STRuctured Matrices PACKage Users' Guide

—

Sparse Package

Pieter Ghysels*, Xiaoye S. Li*, François-Henry Rouet*

Version 1.0.4, August 2016

Contents

[1]Lawrence Berkeley National Laboratory, Computational Research Division, MS 50F-1650, One Cyclotron Road, Berkeley CA94720. {pghysels,xsli,fhrouet}@lbl.gov

1 STRUMPACK Overview

STRUMPACK – STRUctured Matrices 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, for example, Hierarchically Semi-Separable matrices (HSS), to speedup linear algebra operations. STRUMPACK has two main components: a package for dense matrix computations (**STRUMPACK-dense**) and a package (**STRUMPACK-sparse**) for sparse linear systems. The dense package is described in detail in [6] while the sparse package is presented in [2]. This Users' Guide describes the sparse component of STRUMPACK. STRUMPACK-sparse 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 the STRUMPACK-sparse package are also the GMRES and BiCGStab iterative Krylov solvers, that use the approximate, HSS-accelerated, sparse solver as a preconditioner for 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.
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-sparse package uses the **CMake** build system. You need CMake >= 2.8. The recommended way of building the STRUMPACK-sparse library is as follows:

```
> tar -xvzf STRUMPACK-sparse-x.y.z.tar.gz
> mkdir STRUMPACK-sparse-build
> cd STRUMPACK-sparse-build
> cmake ../strumpack-sparse -DCMAKE_BUILD_TYPE=Release \
    -DCMAKE_INSTALL_PREFIX=/path/to/install \
    -DCMAKE_CXX_COMPILER=<C++ (MPI) compiler> \                          # this and below are optional,
    -DCMAKE_C_COMPILER=<C (MPI) compiler> \                              # CMake will try to autodetect
    -DCMAKE_Fortran_COMPILER=<Fortran77 (MPI) 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 examples   # optional
> make doc        # optional, needs doxygen
> 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 ../STRUMPACK-sparse-x.y.z -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. If CMake cannot locate these libraries, you can specify their path by setting the environment variable `$SCALAPACKDIR` or by specifying the libraries manually:

  ```
  > cmake ../STRUMPACK-sparse-x.y.z -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 ../STRUMPACK-sparse-x.y.z -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 ../STRUMPACK-sparse-x.y.z -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 8.

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 library `libstrumpack_sparse.a`.

3 Algorithm

The algorithm used in STRUMPACK-sparse is described in detail in [2], and is based on the work by Jianlin Xia [8]. Here we summarize the main algorithm features. Section 5 has some 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 [1]. 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 Usage

This section gives an overview on the basic usage of STRUMPACK-sparse. Additionally, we refer to the online automatically generated Doxygen pages at

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

for a complete and up-to-date documentation of the STRUMPACK-sparse API. Also, always pass command line options to the solver and run with `--help` or `-h` to get a list of options.

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

The STRUMPACK-sparse package is written in C++, and offers a simple C++ interface. See Section 7 if you prefer a C interface. STRUMPACK-sparse has three different solver classes, all interaction happens through objects of these classes:

- **`StrumpackSparseSolver<scalar,real,integer>`**
  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,real,integer>**
  This solver has (mostly) the same interface as `StrumpackSparseSolver<scalar,real,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`.

- **StrumpackSparseSolverMPIDist<scalar,real,integer>**
  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 three template parameters `<scalar,real,integer>`: the type of a scalar, the type of the corresponding real number and an integer type. It is recommended to first try to simply use the default `int` type for this last template parameter, unless you run into 32 bit integer overflow problems. In that case one can switch to for instance `int64_t`. The supported combinations of `<scalar,real>` are: `<float,float>`, `<double,double>`, `<std::complex<float>, float>` and `<std::complex<double>, double>`.

