FastPILSS
Fast parallel verified linear (interval) system solvers

Quick Overview

Version 0.4.2

Contents

1 Introduction

The FastPILSS (Fast Parallel Interval Linear System Solver) software is a set of parallel linear (interval) system solvers that compute a verified (interval) solution of the system. This means the solvers compute a verified enclosure of the solution to the problem

$$Ax = b,$$

where A is a square matrix of dimension $n \times n$, b is a given right and side vector of dimension n and x is the solution vector of dimension n. The solver is based on the well known method by Rump [8, 9].

The elements of A and b can be real numbers, intervals, complex numbers or complex intervals. The elements of the solution x will then be (tight) intervals (or complex intervals) that have been verified to contain the actual solution of the linear system (for interval systems, the solution is an enclosure of the interval hull of the solution set.

The solvers use the powerful C-XSC library for scientifing computing as a backbone, which provides most of the necessary tools (interval arithmetic, accurate dot products etc.). BLACS and ScaLAPACK [2] are used where possible to obtain optimal speeds.

The solvers are fast compared to many other software solutions for this problem while maintaining a very high accuracy in its results. Due to the distributed memory parallelization it allows to compute the verified solution of very large linear systems (Dimension 100000 or even higher) by distributing the data accordingly. The dimension of the system is limited only by the size of the accumulated memory of the cluster used.

This document intends to give a quick overview on how to compile and use the solvers. Please keep in mind that this software should still be regarded as being in beta stage. If you have questions, suggestions or would like to report a bug, please feel free to mail us at `xsc@math.uni-wuppertal.de`.

Remark: The serial solvers that were included in this package before are now part of the C-XSC library itself. They can be used by including the header file `fastlss.hpp` and calling the solver function as before.

2 Installation

This chapter details the installation of the solvers.

2.1 Requirements

The requirements for the solvers are:

- A modern C++ Compiler (GNU Compiler 4.6.x or higher or Intel Compiler 12.x or higher recommended)
- C-XSC Version 2.5.2 or higher
- BLACS and ScaLAPACK libraries
- MPI environment

2.2 Quick install guide

Here is a quick rundown of the steps necessary to compile and use the solvers. More detailed explanations follow in the next section.

- Open the file `lss.inc` in the root directory. Edit the file according to your system.
- Type `make all` to compile everything.
- Run the programm `tester` from the `bin` directory to check if correct results are computed.
- After compilation you can find the static library `libcxscplss.a` containing the parallel solvers in the `lib` directory. The according header file can be found in the `include` directory.
- You can now use the solvers by including the header from the include path and linking to the solver library and according BLAS, BLACS and ScaLAPACK libraries. See Chapter 3 for instructions on calling the solvers.

2.3 Detailed installation procedure

In this section the installation procedure is explained in more detail.

2.3.1 Editing the `lss.inc` file

The file `lss.inc` is used to configure the installation step for your system. The following list explains the entries of this file in more detail:

