Laboratory exercises 2: A Parallel Finite Element Problem

Xiaohui Wang (X.Wang-13@tudelft.nl)
Cong Xiao (C.Xiao@tudelft.nl)

(2020 version adapted dr. A.G.M. van Hees)

In this exercise, we continue with the Poisson problem from the previous exercise, but more general grids are used. These grids are in principle unstructured and are formed by a triangulation of the domain where a solution is sought. The problem is treated with the finite element method using triangles that are often used for irregular and/or unstructured grids in many realistic applications. Little attention is paid to the details of how to generate the relevant equations. They just are a set of sparse linear equations, that can be solved in principle with standard numerical software. We investigate the complications that arise due to these generalizations and the impact it has on performance as well as on interprocess communication in a parallel program. The solution strategy used in this exercise is the Conjugate Gradient method.

1 Introduction

In this exercise, we continue with Poisson's equation. An important difference with the first exercise is that the grid is now no longer regular. The grid cells are of a triangular shape instead of the rectangular grid cells in the previous exercise. These types of grids are often used in finite element calculations. This generalization has significant consequences on how we have to organize our software.

First of all, due to the arbitrary grid, we have to read the grid information from an input file. This information is generated with some external grid-generating tool. It is out of the scope of this course on how to generate problem-fitted grids, or how to place grid points at 'optimal' positions in space. In the first exercise, the grid was implicitly defined by just specifying the number of grid points in both the x- or the y-direction. Now there will be some data files that specify the grid uniquely.

The problem we encounter here is how to efficiently distribute the grid or the gridpoints over a number of processes. This partitioning phase can be performed with public domain tools (e.g., METIS, see http://glaros.dtc.umn.edu/gkhome/metis/metis/overview) or any reasonable ad hoc partitioning can be attempted. The topic of graph and grid partitioning is discussed in the course. With the partitioned grid information available it now becomes the responsibility of each process to obtain the required information. Processes should know their neighbors. Not every process has North, East, South, and West (or Top/Bottom/Left/Right) neighbors as in the previous exercise, but an arbitrary number. Grid points along each side of the boundary of two subdomains are no longer in a row or column of a matrix formed by the grid, but

they may be positioned rather arbitrarily. However, as far as computation is concerned, not much has changed.

The problem to be solved still is

$$\mathbf{Ax} = \mathbf{b} \tag{1}$$

where we want to solve for $\mathbf{x}$, i.e., to find $\mathbf{x}$ such that

$$\|\mathbf{Ax} - \mathbf{b}\| < \epsilon \tag{2}$$

The matrix $\mathbf{A}$ no longer has the simple structure with 4's along the diagonal and -1's on four off-diagonals. It has become more general, but it still is a sparse matrix. Each row of the matrix corresponds with a gridpoint. For each row in the matrix, there are as many non-zero off-diagonals as there are neighbors of that gridpoint. The matrix-vector multiplication can therefore be performed by a loop over the neighboring grid points.

The matrix elements of $\mathbf{A}$ contain only geometrical information and are evaluated only once, outside the iteration loop.

From a high-performance computing point of view, their evaluation is not something to pay much attention to, since it has to be done only once. Only if we have to decide whether to calculate and store these matrix elements once, or whether to evaluate them over and over again each iteration, this is an argument to consider. When we try to optimize a parallel code on any machine, we often have to decide between the replication of computation to save memory, or the use of additional memory to store data that can be reused later.

2 The finite element problem

In the previous problem we used a Cartesian grid with a fixed distance between lattice points. Such a grid is easily obtained. If the spatial domain in which one is interested has a rectangular shape, and if there is no reason to have more gridpoints in one area than in another, such a grid may be adequate. However, in practical situations, domains do not always have a rectangular shape. Also, the solution may have quite different behavior in different areas (or at different times).

We choose to expand the discretized solution in terms of basis functions that are 1 at a grid point and drop linearly to 0 at the nearby gridpoints, and moreover are linear within each triangle. It is the most simple type of finite element calculation. The problem we want to solve is essentially the same as in the first exercise, i.e., the Poisson equation with the solution kept fixed at some internal points of the domain, and kept fixed at zero along the boundary. The numerical solution strategy we chose is the conjugate gradient method.

The discretization and partitioning of the domain into a number of subdomains is completely separated from the parallel finite element solver. There are even two programs. The first program specifies a grid with triangular gridcells and takes care of the partitioning of the domain into the desired number of subdomains. This program is not a parallel program. We will only use it to create some specific situations. What this first program gives is actually everything that is necessary for the parallel finite

element program to do its work properly. For each subdomain that has to do work these required data are:

- The coordinates of the gridpoints and their index.
- The three indices of the gridpoints that form a triangular gridcell.
- For each neighboring domain there are two lists with indices of gridpoints. One list contains the indices of points for which information has to be sent to the neighbor. This information is just the value at the indicated point. The other list contains the indices of gridpoints for which information is received from the neighbor.

With this information available, each subdomain is able to manage its own work. This work may be either sending/receiving data or doing computations. The other program is the parallel finite element solver, where the actual work is done. We will now discuss its structure in more detail.

