Introduction to Supercomputing

TMA4280 · Project II

This project is mandatory, counts for 30% of the final grade and can be done in pairs or alone.

Instructions

The deadline is set on the **19th of April 2017**.

The deliverable consists of:

1. a report describing your solution, handed out by email in PDF format,
2. the source code developed to perform the computations stored in the same GIT repository as Project I,
3. a 5 minute demonstration of the code on the 19th of April highlighting the main results obtained.

Practical requirements regarding the code:

1. it should be hosted on Github,
2. it should be structured in different subdirectories addressing the different questions and results gathered in text files,
3. it must be written in C/C++ (without use of `std::vector`) or FORTRAN,
4. it must use double precision,
5. it must compile and run using Makefile targets,
6. results presented in the report have to be reproducible.

As soon as you have decided whether you want to work in pairs or alone:

- create a Github repository (one per pair) named `TMA4280LABS` if it is not the case already,
- send your name(s) and the link to the repository by email.

Problem description

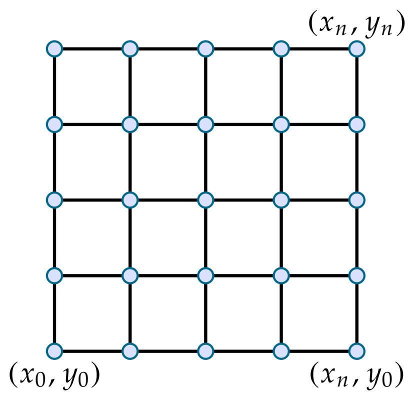
This project deals with the parallelization of the Fast-Diagonalization method introduced in Chapter 9 of the Lecture Notes. Consider the solution u to the two-dimensional Poisson problem with homogeneous Dirichlet boundary conditions,

$$-\Delta u(\boldsymbol{x}) = f(\boldsymbol{x}) \qquad \boldsymbol{x} \in \Omega = (0,1) \times (0,1), \tag{1}$$

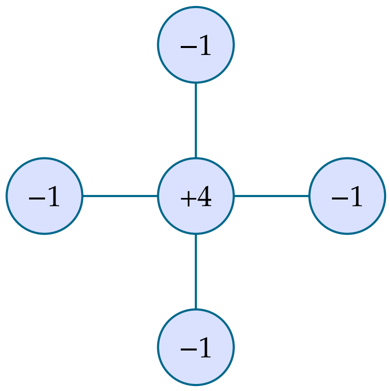
$$u(\boldsymbol{x}) = 0 \qquad \boldsymbol{x} \in \partial\Omega. \tag{2}$$

with f a given right-hand side.

Problem (1)–(2) is approximated by a finite difference method on a regular grid with $(n+1)$ points in each spatial direction, with the grid size $h = 1/n$,



and the Laplace operator is discretized by the standard 5-point stencil,



for which the discrete equations read:

$$-\frac{u_{i+1,j} - 2u_{i,j} + u_{i-1,j}}{h^2} - \frac{u_{i,j+1} - 2u_{i,j} + u_{i,j-1}}{h^2} = f_{i,j} \tag{3}$$

where $v_{i,j} = v(x_i, y_j)$ denotes the degree of freedom located at node (i, j). The solution vector can be represented in a matrix form

$$\mathrm{U} = \begin{bmatrix} u_{1,1} & \cdots & \cdots & \cdots & u_{1,n-1} \\ \vdots & \ddots & & & \vdots \\ \vdots & & u_{i,j} & & \vdots \\ \vdots & & & \ddots & \vdots \\ u_{n-1,1} & \cdots & \cdots & \cdots & u_{n-1,n-1} \end{bmatrix}$$

and the one-dimensional Laplace operator reads

$$\mathrm{T} = \begin{bmatrix} 2 & -1 & 0 & \cdots & 0 \\ -1 & 2 & -1 & & \vdots \\ 0 & \ddots & \ddots & \ddots & 0 \\ \vdots & & -1 & 2 & -1 \\ 0 & \cdots & 0 & -1 & 2 \end{bmatrix}$$

so that in matrix form the two-dimensional problem may be written:

$$\mathrm{TU} + \mathrm{UT} = \mathrm{G} \tag{4}$$

with G the matrix representing the right-hand side.

