Parallel Poisson's Equation solver

Tools of High Performance Computing

Noora Mäkelä

May 25, 2018

Contents

1 Introduction

The problem was to solve partial differential equation in parallel, in this case the Poisson's equation in two dimensions. In unit square the Poisson's equation is

$$\frac{\partial^2}{\partial x^2} f(x,y) + \frac{\partial^2}{\partial y^2} f(x,y) = g(x,y), (x,y) \in [0,1]^2 \tag{1}$$

The function g is known. This is discretized so that the unit square is divided to N part in two directions:

$$f_{i,j} = f\big(\frac{i}{N}, \frac{j}{N}\big), 0 \leq i,j \leq N \tag{2}$$

$$g_{i,j} = g\big(\frac{i}{N}, \frac{j}{N}\big), 0 \leq i,j \leq N \tag{3}$$

These can be divided into a system of $(N-1)^2$ equations and $(N-1)^2$ unknowns. This can be solved using iterative methods and also in parallel using for example using Jacobi over relaxation method or Successive over relaxation method. Here I have used the Successive over relaxation method.

2 Algorithms

So I chose to use Successive over relaxation method (SOR). It uses the Gauss-Seidel iteration algorithm:

$$f_{i,j}(t+1) = (1-\gamma) f_{i,j}(t) + \frac{\gamma}{4}\big[f_{i+1,j}(t) + f_{i-1,j}(t+1) + f_{i,j+1}(t) + f_{i,j-1}(t+1)\big] - \frac{\gamma}{4N^2} g_{i,j}. \tag{4}$$

Here γ is called over relaxation parameter. It's value is found by testing the converging of the program. Here t is the iteration index. In this algorithm the calculation order is important, because there are components with t+1 and t. This just means that the data of t+1 should be newest available and the data with t should be from the previous interval. This algorithm also converges fast. This is a serial algorithm, so it cannot be parallelized as it is. For this purpose I have used the red-black algorithm, which makes the parallelization possible.

Figure 1: The grid in red-black algorithm

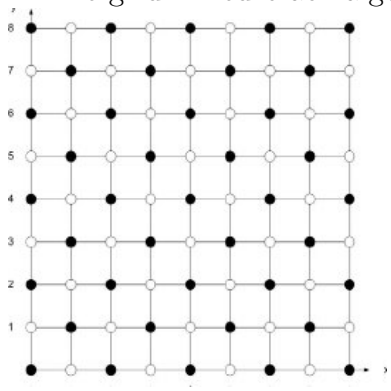


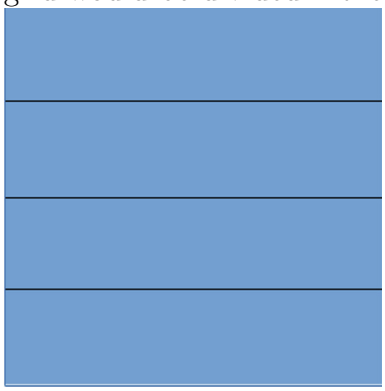
Figure 1: https://www.researchgate.net/figure/Discrete-solution-domain-for-half-sweep-scheme-Red-Black-ordering-strategies-have-been_fig1_221909640

The red-black algorithm divides the lattice into two sublattices. It is called red-black algorithm, because in it every other point is colored with black and every other with red like in the Figure 1. The white points in the figure can be considered red and the blacks, well black. So the basic idea is that the black points can be calculated just with red points and the red points can be calculated just with black points. The calculation is done an alternating fashion for each color at a time. This fits also well with the parallelization, because every process can calculate these in it's own area of the grid. And if their if need for a point in other process's area, they can communicate and change information.

3 Principles of Parallelization

The Gauss-Seidel iteration algorithm is not good for parallelization as it is, so the parallelization of it is done with the red-black algorithm, which I described in the previous section. The parallelization in the whole program is done with Message Passing Interface (MPI). Here we have an unit square, which I call here the grid. The grid is divided, so that every process gets a part of the grid. The parts are all same size. The program calculates how many rows of the grid every process gets, so grid is divided so that every process gets a certain number of rows of it.

Figure 2: How the grid would be divided if there were 4 processes



In the calculations the processes communicate with their neighbour processes by sending points to them. Those are the processes that are above and below a process. Processes send those point with each other that have a neigboring point that belongs to a different process. This makes it possible that the algorithm can be solved in parallel.