- `PREFIX`: The install directory of the C-XSC library.
- `ROOTDIR`: The directory containing the `src` folder of the solvers and the `lss.inc` file.
- `CPP`: The command to invoke the C++ compiler you want to use, for example `g++`.
- `MPICPP`: The command to invoke the C++ compiler for MPI programs. Most MPI distributions have a convenience wrapper that calls the normal C++ compiler and automatically sets all compiler options necessary to compile MPI programs. In most cases it will be called something like `mpiCC` or `mpic++`. If your system does not provide such a command, set this entry to your standard compiler with the necessary compiler options for MPI programs.
- `OPTIMIZATION`: Sets the compiler optimization options. On most machines the default setting will work fine. Depending on your system, you might want to set some additional compiler options to improve inlining (for example the `-ipo` option of the Intel Compiler or options extending the inlining limits). Please refer to your compiler documentation for more information on the available options.
- `BLASVERSION`: Link options for the LAPACK and BLAS libraries you want to use. Remember that the order of the link commands can be important! Please refer to the documentation of your BLAS/LAPACK library for the correct link options.
- `SCALAPACKLINK`: Link options for the ScaLAPACK and BLACS libraries.
- `FORTRAN`: Depending on the BLAS library you use, you might have to link against the standard FORTRAN library. Set this entry accordingly (for example to `-lgfortran` if you use the GNU Fortran compiler).
- `FORTRANNAMING`: Naming conventions when calling FORTRAN functions from C or C++ code. The solvers call FORTRAN functions from ScaLAPACK directly. Depending on your FORTRAN compiler, the naming conventions for calling these functions from C++ can be different. Possible choices are `Add_` if an underscore is added to the name (`dgemm` becomes `dgemm_`), `Unchanged` if the name stays the same and `Uppercase` when the name is changed to upper case (`dgemm` becomes `DGEMM`).
- `CXXRPATH`: Sets the search path for the dynamic C-XSC library. If you did not compile C-XSC into a dynamic library (which is the default), you can comment this entry out or leave it blank. Depending on your operating system and compiler you might need to use another option or set an environment variable instead.

- `LIBTOOL`: The tool to create a static library from object files with the appropriate parameters. For Linux the default value `ar rcs` should normally work, however if you are using the Intel compiler with multi file optimization (compiler flag `-ipo`) you should set this to `xiar cru`.
- `MAKETOOL`: The command to invoke make. Normally this will be `make` or `gmake`.

2.3.2 Compiling the solvers using `make`

After setting the values in the `lss.inc` file correctly, you can start compilation by simply typing `make target` in the root directory, where `target` can be one of the following:

- `all`: Compile solver library, examples and test program.
- `library`: Only compile solver library.
- `examples`: Compile all examples.
- `tester`: Compile test program.

After compilation you can find the libraries in the `lib` directory and all executables (tester program and examples) in the `bin` directory.

2.3.3 Testing and examples

After successful compilation, you should run the `tester` program to check if all computations are performed correctly. This program does not use MPI and can be called directly. Possible problems are wrong results when computing dot products (then you have to set the precision to $K = 0$ or $K = 1$ when invoking the solvers) and incorrect results when using BLAS routines. This can happen if your BLAS library interferes with the floating point control word (especially the rounding mode setting) of the processor. You then have to use a different BLAS library (ATLAS BLAS is recommended).

If you compiled the example programs, you can find them in the `bin` folder. The programs `plss, pilss, pclss, pcilss` are the parallel example programs for real, interval, complex and complex interval systems, respectively. These, of course, use MPI hand have to be called appropriately (for example by using a command like `mpirun`).

The parallel examples all by default compute the solution to an example system of dimension 1000×1000 with one right hand side. The precision K, the dimension and the number of right hand sides can be given to the solver as program arguments (for example `plss 2 5000 10` computes the solution to the example system of size 5000×5000 using 2-fold double precision for the computation of the residual and a right hand side of dimension 5000×10). If you want to change the system, use a different right hand side or make other changes, you can alter the source code directly. The source code of the examples is located in `src/examples`. When running the parallel examples all nodes will put out log messages into a file `outputID.txt` located in the directory from which you called the example program, where `ID` is the MPI ID of the process.

3 Working with the solvers

This chapter explains how to use the solvers once they have been compiled. You can also take a look at the example programs in `src/examples` to show you how to call the solvers. Theres one example program for each solver (each basic data type: real, interval, complex, complex interval).

The solvers are compiled into the static library `libcxscplss.a` which can be found in the `lib` directory. To use them in your program, you have to include the file `cxscplss.hpp` from the `include` folder.

Only the solver for real interval systems (`pilss`) is explained here, the other solvers `plss, pclss, pcilss` work the same way, only the datatypes of the matrices are different.

