Parallel Solution of Laplace's Equation

Leo Unbekandt

November 30, 2013

Contents

Abstract

The main goal of this work is to develop a parallel way to solve numerically Laplace's Equation by using OpenMPI, and to analyze how this process accelerates the computation compare to the serial way to do, and how exact are the results compare to the analytical solution. The using method in this project must have been Jacobi with red-black ordering. However there is no interest in doing red-black ordering with Jacobi as we have to keep the old matrix in memory, that's why I've decided to use Gauss Seidel method associated with red-black ordering.

The complete source code of this project may be found on Github: https://github.com/Soulou/HPC_Assignment It includes the analytical, the serial and the parallel methods, and additionaly the LaTeX source code of this report.

1 The analytical solution

1.1 Solving the equation by using Fourier series

Finding the analytical solution basically consists in solving: 2

$$\frac{\partial^2 \phi}{\partial y^2} + \frac{\partial^2 \phi}{\partial x^2} = 0 \qquad x \in [0,1], y \in [0,1]$$

The homogeneous boundary conditions are:

$$\phi(x,0)=0 \qquad \phi(x,1)=0 \qquad \phi(1,y)=0$$

The inhomogeneous boundary condition is:

$$\phi(0,y)=sin^2(\pi y)$$

The separating the variables, we obtain $\phi(x,y)=X(x)Y(y)$, so Laplace's equation becomes:

$$\frac{1}{X}\frac{\partial^2 X}{\partial x^2}+\frac{1}{Y}\frac{\partial^2 Y}{\partial y^2}=0$$

Let:

$$\frac{1}{Y}\frac{\partial^2 Y}{\partial y^2}=-k^2 \Leftrightarrow Y(y)=A_1\cos(ky)+B_1\sin(ky)$$

So:

$$\frac{1}{X}\frac{\partial^2 X}{\partial x^2}=k^2 \Leftrightarrow X(x)=A_2\cosh(kx)+B_2\sinh(kx)$$

$$Or: X(x)=A_2\cosh(k(x-1))+B_2\sinh(k(x-1))$$

This second expression works better with $X(0)=0$ as boundary condition. As a result we have:

$$\phi(x,y)=[A_1\cos(ky)+B_1\sin(ky)][A_2\cosh(k(x-1))+B_2\sinh(k(x-1))]$$

Thanks to our boundary conditions we deduce that

$$Y(0)=0\Rightarrow A_1=0 \qquad Y(1)=0\Rightarrow k=n\pi \qquad X(1)=0\Rightarrow A_2=0$$

Finally we get:

$$\phi_n(x,y)=B_n\sin(n\pi y)\sinh(n\pi(x-1)) \qquad for\ \ n\in\mathbb{N}^{+*}$$

$$\phi_n(x,y)=\sum_{n=1}^{\infty}B_n\sin(n\pi y)\sinh(n\pi(x-1)) \qquad (by\ \ superposition)$$

Thanks to the inhomogeneous boundary condition we have:

$$\phi(0,y)=\sum_{n=1}^{\infty}B_n sin(n\pi y)\sinh(-n\pi)$$

The Fourier coefficient is now $B_n\sinh(-n\pi)$:

$$B_n\sinh(-n\pi)=\int_0^1\sin^2(\pi y)\sin(n\pi y)dy$$

$$B_n=\frac{2(\cos(\pi n)-1)}{\pi(n^3-4n)\sinh(-n\pi)}$$

1.2 Implementation of the solution

The implementation has been done with the C programming language. The only constraint of using this language is that the result formula contains $\sinh(-n\pi)$. When n gets bigger, a primitive variable type (double, long double) is not able anymore to have enough precision to compute correct results. This is why I've used the library GNU MPFR which is used to manipulate with high precision floating point numbers.

1.3 Illustration of the analytical solution

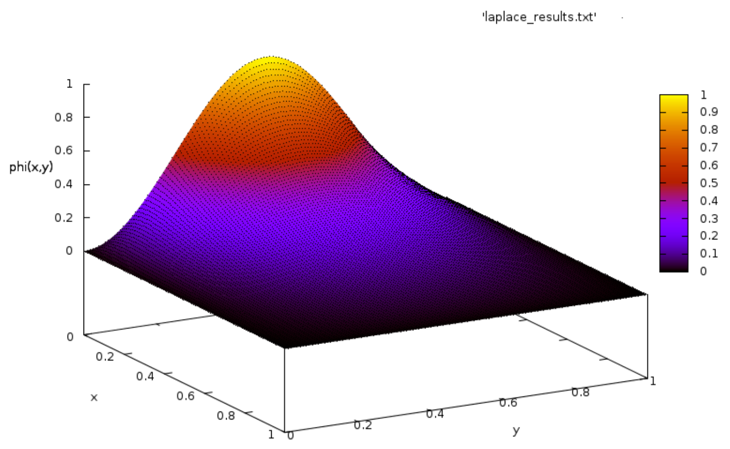


Figure 1: Representation of $\phi(x, y)$ for a stepsize of 1/120

2 The numerical solution - serial computation

2.1 Implementation

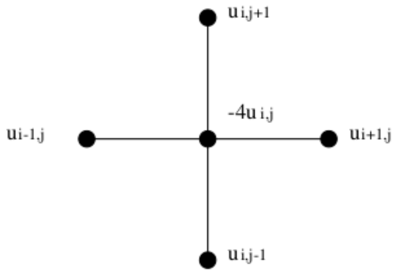
The implementation of Gauss-Seidel method consists in a successive number of iterations of the following formula.

