Uppsala Universitet

Parallel and Distributed Programming 1TD070

Final Project - Parallel Poisson equation solver

Agelii, Carl.

June 18, 2024

Contents

1 Introduction

The Poisson equation is a partial differential equation.

$$\nabla^2 u(\boldsymbol{x}) = b(\boldsymbol{x}), \tag{1}$$

where $u(\boldsymbol{x})$ is the solution and $b(\boldsymbol{x})$ is a given source function. Here, we look at the case where the domain is given by the 2-dimensional plane $[0,1] \times [0,1]$ and the source function:

$$b(x,y) = 2x(1-x) + 2y(1-y). \tag{2}$$

Now, the plane is discretized into a mesh of $(n+2) \cdot (n+2)$ points, with a distance of $h = \frac{1}{n+1}$ between each point. The points are given by the index i and j in the mesh and $x = ih$, $y = jh$. $i, j = 0, 1, 2 \ldots n+1$. The boundary conditions are homogeneous dirichlet conditions, which means that only $n^2 = N$ points need to be calculated. The Laplace operator is also discretized using finite differences:

$$(\nabla^2 u)_{ij} = \frac{1}{h^2}(u_{i+1,j} + u_{i-1,j} + u_{i,j+1} + u_{i,j-1} - 4u_{i,j}). \tag{3}$$

This gives a system of N linear equations:

$$A\boldsymbol{u} = \boldsymbol{b}, \tag{4}$$

where A is a $N \times N$ matrix and both $\boldsymbol{u}$ and $\boldsymbol{b}$ are N-long column vectors. By moving the h^2 to the right side of the equation, the entries to the $\boldsymbol{b}$ vector can be calculated as:

$$b_{ij} = 2h^2(x(1-x) + y(1-y)). \tag{5}$$

2 Algorithm

2.1 Sequential

In order to get the results, the linear system 4 needs to be solved. This can be done in a number of ways, but here, we look at the iterative conjugate gradient method. This algorithm looks like:

Algorithm 1 The CG-method

```
 1: Initialize $\boldsymbol{u} = 0, \boldsymbol{g} = -\boldsymbol{b}, \boldsymbol{d} = \boldsymbol{b}$
 2: $q_0 = \boldsymbol{g}^T\boldsymbol{g}$
 3: while $q_i >$ Tolerance do
 4:     $\boldsymbol{q} = A\boldsymbol{d}$
 5:     $\tau = q_i/(\boldsymbol{d}^T\boldsymbol{q})$
 6:     $\boldsymbol{u} = \boldsymbol{u} + \tau\boldsymbol{d}$
 7:     $\boldsymbol{g} = \boldsymbol{g} + \tau\boldsymbol{q}$
 8:     $q_{i+1} = \boldsymbol{g}^T\boldsymbol{g}$
 9:     $\beta = q_{i+1}/q_i$
10:     $\boldsymbol{d} = -\boldsymbol{g} + \beta\boldsymbol{d}$
11: end while
```

2.2 Parallelization

Next, we want to make the code faster by implementing algorithm 1 in parallel. By looking at algorithm 1, there are some caveats to doing so. First off, the initialization can be parallelized, so long as q_0 is computed globally afterwards. The b vector is split into equal or almost equal parts (if n does not divide p) and then the individual components are calculated. Next, step 4 is a matrix vector multiplication. However, we can exploit the nature of the matrix A, since the multiplication can be done as a stencil operation. For this, overlapping values between processes need to be communicated between them, but only the first and last row need these values. Therefore, it was deemed smart to use non-blocking communication and start the calculation on the middle rows, as to avoid idle processes. This was done using MPI_Isend and MPI_Irecv. Next, global communication was performed to reduce the vector multiplication in step 5, and later in step 8. This was done using MPI_Allreduce along with the MPI_SUM reduce-type. First the local vector multiplication was done and then the total was summed up. The other steps were done in parallel without issue.

3 Experiments and results

In order to evaluate the algorithm, some experiments were conducted. These are split into *validity* and *performance* experiments. The *validity* experiments checked whether the numerical solution was logical and correct. The *performance* experiments timed the algorithm and checked how the parallelization scaled with different problem sizes.