3 Parallelization of the finite element method

We may look at the parallelization process of a finite element solver in two different ways. First, one may start from the sequential code and discuss the steps that are necessary for the various processes to communicate the relevant data between each other. The other way is to look at the parallel program that was built in the previous exercise and discuss the differences and similarities. We chose for the latter option because that better illustrates in what respects this problem and corresponding program deviates from the Poisson solver in the previous exercise.

3.1 Required files

In order to do this exercise you need to have access to the following files and need to copy them into a directory of your own.

GridDist.c: A tool that organizes the distribution of the grid over processes.

grid.c: Included in **GridDist.c**. Part of the grid distribution tool.

sources.dat: A small file containing points that are to be kept fixed; used by **GridDist**. These points are the same as in the previous exercises.

MPI _ Fempois.c: The parallel finite element solver.

Makefile: used to generate the executables **GridDist** and **MPI _ Fempois**.

input.dat: The input data used by **MPI_Fempois**. It contains 2 lines that tell how to end the program: either when a convergence criterion is satisfied or when a certain maximum number of iterations is performed.

The commands you need to execute to run the required parts are the following:

1. **make**: This is the compilation phase, and should be executed whenever changes in the code are made. It makes new versions of **GridDist** and **MPI _ Fempois**

whenever needed.

2. **GridDist**: (with appropriate command line arguments, e.g., 2 2 100 100) The main function of this program is the generate an unstructured grid. In the example used in this exercise, the domain of interest is still the unit square, and actually, the problem is the same as in the previous exercises. With **GridDist** a couple of files **inputX_Y.dat** are generated that contain information about the grid that is read by the parallel finite element Poisson solver. The information in **inputX_Y.dat** is the following, First the number of vertices (gridpoints) that process Y is going to work on is given. For each point, the coordinates are given as well as an integer that denotes the type of vertex. Note, by looking at the beginning of these files that the total number of vertices in the files generated is larger than the number of points in the grid. The reason is that also the ghost- points are included in these files Subsequently information about the elements (the triangles) is given. For each triangle, its own index, as well as the indices of its three corner points are given. This fixes the topology of the grid. For parallel computing, the most important information is given at the end of these files. Here the indices of the vertices that are to be communicated are given. The format is illustrated in the following example

   neighbors: 2
   from 2 : 2550 2551 2552 2553 2554 2555 2556 2557 .....
   to 2 : 2499 2500 2501 2502 2503 2504 2505 2506 2507 .....
   from 1 : 50 101 152 ....
   to 1 : 49 100 151 ....

   Fist the number of neighboring process is given, followed by a from/to block for each neighboring process. In a from (or to) list each process gives its own indices. So the points in the from-line are supplied with the data that are received, whereas the points in the to-line are to be sent to that neighbor. Apart from the **inputX_Y.dat**, there is also a file created called **mappingX.dat** that contains the information about the process topology of the X processes.

3. **srun**: Use **srun** in the usual way, give **MPI_Fempois** as the program to be run. This is the standard way to execute an MPI job. Note that the number of processes should correspond with the input files that are provided. The structure of the program **MPI_Fempois** is similar to that of **MPI_Poisson**. The main difference is that **MPI_Fempois** does not use any arguments. All information needed is read from files.

At the end, if things work fine, **MPI_Fempois** generates a couple of output files called **outputX_Y.dat**, where X indicates the total number of processes and Y the rank of the process. Data in these files are written as (*x*, *y*, *value*) because the position of the gridpoints is not known by a simple formula. Corresponding output files can be concatenated to one file (with *cat*) or sorted (with *sort*) or processed otherwise for visualization or any other purpose.

3.2 Input and output

In the previous exercise, there was the opportunity to specify the number of processes in either direction. Now this is no longer an option since the domains are not necessarily arranged in a Cartesian grid. Hence all the information about grids is provided externally, by means of files. We have made the choice that the **GridDist** tool generates 'personalized' datafiles for each domain. How these domains are created, and why each domain covers a specific area is not the responsibility of the finite element Poisson solver **MPI_Fempois**, but of **GridDist**.

The only thing the user should verify is that the number of MPI processes that are started is identical to the number of domains that are generated with the grid distribution tool **GridDist**. Each process reads in its own data file.

Of course, it is also possible to concatenate all these input files together and read the combined one in only one process that sends the various pieces of input data to the process where it belongs. We have discussed these 2 options also in the previous exercise. Here we have chosen the option that each subdomain reads its own specific data. In this way, the similarity with the sequential code becomes more clear. The same holds for output. Each process writes its own piece of the 'solution' to output. Since besides the solution at a gridpoint the coordinates of that gridpoint are written, it does not matter in which sequence the resulting solution is written to file. Output files can easily be concatenated together or sorted. Post-processing of the resulting data is not the responsibility of the finite element Poisson solver either.

It should be noted that code that makes use of input- and output files in this way is not portable. It is not guaranteed that each process may do its own I/O.

3.3 Point-to-point communication

In the previous exercise each subdomain is arranged in a Cartesian grid, where each subdomain has 4 neighbors: N, E, S, W. If some of those neighbors do not exist (since a subdomain is at a border of the computational domain) there is no problem. Communication with non-existing neighbors implies that nothing has to be communicated. This does not mean that something is wrong.

