via

```
--sp_hss_min_front_size int (default 1000)
--sp_hss_min_sep_size int (default 256)
```

or via the C++ API as follows

```
void SPOptions<scalar>::set_HSS_min_front_size(int s);
void SPOptions<scalar>::set_HSS_min_sep_size(int s);
int SPOptions<scalar>::HSS_min_front_size() const;  // get the current value
int SPOptions<scalar>::HSS_min_sep_size() const;
```

The routine `set_HSS_min_front_size(int s)` sets the minimum size of the entire front, while `set_HSS_min_sep_size(int s)` refers to the size of the top-left block of the front only. This top-left block is the part that corresponds to a separator, as given by the nested dissection reordering algorithm. This top-left block is also referred to as the block containing the fully-summed variable. Factorization (LU in the dense case, ULV in the HSS case) is only applied to this top-left block. ***Tuning the values for the minimum front and separator sizes can have a big impact on performance and memory usage!***

The above options affect the use of HSS within the multifrontal solver. There are more, HSS specific, options which are stored in an object of type `HSS::HSSOptions<scalar>`. An object of this type is stored in the `SPOptions<scalar>` object stored in the `StrumpackSparseSolver`. It can be accessed via the `HSS_options()` routine as follows:

```
StrumpackSparseSolver<double> sp;               // create solver object
sp.options().enable_HSS();                      // enable HSS compression in the multifrontal solver
sp.options().HSS_options().set_leaf_size(256);  // set the HSS leaf size
```

In STRUMPACK, HSS matrices are constructed using a randomized sampling algorithm [5]. To construct an HSS approximation for a matrix A, sampling of the rows and columns of A is computed by multiplication with a tall and skinny random matrix R as follows: $S^r = AR$ and $S^c = A^*R$. Ideally, the number of columns in the matrix R is $d = r + p$, with r the maximum off-diagonal block rank in the HSS matrix and p a small oversampling parameter. Unfortunately, the HSS rank is not known a-priori, so it needs to determined adaptively. The adaptive sampling scheme used in STRUMPACK starts with an initial number of random vector d_0, and increases this in steps of Δd, until the compression quality reaches the desired user specified tolerance, or until the maximum rank is reached. ***The compression tolerances can greatly impact performance.*** They can be set using:

```
--hss_rel_tol real (default 0.01)
--hss_abs_tol real (default 1e-08)
```

or via the C++ API

```
void HSSOptions<scalar>::set_rel_tol(real rel_tol);
void HSSOptions<scalar>::set_abs_tol(real abs_tol);
real HSSOptions<scalar>::rel_tol() const;    // get the current value
real HSSOptions<scalar>::abs_tol() const;
```

5.1 `HSSOptions<scalar>` Interface

Other options are available to tune for instance the initial number of random vectors d_0, the increment Δd, the random number generator or the random number distribution. The complete public interface for the `HSSOptions<scalar>` class is:

```
template<typename scalar> class HSSOptions {
public:
  /* relative compression tolerance                       */
  void set_rel_tol(real rel_tol);           real rel_tol() const;
  /* absolute compression tolerance                       */
  void set_abs_tol(real abs_tol);           real abs_tol() const;
  /* size of the smallest blocks in the HSS hierarchy     */
  void set_leaf_size(int leaf_size);        int leaf_size() const;
  /* initial number of random vectors used in the
     adaptive randomized compression algorithm            */
  void set_d0(int d0);                      int d0() const;
  /* number of random vectors added in each step of the
     adaptive randomized HSS compression algorithm        */
  void set_dd(int dd);                      int dd() const;
  /* currently not used                                   */
  void set_q(int q);                        int q() const;
  /* maximum rank in the HSS representation               */
  void set_max_rank(int max_rank);          int max_rank() const;
  /* random engine/generator to use, see below            */
  void set_random_engine(random::RandomEngine random_engine);
  random::RandomEngine random_engine() const;
  /* the random number distribution, see below            */
  void set_random_distribution
  (random::RandomDistribution random_distribution);
  random::RandomDistribution random_distribution() const;
  /* the compression algorithm to use                     */
  void set_compression_algorithm(CompressionAlgorithm a);
  CompressionAlgorithm compression_algorithm() const;
  /* for expert users                                     */
  void set_user_defined_random(bool user_defined_random);
  bool user_defined_random() const;
  /* for expert users                                     */
  void set_synchronized_compression(bool sync);
  bool synchronized_compression() const;
  /* currently not used                                   */
  void set_log_ranks(bool log_ranks);       bool log_ranks() const;
  /* print statistics?                                    */
  void set_verbose(bool verbose);           bool verbose() const;

  /* parse options in argc/argv                           */
  void set_from_command_line(int argc, char* argv[]);
  /* print description of command line options            */
  void describe_options() const;
};
```

