Uppsala University

Parallel and Distributed Programming

1TD070

Individual project - report

Parallel implementation of the Conjugate Gradient method with stencil-based matrix-vector multiplication

Author:
Csongor Horváth

June 27, 2023

Introduction

In this project I made a parallel implementation of the Conjugate Gradient method for a special application case, where the matrix of the linear equation comes from a 2D 5 point stencil problem.

First I would like to provide a theoretical background about the Conjugate Gradient (CG in the followings) method.
The CG method is an iterative algorithm for the solution of linear equations, where the linear equation has a symmetric positive definite matrix.
There are deterministic methods, the most common being the Gaussian elimination for solving linear equation, but it is scaling badly for large matrices. So it can't be used for a lot of real world application, where we need to solve linear equations with large matrices.
In this cases the different iterative methods can be useful for solving linear equations. The choice of the the iterative method can be made based on the given matrix properties and the need of the convergence rate as different methods guarantee convergence for different type of matrices with different convergence rate.

In our case we have some further advantage using the iterative CG method instead of a direct method. First there is no need to factor the coefficient matrix. Secondly as this method only needs the computation of Ax for arbitrary x vector, so in our application we don't have to store A, as we will do this calculation with a stencil application as the matrix is coming from there. But in a lot of other cases there are also a possibility to store A in small space.

I choose to work on this project, as I am interested in areas where the CG method in general can be useful. Some of it's real world application include image reconstruction [Kno07], facial recognition [HS13], solving least square problem [JL17], solving PDE's [CGO76] and lastly it can be used for different optimization processes including applications in machine learning [YLW09; JDG91].

Algorithm 1 The CG algorithm for solving $Ax = b$

Input: A, b, $x_0 = 0$ initial guess, ϵ sensitivity
Output: $x := x_i$

Init $r_0 = b - Ax_0,\ p_0 = r_0,\ i = 0$
do
$\alpha_i = \frac{r_i^T r_i}{p_i^T A p_i}$
$x_{i+1} = x_i + \alpha_i p_i$
$r_{i+1} = r_i - \alpha_i A p_i$
$\beta_i = \frac{r_{i+1}^T r_{i+1}}{r_i^T r_i}$
$p_{i+1} = r_{i+1} + \beta_i p_k$
$i++$
while $\|r_i\| > \epsilon$

Mathematical background on the CG method

Now I will give an informal mathematical introduction to the general CG method based on [Igo09], so using different notation as in the assignment. Note that my goal here is not to be precise mathematically, but to give an overview what is the motivation behind the iterative method and why it is equivalent with solving the linear equation $Ax = b$.

The first step is to reformulate the problem to $min_x f(x) = \frac{1}{2}x^T Ax - b^T x$. The reason for this is that if we look at the gradient of f we get $\nabla f(x) = Ax - b = 0$ at the place of a minimum. So each solution of the minimization problem is a solution of the linear equation. And we know that there exists a unique minimizer as the hessian of the second derivative $\nabla^2 f(x) = A$ is a symmetric positive-definite matrix.

As each iterative method the CG is starts with a x_0 starting value and at each step until a convergence is detected we do the following steps $x_{i+1} = x_i + \alpha_i p_i$. At the CG method we want the p_i vectors to be special in a way that they are conjugate of each other respect to the matrix A, so $p_i A p_j = 0 \quad i \neq j$. Otherwise it is similar to the Gradient method.
The conjugate property means the following. If we search the optimum as a linear combination of p_i with α_i weights, then at the minimum we get

$\alpha_i = \frac{b^T p_i}{p_i^T A p_i}$. So we get a step size. Note that if given direction p_i, then this value correspond to the result of the exact line search.

Now we need the direction. At step i let the residual be $r_i = b - Ax_i$. Now as we saw it r_i is the negative gradient of f at x_i. So the gradient method would make a step in this direction using step size from a line search. But now we want to have conjugate directions. To this end we can use the Gram-Schmidt orthonormalization to get direction $p_i = r_i - \sum_{j=0}^{i-1} \frac{p_i^T A r_i}{p_i^T A p_i} p_i$.

Now we have motivation for the step direction p_i, and the step size α_i.
Using this motivation we can come up with Algorithm 1, the CG method. It can be shown that the direction vectors of the method $\{p_i\}$ are really conjugate to each other respect to A. Furthermore the CG method has a liner convergence and using exact arithmetic it would converge in at most n steps.

With this I end the introduction to the theory behind the CG method.

Problem description

Algorithm 2 The CG algorithm for solving $Ax = b$

Input: A, b
Output: $x := u$

1: Init $u = 0, g = -b, d = b$
2: $q_0 = g^T g$
3: **while** convergence not reached ($\|g\| < \epsilon$) **do**
4: $\quad q = Ad$
5: $\quad \tau = \frac{q_0}{d^T q}$
6: $\quad u = u + \tau d$
7: $\quad g = g + \tau q$
8: $\quad q_1 = g^T g$
9: $\quad \beta = \frac{q_1}{g_0}$
10: $\quad d = -g + \beta d$
11: $\quad q_0 = q_1$
12: **end while**

From this point let's use the notation given in the assignment description. The algorithm with this notation can be seen in algorithm 2. Note that here the different symbols have the following dimensions, where n is the length of one side of the $2D$ square mesh and $N = n^2$: $b, u, g, d \in \mathbb{R}^N$ vectors, $A \in \mathbb{R}^{N \times N}$ symmetric and positive-definite matrix (it comes from a stencil application) and everything else is scalar.

Now the tasks exact problem setting is that we have a two-dimensional mesh, with coordinates $x_i = ih, y_j = jh,\ i, j = 0, 1, 2, \ldots n + 1, h = \frac{1}{n+1}$ and the boundary is fixed to be 0, therefore the important mesh points are the one where $i, j = 1, 2, \ldots n$, which is $N = n^2$ mesh points. And the vectors b, u are associated with these mesh points, but we can't forget the boundary while working on the application.

The b solution vector is defined as the following $b = \{b_{ij}\}$, $b_{ij} = 2h^2(x_i(1 - x_i) + y_i(1 - y_i)),\ i, j = 1, 2, \ldots n$ and 0 at the boundary in accordance with the result vector, which won't be stored during the algorithm.

$(j+1)$		-1	
(j)	-1	4	-1
$(j-1)$		-1	
	$(i-1)$	(i)	$(i+1)$

Table 1: 5 point stencil defining A.

