Name: Joonas Kylliäinen
joonas.kylliainen@helsinki.fi
Student number: 013873703
Date: Monday 21st December, 2015

Solving Poisson's Equation Using parallel SOR-algorithm

Tools Of High Performance Computing

Contents

1 Introduction

Poisson's equation is a linear partial difference equation. It can be used for example in electrostatics, mechanical engineering and theoretical physics. It is used, for instance, to describe the potential energy field caused by a given charge or mass density distribution. The equation is named after the French mathematician, geometer, and physicist Siméon Denis Poisson. Poisson's equation in the unit square is [1]:

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

In this case g is a known function. The objective of this work is to find f when we also know its boundaries.

The first step is to discretisize the functions in N parts so that

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

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

We can use central differerence approximation for the second derivative of f

$$\frac{\partial^2}{\partial x^2} f(x,y) \approx \frac{1}{\Delta^2} \left[f(x+\Delta,y) - 2f(x,y) + f(x-\Delta,y) \right]. \tag{4}$$

And similarly for $\frac{\partial^2}{\partial y^2} f(x,y)$. This way we get

$$f_{i,j} = \frac{1}{4} \left(f_{i+1,j} + f_{i-1,j} + f_{i,j+1} + f_{i,j-1} \right) - \frac{1}{4N^2} g_{i,j}, \quad 0 < i,j < N \tag{5}$$

2 Algorithms

We can iteratively solve this equation for example using Jacobi's method but we can improve our solution by using relaxation factor γ which tells us that if $f_{i,j}$ is a good direction in which to move, one should move even farther to that direction by that factor. This way we get Jacobi over relaxation (JOR) method:

$$f_{i,j}(t+1) = (1-\gamma) f_{i,j}(t) + \frac{\gamma}{4} \left[f_{i+1,j}(t) + f_{i-1,j}(t) + f_{i,j+1}(t) + f_{i-1,j}(t) \right] - \frac{\gamma}{4N^2} g_{i,j} \tag{6}$$

where t is iteration index. The factor γ must be found by experimenting. This method converges relatively slowly so we can further improve it by

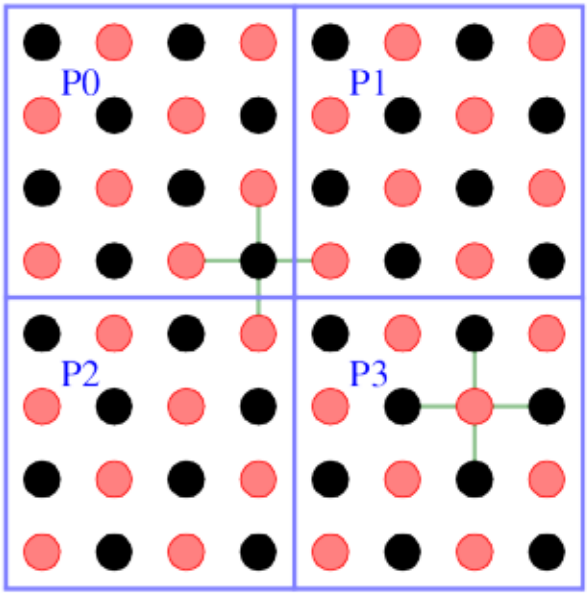

Figure 1: Parallelization [2]

taking account that every time an index i,j is calculated, we can use it in the same iteration. This way we get the Successive over relaxation (SOR) that uses the Gauss-Seidel iteration algorithm:

$$f_{i,j}(t+1) = (1-\gamma)f_{i,j}(t) + \frac{\gamma}{4}\left[f_{i+1,j}(t) + f_{i-1,j}(t+1) + f_{i,j+1}(t) + f_{i-1,j}(t+1)\right] - \frac{\gamma}{4N^2}g_{i,j} \tag{7}$$

3 Principles Of Parallelization

First we can use domain decomposition of the unit square by giving each processor a small rectangle for computation. This way we have to however take into account those indexes that are in other processes' or threads' rectangles and this need communication between them. But when using SOR, we can divide the workload by by choosing every second point for one iteration. So we loop through the grid points in the red-black order(named after the resemblance to a chess board). This is better understood from the picture 1

The idea is to update all the black points, followed by all the red points. The black grid points are connected only to red grid points, and so can all be updated simultaneously using the most recent red values. Then the red

values can be updated using the most recent black values. This yields the following algorithm:

```
for all black (i,j) grid points
      f(i,j)(t+1) = (f(i-1,j)(t) + f(i+1,j)(m) +
                     f(i,j-1)(t) + f(i,j+1)(t) + g(i,j))/4
end for
for all red (i,j) grid points
      f(i,j)(t+1) = (f(i-1,j)(t+1) + f(i+1,j)(t+1) +
                     f(i,j-1)(t+1) + f(i,j+1)(t+1) + g(i,j))/4
end for
```

When adding the relaxation factor to this, we get the final method.

4 Presentation Of The Code

The code first sets the g-function as a function with an known solution i.e.

$$g = \pi e^{(x+y)} \left[\cos\left(\frac{pi}{2}e^{(x+y)}\right) - \frac{\pi}{2}e^{(x+y)}\sin\left(\frac{\pi}{2}e^{(x+y)}\right)\right] \tag{8}$$

$$- \pi e^{(y-x)} \left[\sin\left(\frac{pi}{2}e^{(y-x)}\right) - \frac{\pi}{2}e^{(y-x)}\cos\left(\frac{\pi}{2}e^{(y-x)}\right)\right] \tag{9}$$

and

$$f = \cos\left(\frac{\pi}{2}e^{(y-x)}\right) + \sin\left(\frac{\pi}{2}e^{(x-y)}\right) \tag{10}$$

It calculates the g-function and then the real solution. From the real solution the boundaries are taken. The code first uses the JOR method without parallelization and measures the time it takes. The it uses SOR method without parallelization and measure the time it takes. For the sake of finding the correct γ-factor, it then uses a function to test how far the solution is fro the actual solution.

Then it calculates parallel using openmp the solution. Using openmp's walltime function we got the elapsed time from it.

5 Instructions For Using The Code

Using the code is simple. You can compile the code with the makefile provided by just giving the command make. It compiles the code using GNU fortran compiler gfortran and openmp-library. The program goes into run folder and it is named as poisson.

You can run the program with "./poisson gamma" command where gamma is the γ-factor between 0 and 1.

6 Results

The γ-factor was tried manually and for a matrix 100x100 it was best for it to be as big as possible. For 10x10 matrix 0.1 was the best. One sample output:

```
JOR time without parallelization:   0.168000013
SOR time without parallelization:    6.39999956E-02
difference:   0.455303162
Running with            4  threads
wall clock time using SOR with omp:    6.25000000E-02
```

Paralel code managed to be quicker, but only slightly. With bigger matrixes it would definetly got faster, but a problem of cache size became a barrier of increasing the matrix size.

7 Conclusions

SOR-algorithm is easy to convert into paralel code and if one fixes the cache size problem it will become very fast.

References

[1] http://web.mit.edu/dimitrib/www/pdc.html
[2] Tools Of High Performance Computing courses final project assignement