Since version 0.4.0, the interface to starting the solvers has changed slightly, and has again been slightly modified for version 0.4.1 and 0.4.2. Many options are now set using a `struct`. Before calling the solvers, you have to create an instance of the struct `plssconfig`, which is shown in the following listing:

```
struct plssconfig {
  int   K;              //Dot product precision
  int   lssparts;       //Solver stages to use
  int   threads;        //Number of OpenMP threads
                        //(probably used by your BLAS library)
  int   maxIterResCorr; //maximum number of iterations during
                        //residual correction
  int   maxIterVer;     //maximum number of iterations during
                        //the verification step
  bool  refinement;     //Perform an iterative refinement?
  int   maxIterRef;     //maximum number of iterations during
                        //the refinement step
  real  epsVer;         //Epsilon for the verification step
  real  epsRef;         //Epsilon fot the refinement step
  int   nb;             //Blocksize for Scalapack
  bool  matrixMode;     //Activate matrix mode, faster for
                        //multiple right hand sides
                        //(forces K=1)

  plssconfig() : K(2), lssparts(LSS_BOTH_PARTS), threads(-1),
                 maxIterResCorr(5), maxIterVer(5), refinement(false),
                 maxIterRef(5), epsVer(0.1), epsRef(1e-5), nb(256),
                    matrixMode(false) {}

};
```

The attributes of the struct have the following meaning:

- `K`: The dot product precision to use in the solver. Note that for speed reasons not all computations are performed with this precision, but only the residual computations. Nevertheless, a higher setting can lead to more accurate results. Possible values are 0 (maximum precision), 1 (pure floating point) or $K \geq 2$ using K-fold double precision. The default value is $K = 2$, which should work fine for most practical applications. However, if you try to solve a system with high condition number using stage two of the solver (explained below), a setting of $K = 3$ often provides noticably better results. If $K = 1$ is used, Matrix Mode is activated automatically (see below).
- `lssparts`: Determines if only stage one (value `LSS_ONLY_PART_ONE`), only stage two (value `LSS_ONLY_STAGE_TWO`) or stage one followed by stage two - if stage one is not succesful - (value `LSS_BOTH_PARTS`) of the solver is performed. Stage one normally works for systems with condition number of up to about 10^{15}, while stage two can solve systems with condition numbers up to about 10^{30}. Be aware that if the system matrix is an interval matrix whose elements are not very thin intervals and stage one does not work, then stage two also is not very likely to work, since the interval matrix then most likely contains very badly conditioned or even singular point matrices. Also note that stage two takes considerably more computing time. The default value for this attribute is `LSS_BOTH_PARTS`.
- `threads`: The number of OpenMP threads to use. Currently, this has only an effect if the used BLAS and LAPACK libraries use OpenMP. Note that this value will override the number of threads set in the environment variable `OMP_NUM_THREADS`. The default value of -1 uses the current default number of threads for OpenMP.
- `maxIterResCorr`: Maximum number of iteration steps for the residual correction. The solver tries to improve the approximate solution computed for the midpoint system at the beginning. The maximum number of iteration steps can be set with this variable. Setting it low will save some time (should be negligible for larger systems), setting it higher might produce better results, especially for point systems.
- `maxIterVer`: Maximum number of iteration steps for the verification. In a final iteration, the solver tries to verify that the computed enclosure contains the true solution. If the enclosure can not be verified, but the matrix is not ill-conditioned, setting this variable somewhat higher might produce a verified result. Normally this should not be necessary.

- `refinement`: When set to true, an iterative refinement step is used to try to improve the computed enclosure somewhat. The cost for this is normally negligible and especially for interval systems the results are often (slightly) improved.
- `maxIterRef`: Maximum number of iteration steps for the refinement step described above.
- `epsVer`: Epsilon for the epsilon inflation used during the verification step. The iterate is blown up a little in each step to try to achieve an enclosure in the interior of the previous iterate.
- `epsRef`: Epsilon for the refinement step. This value dictates how large the maximal distance between two successive iterates should be for the refinement iteration to stop. Setting this value lower and the number of refinement steps higher might produce more accurate results.
- `nb`: The block size for ScaLAPACK. The block size can have a huge effect on the efficiency of the ScaLAPACK routines. The optimal setting depends on the used system. The default value is 256, which provided good results on most test systems. You should experiment with this setting to find the best value for your system.
- `matrixMode`: Enables the so-called matrix mode, which does not loop over multiple right hand sides but instead treats them as a matrix. For many right hand sides it is suggested to activate this mode. Note that enabling matrix mode forces $K = 1$ and the use of only stage one of the solver. Also, inner enclosures can not be computed in this mode.