Furthermore we know that the matrix A comes from the above stencil application. This stencil application correspond to the first order approximation of the Laplacian at given point as $\nabla^2 f = \sum_{i=1}^{dim} \frac{\partial^2 f}{\partial x_i^2}$

We know the Taylor expansion of $\frac{\partial^2 f}{\partial x^2}$ and $\frac{\partial^2 f}{\partial y^2}$, which is

$$\frac{\partial^2 f}{\partial x^2} = \frac{f(x+\Delta x, y) + f(x-\Delta x, y) - 2f(x,y)}{\Delta x^2} - 2\frac{f^{(4)}(x,y)}{4!}\Delta x^2 + \cdots$$
$$\frac{\partial^2 f}{\partial y^2} = \frac{f(x, y+\Delta y) + f(x, y-\Delta y) - 2f(x,y)}{\Delta y^2} - 2\frac{f^{(4)}(x,y)}{4!}\Delta y^2 + \cdots$$

Now using this and choosing $\delta x = \delta y = h$ we get the following

$$\begin{aligned}
\nabla^2 f &= \frac{\partial^2 f}{\partial x^2} + \frac{\partial^2 f}{\partial y^2} \\
&= \frac{f(x+h,y) + f(x-h,y) + f(x,y+h) + f(x,y-h) - 4f(x,y)}{h^2} - 4\frac{f^{(4)}(x,y)}{4!}h^2 + \cdots \\
&= \frac{f(x+h,y) + f(x-h,y) + f(x,y+h) + f(x,y-h) - 4f(x,y)}{h^2} + O\left(h^2\right)
\end{aligned}$$

So we obtained the the following approximation $\mathcal{O}(h^2)$

$$\nabla^2 f(x,y) \approx \frac{f(x-h,y) + f(x+h,y) + f(x,y-h) + f(x,y+h) - 4f(x,y)}{h^2}$$

So $A \approx -h^2 \nabla^2 f(x,y)$. Therefore the program gives a numerical approximation for the solution of the following 2 dimensional differential equation for f with homogeneous Dirichlet boundary condition:

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

$$f(x, y) = 0, \quad \text{on the boundary, so if } x \text{ or } y \text{ is 0 or 1}$$

This means that the values of the resulting $u = \{u_{ij}\}$ correspond to the values of $f(x, y)$ in the following way $f(ih, jh) \approx u_{ij}$, where $f(x, y)$ is the solution of the above differential equation.

So now we know the basic theory behind the CG method and the motivation for our exact problem setting. Now before start explaining my implementation I want to mention some relevant information and requirement from the assignment, which either makes my task easier or limit me in the implementation.

First as the task said I assumed that the used number of processes p is a square number, even though my code works for arbitrary p. This way I can create a $\sqrt{p} \times \sqrt{p}$ logical architecture. And the vectors should be fully distributed between the processes, so only approximately N/p long parts of the vectors should be stored locally in each process.
Secondly only one parameter n is needed for this application as the stencil and the corresponding right hand side result vector are given, but it should work for arbitrary n. Also as it is required I will implement a stencil application instead of a matrix multiplication as this require much less memory and probably result in faster run times.
Finally instead of iterate until a certain value in precision is reached the task ask to make exactly 200 iteration and the norm $\sqrt{g^T g}$, so the precision which is reached in the simulation should be output.

The task was to implement this special application in a parallel way using C and MPI and make scaling measurements for it. This will be discussed in the following sections.

Implementation

Analysis of the pseudo code

First let's look at the pseudo code, which should be implemented. Sometimes I will refer to different part of the pseudo code as line numbers (corresponding to Algorithm 2). This is also used in the comments in the code.
This section conclude only general idea of the algorithm and how costly different parts are, details about my implementation will come later.
Using MPI we need to have a virtual structure of the processes with each process covering a part of the 2D mesh. Once we set up this virtual structure and make them associated with the mesh grid, then the initialization can be made locally, as the given b only depends on the position on the mesh.

Now the algorithm consist of 4 different type of task. One is the scalar product (line 2, 5, 8). This requires global communication between all the processes. As the dot product of a given vector $v = \{v_i\}_{i=1}^n$ is calculated as $\sum_{i=1}^n v_i^2$, so an efficient implementation of the dot product computes the local part of the sum, so which comes from the locally stored part of the vector and only communicates these partial sums to compute the final value of the dot product. Here the local sum is fast to compute due to vectorization. But the communication can be time consuming as all process need to exchange data, so they have to wait to each other and be at the communication point at the same time.
Then the second operation, is the scalar multiplication/division and assigning values, which basically takes no time compared to the other parts.
The third different task is a vector update, which is done by updating one vector to a linear combination of the given vector and one other. These task can be done locally if the vectors have the same part of them stored in the same process which will be used as it makes sense since there are no task which would benefit from using other storage. This task is fast again, because vectorization can be used.
And lastly the matrix vector multiplication which will be done using a 2D stencil application as the task said. Here there are need of communication to exchange the border values between processes to calculate the border values in the stencil application. And there are considerable amount of local computation.

So at first glance the crucial part of the algorithms are the following: stencil application and communication for the dot product.

Data dependency in the algorithm

Now let's discuss the data dependencies of the algorithm and the problems it may cause. Shortly we have almost no freedom to change the order of the lines/tasks in the given pseudo code. A bit more detail. Inside the loop the following should hold. Line 4 computed earlier then the computation of line 5 starts. Line 5 should be computed earlier then line 6,7. Line 7 should be computed earlier then line 8. Line 8 should be computed earlier then line 9. Line 7 and 9 should be computed earlier then line 10. Line 11 should be computed after line 10 and before the next cycle line 5 computation.
So we have very little freedom in changing the order of the tasks. It means that we can't merge communications together, so we will have 3 different communication, 2 of which is global communication and between there are relatively few tasks to do. Also the advantage of the non-blocking communication can't be used for the dot product necessary for computing τ, as the task before need to be finished to start computing it, and it is used right after. For the other two communication there is a possibility to use non-blocking communication and do some tasks while the communication can happen in the background.
So the data dependencies cause us to have a relatively fixed order of the execution of the commands. It makes us have 3 different communication per loop cycle or at least there is no easy workaround to avoiding it.

