A Friendly and Free Parallel Block-Tridiagonal Solver: User Manual

Pascale Garaud[1] & Jean-Didier Garaud[2]
[1]Applied Mathematics and Statistics, Baskin School of Engineering, UCSC
[2]ONERA, Paris.

October 1, 2007

1 Motivation

We have developed this FORTRAN parallel algorithm for solving block-tridiagonal problems using MPI. It is specifically designed to be as easy-to-use as possible, and requires practically no knowledge of MPI (a vague idea of how MPI works is useful, but not essential). We find that it scales quite well, and has good stability properties. It has been tested on a few problems and seems to give the right answers. Note that the documentation on the algorithm will be provided shortly.

2 What problem does it solve?

The algorithm solves regular block-tridiagonal problems of the kind

$$MX = B \tag{1}$$

where M is a block-tridiagonal matrix, and B is the right-hand-side vector. Each block in M is of size `maxi`×`maxi`, and there are `mlines` block-lines in M. The size of B is therefore `maxi`×`mlines`. Here is an example of M with `maxi`=3 and `mlines`=5:

$$M = \begin{pmatrix}
X & X & X & X & X & X & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
X & X & X & X & X & X & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
X & X & X & X & X & X & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
X & X & X & X & X & X & X & X & X & 0 & 0 & 0 & 0 & 0 & 0 \\
X & X & X & X & X & X & X & X & X & 0 & 0 & 0 & 0 & 0 & 0 \\
X & X & X & X & X & X & X & X & X & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & X & X & X & X & X & X & X & X & X & 0 & 0 & 0 \\
0 & 0 & 0 & X & X & X & X & X & X & X & X & X & 0 & 0 & 0 \\
0 & 0 & 0 & X & X & X & X & X & X & X & X & X & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & X & X & X & X & X & X & X & X & X \\
0 & 0 & 0 & 0 & 0 & 0 & X & X & X & X & X & X & X & X & X \\
0 & 0 & 0 & 0 & 0 & 0 & X & X & X & X & X & X & X & X & X \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & X & X & X & X & X & X \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & X & X & X & X & X & X \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & X & X & X & X & X & X
\end{pmatrix} \tag{2}$$

where X denotes a non-zero entry.

Two versions of the code exist, where M and B are real or complex, respectively. Choose carefully which one you need to use.

3 Where to start?

3.1 Installation

The code and template examples are freely available upon request. Please email: `pgaraud@ams.ucsc.edu`

Instructions:

- Download the source file, which is `TriSolve.tgz`
- Uncompress and untar the file, using for example `tar -zxvf TriSolve.tgz` or `gunzip TriSolve.tgz` followed by `tar -xvf TriSolve.tgz`

This creates the following directories and files:

`TriSolve`:

- `src`:
  - `src_driver`:
    - `bfunc-cplx.f`
    - `bfunc-real.f`
    - `main-cplx.f`
    - `main-real.f`
    - `mfunc-cplx.f`
    - `mfunc-real.f`
  - `src_trisolve`:
    - `Tri-Solve-cplx.f`
    - `Tri-Solve-real.f`
- `workdir`:
  - `init_tri.h`
  - `Makefile-Pleiades`
  - `Makefile-gfortran-openmpi`
  - `solution-real-ref.dat`
  - `solution-cplx-ref.dat`

The directory `workdir` is the working directory, from which the code should be compiled and run. The `src` directory contains all the source codes. The `src_trisolve` directory contains the main solver, which the user should avoid trifling with. The `src_driver` directory contains the main driver as `main-****.f`, as well as two user-input functions `mfunc-****.f`

and `bfunc-****.f` which return the input coefficients of the matrix M and vector B of the problem to be solved. Simple template examples of these are provided. Two versions of each file exist, one for the real-version and one for the complex-version. Reference outputs are provided in `workdir`.

3.2 Compiling

Compiling the code will depend on the system. The `Makefile-Pleiades` provided in `workdir` is a template, which can be used *as-is* on Pleiades at UCSC. Another template is provided for use with OpenMPI and gfortran. The user should modify it for other platforms, by specifying the version of Fortran used, as well as the relevant compiler options.

The entry `NPROC` is the number of processors to which the job is submitted. Note that if `NPROC` in `Makefile` is different from $2^{\mathbf{nsize}} - 1$ then the code exits.

Two executables can be constructed: `make Tri-r` builds the real-version of the code from all the files `*-real.f`; `make Tri-c` builds the complex-version of the code from the files `*-cplx.f`. The executable is called `TriSolve`. Once it has been generated, to run the code, type `make run`.

