Parallel solution of the 2D Poisson's equation using OpenMPI

Roni Koitermaa

August 1, 2022

1 Introduction

Parallelization of numerical algorithms allows us to speed up the execution of our programs, which increases the size of systems we can study. Physical laws often take the form of *partial differential equations* (PDEs), so in order to study physical systems numerically, we need to be able to solve these equations on a computer. Solving these equations can be computationally intensive, so we want to leverage all of the capabilities of our hardware. Using multiple processors to divide the work is a central goal in the parallelization of PDE solvers.

Poisson's equation

$$\nabla^2 f = g \tag{1}$$

is a ubiquitous PDE found in many areas of physics, e.g. in electrostatics. In many cases, we want to solve f for some g, e.g. when we want to know the electric field for some charge density distribution in the case of electrostatics. While solving f analytically can be very difficult in most cases, the equation can be easily solved numerically. This can be done by *discretizing* the equation, i.e. by dividing the solution domain into finite pieces. This way the discrete form of the equation can be solved on a computer using some numerical algorithm.

In this work, we solve Poisson's equation in the unit square using domain decomposition to parallelize the successive over-relaxation method. We are interested in the scaling behavior of the methods used, so we study different system sizes solved on different numbers of processors. Since for a larger number processors the amount of work done by each processor is smaller, we would expect the solution time to decrease.

2 Methodology

2.1 Algorithms

In 2 dimensions, we have the equation [1]

$$\frac{\partial^2}{\partial x^2} f(x,y) + \frac{\partial^2}{\partial y^2} f(x,y) = g(x,y) \tag{2}$$

defined in the unit square, which can be discretized using the *finite difference method* by dividing the domain into an $N \times N$ matrix:

$$f_{ij} = f\left(\frac{i}{N}, \frac{j}{N}\right),\ i,j \in [0,N].$$

The central difference approximation [1]

$$\frac{\partial f}{\partial x} \approx \frac{f(x+1/2N) - f(x-1/2N)}{N}$$

can be used to to obtain the derivatives of f. This gives the LHS second derivatives using three points [1]

$$\frac{\partial^2}{\partial x^2} f(x,y) \approx \frac{1}{N^2}\left(f(x+1/N,y) - 2f(x,y) + f(x-1/N,y)\right),$$
$$\frac{\partial^2}{\partial y^2} f(x,y) \approx \frac{1}{N^2}\left(f(x,y+1/N) - 2f(x,y) + f(x,y-1/N)\right),$$

so we get the equation [1]

$$f_{ij} = \frac{1}{4}\left(f_{i+1,j} + f_{i-1,j} + f_{i,j+1} + f_{i,j-1}\right) - \frac{g_{ij}}{4N^2}.$$

For Dirichlet boundary conditions defined on the edges of the square, this is an $(n-1)\times(n-1)$ system of equations, which can be solved using many methods that are used in matrix solving. *Successive over-relaxation* (SOR) is a class of methods (containing, e.g. Gauss-Seidel) which has the iteration $k+1$ [2, p. 866]

$$f_{ij}^{(k+1)} = (1-\gamma)f_{ij}^{(k)} + \frac{\gamma}{4}\left(f_{i-1,j}^{(k+1)} + f_{i,j-1}^{(k+1)} + f_{i+1,j}^{(k)} + f_{i,j+1}^{(k)}\right), \tag{3}$$

for an *over-relaxation parameter* $\gamma \in]1,2[$, which determines how strongly solution values are influenced by their neighbours. Compared to the Jacobi over-relaxation method (JOR), SOR uses updated $k+1$ values as soon as they are available, which means values in a matrix can be overwritten, leading to better space complexity. Use of SOR is desirable because it results in faster convergence than JOR for optimal γ.

2.2 Parallelization

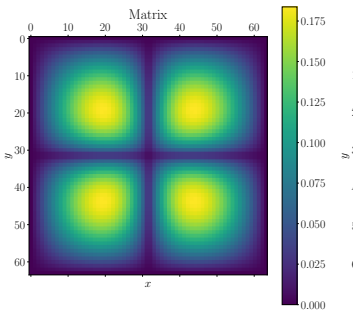

Figure 1: Domain split into $N_p = 4$ processes.

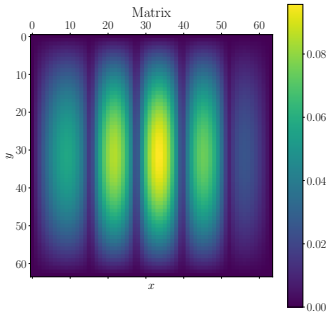

Figure 2: Domain split into $N_p = 5$ processes.