Final implementation

Now let's talk about my original and final implementation and later I will address my efforts for optimization. I stick to this original version as this is the simplest and I was unable to obtain measurable improvements in performance, but for different application the changes mentioned in the optimization can be considered. For the code I tried to use functions for every steps, where it made sense and in the main part of the code I mostly made function calls. I also commented which part of the algorithm each function calculates. I would like to note here that the time measurements starts after the initialization of the starting data, so at line 2. And ends after the 200 iteration and the computation of the final dot product, which shall be plotted. And

optimization efforts were only made for the measured part of the code.
Also this code works for arbitrary p value, even though in a lot of place I only talk about square p values and I did the experiments only with those numbers.

Used data distribution and data storage

I used a virtual 2D non-periodical Cartesian square grid with $\sqrt{p}$ process in one side (assuming p is square). I used `MPI_Cart_Create` for this instead of manually implementing it, as this is easier and for the dimensions I used `MPI_Dims_create` to have a well balanced grid in cases p is not a square number. I allowed the create to reorder the processes as this doesn't matter for the other part of the code and it may allowed MPI to do optimization.
I pair to each process a part of the 2D mesh. Each process get an approximately $\frac{n}{\sqrt{p}} \times \frac{n}{\sqrt{p}}$ size rectangular part of the mesh, so that the whole mesh is distributed and the sides of the local rectangles didn't differ more then 1. At each process I calculate the local vertical and horizontal length of the rectangular part of the mesh which will be stored in a 1D vector locally and I also calculated the bottom left cells coordinates in the global mesh (the smallest index in both coordinate).
Now that the position are known locally, I allocate memory for the vectors. For each local vector (g, d, u, q) I used a 1D double array representing the *horizontal size* $\times$ *vertical size* rectangular variable part. I additionally used *horizontal size* and *vertical size* length double arrays to store the borders of the local rectangular mesh. So these used for the stencil computation.

After this the initial data was computed locally as it is given in a deterministic way, which can be computed knowing the vertical and horizontal global index corresponding to a given array element. During the above set up I calculated every value needed for the knowledge of global position.

The next part will be the implementation of the algorithm. I basically tried to make it as straight forward translation of the pseudo code as it seemed possible. In the following I will address the parts which may not be trivial (like the $q_0 = q_1 \Rightarrow q0 = q1$; transition is being trivial).

The dot product

For the dot product I calculated local sums in a for loop, and I used `MPI_Allreduce` with `MPI_SUM` to get the global dot product at each process. This was exactly the case for the line 2 and 5 dot product. A slight change for line 8 product is that there the local sums are calculated in a for loop together with the line 6 and 7 vector update calculations.
I chose Allreduce to communication as this is created exactly for this kind of communications therefore hopefully well optimized. As far as I could find it it is equivalent to an `MPI_Gather` and `MPI_Scatter` with calculation between, which is exactly what I would have done if I couldn't use the Allreduce. This use $2(p-1)$ communication between threads, which can't be improved.

Vector update

For vector update we only need to calculate the local vector updates, This can be done in a single for loop going through the length of the arrays and make the update at each step. This is the exact method for line 10. For line 6 and 7 I used a single loop together with the local dot product from line 8 as this is better then doing them in separate loops.
Note that these calculations are well vectorizable therefor they are very fast using auto-vectorization, which is included in the `-O3` flag.

Stencil application

For the stencil application there is a need for communicating the borders. This is done using the defined 2D Cartesian grid. When creating the process grid I used `MPI_Cart_shift` to get the up, down, right and left neighbours address. As the grid was defined non-periodic, so at the border of the global mesh the neighbouring process will get `MPI_PROC_NULL`. This way I don't have to test if a send or receive is outside the process grid as this is handled by MPI. So as the global border should always be zero for calculation, so instead of is statements to test if we are at the global border, I used to declare all border zero, and as to this direction there will be no communication it stays 0 at the global border.
So the stencil application implemented as follows. First I use `MPI_Sendrecv` to communicate to all 4 neighbours, as just mentioned at global borders these communication ignored, and store the received border data in lists for all 4 border. Now I have all the data, so I could make the stencil computation

locally. First I compute the inner stencil applications, so the ones, which don't require any local border data. This is done in a double loop and in each iteration I compute one cell update as defined from the stencil.
Then I compute the local border updates, first the 4 corner, then the left, right border and up, top borders in 2 separate loop.
Note that there here are a lot of freedom in the implementation as it will discuss. Like we can change the communication to a non-blocking one, or change how the stencil application is computed.

Now I addressed all part of the algorithm implementation, finally after 200 iteration I compute $|| \ g \ || = \sqrt{g^T g}$ and share it with Allreduce as done for the other dot product. And then I plot from the root process.
Note that as it wasn't the aim I didn't included the gathering of the resulting u vector in the measured part.

Additional functionalities

I also have added some additional features to my code. First there is a print function for printing the local variables, which was used for debugging purposes.
Secondly I added variables based on which in the end the norm and run time may be printed to the screen. This was used to time measurements, as in the final version only the norm is plotted as it is said in the project description.
Next I added a variables based on which the program may run until a certain precision is reached instead of 200 iteration. This can be useful, if you want to use the program really for numerically approximating the solution of the PDE mentioned in the introduction.
Finally I added a functionalities to optionally save the final result. This means gathering all local data and saving it to a txt. This is also useful for real application. These last 2 options are used for the results in the *Results of the numerical modelling* section.

Optimization efforts and future possibilities

Now let's talk about the optimization effort I made to make my above original implementation faster. Let's mention that I only tried to sped up the measured part of the code.

First mention some fact. As in data dependency section it was mentioned there are a fixed order in the steps of the algorithm. Without making major changes these can't be resolved, but this way it is unlikely to expect too well scalability as there are global communications too frequently in the code, which results in a large amount of waiting.
For load balancing I didn't made any extra effort, for the following reasons. At the measured part of the code the load between the processes already well balanced. As each process has a local vector parts corresponding to a $(\frac{n}{\sqrt{p}} \pm 1) \times (\frac{n}{\sqrt{p}} \pm 1)$ size of the global mesh, which result in almost equal size for cases where $\sqrt{p} \mid n$ or $\frac{n}{\sqrt{p}}$ is large. As the local amount of work only depend on the size of the local vectors, this will result almost equal amount of work in each process.
Also it is important that not only the overall amount of work is equal, but between the communications the different processes do similar amount of work, so they doesn't have to wait for long for the communication. But in my implementation that is the case in the measured part of the code as between two communication the processes do the "same work", so they do the same computation on their local vectors, which are almost the same size. There is a slight load imbalance in the `MPI_Allreduce`, as the sum is computed in one process.
So the implementation has a good theoretical load balance, if we don't consider the waiting coming from the frequent communication. But due to the small differences in computing very similar tasks result in a significant waiting time in the communication steps, which as said can't be avoided due to data dependency.
So the final load balance is not so good, as a big part of the run times comes from MPI processes, large part of which are spent on waiting.