Now it is much more general. Each subdomain must know which domains are neighbors. Therefore it uses a list of neighbors. The information about which subdomain is a neighboring subdomain is extracted from the input files, i.e. **mappingX.dat**. Apart from the knowledge to which subdomain information is to be sent, each process must also know which information it has to send. For the information that it receives, it also needs to find out in which locations that information has to be stored.

How was this solved in the previous exercise? For example, data that was received from the North neighbor was stored in the top row of a matrix, whereas the data that resides one row below the top was sent to the North neighbor. Since these data elements are stored in a matrix it is rather easy to calculate the address of all elements in a row or column. In MPI it is possible to create a special datatype for it with **MPI_Type_vector**.

Now we have a much more generic situation. There is no longer a 2-dimensional matrix with elements $\phi_{i,j}$, but only a one-dimensional list ϕ_k. Each index k has an x- and y-coordinate associated to it. However, it is not clear from just looking at the index, which indices correspond to border points and which ones correspond to ghost points. This is not necessary either, since this information is provided by the **GridDist** tool. It simply tells for which indices k the value ϕ_k has to be transmitted to each of the neighbors. For this purpose, there is a list of k-values for each neighboring subdomain. Similarly, there is another list that specifies which are the k-values of the (ghost)points for which the value k is received from the neighbor.

There are two things that are important in this respect

1. The number of items n^s that process A sends to a neighbor B should be identical to the number of items n^r that process B expects to receive from process A. This consistency is again the responsibility of **GridDist**.
2. There exists a global criterion to sort the gridpoints, and the gridpoints of each subdomain are sorted according to this criterion. This implies that if process A sends n^s elements to process B, the one with the lowest index k first, then process B knows that the first element received corresponds with its lowest index k. Hence, though the index of the same gridpoint in the 2 subdomains is different, there is no problem with moving the received data to the right locations.

In the code, the MPI function **MPI_Type_indexed** is used to create a datatype for each process to which a process sends and from which it receives data. See the function **Setup_MPI_Datatypes**. In the calculation, the communicating processes thus only have to exchange one element of this complicated data type. The alternative is to send (many) data of a primitive datatype like DOUBLE, between processes. The advantage of the present approach is that the data exchange in **Exchange_borders** has become very simple.

3.4 Global communication

Inherent with the conjugate gradient method is that the dot product of 2 vectors has to be calculated. The vectors are distributed over all processes. However, it is irrelevant how the gridpoints are distributed over the processes. Each process calculates its own local dot product. The local result is with an **MPI_Allreduce** processed such that at the end all processes have the global value of this dot product. Hence for global communication operations, it does not matter how the processes or subdomains are arranged, or how gridpoints are distributed over the domains.

4 Exercises, and performance aspects of the finite element code

4.1 Exercise 4.1

Read the preceding sections in this document carefully. Make sure you understand the essential differences between this finite element code and the Poisson solvers in the previous exercises. Briefly inspect the source code of **MPI_Fempois.c**. Specifically, in the section of data communications among the neighboring processors, you need to finish the code to realize the purpose of data communications.

4.2 Exercise 4.2

Analyze the time spent in the various phases of the finite element code. Distinguish between the following phases.

◊ Process is doing computations.
◊ Process is exchanging information with neighbors.
◊ Process is doing global communication.
◊ Process is idle.

Measure or collect these times in a number of runs with 4 processes (configurations 414 and 422) , with problem sizes 100×100, 200×200, and 400×400.

4.3 Exercise 4.3

Calculate the amount of data that has to be sent (and received) each iteration between a process and all its neighbors. Assume a uniform triangulated grid is partitioned, stripe-wise or block-wise, and distributed over P processes. Give approximate expressions that illustrate the dependence on problem size (n^2 grid points) and number of processes P.

4.4 Exercise 4.4

You may have noticed that with a 2×2 grid of processes there is some imbalance in the communication. Two processes are communicating with two neighbors, whereas the two other processes are communicating with three neighbors. You can see this by looking at the last few lines of the **inputX_Y.dat** files for a 2**×**2 process grid. Where does this asymmetry come from? How many neighbors communicate with the 'central' process in a 3×3 process grid? How many neighbors communicate with a 'corner' process in a 3×3 process grid?

4.5 Exercise 4.5

Estimate with a fixed number of processes, let's take 4, how large the problem size should be to spend the same amount of time on communication as to computation. Estimate for a fixed (big) problem size, let's take a 1000x1000 problem, the number of processes for which the communication and computation time are about equal. Perform the computations that you think are needed to make a reliable estimate. Do not exceed 2 minutes of processor time!

4.6 Exercise 4.6

Use the **GridDist** tool to generate a grid that has more gridpoints in regions near the 3 fixed points (source points). Use the keyword *adapt* as the last argument in the command line of **GridDist**. Does such a distorted grid lead to faster convergence? Does it affect the speed of convergence? The amount of computing time? Do this for sizes 100×100, 200×200, 400×400, etc.