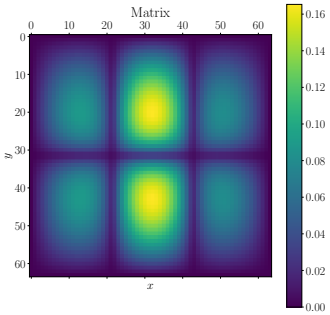

Figure 3: Domain split into $N_p = 6$ processes.

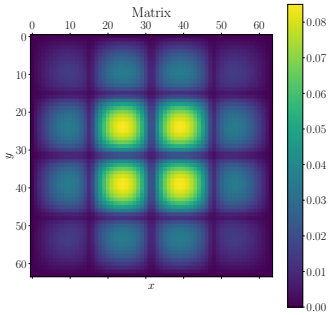

Figure 4: Domain split into $N_p = 16$ processes.

Using domain decomposition, we can split a boundary value problem into many smaller boundary value problems. A simple way to do this in a physical system is to divide the problem domain spatially among processors, so that each processor takes care of a localized part of the system. Each sub-problem can be solved in parallel, while making sure each has the appropriate boundary conditions. In order to make sure we solve the larger problem correctly, there needs to be inter-process communication.

In figures 1, 2, 3, 4 we demonstrate the spatial splitting of a solution between different numbers of processors. In these figures, there is no inter-process communication, so each part of the system is solved entirely separately. When we assemble the final results together, the solution to the larger problem is not correct. Instead, in order to get the correct solution, each process needs to get its boundary conditions from the neighbouring processes.

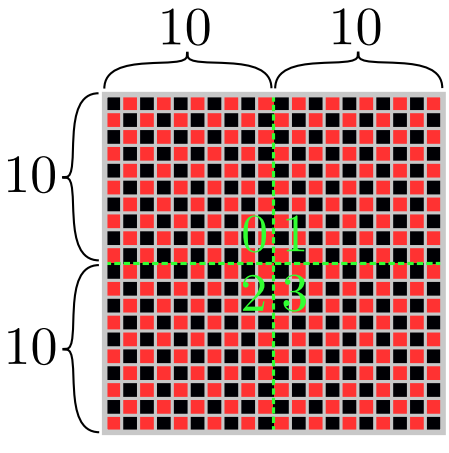

Figure 5: Lattice division for even case.

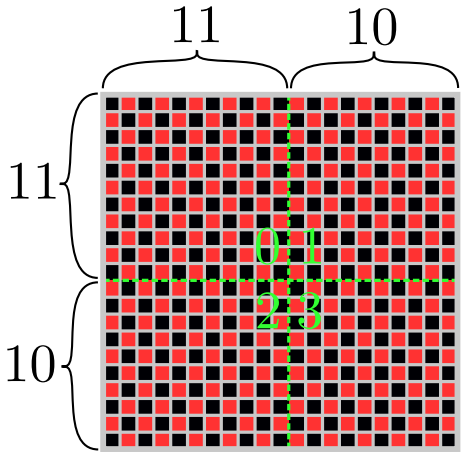

Figure 6: Lattice division for odd case.

To parallelize the SOR method, we need to make sure that $k+1$ iteration values in equation 3 are always available when needed. In the serial case this is not an issue, but when parallelizing the method, problems appear at the edges of each process domain. At the edges, it is not guaranteed if the values on the neighbouring processes

are $k+1$ or k, since they are currently worked on by the neighbouring process. This problem can be solved by dividing the grid into alternating sublattices using the "red-black algorithm" [3]. The entire domain is colored using "black" and "red" points, as shown in figure 5. To solve the "black" sublattice, we only need the "red" neighbour values. This means that we do not need to know the "black" neighbour values when solving the sublattice. To solve the whole lattice, we first solve one of the lattices (red or black), get neighbouring values from other processes, and then solve the other lattice. This process ensures that the $k+1$ iteration values are available each time the system is solved.

When dividing the lattices, we also need to take into account how the processes are divided. Essentially, we divide the process grid into red and black domains depending on where the processes are located. In figures 5 and 6, the process boundaries are shown in green. For a lattice with process domains of even size ($N_0 = 10$), the first element f_{00} belongs to the black sublattice for all processes. However, if process 0 has size $N_0 = 11$, process 1 can't start with the black sublattice since we would have black and red points lining up. This means that we have solve the black sublattice for process 0 and the red sublattice for process 1 to ensure the algorithm works. In general, if the sum of the i_0 and j_0 offsets of the process is even, we solve the black sublattice and for the odd case we solve the red sublattice.