I did a map analysis for the code to see the origin of the work in the code. If not stated otherwise I am referring to analysis done on the final implementation with $n = 4096,\ p = 9$.

Results of the map analysis

The results of the different map analysis will be discussed later on in the relevant parts, but now I want to give an overview and a detailed line based introduction as later I won't investigate the times spent on each line. Some statistic can be seen on Table 2. Note that as there are limited samples there

can be major difference between running's.
Briefly a large proportion of the CPU usage comes from MPI processes mostly when there are relatively large number of process and small amount of overall work.
The MPI calls can't be line by line investigated, as the only information is that they call other functions. So let's speak about the computational part. The relevant parts are the vector update and the local stencil calculation. Based on the problem size and and processes these use $10-80\%$ of the CPU time with different parameter. The tendency is that the ratio of computational work grows as the problem size grows and also as the number of process are decreased. The most time consuming part of the computations are the memory access with approximately $80-90\%$ for vector update and $60-75\%$ for the inner stencil application (and the ratio is slightly larger for larger p or n). The dot product calculations doesn't use much memory, but can be significant amount of work. The outer stencil takes small time due to the short loops, so we can leave it out from the investigations as it doesn't play a major role in the overall run time.

	`Allreduce`	`Sendrecv`	inner stencil	vec update 1	vec update 2	d^Tq prod
4096 $-n$ 16	31%	10%	15%	24%	11%	7%
4096 $-n$ 9	34%	17%	7%	24%	9%	10%
4096 $-n$ 4	14%	4%	18%	32%	14%	17%
2048 $-n$ 16	68%	22%	1%	7%	2%	2%
2048 $-n$ 9	75%	11%	1%	10%	2%	1%
2048 $-n$ 4	28%	13%	8%	24%	12%	12%
1024 $-n$ 9	74%	18%	2%	4%	1%	1%
1024 $-n$ 4	55%	12%	5%	16%	6%	8%

Table 2: map analysis

Serial optimization

Let's discuss my efforts for serial optimization. The parts which takes measurable amount of serial work is the stencil application and the vector update / dot product.
For vector update and dot product I used a single for loop through the array

length and if it was possible I made more computation in the same loop. Using `-O3` compiler flag I got an efficient code for these task. I tried some method to archive speed up. The only one left in my implementation are the usage of `restrict` keyword as this resulted in a small, but measurable speed up. I also tried to use `__builtin_assume_aligned`, to make the vectorization better, which doesn't resulted in further speed up. Additionally I tested splitting the merged calculation into multiple for loop and do loop unrolling manually, but these doesn't have any positive effect on the run time, when I tested it separately.
In the map analysis I got that all of the above process can take up to 30% of the CPU time, so it is crucial to have a fast implementation, but as explained most of the resources spend on accesing memory, which I don't see a way to make faster.

Now about the stencil application. There would be possibilities to store the data differently, as will be discussed later, now focus on optimization with this data distribution. One option would be to make one double for loop and inside test if it is a border element. This is slower. In the separated implementation there are options to try to achieve speed up by changing the loops. I try the followings: separate the inner stencil calculation to better cache usage, merge the border computation into one for loop. But none of these resulted in speed up.
Note that in the map analysis I get 7.2% of CPU usage for the inner stencil and 0.5% CPU usage for the outer stencil. So small speed up here doesn't measurable in the final run time. And most of the run time comes from the memory access in these task as well, which I couldn't really reduce.
Finally I tried using `inline` as I wrote every part of my code to a function, but that doesn't resulted in speed up either.

It could be expected as these computation takes up only a small proportion of time to not get any speed up here. Also most of the methods for optimizing these easy computations are done better by the compiler optimization, so it was unlikely to achieve speed up by hand made manipulations.

Parallel communications

After doing the map analysis ($n = 4096,\ p = 9$) I saw that 43.5% of the CPU usage come from MPI processes (30% from `MPI_Allreduce`, 13.5% from

`MPI_Sendrecv`). This means that a large proportion of the resources goes to the communication processes, therefore it may be crucial to use sufficient communication protocols.
There are some place, where in theory some speed up can be achieved with changing the communication, those ain't in the submitted version, because they didn't resulted in measurable speed up, but I implemented and tried the following changes in communication.
Note that it would reduce the resources if we could reduce the number of communications. For the dot product it is unlikely due to the need of the global distribution of the data and the discussed data dependencies. Also `MPI_Allreduce` use $2p - 2$ communication, which can't be reduced.
For the border communication if we want to communicate the border in every loop cycle, then we can't reduce the number of communication. One theoretical way to not do communication in every step will be presented later.
So all I can change is the usage of blocking / non-blocking communication. In this implementation all communication are blocking. The change to non-blocking would allow the processes to do the communication in the background, while doing other calculation, therefor it may result in speed up and better load balance.
For line 5 I couldn't gain anything as there ain't any computation, which can be done while that communication.
For the other for product it was possible to do the communication, while doing the calculation of line 6. Note that it means that I only did 2 computation in the first for loop (line 7, 8) and calculated line 6 in a separate loop while doing communication using `MPI_Iallreduce` in the background.
I was hoping the most change from the following version, but it also doesn't resulted in measurable improvement. So in the stencil application when I used non-blocking communication in the background I could do the computation of the inner stencil, and I also pay attention, so while I compute the rows and columns differently to place the `MPI_Wait` between the 2 loop for which it can be communicated there and also did the columns computations earlier as they takes longer due to more cash misses.
Sadly I didn't received improvement from any of this changes, so I didn't keep them.

I did a map analysis with the last change and using the same parameters. I got the following results. The overall MPI CPU usage is a bit less 41% with still 30% coming from the `MPI_Allreduce` and 9% from the `MPI_Wait`.

Also the inner stencil application use a bit more resources 13%. Note that these small changes may only occurs as different analytical runs has a slightly different result. So the most significant result here is that the `MPI_Wait` will takes almost as much resource as the `MPI_Sendrecv` did previously, meaning that the waiting/synchronization is the most time consuming part of the MPI processes and not the real communication, but it can't be solved with non-blocking communication, as the processes should wait for all at a given points.