Create an instance of the struct and overwrite the values you want to change (if any) or use the values sett by the default constructor. Then pass the instance as the last argument of the solver function call (see also the example programs).

Before starting the solvers, you have to also initialize the MPI environment and retrieve the process ID and number of processes yourself using the corresponding MPI functions. After that, you can start the solver by invoking the function:

```
  void pilss(imatrix& A, imatrix& b, imatrix& x, int m, int n, int procs,
int mypid, int& errc, ofstream& out, struct plssconfig cfg)
```

The parameters have the following meaning:

- `A`: The system matrix. When starting the solver, the complete system matrix must be stored in process 0. It will then be distributed to the other processes and deleted afterwards.
- `b`: The right hand side(s). When starting the solver, the complete matrix must be stored in process 0. It will then be distributed to the other processes and deleted afterwards.

- `x`: The solution matrix. After the solver is finished and if not in matrix mode (see above), the solution will be stored column cyclically distributed between the processes (A process p stores all columns j with $j \mod P = p$, where P is the number of processes used). The solution matrix must already have sufficient size to hold the result when starting the solver. If in matrix mode, process 0 will store the complete result matrix.
- `m`: The number of rows of the system matrix.
- `n`: The number of columns of the system matrix.
- `procs`: The number of processes used.
- `mypid`: Own MPI process id.
- `errc`: An error code. If `errc` equals 0, no error occured. Otherwise the function `std::string LinSolveErrMsg(int)` can be used to retrieve the error message corresponding to the error code.
- `out`: An output file stream to which the log data should be written.
- `cfg`: The configuration struct as explained above.

If you want to solve very large systems, the version of the solver described above might not work because the whole system matrix has to be stored by one process in the beginning. For such cases you can use a different version, where A and b are not given as matrices, but as function pointers. For the real interval case, this must be a pointer to a function like this:

```
void makeA(int i, int j, interval& r)
```

This function will be called by the solver while distributing the matrix. It should overwrite `r` with the entry at position (i, j) of the matrix. The other solvers require a function where r is of type `real`, `complex` or `cinterval`. The function can for example compute the entry, if there is some formula for the matrix entries, or read the appropriate value from a text file. This way, the system matrix never has to be stored completely by one process. Still, the larger the system you want to solve, the more memory is needed. In these cases you should give the solver as much ressources (nodes) as possible in your environment.

This second version is called as follows:

```
void pilss(Get_imatrix& A, Get_imatrix& b, imatrix& x, int m, int n, int
rhs, int procs, int mypid, int& errc ofstream& out, struct plssconfig cfg)
```

The parameters have the following meaning:

- `A`: Pointer to a function that can compute element (i, j) of the system matrix.

- `b`: Pointer to a function that can compute element (i, j) of the right hand side.
- `x`: The solution matrix. After the solver is finished and if not in matrix mode (see above), the solution will be stored column cyclically distributed between the processes (A process p stores all columns j with $j \mod P = p$, where P is the number of processes used). The solution matrix must already have sufficient size to hold the result when starting the solver. If in matrix mode, the solution will be distributed among the processes in ScaLAPACKs two-dimensional block cyclic distribution according to the block size set in the configuration struct. Consult the ScaLAPACK manual [2] for more information.
- `m`: The number of rows of the system matrix.
- `n`: The number of columns of the system matrix.
- `rhs`: The number of right hand sides.
- `procs`: The number of processes used.
- `mypid`: Own MPI process id.
- `errc`: An error code. If `errc` equals 0, no error occured.
- `out`: An output file stream to which the log data should be written.
- `cfg`: The configuration struct as explained above.