Since we assume the lattice to be square in equation 3, we have to make sure that the processes evenly divide the grid. In figures 2 and 3 the process domains are not squares, while in figures 1 and 4 they are. Using rectangular grids negatively impacts the convergence of the algorithm, so we want to divide the grid as evenly as possible. Safe choices for processor number are the squares: $N_p = N_{p,0}N_{p,1} = 1, 4, 9, 16, 25, 36, 49, 64, \ldots$ Ideally we would also want the system size N to be divisible by by $N_{p,0}$ so that all process domains are square and none of them have to be rectangular. For example, in figure 6 process 1 would be 11×10.

3 Documentation

3.1 Description of code

Code was written in C solve Poisson's equation in parallel in the unit square. We use OpenMPI for parallellization of the SOR method. The solver consists of the files `solver.c` and `solver.h`, which contain the MPI part of the code. The files `poisson.c` and `poisson.h` contain the solver for Poisson's equation, which uses SOR. The Poisson solver operates within individual processes. Utilities, e.g. for manipulating matrices, are contained in file `util.h`.

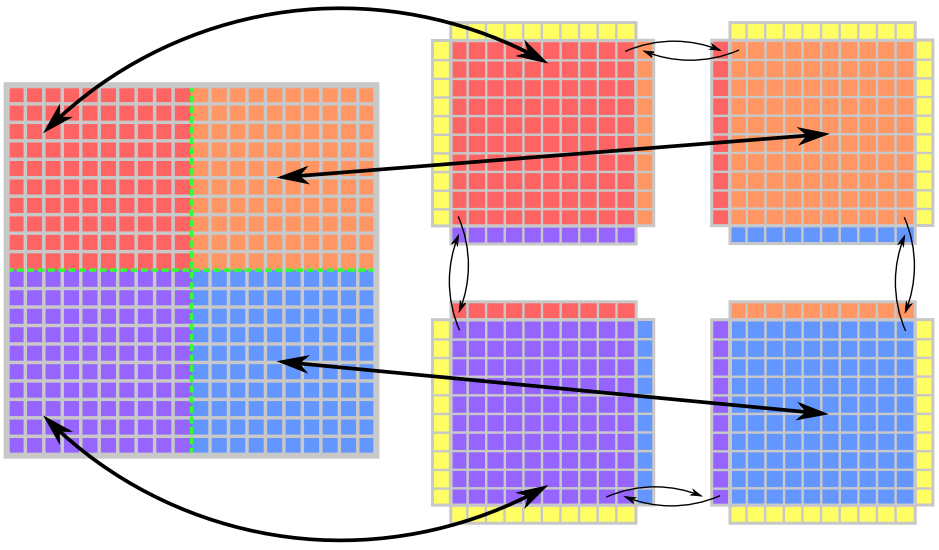

Figure 7: Communication between process domains.

The code starts by taking in command line arguments from process 0 and distributing these among all processes in fuction `initSolver`. The command line arguments specify the input/output files, system size N, over-relaxation parameter γ and convergence criterion C. Process 0 reads the input file, which is an $N \times N$ matrix that specifies the RHS g. The edges of this matrix are the Dirichlet boundary conditions for f.

Depending on whether the program is run in serial or parallel mode, the function `decompose` is run, which performs domain decomposition for the system. This function makes a Cartesian 2D $N_p = N_{p,0} \times N_{p,1}$ grid using

the processors (`MPI_Dims_create`), making the grid as square as possible ($N_{p,0} = \lfloor\sqrt{N_p}\rfloor$). Boundaries of each process domain are set to fixed (not periodic). A Cartesian communicator is created(`MPI_Cart_create`), and all inter-process communication is done in this communicator. The Cartesian id of the root process is shared among all processes using `MPI_Bcast`. We get the coordinates of each process domain in this grid using `MPI_Cart_coords`, and calculate the sizes of each by dividing $N \times N$ into the $N_{p,0} \times N_{p,1}$ process grid. In the case that system size N is not divisible into the processes equally, we place the remaining points into the left/up corner processes (with coordinates $x_{p,\mathrm{id}} = y_{p,\mathrm{id}} = 0$). Each process domain is of size $n \times m$, so they are not restricted to squares. Using the Cartesian communicator, each process gets its x and y neighbour process ids (`MPI_Cart_shift`). The program prints the process grid layout in the log file under "process topology".