Virtual structure of the processes

I made an implementation to use a 1D grid distribution to the virtual structure, but it was measurably slower. But it has advantages compared to the square grid, like it can be used for any p value.
Also instead of the square process grid another rectangular grid could be used. This is the case for my code if you run it using a non square p value, but even for square numbers there are other forms to consider other then square or 1D. But they won't result in any speed up.
According to my measurements the square grid seems to be the optimal choice for square p values.

Data storage

So the implemented data storage was discussed. It has advantages for the computation of dot product and vector update, but it is not so good for the stencil application. So I would like to suggest one other storage structure, which would be more optimal for the stencil application and less optimal for the other calculations. Note that this change ain't major.
So to storing the local vectors and the neighbours I could use one 1D array of size$\mathit{horizontal\ size + 2} \times \mathit{vertical\ size + 2}$. In this way after communication the stencil application would look like a simple double for loop, but for other calculations I always should check to be inside boundaries, which may be done using a double for loop instead of a single, this way there is no need for if statement inside the for loop, and the number of cache miss doesn't grow much.
So overall it would result a simpler and bit faster code for the stencil application and a bit complicated and a slightly slower code for the other parts.

These changes mostly immeasurable during the 200 iteration, as the speed up may come from the outer stencil application, which is less then 0.5% of the CPU usage.

Other approaches

There are several other approach to solve the original problems, either if we think about is as solving $Ax = b$, or solving the arising PDE from the problem setting. But those I am not discussing here. As the task was to implement the given pseudo code. There are ways to make more complicated implementation, one of which will be discussed here, but I chose to do the most straight forward implementation, as I couldn't find any change which would result in significant speed up.

So I just mention one possibility to avoid some of the communication in the stencil application. As the stencil application only needs local information it is possible there, while I couldn't find a way to get rid of any global communication for the dot product using distributed vectors.
I didn't implemented it, but an idea would be to communicate less frequently, but instead of the local $a \times b$ vector and the communicated one row / column next to this, store a larger neighbourhood like $a + 2x \times b + 2x$ grid, so x neighbouring rows / columns in every direction and compute for some steps the neighbouring cells stencil application locally as well. This way we could reduce the communication for the stencil application, which was measured to use 13.5% of the CPU time. This may result in speed up, as I would expect that with less frequent bigger communication the amount of work may be reduced. And as the stencil application currently use less energy $< 7\%$, and hopefully the extra work wouldn't result in seriously more work. Approximately $4x(\frac{n}{\sqrt{p}}) + x^2$ more stencil approximation would be computed instead of $\frac{n^2}{p}$ in each iteration, so the stencil application run time shouldn't grow much for large n.
Note that these are just speculations as I didn't implemented this idea to test. And as the global communication still happens in each iteration, so there are frequent synchronization between all the process, so this change in communication may only lead to larger waiting times at the calculation of the next dot product.

Experiment and analysis

I did the following experiment. I run the code for some 2 powers, $n = 256, 512, \ldots 16384$ and tested the run time for the square number of processes $p = 1, 4, 9, \ldots 64$ as the structure and the load is clear in this cases. Note that I didn't run the timing a lot of time, not to use too much of the Uppmax resource, but this way there are more uncertainty in the measured times. All of the time measurements can be found in Table 3.
I also did a serial implementation and run that to have a base time. As it can be seen my parallel implementation ain't any slower then my serial (that may be possible to optimize more), but it makes sense, as for 1 process the communication goes fast, using map it doesn't even show any MPI resource usage. So this means that the parallel implementation in 1 core is an efficient serial implementation.
Let's note that for "small" n values, so smaller than 2048 there can be seen no real scaling at least for the large number of processes. The reason should be that the overhead of MPI and MPI communication, which are causing most of the work for small n with large number of process. This can be seen in the map analysis that for $n = 512$, $p = 16$, the MPI processes takes up around 90% of all resource usage, which would be even higher for larger number of process, so in this case the computational task ain't matter much compared to the MPI processes.
So for small n values it is best to use a serial implementation or not too large number of processes. And for the scaling analysis the measurements the times for $n = 256, 512$ and 1024 won't be used.

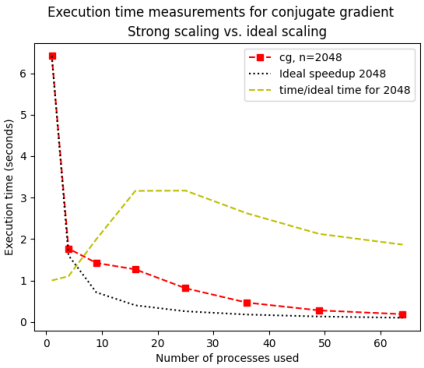
Also note that for this part, so the scaling experiment is the 200 iteration crucial, as it is needed to guarantee the same amount of work in some way and the fixed number of iteration is an easy way to do it.

Strong scaling

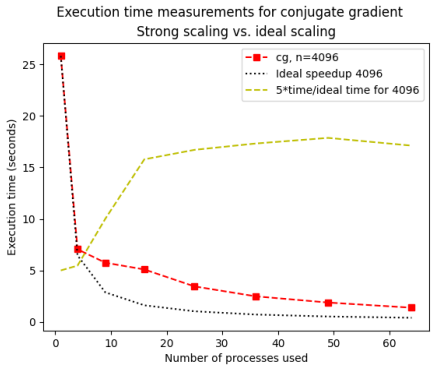
For the strong scaling experiment we have to fix the amount of work and increase the number of processes, so we need to look at one row at the run time table. For rows, where $n \geq 2048$, I plotted the measured run times and the ideal speed up based on the run time of my code in 1 core. And I also plotted the ratio between the two.
The plots can be seen in Figure 1.

n	serial	-n 1	-n 4	-n 9	-n 16	-n 25	-n 36	-n 49	-n 64
256	0.0965	0.0961	0.0235	0.0114	0.0072	0.0729	0.1321	0.1207	0.1073
512	0.3622	0.3310	0.0880	0.0416	0.0201	0.0821	0.1302	0.1214	0.1107
1024	1.4018	1.4942	0.3555	0.1636	0.0780	0.1302	0.1588	0.1483	0.1213
2048	6.5385	6.4169	1.7690	1.4232	1.2680	0.8133	0.4674	0.2782	0.1870
4096	25.0650	25.8173	7.0524	5.7472	5.0936	3.4501	2.4842	1.8826	1.3807
8192	105.5433	104.8521	28.1233	23.3686	20.1396	13.9072	9.5742	7.1326	5.8683
16384	427.5420	424.9189	113.1142	92.7977	80.9599	54.1834	51.5656	40.4459	29.2682