Let us recall that such diagonalization methods consists of two *matrix–vector* multiplications and one *vector* operation in one dimension of space, while it requires four *matrix–matrix* operations and one *matrix* operation in two dimensions of space:

1. $\tilde{\mathrm{G}} = \mathrm{Q}^T\mathrm{G}\mathrm{Q}$
2. $\Lambda\tilde{\mathrm{U}} + \tilde{\mathrm{U}}\Lambda = \tilde{\mathrm{G}}$
3. $\mathrm{U} = \mathrm{Q}\tilde{\mathrm{U}}\mathrm{Q}^T$

with λ the matrix of eigenvalues, and Q the matrix of (orthonormal) eigenvectors, where $\mathrm{Q}^{-1} = \mathrm{Q}^T$. The algorithm takes advantage of the fact that eigenvalues and eigenvectors can be computed explicitly. To optimize further the algorithm the Discrete Sine Transform (DST) can applied to compute $\tilde{\mathrm{U}}$ and $\tilde{\mathrm{G}}$, with the substitution described in Chapter 9:

$$\mathrm{Q} = \sqrt{\frac{2}{n}}\mathrm{S}^{-1}$$

The problem was shown to be computable in $O(n^2 \log n)$ floating-point operations.

Question 1. Write a program to solve the Poisson problem with P processes using the algorithm described. You can choose $f \equiv 1$ for the initial development.

Use the provided routines to compute the DST and its inverse based on the FFT. See Appendix A for details on compiling, linking and running the serial version of the Poisson solver. Use the MPI communication library to develop your program for a distributed memory parallel computer, and OpenMP to use t threads on each MPI process. See Appendix C for comments on the transpose operation.

Question 2. Run your parallel program on *Lille*. Obtain detailed timing results for different combinations of $n = 2^k$ and P, t. In particular, demonstrate that your program functions correctly for selected values of P in the range $1 \leq P \leq 36$. Follow the procedure described in Appendix B.

Note: Having a program that runs quickly is useless unless the answer is correct.

Note: It is sufficient and strongly recommended that you test the correctness of your program on a small problem size before solving larger problems.

Question 3. Run your program with $n = 2^{14}$ and $pt = 36$. Does the hybrid model work better, worse or equivalent to the pure distributed memory model? Explain your observations.

Question 4. Report the speedup S_p as well as the parallel efficiency η_p for different values of n and p. Th parallel efficiency is defined as $\eta_p = S_p/p$.

How do your timing result scale with the problem size n^2 for a fixed number of processors? Is it as expected? Do you see an improved speedup if you increase the problem size?

Note: To obtain consistent and reliable timing results, submit individual runs as different jobs.

Question 5. Modify the given f to be a function of your own choice. As an example, you could choose f to be a smooth function like

$$f(x, y) = \mathrm{e}^x \sin(2\pi x) \sin(2\pi y).$$

Another example is to let f represent point sources, e.g. $f \equiv 0$ in the whole domain except at two chosen gridpoints where $f = -1$ and $f = 1$ respectively.

Run your program with the new right hand side f for a particular n and p. Do you have to modify anything related to the parallel implementation when you change f, i.e. when solving a different Poisson problem?

Question 6. Discuss how you would modify the numerical algorithm to deal with the case where $u \neq 0$ on $\partial\Omega$, i.e. for non-homogeneous Dirichlet boundary conditions. You do not have to implement this.

Question 7. In the exercise and lectures we have assumed that the domain is the unit square, i.e. $\Omega = (0, 1) \times (0, 1)$. Discuss how you would modify the Poisson solver based on diagonalization techniques if the domain instead is a rectangle with sides L_x and L_y. You can still assume a regular finite difference grid with $n + 1$ points in each spatial direction. Does this extension of the original method change anything in terms of your parallel implementation? You do not have to implement this.

General comments

The program should be organized, easy to understand as well as well parallelized and load balanced.

A report describing the results of parallelization of a scientific problem typically contains

- a description of the problem;
- a discussion of possible solution strategies;
- a brief explanation of the finished program;
- a description of the computer on which the numerical results were obtained, which compiler (with version) and compiler options you used and other relevant information, such as libraries used and their versions;
- numerical results (preferably plots)
- analysis (theoretical and experimental) of the performance of the algorithm and its implementation (time usage, speedup and efficiency as functions of problem size and number of processes or threads);
- a discussion of bottlenecks and possible improvements; and
- possibly a listing of relevant parts of the source code.

A Compiling and linking

The serial code ships with a CMake build system. You can generate a build system using

```
module load gcc
module load openmpi
module load openblas
CC=mpicc FC=mpif90 cmake . -DCMAKE_BUILD_TYPE=Release
```

assuming you are located in the folder with the `CMakeLists.txt`. On success this will generate a makefile, and you can then build and run the program using

```
make
./poisson 128
```

The first statement builds the program, while the second runs it on a single processor with $n = 128$. Note that n must be a power of two.

By default CMake does not show you the compiler commands. You can see them by doing

```
make VERBOSE=1
```

It is recommended to perform out-of-tree builds. For example, using a `build` folder:

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

The CMake setup has options for compiling with MPI and OpenMP. They are on by default, but can be disabled at the configuration stage:

```
cmake . -DENABLE_OPENMP=0 -DENABLE_MPI=0
```

B Verification method

Computational codes should be tested at each stage of their development:

- unit testing: logic of the implementation (satisfaction of invariants, pre-conditions, post-conditions, ...)
- verification: numerical properties of the algorithm (convergence rate, stability, ...)
- validation:consistence of the model with experimentation (benchmarks)

One way to verify that the code works correctly is to do a *convergence test* using analytical solutions or by the method of manufactured solutions. Following this approach, let us assume an exact solution to the Poisson problem. For example, the exact solution may be given as

$$u(x, y) = \sin(\pi x)\sin(2\pi y).$$

This solution satisfies the boundary conditions.