Every process calculates an alternating fashion the red and black points in it's own area in the grid. In the program I have used non-blocking communication between the processes. Non-blocking communication is the use of `MPI_Isend` and `MPI_Irecv`. With these the program does not completely get blocked. Every process has two grids, a grid with the newest information and a grid that has the information from the previous interval. In this way the data with t+1 and t is easy to maintain and access.

When every process has calculated their part of the grid for good amount of intervals, the processes send their grids to the process 0. The process 0 receives all of these and updates it's grid with these received grids. I have some strange bug in the code, that the process 0 does not receive the last row of the grid and half of the second last row. These appear to be just zeros after the process 0 has received them. Although the calculation in every process is done right, and the last rows in every grid are not just zeros before the sending to process 0.

4 Presentation of the code

My program is written in c++ and it contains 6 functions and a main routine. The functions are for initializing the grids, allocating them, printing them,

calculating red points, calculating black points and calculating the function g. The program starts by initializing the MPI and the grids. The grids are initialized to contain random numbers between 0 and 1. These random numbers are generated with Mersenne Twister random number generator. There are also boundary conditions, that can be changed. The program places that condition to the boundaries of the grid. In the beginning the initial grid is also printed to screen. Here every process has two grids, one that holds the newest information and one that holds the information from the previous interval. The grids here are allocated, so that they have continuous memory. This way they can be send and received in MPI without problems.

The program calculates how many rows every process calculates, what is the highest and lowest row of the part that the process is responsible for. In that way, we can make sure that every process calculates only it's own part. Then every process enters a for-loop where the Gauss-Seidel algorithm with red-black algorithm is used to calculate the Poisson's equation. First every process calculates red points and then black points. Then the grid with the newest information is copied to the grid, that has the older information, before the program enters a new iteration.

In the red point and black point functions (redDots and blackDots) the program first checks that the limits are correct. The boundaries of the grid are not calculated, so that has to be taken into account. Then the processes start to calculate the points in the grid. Every other row the points are calculated starting from the second point and every other starting from the third point. This is taken into accout with the if-loop that checks if the row number is divisible with 2. In the functions the processes send those points, that has a neighbour that belongs to some other process, to those neigbouring processes. This communication makes sure that the processes have enough information about the points that they need for their own calculations but do not belong to those processes. In the program the function g is just a sum of it's x and y coordinates, which were x=i/N and y=j/N. Here i and j are the coordinates in the grid and N is the size of the side of the grid.

When the processes have gone through enough intervals, every process except 0 sends their grid to process 0. Like I said in the previous sections, here was some bug, that prevents sending the last rows if the grids, although they are calculated right. Because of this the program also prints the grid of the last process to show that the last rows with not-zero values are there. Then process 0 receives all of those grids, and updates it's own grid so that every part of the grid now contains the newest information. Also in the end

the process 0 prints to screen the solved grid.

5 Instructions for using the code

The code is written in c++ and it uses MPI and random libraries so it must be compiled with "mpic++ -std=c++11 ParallerPoisson.cpp". The size of grid can be changed in the program by changing the constant N. The constant N is the lenght of the size of the grid, so the size of the grid is then NxN. Then the program can be ran with "mpirun -np 2 ./a.out". The size of the grid must be divisible with the number of processors. In this way every process get an equal amount of rows of the grid.

The program prints to the screen in the beginning the initial grid and the solved grid in the end of the program. The printed solved grid is not fully correct because of a bug, that did not let the processes send the last rows of the grid to the process 0. This is why the last and half of the second last row are full of zeros. The calculations still work, the only problem is with the sending or receiving in the end. To prove that the last process that is responsible for the lowest part of the grid prints its content to the screen before the sending to the process 0. In that grid we see that the last rows are also calculated and not full of zeros.

Here I do not have any input.dat or output.dat, since the program does not need any input data nor does it produce any data, except just printing to the screen.

6 Principles and results of benchmarking

I was not able to find the optimal value of the over relaxation parameter, although it would have been very important. This is because I was not able to handle the case where the different processes start to converge at different points of time.

The calculations took longer time with more processes and with larger grid size. This is quite obvious results, because those mean more processes that take time. With 400 as a grid size, 400 iterations and 2 processes the calculations took 0.021467 seconds. With 10 processes, 400 as a grid size and 400 iterations the calculations took 0.05685 seconds in CPU time. This still is not a long time, but the difference is relevant compared to the 2 process

case. And with more processes the CPU time grows. The program is not very scalable, so very big grids in the program take a lot of time to execute.