In both cases, an inner enclosure of the solution can be computed by passing an additional parameter `imatrix y` after the parameter x:

```
  void pilss(imatrix& A, imatrix& b, imatrix& x, imatrix& y, int m, int n,
int procs, int mypid, int& errc, ofstream& out, struct plssconfig cfg)

  void pilss(Get_imatrix& A, Get_imatrix& b, imatrix& x, imatrix& y, int m,
int n, int rhs, int procs, int mypid, int& errc ofstream& out, struct plssconfig
cfg)
```

An inner enclosure can be used to determine the quality of the computed outer enclosure. The true result is guaranteed to lie between the inner and outer enclosure. Size and data distribution of the inner enclosure are the same as for the outer enclosure. Note: Sometimes, no inner enclosure can be computed (especially for point systems). The corresponding elements of the inner enclosure are then set to `SignalingNaN`. Also note that when using $K = 1$ (matrix mode), no inner enclosure can be computed.

Note: You have to link your program not only against the solver library, but also against the BLACS, ScaLAPACK, BLAS, LAPACK and C-XSC libraries!

Bibliography

[1] L.S. Blackford, J. Demmel, J. Dongarra, I. Duff, S. Hammarling, G. Henry, M. Heroux, L. Kaufman, A. Lumsdaine, A. Petitet, R. Pozo, K. Remington, R. C. Whaley: An Updated Set of Basic Linear Algebra Subprograms (BLAS). ACM Trans. Math. Soft., 28-2 (2002), pp. 135–151.

[2] Blackford, L. S. and Choi, J. and Cleary, A. and D'Azevedo, E. and Demmel, J. and Dhillon, I. and Dongarra, J. and Hammarling, S. and Henry, G. and Petitet, A. and Stanley, K. and Walker, D. and Whaley, R. C.: ScaLAPACK Users' Guide. Society for Industrial and Applied Mathematics, 1997, Philadelphia, PA, ISBN = 0-89871-397-8

[3] Grimmer, M.: Selbstverifizierende Mathematische Softwarewerkzeuge im High-Performance Computing. Konzeption, Entwicklung und Analyse am Beispiel der parallelen verifizierten Lösung linearer Fredholmscher Integralgleichungen zweiter Art. Logos Verlag, 2007.

[4] Hofschuster, W.; Krämer, W.: C-XSC 2.0: A C++ Library for Extended Scientific Computing. Numerical Software with Result Verification, Lecture Notes in Computer Science, Volume 2991/2004, Springer-Verlag, Heidelberg, pp. 15 - 35, 2004.

[5] Hölbig, C.; Krämer, W.: Selfverifying solvers for dense systems of linear equations realized in C-XSC. Technical Report BUW-WRSWT 2003/1, 2003.

[6] Klatte, Kulisch, Wiethoff, Lawo, Rauch: C-XSC - A C++ Class Library for Extended Scientific Computing. Springer-Verlag, Heidelberg, 1993.

[7] Krämer, W., Zimmer, M.: Fast (Parallel) Dense Linear System Solvers in C-XSC Using Error Free Transformations and BLAS. Accepted for publication 2008, Springer Lecture Notes in Computer Science.

[8] Rump, S.M.: Kleine Fehlerschranken bei Matrixproblemen. PhD Thesis, Universität Karlsruhe, 1980.

[9] Rump, S.M.: Verification methods for dense and sparse systems of equations. In J.Herzberger, editor, *Topics in validated numerics*, Studies in Computational Mathematics, pages 63-136. Elsevier, Amsterdam, 1994.

[10] Zimmer, Michael: Laufzeiteffiziente, parallele Löser für lineare Intervallgleichungssysteme in C-XSC. Master Thesis, University of Wuppertal, 2007.