Next, evaluate $-\Delta u$, which should be equal to the corresponding source term f, i.e.

$$f(x, y) = -\Delta u = 5\pi^2 \sin(\pi x) \sin(2\pi y).$$

Assuming the given data f we now solve the Poisson problem numerically, and compare the computed solution with the exact solution at the grid points. The maximum pointwise error should decrease to zero as $O(h^2)$the given data f we now solve the Poisson problem numerically, and compare the computed solution with the exact solution at the grid points. The maximum pointwise error should decrease to zero as $O(h^2)$. Remember that finding the maximum pointwise error will require communication.

C Comments on the transpose operation

The implementation of the transpose operation is trivial in a serial context. In a parallel context, using a distributed memory programming model, it is quite tricky. In this case the transpose of a matrix will involve all-to-all communication.

Part of the solution of this exercise requires a parallel implementation of the transpose operation. We are given a matrix of a certain dimension. We can distribute the matrix so that each process is responsible for a certain number of columns (or rows).

The most convenient way to implement the transpose operation is to use the MPI function `MPI_Alltoallv`. For a detailed description of this function, please use Google.

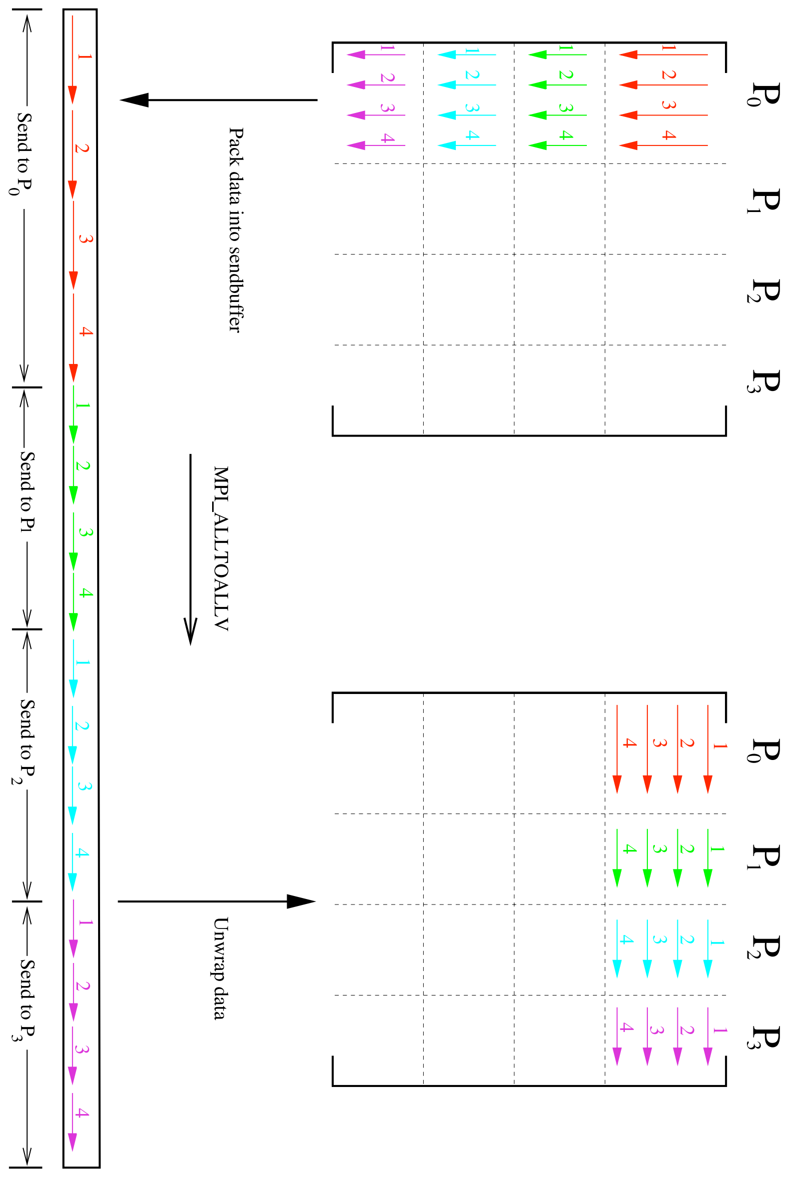


Figure 0.1: The transpose operation using message passing: the packing and unpacking of data. The figure is due to Bjarte Hægland.