Next, we determine the boundary conditions for each process. Some processes are located on the edge of the entire domain, while some are on the inside, surrounded on all sides by other processes. The Dirichlet boundary conditions need to be changed depending on the location of the process. In figure 7, we have visualized how process domains are split (among 4 processes in this example) and how boundary conditions are determined. On the left, we have the entire problem domain, while on the right we have the decomposed process domains. In the figure, each process domain is colored uniquely, with boundary conditions colored with the matching domain. On the right, the boundary conditions in the middle are determined by the neighbouring processes. The outside boundaries are determined from the exterior boundary conditions defined in the input file, colored yellow. Arrows denote how processes communicate between each other. This communication occurs by passing rows and columns (of size n and m) to the boundaries of neighbouring processes, which determine the Dirichlet boundary conditions for each process domain. To make this communication easier, we define MPI datatypes for rows and columns in these matrices, which makes sending and receiving them simpler.

Since we have arbitrarily sized $n \times m$ process domains, each process needs to know the coordinates and sizes of other processes. Each process sends its coordinates $(x_{p,\mathrm{id}}, y_{p,\mathrm{id}})$ and size (n, m) to the root process using `MPI_Gather`. The root process gathers the values into arrays of size N_p (indexed by Cartesian id) and calculates the offsets of each process domain on the entire domain by using the sizes and coordinates. Knowing these offsets is required when we need to communicate process matrix values. The root process shares all of these arrays with the other processes using `MPI_Bcast`.

In order to set up the solvers for each process domain, we need to divide the full f and g into the process domains. Figure 7 illustrates how the domain is divided, and this process is done for both f (containing boundary conditions) and g. The root process has the full f and g matrices, so first it sets its own $n \times m$ submatrices from the full ones to its own Poisson solver. Then it iterates through all processes in the Cartesian communicator and gets the submatrices for each process depending on their sizes and offsets. The root process sends the matrices to each process using `MPI_Send`, and every other process receives these using `MPI_Recv`. At the end, each process has its own part of the full system.

Solving Poisson's equation using succesive over-relaxation proceeds by updating the values in the matrix f using equation 3 in a loop. The solver keeps updating f as long as the system has a *residual* δ that is less than the convergence criterion C. When $\delta \leq C$, we consider the solution to have converged and stop iterating. The Poisson solver in `poisson.c` has two types of SOR iteration functions: one for the serial version (`solveSOR`) and two for the parallel one (`solveSORR` for red sublattice and `solveSORB` for black sublattice). In the serial version, the entire lattice is solved at once using the Gauss-Seidel iteration for SOR as shown in equation 3 (function `iterateGS`). For the parallel version, the red-black algorithm from section 2 is used. The black sublattice function starts at f_{00} and solves for every other point onwards in an alternating fashion as shown in figure 5. The red sublattice function starts at f_{01} and proceeds the same way. When solving an SOR iteration in parallel, the process decides whether to start with the red or black sublattice first as discussed in section 2. The SOR update functions return the residual for the iteration, which is calculated as the difference between the old values f'_{ij} and the new values f_{ij} as

$$\delta = \sum_{i,j} |f_{ij} - f'_{ij}|.$$

Since the red-black algorithm divides the iteration in two parts, to calculate the total residual in the parallel version, we add the red and black residuals together: $\delta = \delta_r + \delta_b$.

Inter-process communication needs to happen after each SOR iteration. The update proceeds as follows (assuming "even" case):

1. Solve black sublattice to get $k + 1$ values for neighbours of red sublattice
2. Communicate black sublattice values to neighbouring processes
3. Solve red sublattice to get $k + 1$ values for neighbours of black sublattice

4. Communicate red sublattice values to neighbouring processes

In the code, steps 2. and 4. are identical, and simply occurs by communicating the entire boundary values to the neighbour processes. Communication is done in function `communicate`, which exchanges edge values between all processes as shown using arrows in figure 7. Since we have defined column and row datatypes, the entire edges can be communicated using `MPI_Recv` and `MPI_Send`. First, the x boundary values are communicated, with even coordinate processes receiving into their BC values and and odd coordinate processes sending from their solution values. After this, the two processes switch to make sure both processes have updated BC values. In the x direction, the first receiving process gets values from its left and the sending process sends to its right. After a two-way update the direction is switched. The y edge communication occurs similarly afterwards. Since residuals are computed only for process domains, we need to add the residuals from all processes to get the total. This is done using `MPI_Reduce` with `MPI_SUM` to calculate the total in the root process. The root process shares the total with other processes using `MPI_Bcast` to make sure they all stop at the same time.