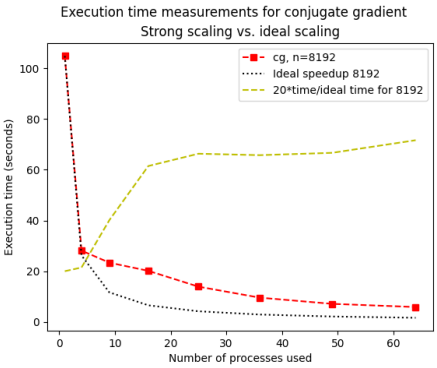
Table 3: Time measurement cg.c



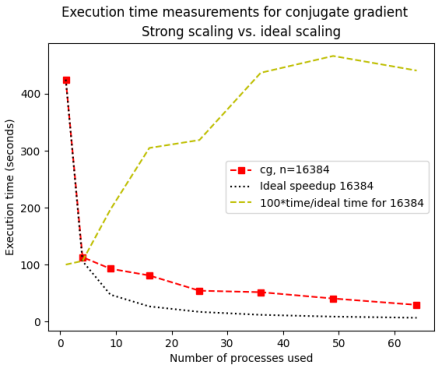
(a) CG $n = 2048$ strong scaling



(b) CG $n = 4096$ strong scaling



(c) CG $n = 8192$ strong scaling



(d) CG $n = 16382$ strong scaling

Figure 1: Strong scaling

Now let's investigate the result of the strong scaling experiment. As it can be seen for small ($p = 4$) size of process the speed up is very good, then it is starting to becoming much worse and the measured time on average around 3 times slower for large number of processes, then the ideal speed up would suggest. This is not a good scaling result, but also not terrible bad. One of the main reason, why the scaling can't be too good is the large amount of communication process, which after a while result in a serious time loss with performance as all the process should wait for each other at all dot product calculation and after the number of processes reaches a critical amount, this will result in a serious loss in performance, as there is a a bigger difference in computation times between the communication, partly caused by the large number of process and partly by a higher $\frac{n}{\sqrt{p}}$ value in cases $\sqrt{p} \nmid n$. Looking at the graphs and using the map analysis tool my suggestion is that the critical mass, where the waiting time increase happens between 4 and 16 process as it can be seen the ration with the ideal scale up grows the much there and then it mostly stays the same after. Also with the map analysis tool for $n = 2048$ I get that MPI has 28% CPU usage for $p = 4$ and 84% for $p = 16$, which is a serious grow in ratio of the resource usage for the communication. Note that this point where the amount of MPI work jumps in a short time it may vary with the problem size.
Another major part of the run times is generated by the memory access time, which in most cases also shows a growth compared to the ideal speed up. This can be concluded using the map analysis data and run time data shown previously. And as it was mention the memory access is the most time consuming part of the rest of the processes. And this also plays a big part of the loss compared to the ideal speed up as it can be seen from the map analysis on $n = 4096$ between $p = 9$ and $p = 16$. In this case the ratio with the ideal speed up still getting significantly worse, but here the time spent on the communication process didn't play a major part in the growth of the ratio. So here the computation causes further time loss, and as mention not only the memory access crucial in this part, but it shows a growing tendency as the number of cores become larger.

So we can see a relatively poor strong scaling, which can be motivated with the problem structure and the sudden jump in the communication time as well as the growth in the memory access time.

Speedup

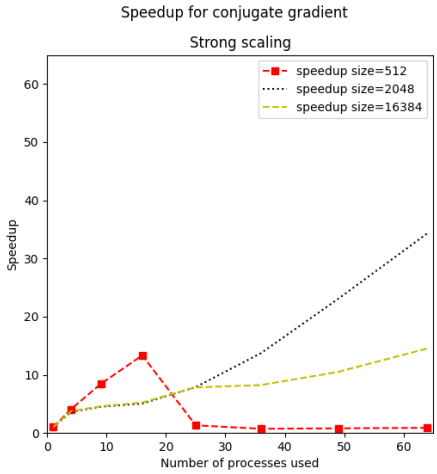


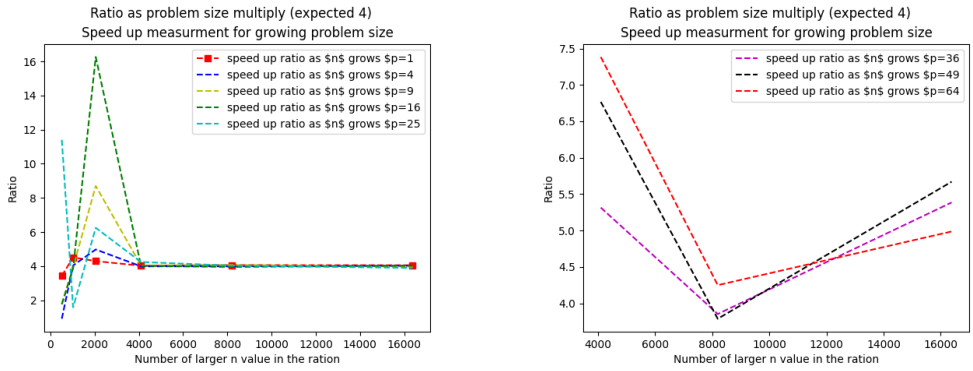
Figure 2: Strong scaling: speedup plot

Based on the above data we can compute the speedup, which is the traditionally used measure for the scaling. $S(n) = \frac{t(1)}{t(n)}$, where n is the used number of processes. As mentioned above with problem size smaller than 2048 the scaling is very bad, almost non existence after a while. Here I will calculate and plot the speed up for problem size 256 to demonstrate this. I also calculate speed up for problem size 2048 and 16382. The calculated speed ups are in table 4 and plotted in figure 2. As it can be seen the data represent the same tendencies as it was discussed in the above section, we have the same underlying data when we calculated the speedup values. So shortly: for small problem size in the beginning the scaling is good, but then it becomes very very poor. Smaller than 1 speedup values, due to the lack of work in each tread per cycles, so the communication becomes the significant part of the tasks and basically the threads spend little time on the real computation. For larger problem size we see a more normal speedup curve. But these curves shows a relatively poor scaling, but not tragic. Which can be expected from the problem settings.