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,
      // some additional constants are defined in the strumpack::params namespace
typedef double scalar;
typedef double real;

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
  scalar* val = ...;          // nnz scalars
  scalar* x = new scalar[N];  // will hold the solution vector
  scalar* b = ...;            // set a right-hand side b

  StrumpackSparseSolver<scalar,real,int> sp(argc, argv);  // create solver object
  sp.set_relative_Krylov_tolerance(1e-10);                // set options
  sp.set_gmres_restart(10);                               // ...
  sp.set_from_options();                                  // 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
}
```

The main steps are: create solver object, set options and 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

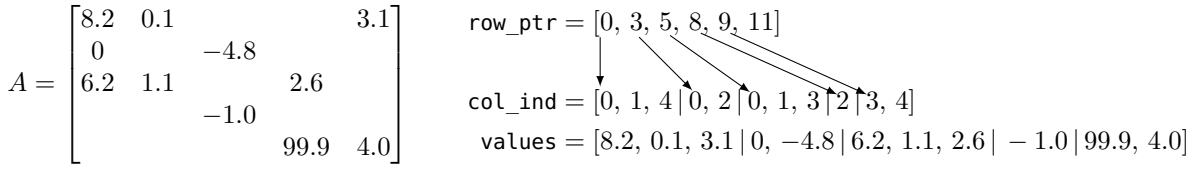

Figure 1: Illustration of a small 5×5 sparse matrix with 11 nonzeros and its Compressed Sparse Row (CSR) or Yale format representation. We always use 0-based indexing! Let $N = 5$ denote the number of rows. The `row_ptr` array has $N+1$ elements, with element i denoting the start of row i in the `col_ind` 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).

called Yale format, with 0 based indices. Figure 1 illustrates the CSR format. In the basic scenario, it is not really 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_sparse` and with the Metis, ParMetis, Scotch, PT-Scotch, BLAS, LAPACK, ScaLAPACK and BLACS libraries.

4.2 StrumpackSparseSolverMPI Example

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

```
#include "StrumpackSparseSolverMPI.hpp"
using namespace strumpack;
typedef double scalar;
typedef double real;

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
    scalar* val = ...;         // nnz scalars
    // allocate entire solution and right-hand side vectors on each MPI process
    scalar* x = new scalar[N]; // will hold the solution vector
    scalar* b = ...;           // set a right-hand side b

    // construct solver and specify the MPI communicator
    StrumpackSparseSolverMPI<scalar,real,int> sp(MPI_COMM_WORLD, argc, argv);
    sp.set_from_options();
```

```
    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,real,integer>` 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;
typedef double scalar;
typedef double real;
typedef int integer;

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
    integer* 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] = ..;
    integer local_n = dist[rank+1] - dist[rank];  // number of rows of the input matrix assigned to me
    integer* row_ptr = new integer[local_n+1];
    .. // set the sparse matrix row pointers in row_ptr
    integer local_nnz = row_ptr[local_n+1] - row_ptr[0];
    integer* col_ind = new integer[local_nnz];
    .. // set the sparse matrix column indices in col_ind
    scalar* val = new scalar[local_nnz];
    .. // set the matrix nonzero value in val
    scalar* x = new scalar[local_n];              // local part of solution
    scalar* b = new scalar[local_n];              // local part of rhs
    for (int i=0; i<local_n; i++) b[i] = ..;      // set the rhs

    StrumpackSparseSolverMPIDist<scalar,real,integer> sp(MPI_COMM_WORLD, argc, argv);
    sp.set_from_options();
    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();
}
```

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

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

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

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 Initialization and Command Line Option Parsing

Let

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

Each of the solver classes has a single constructor:

```
Sp::StrumpackSparseSolver(int argc, char* argv[], bool quiet=false);
SpMPI::StrumpackSparseSolverMPIDist(MPI_Comm mpi_comm, int argc, char* argv[], bool quiet=false);
SpMPIDist::StrumpackSparseSolverMPIDist(MPI_Comm mpi_comm, int argc, char* argv[], bool quiet=false);
```

where `argc` and `argv` contain the command line options and the `quiet` option can be set to `true` to suppress output of the solver. Once this object is created, options can be set on it, as discussed in the next few subsections. Then, calling the function `StrumpackSparseSolver::set_from_options()` will parse the command line options from `argv`, possibly overwriting any options defined on the `StrumpackSparseSolver` before this point. When one of the options is `--help` or `-h`, a list of all possible options is printed and the code exits. 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 can be given in either compressed sparse row or compressed sparse column format [7]:

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

Internally, the matrix is copied, so it will not be modified. The CSR format is generally somewhat faster in the iterative solver, since the CSR sparse matrix-vector product (SpMV) performs better in parallel than the CSC SpMV. However, CSC might be slightly faster in the factorization because it is column based storage and the dense matrices in the factors are also stored column major. 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 as follows:

```
void SpMPIDist::set_csr_matrix(integer_t N, integer_t* row_ptr, integer_t* col_ind,
                               scalar_t* values, bool symmetric_pattern=false);
void SpMPIDist::set_csc_matrix(integer_t N, integer_t* col_ptr, integer_t* row_ind,
                               scalar_t* values, bool symmetric_pattern=false);
void SpMPIDist::set_distributed_csr_matrix(integer_t local_rows, integer_t* row_ptr, integer_t* col_ind,
                                           scalar_t* values, integer_t* dist, bool symmetric_pattern=false);
```

The `SpMPIDist::set_csr_matrix` and `SpMPIDist::set_csc_matrix` routines take as input a sequential CSR or CSC matrix, but this matrix will internally be scattered over the different processes. The `set_distributed_csr_matrix` routine takes a block-row distributed CSR matrix as input as illustrated in the example above. Input of distributed CSC is not supported.

4.6 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 done by calling

```
params::returnCode Sp::reorder();
```

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

- `params::SUCCESS` success
- `params::MATRIX_NOT_SET` the matrix was not set
- `params::REORDERING_ERROR` problem with nested dissection or matrix reordering.

4.6.1 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 Sp::set_mc64job(int job);
int Sp::get_mc64job();
```

where `get_mc64()` queries the currently selected strategy (the default is **5:** maximum product of diagonal values plus row and column scaling). The command line option

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

can also be used.

4.6.2 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 Sp::set_matrix_reordering_method(params::MatrixReorderingStrategy m);
params::MatrixReorderingStrategy Sp::get_matrix_reordering_method();
```

The options for `MatrixReorderingStrategy` are

- `params::METIS`
- `params::SCOTCH`
- `params::GEOMETRIC`

When the solver is an object of the `Sp` or `SpMPI` classes, nested dissection will use Metis or Scotch if `METIS` or `SCOTCH` are set respectively. However, if the solver is an `SpMPIDist` object, setting `METIS` will result in a call to ParMetis and setting `SCOTCH` will call PT-Scotch.

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

```
params::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|scotch|geometric]
```

4.7 Factorization

Compute the factorization by calling

```
params::returnCode Sp::factor();
```

where the return value are the same as for `Sp::reorder()`. If `Sp::reorder()` was not called already, it is called automatically.

4.8 Solve

Solve the linear system $Ax = b$ by calling

```
params::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()`.

When `Sp::solve` is called on a `SpMPIDist` solver object, the right-hand side and solution vectors should only point to the local parts!

4.9 Command Line Options

To get a list of all available options, make sure to pass "`int argc, char* argv[]`" when initializing the StrumpackSparseSolver 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.

`--sp_gstype modified|classical` Gram-Schmidt type for GMRES

--sp_Krylov_solver auto|direct|refinement|pgmres|gmres|pbicgstab|bicgstab default: auto (refinement when no HSS, pgmres (preconditioned) with HSS compression). The gmres and bicgstab methods are NOT preconditioned. Use pgmres and pbicgstab to use the multifrontal+HSS preconditioner inside gmres/bicgstab.