3.1 Validity

The first experiment that was done tested the validity of the $\boldsymbol{b}$ vector. By integrating over the whole domain, we get the equation:

$$b_0 = \int_0^1 \int_0^1 2(x(1-x) + y(1-y))dxdy = 2/3 \approx 0.667. \tag{6}$$

Theoretically, the same value should be approximated when summing the entries of the b vector when using equation 5. By doing so with $n = 512$, a value of $b_{sum} = 0.665...$ was achieved.
Now, it is time to check the solution. Figure 1 plots a solution when $n = 256$.

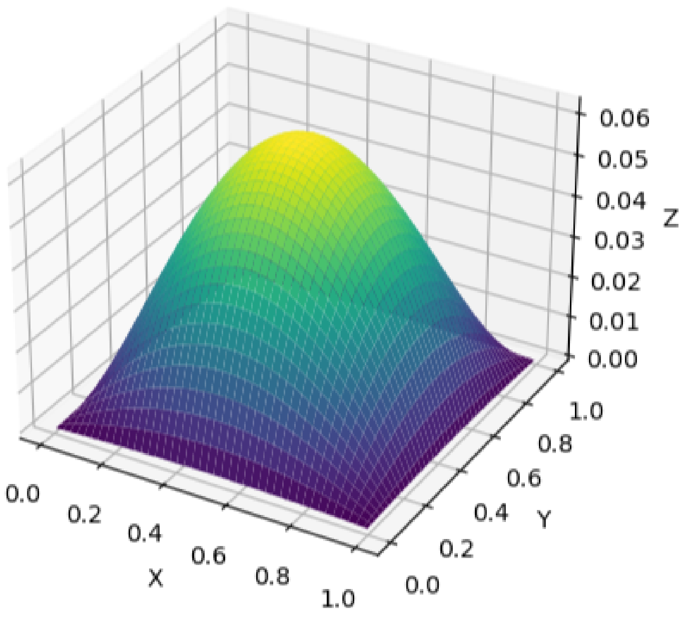

Figure 1: Numerical solution, $n = 256$ with 200 CG iterations.

By solving equation 1 analytically, the solution is:

$$u(x, y) = x(1 - x)y(1 - y). \tag{7}$$

Now, we calculate the analytical solution using equation 7 for a grid with $n = 256$ and compare the error norm to the numerical solution. This yielded an error of ≈ 0.0075.

3.2 Performance

Now that the validity of the solution has been tested, the performance can be examined. Both fixed size and weak scalability experiments were conducted on *snowy*. Firstly, the problem size was fixed to $n = 1024$ which means $n^2 = N = 1048576$. Then, the number of processes varied as $p = 1, 2, 4$ and 8. Table 1 shows the results.

Number of Processes	**Execution Time (seconds)**
1	1.488083
2	0.839849
4	0.351128
8	0.174050

Table 1: Execution Time for Different Numbers of Processes when $n = 1024$.

The speed up for these execution times can be seen in figure 2.

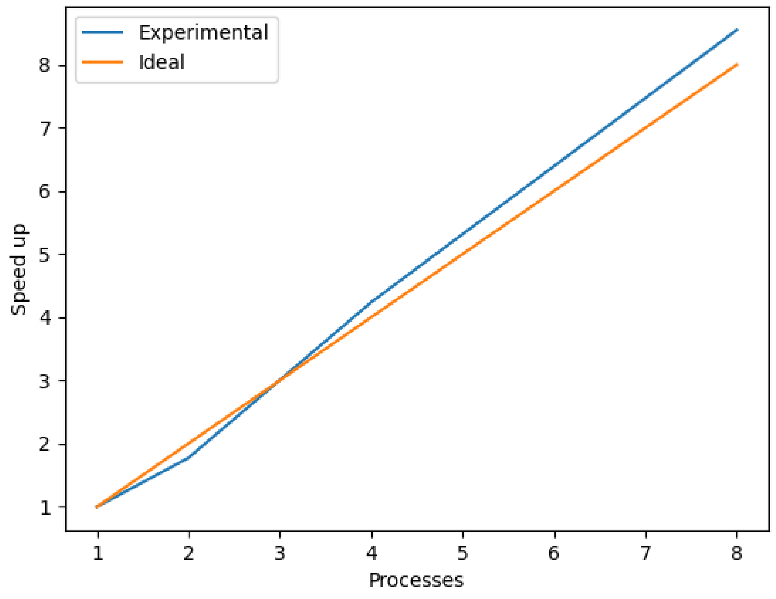

Figure 2: Speed up for different processes when $n = 1024$.