When I run the program with 1000 iterations,100 as a grid size and 5 processes my initial grid is:

```
3.000   3.000   3.000   3.000   3.000   3.000   3.000   3.000   3.000   3.000
3.000   0.900   0.226   0.524   0.503   0.887   0.380   0.850   0.172   3.000
3.000   0.513   0.825   0.180   0.231   0.889   0.352   0.513   0.605   3.000
3.000   0.171   0.240   0.134   0.260   0.768   0.949   0.862   0.522   3.000
3.000   0.199   0.435   0.805   0.879   0.638   0.145   0.701   0.789   3.000
3.000   0.303   0.399   0.116   0.121   0.347   0.324   0.591   0.344   3.000
3.000   0.048   0.298   0.919   0.243   0.211   0.030   0.651   0.358   3.000
3.000   0.207   0.099   0.999   0.421   0.960   0.855   0.937   0.760   3.000
3.000   0.834   0.820   0.432   0.834   0.174   0.551   0.426   0.121   3.000
3.000   3.000   3.000   3.000   3.000   3.000   3.000   3.000   3.000   3.000
```

And the solved grid is :

```
3.000   3.000   3.000   3.000   3.000   3.000   3.000   3.000   3.000   3.000
3.000   1.184   1.225   1.076   1.123   1.262   1.161   1.291   1.651   3.000
3.000   1.409   0.815   0.745   0.709   0.804   0.709   0.981   0.904   3.000
3.000   0.513   0.441   0.515   0.511   0.458   0.392   0.714   1.173   3.000
3.000   1.033   0.590   0.419   0.368   0.459   0.560   0.835   0.796   3.000
3.000   0.522   0.572   0.442   0.279   0.310   0.374   0.744   1.112   3.000
3.000   0.961   0.433   0.194   0.150   0.163   0.193   0.296   0.379   3.000
3.000   0.541   0.370   0.228   0.285   0.156   0.286   0.399   0.875   3.000
3.000   1.552   1.163   1.290   0.000   0.000   0.000   0.000   0.000   0.000
0.000   0.000   0.000   0.000   0.000   0.000   0.000   0.000   0.000   0.000
```

Here the last and second last rows full of zeros are because of the bug. This can be shown by printing the grid of the last process, which takes care of the last two rows, before sending it to process 0. The grid of the last process is:

```
3.000   3.000   3.000   3.000   3.000   3.000   3.000   3.000   3.000   3.000
3.000   0.900   0.226   0.524   0.503   0.887   0.380   0.850   0.172   3.000
3.000   0.513   0.825   0.180   0.231   0.889   0.352   0.513   0.605   3.000
3.000   0.171   0.240   0.134   0.260   0.768   0.949   0.862   0.522   3.000
3.000   0.199   0.435   0.805   0.879   0.638   0.145   0.701   0.789   3.000
3.000   0.303   0.399   0.116   0.121   0.347   0.324   0.591   0.344   3.000
3.000   0.048   0.298   0.919   0.243   0.211   0.030   0.651   0.358   3.000
3.000   0.207   0.099   0.999   0.421   0.960   0.855   0.937   0.760   3.000
```

```
3.000   1.552   1.163   1.290   1.178   1.284   1.285   1.305   1.266   3.000
3.000   3.000   3.000   3.000   3.000   3.000   3.000   3.000   3.000   3.000
```

We have to remember that in this grid the process 4 only calculated the last two rows, so the other rows are the same than in the initial array. Here we can see that the last rows have values, so something have gone wrong when sending the last process's grid to process 0. Here the boundary value was 3. This calculation took 0.065 seconds CPU time.

7 Conclusions

So in the program I solved Poisson's equation in two dimensions in parallel using Successive over relaxation method, Gauss-Seidel algorithm and red-black algorithm. To parallelize the program I used MPI. In the program I divided the grid to parts between the processes and every process took care of the calculations in it's own area. Processes communicated between each other, so that every process got the information they needed from other processes. This way I got an program that solves the Poisson's equation in parallel.

Although, I had a bug in the code, the processes were not able send their grids correctly to the process 0, collected the results. The last and half of the second last rows stayed empty, although the sended grids had values in those positions. I was not able to figure out why this is happening. This would have been important thing, because now the user is not able to see the full solved grid when it is printed. I was also not able to determine the ideal value of over relaxation parameter. This would have been very important, because then the program would have become much more faster and efficient. These are improvements that should be done. Also the code could be more efficient and clear.