--sp_reordering_method metis|scotch|geometric Code for nested dissection. Geometric only works on regular meshes and you need to provide the sizes. When using the sequential or MPI (replicated input matrix) interfaces, `metis` and `scotch` will use Metis and Scotch respectively, while the MPIDist fully distributed interface will use ParMetis or PT-Scotch respectively.

--sp_atol real_t (default 1e-10) Krylov absolute (preconditioned) residual stopping tolerance.

--sp_rtol real_t (default 1e-06) Krylov relative (preconditioned) residual stopping tolerance.

--sp_rctol real_t (default 0.01) See Section 5. HSS relative compression tolerance.

--sp_actol real_t (default 1e-10) See Section 5. HSS absolute compression tolerance

--sp_maxit int (default 500) Maximum Krylov iterations.

--sp_restart int (default 30) GMRES(m) restart length.

--sp_hss_front_size int (default 512) Minimum size of front for HSS compression, see Section 5.

--sp_nd_param int (default 8) Stop nested dissection recursion when separators become smaller than this value.

--sp_rank_offset int (default 128) See Section 5.

--sp_max_rank int (default 2000) See Section 5.

--sp_mc64job int [0-5] (default 5) See Section 4.6.1.

--sp_print_ranks filename (default no) Print out some info about the ranks encountered in the HSS matrices, only for sequential fronts.

--sp_q_power int (default 0) See Not supported yet.

--sp_separator_ordering_level (default 1) Only 0 and 1 are supported. When set to 1, the separator graph will be augmented with extra length-2 connections before being passed to the graph partitioner in order to determine the HSS partitioning. When set to 0, these extra connections are not added. Adding the connections can lead to much smaller ranks.

--sp_hss (default no) Use HSS compression of fronts, see Section 5.

--sp_rank_pattern adaptive|constant|sqrtN|sqrtNlogN|bisectcut default: `sqrtNlogN` See Section 5.

--sp_rank_factor float (default 1) See Section 4.6.1.

--sp_log_etree (default no) Print out the nested dissection tree.

--sp_log_ranks (default no) Print out info about HSS front sizes, nr of random vectors and HSS ranks.

--help or `-h`. Print info about the available command line options.

--sp_verbose or `-v` Print some output about the different steps in the algorithm.

--sp_quiet or `-q`. Suppress output.

--sp_task_level integer The number of recursion levels on which new OpenMP tasks are generated. The default option is $\log_2(\#\text{threads}) + 3$, which seems to give a reasonable level of task granularity on a range of number of threads.

--sp_random_blocksize integer The initial number of random vectors, is set to 128. The number of random vectors is adapted automatically, but if you have a good idea of the maximum HSS-rank for your application, for instance it is around 400, then you can set `--sp_random_blocksize 500`, where $500 = 400 + 100$ is somewhat larger than the actual HSS-rank 400 in order to get a better randomized sampling.

--sp_rank_offset integer The oversampling parameter p, default value is $p = 64$. This means that if the rank-revealing factorization detects a rank r, such that $r + p > d$, with d the number of random vector, the compression is rejected and new random vectors are added. To get good quality HSS compression, this parameter should not be too small. This parameter is also closely related to `--sp_random_blocksize`.

--sp_random_distribution uniform|normal The random number distribution. Default is to use a normal $\mathcal{N}(0, 1)$ distribution (mean 0 and standard-deviation 1). The other option is the uniform $[0, 1)$ distribution: `--sp_random_distribution uniform`. Uniform distribution requires approximately 7 floating point operations per random number, compared to 23 for the normal distribution. However, convergence results are generally better when using normal distribution.

--sp_random_engine linear|mersenne Default value is to use `minstd_rand`, the linear congruential engine [5]. Alternatively, the Mersenne twister `mt19937` [4] higher quality random number, but it has a much larger internal state, making it more expensive to seed. In our algorithm, we need to seed it frequently.