When the convergence criterion is satisfied, the iteration is stopped and the solution f is written to a file by the root process. The root process only has its own submatrix, so the solutions from other processes need to be communicated back to the root process. This is done similarly as when distributing f from the root process to the other processes, except in reverse. The root process receives the submatrices from every other process (communicated using `MPI_Send`/`MPI_Recv`) and extracts the solutions from them, excluding the BCs. These submatrices are set to the final $N \times N$ matrix, which is printed to the output file.

The program prints various information about the process configuration and solution process to stdout, which can be directed to files. The residual is printed at regular intervals, and at the end, the total number of iterations is shown along with the wall time. If the solution didn't converge within a maximum number of iterations or the residual increased beyond its maximum (solution diverged), the log file indicates this.

3.2 Instructions for using the code

The code in directory `src` can be compiled using GCC with a makefile in the directory above. To run the program, the shell script `run.sh` can be used to perform the runs shown in the following section. This script generates the input files by running `run/input.sh` (awk), finds optimum values for γ by running `run/rungamma.sh` and tests the scalability by running `run/run.sh`. Plotting can be done using the script `./plot.sh`, which plots γ curves using `plotgamma.py` and scaling curves using `plotscaling.py`. Matrices from `.dat` files can be plotted using `./matplot.py <out.dat>`.

The (serial) code itself is run using:

```
./parps <infile> <outfile> <n> <gamma> <crit>
```

The arguments specify the input file for g, the output file for f, system size N for the $N \times N$ matrix, over-relaxation parameter γ and convergence criterion C. On PCs the parallel version can be run using:

```
mpirun -np <number of processes> ./parps <infile> <outfile> <n> <gamma> <crit>
```

On Turso this is:

```
srun --mpi=pmix ./parps <infile> <outfile> <n> <gamma> <crit>
```

4 Benchmarking

Development and initial testing was done on a 4-core Intel i7-4770 PC with shared memory. Software on the PC was GCC 12.1 and OpenMPI 4.1. Scalability testing was done on the University of Helsinki Turso cluster, specifically Ukko2 with 32-core Intel Xeon E5540. On Turso we chose GCC 8.3 and OpenMPI 3.1. Runs on Turso were done both with 1 and 2 Ukko2 nodes, with InfiniBand for MPI message passing.

To test that the serial Poisson's equation solver is working correctly, we want to test it with a g for which an analytical solution is known. In the 2D unit square, one simple solution is obtained for [5]

$$g_a = g(x, y) = -2\pi^2 \sin(\pi x) \sin(\pi y),$$

which gives the solution

$$f(x, y) = \sin(\pi x) \sin(\pi y).$$

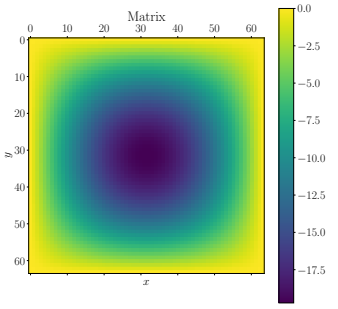

Figure 8: RHS g_a with $N = 64$.

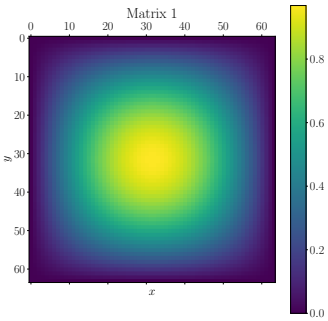

Figure 9: Solution for g_a with $N_p = 1$ processes.

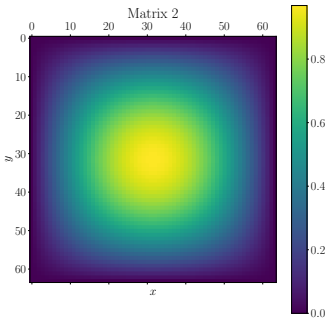

Figure 10: Solution for g_a and $N_p = 4$ processes.

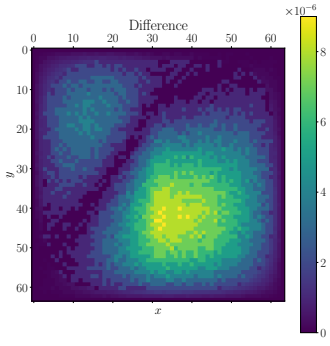

Figure 11: Difference between serial and parallel versions.