5.2 HSS Command Line Options

The HSS specific command line options are:

```
HSS Options:
  --hss_rel_tol real (default 0.01)
  --hss_abs_tol real (default 1e-08)
  --hss_leaf_size int (default 128)
  --hss_d0 int (default 128)
  --hss_dd int (default 32)
```

```
--hss_q int (default 0)
--hss_max_rank int (default 5000)
--hss_random_distribution normal|uniform (default normal(0,1))
--hss_random_engine linear|mersenne (default minstd_rand)
--hss_compression_algorithm original|stable (default stable)
--hss_user_defined_random (default false)
--hss_enable_sync (default true)
--hss_disable_sync (default false)
--hss_log_ranks (default false)
--hss_verbose or -v (default false)
--hss_quiet or -q (default true)
--help or -h
```

6 HSS Approximation of Dense Matrices

The HSS code can be found in the `src/HSS/` subdirectory. All HSS code is in a namespace called `HSS`. The class for a sequential/multithreaded HSS matrix is `HSSMatrix<scalar>`, while the distributed memory HSS class is `HSSMatrix<scalar>`. For examples of the usage of these classes, see the test code in `test/test_HSS_seq.cpp` and `test/test_HSS_mpi.cpp` respectively. There is also one sequential example in `examples/MLkernel.cpp`, which uses HSS compression for kernel matrices as used in certain machine learning applications, see for instance [1]. This part of the code will be better documented in future STRUMPACK releases.

7 Examples

A number of examples are available in the `examples/` folder. This folder, with the sources, is copied to the build directory when running **`cmake`**. However, the examples are not part of the CMake build system, so they will not be compiled and linked when running **`make`**. A simple `Makefile` is generated in the `examples/` folder in the build directory, with the compiler setting based on what CMake has discovered. However, this Makefile is rather simplistic and in certain cases it might need some manual edits. However, it can serve as an example for how to compile code using STRUMPACK. Check the README file in the `examples/` directory for more details.

8 C Interface

The C interface is defined in the header file `StrumpackSparseSolver.h` and is very similar to the C++ interface. For example usage see the programs `sexample.c`, `dexample.c`, `cexample.c` and `zexample.c` in the `examples/` directory, for simple single and double precision real and complex example programs. Note that since the STRUMPACK code is written in C++, even when using the C interface you should link with a C++ aware linker or link with the standard C++ library. For instance when using the GNU toolchain, link with `g++` instead of `gcc` or link with `gcc` and include `-lstdc++`.

9 Advanced Usage Tips

- It is recommended to link with the `TCMalloc` library (`-ltcmalloc`). `TCMalloc` replaces the default memory allocator (C++ `new`) with a more scalable implementation. Alternatively, you can link with the Intel® TBB Scalable Allocator (`-ltbbmalloc`), in which case you also need to configure with `CPPFLAGS=-DUSE_TBB_MALLOC`.

- To keep track of the number of floating point operations performed in the STRUMPACK Sparse Solver, you can run configure with `CPPFLAGS=-DCOUNT_FLOPS`. Then, when running, do not set the `quiet` flag in the

StrumpackSparseSolver constructor or on the command line and the solver will print some statistics. This will also enable a counter for data movement in the solve phase, from which the (approximately) attained bandwidth usage is derived. This is done because the solve phase is typically bandwidth limited, while the factorization is flop limited.