--sp_minpart integer The minimal size of a partition in the HSS tree. Default value is 128. Smaller values might lead to more parallelism but less efficient dense matrix operations. The value can also impact the HSS-rank.

5 Tuning the Preconditioning Strategy

The sparse matrix factorization algorithm used in STRUMPACK relies on a nested dissection reordering of the matrix in order to reduce fill-in in the LU factorization. Nested dissection recursively computes vertex separators from the graph of the sparse matrix. A vertex separator is a set of nodes, which, when removed from the graph split the graph into two unconnected components. Nested dissection is applied recursively to each of these components and this recursion defines a tree (typically binary) of separators. The first separator is called the root separator as it corresponds to the root node of the separator tree. In practice, the separators close to the root of the separator tree are the larger ones and separators become smaller as the recursion depth increases.

With each separator, a dense matrix, a so-called frontal matrix, is associated. The STRUMPACK sparse preconditioner will use low-rank compression (using structured matrices, specifically HSS matrices). Figure 3 illustrates the HSS matrix format. This low-rank technique asymptotically reduces memory usage and floating point operations, while introducing approximation errors. HSS compression is not used by default (the default is to perform exact LU factorization), but can be turned on via the command line:

```
--sp_hss   (no argument)
```

or via the C++ API as follows

```
void Sp::use_HSS(bool h);   // enable HSS if h == true
bool Sp::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 4.8.

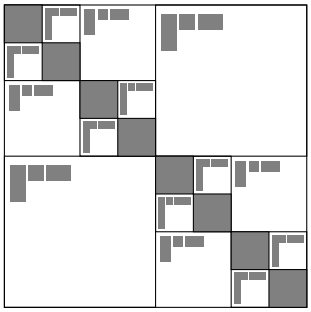

Figure 3: Illustration of a Hierarchically Semi-Separable (HSS) matrix. Gray blocks are dense matrices. Off-diagonal blocks, on different levels of the HSS hierarchy, are low-rank. The low-rank factors of off-diagonal blocks of different levels are related.

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. Moreover, a frontal matrix will only be compressed using low-rank if its parent in the separator tree is also compressed using low-rank. The minimum dense matrix size for HSS compression is set via the command line via

```
--sp_hss_front_size int (default 512)
```

or via the C++ API as follows

```
void Sp::set_minimum_HSS_size(int s);   // set minimum frontal matrix size for HSS compression
int  Sp::get_minimum_HSS_size();        // get minimum frontal matrix size for HSS compression
```

In STRUMPACK, HSS matrices are constructed using a randomized sampling algorithm [3]. 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 $S^c = A^T R$. Ideally, the number of columns 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 this needs to be estimated or computed. Finding a good estimate for the number of random vectors (i.e., the number of columns in R) is crucial for performance. STRUMPACK provides a few strategies to guess the number of random vectors:

- SQRTNLOGN: (this is the default) set $d = \alpha\sqrt{N}\log\sqrt{N} + \beta$, where N is the size of the separator. This is inspired by the idea that for a 3-dimensional $k \times k \times k$ mesh, the separator is $N = k \times k$, and there is some theory stating that for certain PDEs, the off-diagonal ranks grow as $\mathcal{O}(k)$.
- SQRTN: $d = \alpha\sqrt{N} + \beta$, same as above but without the logarithmic term.
- CONSTANT: $d = \beta$, take the number of random vectors constant, the same for all frontal matrices/separators.
- ADAPTIVE: Adaptive determination of the rank: $d_0 = \beta$, $d_{k+1} = 2d_k$. More random vectors are added until a certain accuracy is reached, see below. This only works for the StrumpackSparseSolver (i.e., the multithreaded) interface, not the MPI distributed code.
- BISECTCUT: Not implemented yet.

The rank strategy can be selected from the command line via:

```
--sp_rank_pattern adaptive|constant|sqrtN|sqrtNlogN|bisectcut (default: sqrtNlogN)
```

```
void Sp::set_rank_pattern(params::RankPattern p);
```

with `enum RankPattern {ADAPTIVE, CONSTANT, SQRTN, SQRTNLOGN, BISECTIONCUT}`.

How to set α and β?

After randomized sampling, a rank-revealing factorization is applied to subblocks of S^r and S^c to determine the actual off-diagonal block ranks and to build low-rank representations. The code currently uses QR with column pivoting as a rank-revealing QR (RRQR) factorization to determine the numerical or ϵ rank. So, the parameter ϵ, which determines the stopping criterion for the RRQR factorization is very important for both performance and quality of the preconditioner. However, ϵ is also closely linked to the number of random vectors. If the estimate for the number of random vectors is too high, the RRQR can still terminate early if ϵ is not too small. If the number of random vectors is too small, the computations are cheaper but RRQR will not reach it's desired accuracy ϵ. However, in this latter case, the resulting HSS approximation might still be used as a preconditioner.

The RRQR factorization in STRUMPACK takes both a relative and an absolute tolerance, which can be set via:

```
--sp_rctol real_t (default 0.0001)
--sp_actol real_t (default 1e-10)
```

or via the C++ API

```
void Sp::set_relative_compression_tolerance(real_t rctol);
void Sp::set_absolute_compression_tolerance(real_t actol);
```

6 Examples

TODO refer to examples folder with examples of:

- Factor once, solve multiple times
- Reorder, factor solve, change matrix values but reuse reordering!
- A complex arithmetic example
- A single precision factorization and a double precision solve??

7 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 test directory, for simple single and double precision real and complex test 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++`.

8 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,fhrouet}@lbl.gov`

9 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.

10 Acknowledgements

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).

11 Copyright notice

NOTICE. This software is owned by the U.S. Department of Energy. As such, the U.S. Government has been granted for itself and others acting on its behalf a paid-up, nonexclusive, irrevocable, worldwide license in the Software to reproduce, prepare derivative works, and perform publicly and display publicly. Beginning five (5) years after the date permission to assert copyright is obtained from the U.S. Department of Energy, and subject to any subsequent five (5) year renewals, the U.S. Government is granted for itself and others acting on its behalf a paid-up, nonexclusive, irrevocable, worldwide license in the Software to reproduce, prepare derivative works, distribute copies to the public, perform publicly and display publicly, and to permit others to do so.

12 License agreement

"STRUMPACK – STRUctured Matrices PACKage, Copyright (c) 2014, The Regents of the University of California, through Lawrence Berkeley National Laboratory (subject to receipt of any required approvals from the U.S. Dept. of Energy). All rights reserved."

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.

2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

3. Neither the name of the University of California, Lawrence Berkeley National Laboratory, U.S. Dept. of Energy nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

You are under no obligation whatsoever to provide any bug fixes, patches, or upgrades to the features, functionality or performance of the source code ("Enhancements") to anyone; however, if you choose to make your Enhancements available either publicly, or directly to Lawrence Berkeley National Laboratory, without imposing a separate written license agreement for such Enhancements, then you hereby grant the following license: a non-exclusive, royalty-free perpetual license to install, use, modify, prepare derivative works, incorporate into other computer software, distribute, and sublicense such enhancements or derivative works thereof, in binary and source code form.

References

[1] 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.

[2] 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).

[3] 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.

[4] M. Matsumoto and T. Nishimura, *Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator*, ACM Transactions on Modeling and Computer Simulation (TOMACS), 8 (1998), pp. 3–30.

[5] S. K. Park, K. W. Miller, and P. K. Stockmeyer, *Remarks on choosing and implementing random number generators, response (technical correspondence)*, Communications of the ACM, 36 (1993), pp. 108–110.

[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.