We generate g_a with size $N = 64$ using the shell/awk script `run/input.sh`. The RHS g is plotted in figure 8. This is solved in serial mode on the PC with $\gamma = 1.5$ and $C = 0.001$ to produce the solution f in figure 9. We can see that this does have the shape of the function we were expecting, with good accuracy. Next, we test the serial version of the program, doing domain decomposition with $N_p = 4$ processes. This also results in the correct solution (figure 10), so we can conclude that inter-process communication is working correctly (compare figures 1, 2, 3 and 4). In figure 11 we have plotted the differences between the serial and parallel versions, which are of the order of magnitude 10^{-6}, much less than the convergence criterion.

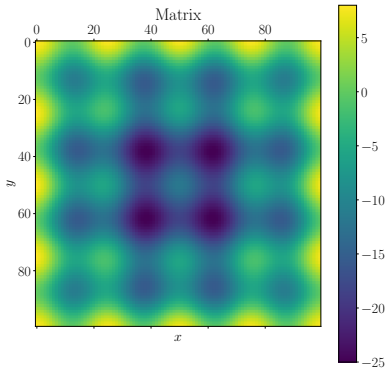

Figure 12: RHS g with sinusoidal edge and $N = 100$.

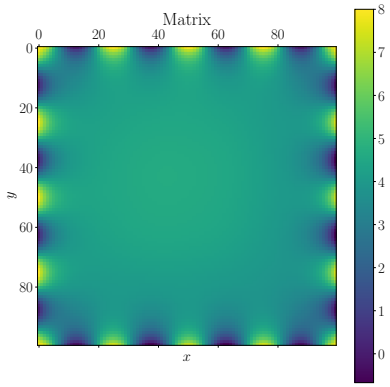

Figure 13: Solution with $N_p = 4$.

In the previous cases the boundary condition is 0, so we also want to see if the solution satisfies the boundary condition in the parallel version. We generate a g with some sinusoids on the edges, which are the Dirichlet boundary conditions the solution should satisfy. This time the system size is $N = 100$ and the run is performed with $N_p = 4$ processes. In figure 13 we have the solution for figure 12, which shows matching boundary conditions.

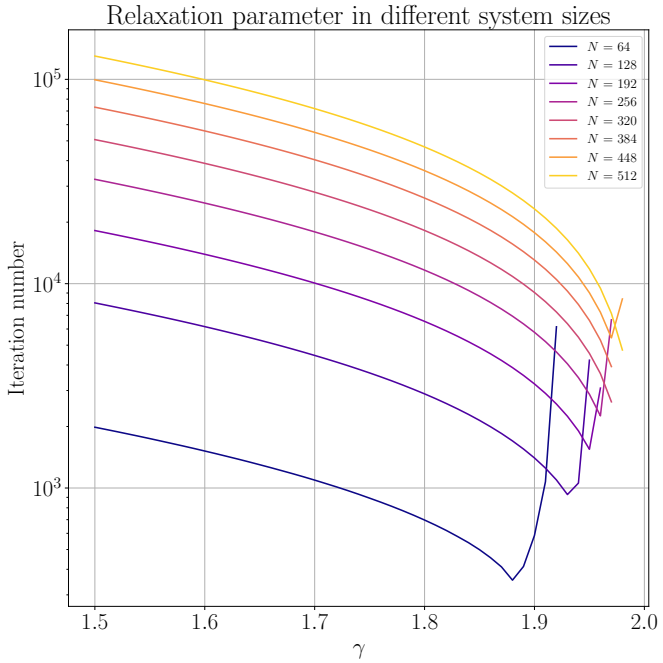

Figure 14: Number of iterations at convergence as a function of γ for various system sizes.

In order to achieve optimal convergence, we need to find the value of the over-relaxation parameter γ, which depends on the system size. This can be done using the shell script `run/findgamma.sh`, which steps γ in the range $[\gamma_0, \gamma_1]$ to determine the number of iterations it takes to reach convergence. For the optimal γ value, we would see fast convergence of the solution, with few iterations required to reach the set criterion.

To find the optimal γ values for many system sizes, we run the script `run/rungamma.sh` on Turso with $N_p = 16$ processes (1 Ukko2 node). In figure 14, the iteration number is plotted as a function of γ in the range [1.5, 1.99] (with 50 steps) with different system sizes from $N = 64$ to $N = 512$. The RHS is g_a with different N. We can see that the minima for different system sizes are different, with the larger systems tending to have γ values closer to 2. These results are similar to what we see in the literature [4].