$$\phi_{i,j}^{k+1} = \frac{1}{4}(\phi_{i+1,j}^{k} + \phi_{i-1,j}^{k+1} + \phi_{i,j+1}^{k} + \phi_{i,j-1}^{k+1})$$

$$i \in [2; k-1], j \in [2; k-1]$$

This is really convenient to implement, because by iterating by line as it

Figure 2: Finite difference stencil



is showed in the following schema, when we calculate the point $o22$ we already have $\phi_{i-1,j}^{k+1}$ and $\phi_{i,j-1}^{k+1}$ because they have been computed previously. The consequence is that we don't need to keep a copy of the old matrix in memory, we can iterate on in directly.

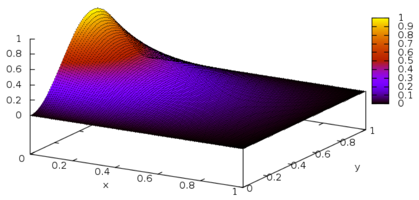
```
 ------------------------------
|                              |
| f   0   0   0   ...   0   0  |
| f  n11 n12 n13  ...  n1k  0  |
| f  n21 o22 o23  ...  o2k  0  |
|                              |
|                              |
|           .........          |
|                              |
|                              |
|                              |
|______________________________|
```

Another important point, is the stopping condition. The following choice has been done: after each iteration we are looking at the norm of the difference between the newly computed matrix and the previous one and we compare it to a certain tolerance the user has to give as argument to the software.

$$||\phi^{k+1} - \phi^{k}|| < \epsilon$$

2.2 Aspect of the results

The results look similar as the results obtained in the analytical computation:



2.3 Validation of the implementation

To know if the results are accurate we need to compare them to the results of the analytical solution. In order to achieve that, I've developed a small script: `diff_results.rb` which compares two result files by calculating the difference of each $\phi(x, y)$ and finally printing the average difference in percents. The following table shows the difference between the analytical implementation and the Gauss-Seidel serial version, for different sizes of domain and different error tolerances [1].

- N: Number of steps/size of the result matrix
- ē: Average error
- t: convergence tolerance

Table 1: Difference between analytical and serial solutions

N	ē ,t = 10^{-2}	ē, t = 10^{-3}	ē, t = 10^{-4}	ē, t = 10^{-5}
20	0.1822	0.1325	0.1234	0.1227
40	0.2170	0.1086	0.0737	0.0708
60	0.2626	0.1091	0.0554	0.0500
80	0.3004	0.1171	0.0465	0.0388
100	0.3206	0.1228	0.0421	0.0320

There are different things we can observe in this table. First, whatever is the size of the Domain, we can observe that the precision of the results is better when the tolerance is decreasing. Then, the results look accurate when the convergence tolerance is small enough. Actually when $t = 10^{-5}$ and $N \geq 60$, the numerical results are less than 5% different from the analytical one. Finally we can also see that the bigger the domain, the small the required tolerance has to be to obtain good results. As a result we can validate this way to measure the error and to detect the convergence.

[1] Data generated by: analytical_serial_diff.sh

3 OpenMPI - parallel computation

3.1 Computational domain decomposition

To divide a matrix, there are two main solutions:

- Panels:

```
 ____________________        ____________________
|      |      |      |      |                    |
|      |      |      |      |                    |
|      |      |      |      |____________________|
|      |      |      |      |                    |
|      |      |      |      |                    |
|      |      |      |      |____________________|
|      |      |      |      |                    |
|      |      |      |      |                    |
|______|______|______|      |____________________|
```

- Squares:

```
 ____________________        ____________________
|          |         |      |      |      |      |
|          |         |      |      |      |      |
|          |         |      |______|______|______|
|          |         |      |      |      |      |
|----------|---------|      |      |      |      |
|          |         |      |______|______|______|
|          |         |      |      |      |      |
|          |         |      |      |      |      |
|__________|_________|      |______|______|______|
```

As shown above, the panels decomposition may be horizontal or vertical. Fondamentaly there is no difference between their characteristics, their are as many value to calculate and to exchange. However their is one difference which is linked to how caching is managed on the computing nodes. In our case the cache is storing lines, that's why it is much better to use horizontal panels than vertical.

The main advantage of the panels compare to the squares is that each process only need to communicate with a maximum of 2 other instances (the one above and the one underneath). Even if by dividing the domain in squares, some processes would need to communicate with 4 other nodes, the amount of data which has to be exchanged by each process is smaller