3.3 Checking your installation

1. Go to the `workdir` directory, and create the `Makefile` according to your system specifications. You may have a look at one of the 2 provided templates (`Makefile-Pleiades` and `Makefile-gfortran-openmpi`)
2. Compile: `make Tri-r` (or `make Tri-c`)
3. Run: `make run`
4. Test: Compare the results with `solution-real-ref.dat` (or `solution-cplx-ref.dat`)

4 The driver

The file `main-****.f` contains a minimal sample driver for the problem. Note the following:

- The file `init_tri.h` is the file where the dimensions of the matrix are defined, as well as other parameters related to the number of processors used (see section 5 on input files).
- The variable `idp` is the number of the processor. Note that in MPI every single processor runs the same code, and needs to know its own identity. Identity assignement is done in the line `call initMPI(idp)` which *must* be included.
- The solver is called with the sequence `call trisolve(sol,bfunc,mfunc,idp)` where
  - `sol` is an array of size `maxi`×`mlines` containing the solution on exit (see section 5 on output files),
  - `bfunc` is a pointer to the function in `bfunc-****.f` (declared as external above) where the coefficients of B are entered (see section 5 on input files) .

  - `mfunc` is a pointer to the function in `mfunc-****.f` (declared as external above) where the coefficients of M are entered (see section 5 on input files).
  - `idp` is the processor number

- On return from `trisolve` every processor knows the full solution vector `sol`. Only one of them needs to write the solution to a file, hence the `if (idp.eq.0)` statement.
- Before exiting the program, each thread needs to exit cleanly, hence the line `call finishMPI(idp)` which *must* be included. Both routines `initMPI` and `finishMPI` are provided with the code.

5 Input/Output

5.1 Problem setup

The file `init_tri.h` is the file where the dimensions of the matrix are defined. It is located in the `workdir` directory and contains the following quantities which must be input by the user.

- `mlines`: as described in section 2, the number of block-lines
- `maxi`: as described in section 2, the size of the square blocks.
- `nsize`: the code can only work with a number of processors that is in the form $N_{\text{proc}} = 2^{\mathbf{nsize}} - 1$ (e.g., 3, 7, 15, 31, 63, 127, ..). Here, `nsize` is entered.

The other parameters in this file are calculated by the compiler.

5.2 Defining the matrix M

The matrix M is entered in the file `mfunc-****.f` through the function `mfunc(char,i,j,n)`. In this calling sequence,

- `n` references the number of the block-line considered.
- `char` is a character that references which block one is considering within the block-line `n`: `'m'` for the middle (diagonal) block, `'l'` for the left-side block and `'r'` for the right-side block.
- `i` and `j`: reference the line and column of the entry *within the block defined by* `n` and `char`.

The function returns the coefficient of M located at that entry. For a real problem, use `mfunc-real.f` and for a complex problem, use `mfunc-cplx.f`

5.3 Defining the vector B

The vector B is entered in the file `bfunc-****.f` through the function `bfunc(i,n)`. In this calling sequence,

- `n` references the number of the block-line considered.

- `i` references the line of the entry *within the block defined by* `n`.

The function returns the coefficient of B located at that entry. For a real problem, use `bfunc-real.f` and for a complex problem, use `bfunc-cplx.f`.

5.4 Example

The example given in the template files `init_tri.h`, `mfunc-real.f` and `bfunc-real.f` corresponds to solving on 3 processors the problem

$$\begin{pmatrix} 4 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 4 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 4 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 4 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 4 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 & 4 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 4 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 4 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 4 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 4 & 1 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 4 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 4 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 4 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 4 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 4 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \\ x_8 \\ x_9 \\ x_{10} \\ x_{11} \\ x_{12} \\ x_{13} \\ x_{14} \\ x_{15} \end{pmatrix} = \begin{pmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \end{pmatrix} \tag{3}$$

Another simple example is provided for the complex version.

5.5 Outputs

The solution is returned in the array `sol(i,n)`, where `i` and `n` reference the position of the entry as the `i`-th entry in the `n`-th block-line. All the processors know the full solution on exit from the routine. From the template driver, the solution is printed in the file `solution.dat` in the directory `workdir`.

The system also prints out a job report for each processor running, to help track down problems. These are named `proc***.out`, where `***` is the number of the processor.

6 Performance & Stability

The current version of the program has only been tested on a few simple cases. We do not guarantee perfect performance nor stability. However, the algorithm used has been found to be significantly more stable than the standard cyclic reduction, with no loss of performance.

Here is an example of performance scaling for runs on the Pleiades cluster at UCSC. For a complex problem, using `maxi = 1000` and `mlines = 1000`, then the running times are

- 31 processors: 73 minutes
- 63 processors: 40 minutes

- 127 processors: 25 minutes

Typically, one may expect very good scaling up to `NPROC = mlines` /10, and deterioration beyond that.

7 Acknowledgments & Disclaimers

Acknowledgments: This project was initiated as part of the construction of a parallel two-point boundary value solver, soon to be freely available too. J.-D. Garaud was supported by the California Space Institute, and P. Garaud is funded by NSF-AST-0607495. The computations and tests were performed on the Pleiades cluster at UCSC, purchased with the NSF grant NSF-MRI-0521566.

Disclaimer: This is *not* a commercial project; we have not yet tested the code particularly extensively, and cannot guarantee it to be fool-proof. Better and more efficient codes probably exist elsewhere. We will only support the code for our collaborators.

Acknowledging our work: If you do find this code useful, and happen to use it for your own research, please send us an email and if you may so be inclined thank us in your own acknowledgments! Happy computing!