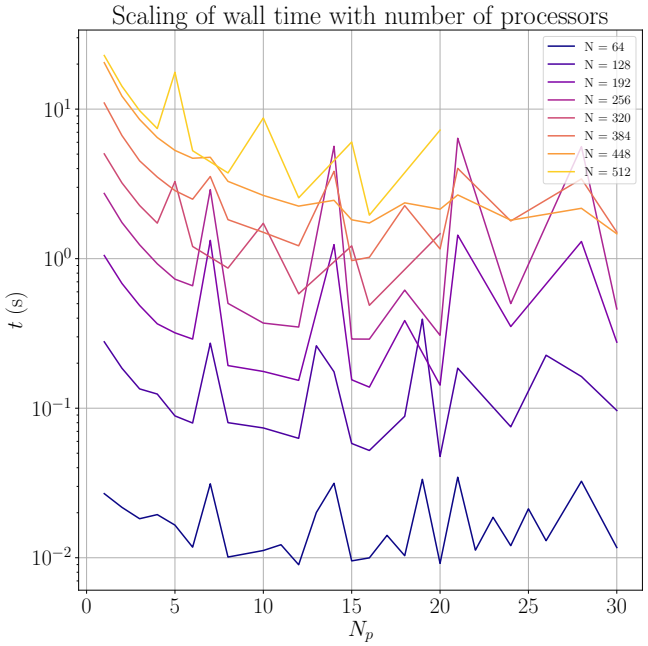

Figure 15: Scaling of solution time, $N_p = 1, 2, 3, \ldots, 32$, run on a single Ukko2 node.

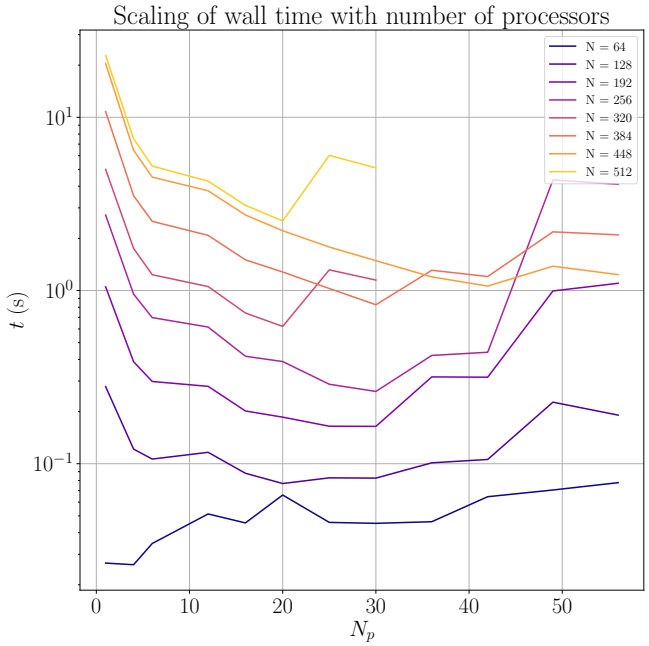

Figure 16: Scaling of solution time, $N_p \in N_s$, run on 2 Ukko2 nodes.

In order to determine how effective the parallelization of our code is, we want to study the scaling behavior of the solution time as a function of the number of processes. At $N_p > 1$ there should be some benefits to using more processors, though there may be overhead when using MPI, so we would not expect the benefits to continue for ever larger numbers of processors.

The choice of processor numbers is important for convergence of the solution. We can't use processor numbers that result in highly non-square decompositions, since our assumption is that each domain is roughly square. We run `run/run.sh` on Turso with 1 Ukko2 node and 32 reserved CPUs. In figure 15 we use all processor numbers in the range $N_p \in [0, 32]$, which contains many prime numbers. In the figure, we can see spikes for certain processor numbers, such as $5, 7, 10\ (2 \times 5), 14\ (2 \times 7), \ldots$ Diverged solutions are not shown in this plot, and for large prime numbers the convergence gets worse and worse. Because of these results, for the next simulations, we will pick numbers such that $N_{p,0}$ and $N_{p,1}$ can be at most 1 apart, i.e. $N_s = \{1, 2, 4, 6, 9, 12, 16, \ldots\}$. This gives a reasonable number of data points while being roughly square.