- There is also some support for PAPI. Compile with `CPPFLAGS=-DHAVE_PAPI` and specify the PAPI include folders and libraries.
- We have added timers all throughout the code. These can be enabled with `CPPFLAGS=-DUSE_TASK_TIMER`. Running the code will generate a file `time.log`. A script to visualize these timings is provided.
- If you compile with MKL or OpenBLAS, you can take advantage of some extra optimized routines by specifying `-D__HAVE_MKL` or `-D__HAVE_OPENBLAS` respectively.
- The code is not completely thread safe at the moment: do not call solve on the same `StrumpackSparseSolve` object from different threads simultaneously.
- For comments, feature requests or bug reports: `{pghysels,xsli,gichavez}@lbl.gov`

10 FAQ

- Help, I get this compilation error:
  ```
  catastrophic error: cannot open source file "chrono"
  #include <chrono>
  ```
  You need a C++11 capable compiler, and also a **C++11 enabled standard library**. For instance suppose you are using the Intel 15.0 C++ compiler with GCC 4.4 headers. The Intel 15.0 C++ compiler supports the C++11 standard, but the GCC 4.4 headers do not implement the C++11 standard library. You should install/load a newer GCC version (or just the headers). On cray machines, this can be done with `module unload gcc; module load gcc/4.9.3` for instance.
- When running **`make test`**, many of the tests fail!

  The parallel execution in ctest is invoked by the `MPIEXEC` command as discovered by CMake. On many HPC clusters, this does not run unless it is executed from within a batch script. In this case all parallel tests will fail. At the moment, a small number of tests still fail. This is normal behavior.

11 Acknowledgements

The new, more robust version of the adaptive randomized HSS compression algorithm was developed in collaboration with Theo Mary from the Université de Toulouse.

The code for the STRUMPACK-sparse is based on the sequential code StruMF, originally developed by Artem Napov. We wish to thank people who sent us test problems and helped testing the code: Alex Druinsky, Yvan Notay and Shen Wang.

Partial support for this work was provided through Scientific Discovery through Advanced Computing (SciDAC) program funded by U.S. Department of Energy, Office of Science, Advanced Scientific Computing Research (and Basic Energy Sciences/Biological and Environmental Research/High Energy Physics/Fusion Energy Sciences/Nuclear Physics). This research was supported by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration.

12 Copyright notice

13 License agreement

References

[1] G. B. D Yu Chenhan, William B March, *An N log N Parallel Fast Direct Solver for Kernel Matrices*, in 2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS), IEEE, May 2017, pp. 886–896.

[2] I. S. Duff and J. Koster, *The design and use of algorithms for permuting large entries to the diagonal of sparse matrices*, SIAM Journal on Matrix Analysis and Applications, 20 (1999), pp. 889–901.

[3] P. Ghysels, X. S. Li, C. Gorman, and F. H. Rouet, *A robust parallel preconditioner for indefinite systems using hierarchical matrices and randomized sampling*, in 2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS), IEEE, May 2017, pp. 897–906.

[4] P. Ghysels, X. S. Li, F.-H. Rouet, S. Williams, and A. Napov, *An efficient multi-core implementation of a novel HSS-structured multifrontal solver using randomized sampling*, Submitted to SIAM SISC, (2015).

[5] P.-G. Martinsson, *A fast randomized algorithm for computing a hierarchically semiseparable representation of a matrix*, SIAM Journal on Matrix Analysis and Applications, 32 (2011), pp. 1251–1274.

[6] F.-H. Rouet, X. S. Li, and P. Ghysels, *A distributed-memory package for dense hierarchically semi-separable matrix computations using randomization*, ACM Transactions on Mathematical Software, (2016). to appear.

[7] Y. Saad, *Iterative methods for sparse linear systems*, Society for Industrial Mathematics, 2003.

[8] J. Xia, *Randomized sparse direct solvers*, SIAM Journal on Matrix Analysis and Applications, 34 (2013), pp. 197–227.