For different problem sizes, the q_0 value was printed at the last iteration. This is a value that is associated with the convergence of the CG-algorithm. This value for different problem sizes for 200 CG iterations can be seen in table 2.

n	q_0
256	3.817e-05
512	4.432e-03
1024	7.275e-03

Table 2: q_0 for different problem sizes.

In order to examine the weak scalability of the program, the problem size need to be fixed on each process. For example, if $n = 256 = 2^8$ for one process, the problem size is $N = n^2 = 2^{16}$. In order to keep the problem size fixed for more processes, we need to follow:

$$n = \sqrt{p \cdot 2^{16}} = \sqrt{p} \cdot 2^8. \tag{8}$$

Therefore, it is impossible to get exact values for many values of p. Only perfect squares yield exact results, i.e. $p = 1, 4, 9$ and so on. Thus, n is chosen as: $n_{p=1} = 256$, $n_{p=4} = 2 \cdot 256 = 512$ and $n_{p=9} = 3 \cdot 256 = 768$. The results of the execution times can be seen in table 3. The efficiency for these experiments can be seen in figure 3.

Number of Processes	Execution Time (seconds)
1	0.084654
4	0.085389
9	0.087523

Table 3: Execution Time for Different Numbers of Processes for weak scalability.

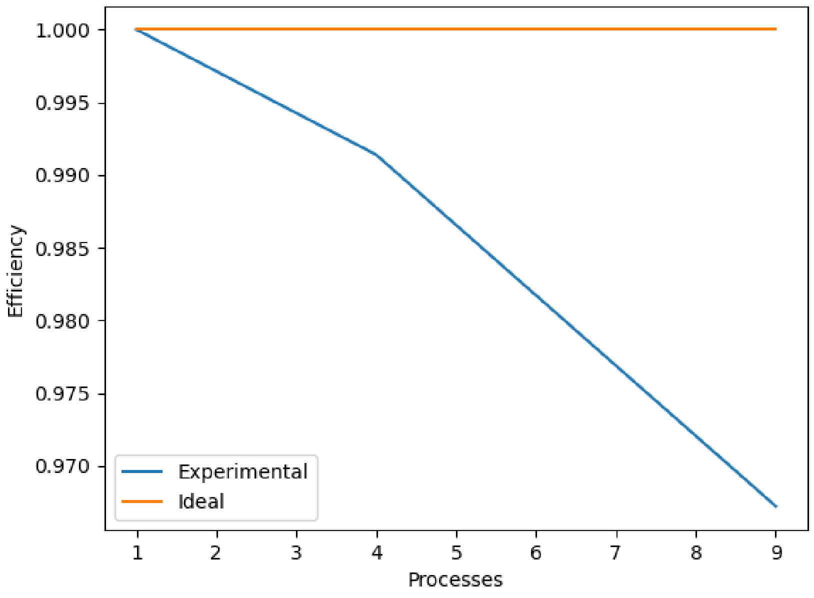

Figure 3: Efficiency plot for the weak scalability experiments.

4 Discussion

The experiments of the algorithm seem to give fairly good results. The plot in figure 2 is peculiar. For two processes, the speed up is expected. However, for 4 and 8 processes, the algorithm exhibits superlinear behaviour. This is certainly unexpected. Several timing experiments were conducted in order to examine if there was a bottle neck for fewer processes, but none were found. The strange behaviour could be because of better cache utility; if the problem size is small enough to fit in the cache for more processes, it could drastically increase the speed of the heavy calculations.
Table 2 indicates that the convergence of the algorithm decreases as the problem size increases. This is expected, since a larger system tends to need more iterations to converge.
Lastly, we look closer at the weak scalability experiments. The plot in figure 3 seems very good. The efficiency is very close to 1 throughout the experiment. This also strengthens the theory of cache utility in the strong scalability experiments, because the efficiency does not go above 1 for more processes, indicating that there is not a bottle neck for 1 and 2 processes.
During the matrix-vector multiplication, there are a lot of conditions. This is because the stencil operation is zero on the boundaries. This is solved using many ? operations during the first and last row on each process. Theoretically, performance could increase

by moving these out of the for loop, but that would make the code way more messy, and it would only affect the first and last rows which are negligible for large values of n. This could probably also have been solved by allocating $(n+2)^2$ values and setting the boundaries to zero, but it was deemed that would lead to allocating redundant values. This would be especially bad for large values of n.