Next, we run on Turso with 2 Ukko2 nodes to see at what point increasing the processor number starts giving diminishing returns. Now message passing can require inter-node communication, which can be slow. The results are plotted in figure 16 up to 64 processors. We can see that for most system sizes, the solution time slows down around 16 processors and starts increasing beyond 32 processors. At this point each domain has only a few points, so most of the time will be spent on inter-process communication.

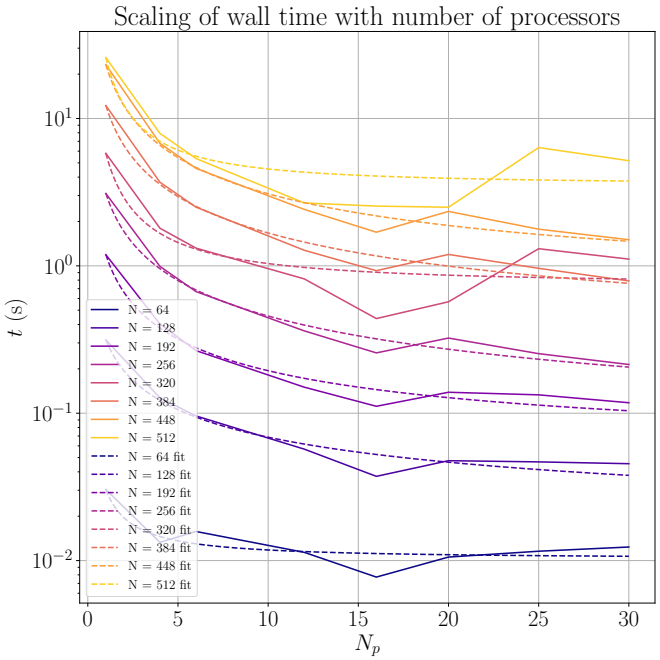

Figure 17: Scaling of solution time, $N_p \in N_s$, run 4 times on a single Ukko2 node.

Finally, we run on a single Ukko2 node up to 32 processors to determine the scaling curves for different system sizes. We would expect parallellization to make more of a difference for larger systems. In figure 17, we have the averaged solution times over 4 runs, with fitted curves for $t \propto N_p^\alpha$. The average fitted value for the power law is $\alpha = -0.9611$, so we can conclude this is a roughly $1/N_p$ dependence. For the large systems the time saving is greater than for small systems.

5 Conclusions

We developed a code to solve Poisson's equation in the unit square in parallel using successive over-relaxation. Domain decomposition was used to do the parallelization with MPI. Some of the challenges in parallelizing the algorithms were discussed and methods for overcoming these were presented. A detailed description of the code was given to show how the discussed methods were implemented in practice.

Our code was tested using an analytically known solution and found to have good accuracy. The parallel version of the code was found to give the same solutions as the serial one, and it was also found to be faster than the serial

one. We determined optimal over-relaxation parameter values for different system sizes, which were used to study the scaling behavior of the parallel code. It was found that the solver works best for square processor numbers, and that there is a limit to how many processors will make the solution time faster. Over all system sizes, the solution time was found to scale with the processor number as roughly $1/N_p$.

Ideally we would like to utilize any number of processors to do the domain decomposition. The MPI part of our code is designed to work with rectangular domains, but the solution algorithms are not. Adding support for rectangular domains would require changes to how we solve the linear system of equations. A further improvement could be also to add support for arbitrarily-shaped domains. The MPI implementation of the code is likely not optimal, and improvements could be made especially in inter-process communication. This is likely a major bottleneck as the number of processes increases, so even small improvements could make a big difference.

References

[1] Tools of high performance computing 2022 - Final project. University of Helsinki, 2022.

[2] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. Numerical Recipes in C: The Art of Scientific Computing. Cambridge University Press, Cambridge, England, 2nd edition, 2002.

[3] Dan Wallin, Henrik Löf, Erik Hagersten and Sverker Holmgren. Multigrid and Gauss-Seidel Smoothers Revisited: Parallelization on Chip Multiprocessors. Proceedings of the 20th annual international conference on Supercomputing, 2006.

[4] Shiming Yang and Matthias K. Gobbert. The optimal relaxation parameter for the SOR method applied to the Poisson equation in any space dimensions. Applied Mathematics Letters, 22(3):325-331, 2009.

[5] Abdon Atangana and Adem Kılıçman. Analytical Solutions of Boundary Values Problem of 2D and 3D Poisson and Biharmonic Equations by Homotopy Decomposition Method. Abstract and Applied Analysis, 2013.