size	-n 1	-n 4	-n 9	-n 16	-n 25	-n 36	-n 49	-n 64
256	1	4.0893	8.4298	13.3472	1.3182	0.7274	0.7961	0.8956
2048	1	3.6274	4.5088	5.0607	7.8899	13.7289	23.0658	34.3150
16384	1	3.7566	4.5789	5.24851	7.8422	8.2404	10.5059	14.5181

Table 4: Speedup values for the code cg.c

Investigation of the column wise speed up



(a) Speed up ration for $p \leq 25$ (b) Speed up ration for $p \geq 36$

Figure 3: Speed up ration for growing multiplying problem size

It is also interesting to look at the scaling we get as the problem size grow, as this can be also important for real application. I am plotted the ratios as I multiply the problem size. This ratio is expected to be 4, that would mean perfect scale up. The plots are shown in Figure 3. *Note that here the ratio is not compared to one base time, but only in pair values. As this has an expected value of 4, which can be seen very nicely for the smaller p values with sufficiently large n.*

In the first figure I plotted the ratios for $p \leq 25$. Here it can be seen in the beginning there is some randomness, then they all are almost exactly 4 later on. The first "chaotic" part are partly caused by the fact that at a given point we reach a point where the whole vectors can be fitted in the cache which result in a significant speed up, so that's part of the reasons why there are big jumps upwards in the plots. Also in the smaller cases the randomness

of the run times may cause a bigger change in the ratio.
In the second plot the large number of process only has the larger problem sizes plotted as for small problems the computation doesn't matter much and therefore there are unrealistic ratios. I would also expect that for the large number of process the ratio would become similarly almost 4 as for smaller number of processes once the problem size becomes big enough, so that the computation parts becomes dominant int the run time.
So the conclusion is that for problem size after a sufficiently large size the code seems to scales well. It can be seen for small number of process and my guess is that it should behave similarly for larger number of processes as a sufficiently large n value is reached as I see no motivation why the larger cases behave differently.

Weak scaling

For weak scaling we have to look at time measurements, where with the resources the amount of work grow at the same tendency. As the amount of computation grows as n^2, so I used $n = 2048, 4096, 8192$ and 16384 with $p = 1, 4, 16$ and 64. The plot can be seen in Figure 4. Next to the measurements I plotted the ideal scale up, which is a line, as this is what weak scaling means.

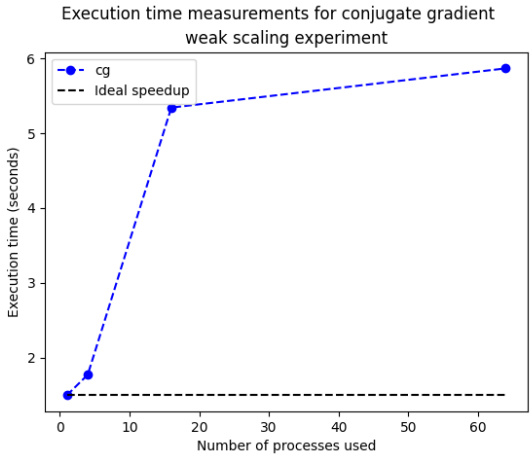


Figure 4: CG weak scaling $n = 2048 - 16382$ with $p = 1 - 4 - 16 - 64$

Now let's investigate the weak scaling result. As it can be seen there is a strange thing, so that for 1 and 4 core and 16 and 64 core the weak scaling

are pretty reasonable separately, but between the two there is a big decreasement in performance. It should be caused by the same reasons as the growth in the ratio was caused in the strong scaling part as we saw that the scaling between different n values are almost perfect. And the change is also appears in the same place between $p = 4$ and $p = 16$. So the main causes is the growth in communication time as well as the growth in memory access time. Of course there are other matters include, and it is expected to don't have perfect scaling, but this extreme jump in run time are needed to be explain with other reasons then the normal time loss, that's why the reasons were mentioned here.

So the conclusion is that there is a reasonable weak scaling between the small process number and the large, but between there is a jump.

Conclusion of the scaling analysis

So the short conclusion is that the problem doesn't scale well to large number of processes, but this is caused by the lot of communication which can be seen using analytic tools and a growth in wait time, which means bad load imbalance is a major cause of the bad scaling. Another reason is the growth in memory access time.
Also for small number of process and large number of processes the problem scales relatively well. Additionally the scaling with the increasement of the problem size is expected to become perfect after a sufficient problem size.

Results of the numerical modelling

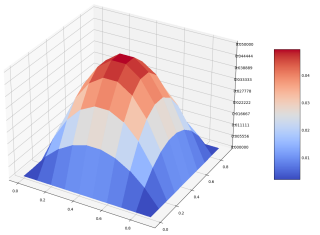
Now I would like to talk a bit about the final norm values and the result vectors. So after 200 iteration the out putted resulting norms can be seen in Table 5.
It can be seen that they are increasing. The reason is that in the discrete norm there are much more number summed as n grows and even they average smaller they can result in a larger norm. Therefore for larger problem size a larger number of iteration is needed for reach the same precision of approximation.
This is a reason that for scaling the fixed number of iteration is a good way to fix computation, as the number of iterations would vary a lot if we would run until convergence reached.

n	norm
256	0.0000381655
512	0.0044322618
1024	0.0072754062
2048	0.0082487287
4096	0.3864326544
8192	0.5603785297
16384	0.6876454245

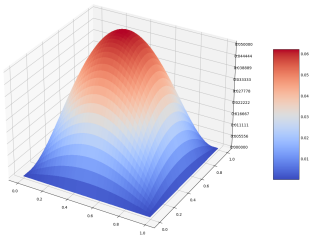
Table 5: Norms created by cg.c

Now I gathered the resulting vector saved it and plotted it to show the numerical solution of the PDE. It is showed in Figure 5 and 6.
The result for 200 iteration and convergence (with $\epsilon = 0.001$ precision) are plotted separately for some different n values.
As it can be seen with small n the approximation is not that good. Also for large n values the 200 iteration are seemingly gives bad results.
After convergence the resulting plots looks a lot alike and it seems like a reasonable solution for the given PDE.

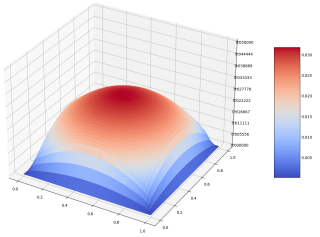
This is a proof of the correctness of the implementation in a way, or at least a guarantee that there ain't major errors, as the result given by it is a reasonable approximation of the solution of the PDE.



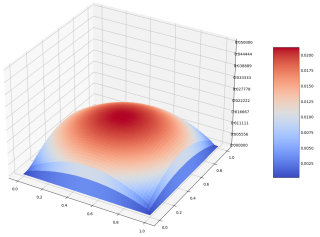
(a) Result vector after 200 iter $n = 8$



(b) Result vector after 200 iter $n = 256$

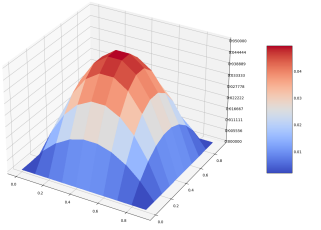


(c) Result vector after 200 iter $n = 2048$

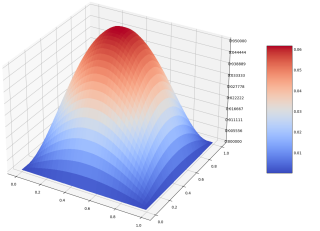


(d) Result vector after 200 iter $n = 4096$

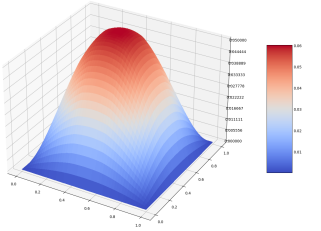
Figure 5: Resulting vector after 200 iteration for different grid side



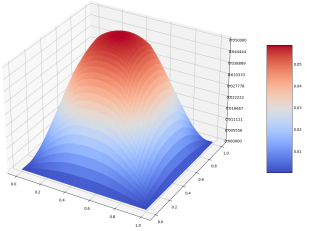
(a) Result vector $n = 8$



(b) Result vector $n = 256$



(c) Result vector $n = 2048$



(d) Result vector $n = 4096$

Figure 6: Resulting vector after convergence for different grid side $\epsilon = 0.001$

Summary

In summary the conjugate gradient method can be useful for solving linear equation and it is very useful when large spare systems arise, which is often the case for solving PDEs numerically.

For reasonable small mesh size the serial implementation is quite powerful, but as the need for very large grids arise, then it becomes more important to make the solution parallel.

For the parallel implementation due to data dependencies there are a need for lot of communication between all processes, which result insufficient scale up as using large number of cores, because there is a larger loss due to communications waiting time.

So we can use this algorithm for several different application, but there is the necessity to optimize the implementation to the given application. Like choosing between serial and parallel implementation, or using different background knowledge of the problem, resulting similar advantages like doing the stencil application instead of matrix multiplication in our case.

References

[CGO76] Paul Concus, Gene H. Golub, and Dianne P. O'Leary. "A GENERALIZED CONJUGATE GRADIENT METHOD FOR THE NUMERICAL SOLUTION OF ELLIPTIC PARTIAL DIFFERENTIAL EQUATIONS". In: *Sparse Matrix Computations*. Ed. by JAMES R. BUNCH and DONALD J. ROSE. Academic Press, 1976, pp. 309–332. ISBN: 978-0-12-141050-6. DOI: https://doi.org/10.1016/B978-0-12-141050-6.50023-4.

[JDG91] E.M. Johansson, F.U. Dowla, and D.M. Goodman. "BACKPROPAGATION LEARNING FOR MULTILAYER FEED-FORWARD NEURAL NETWORKS USING THE CONJUGATE GRADIENT METHOD". In: *International Journal of Neural Systems* 02.04 (1991), pp. 291–301. DOI: 10.1142/S0129065791000261.

[Kno07] Potts D. Knopp T Kunis S. "A note on the iterative MRI reconstruction from nonuniform k-space data." In: *Int J Biomed Imaging.* (2007). DOI: 10.1155/2007/24727.

[Igo09] Ariela Sofer Igor Griva Stephen G. Nash. *Linear and Nonlinear Optimization*. George Mason University Fairfax, Virginia. Society for Industrial and Applied Mathematics, 2009. ISBN: 978-0-898716-61-0.

[YLW09] Gonglin Yuan, Xiwen Lu, and Zengxin Wei. "A conjugate gradient method with descent direction for unconstrained optimization". In: *Journal of Computational and Applied Mathematics* 233.2 (2009), pp. 519–530. DOI: https://doi.org/10.1016/j.cam.2009.08.001.

[HS13] M. Malekzadeh H. Azami and S. Sanei. "A New Neural Network Approach for Face Recognition Based on Conjugate Gradient Algorithms and Principal Component Analysis". In: *Journal of Mathematics and Computer Science* 6.3 (2013), pp. 166–175. DOI: 10.22436/jmcs.06.03.01..

[JL17] Hao Ji and Yaohang Li. "Block Conjugate Gradient algorithms for least squares problems". In: *Journal of Computational and Applied Mathematics* 317 (2017), pp. 203–217. ISSN: 0377-0427. DOI: https://doi.org/10.1016/j.cam.2016.11.031.

Appendix

Final implementation of CG (cg.c)

```c
#include <mpi.h>
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <unistd.h>


#define save 0 // for saving resulting u value
#define root 0 // for I/O processes

#define print_time 0
#define print_norm 1

#define is_precise 0 // for iteration untill precision
#define eps 0.001
#define MAX_ITER 10000

int write_output(char *file_name, const double *matrix, int n) {
        /*
        A function to output the gathered global u value for plotting purposes
        This function can use to plot any 2d array with a 0 valued border
    */
    FILE *file;
        if (NULL == (file = fopen(file_name, "w"))) {
                perror("Couldn't open output file");
                return -1;
        }
    //first row with 0s
        if (0 > fprintf(file, "%.6f ", 0.0)) {perror("Couldn't write to output file");}
        for (int j = 0; j < n; j++) {
                if (0 > fprintf(file, "%.6f ", 0.0)) {perror("Couldn't write to output
                    file");}
        }
        if (0 > fprintf(file, "%.6f\n ", 0.0)) {perror("Couldn't write to output file")
            ;}
    //matrix body
        for (int i = 0; i < n; i++) {
        if (0 > fprintf(file, "%.6f ", 0.0)) {perror("Couldn't write to output file");}
        for (int j = 0; j < n; j++) {
                if (0 > fprintf(file, "%.6f ", matrix[i*n+j])) {perror("Couldn't write
                    to output file");}
        }
        if (0 > fprintf(file, "%.6f\n ", 0.0)) {perror("Couldn't write to output file")
            ;}
        }
    //last row with 0s
        if (0 > fprintf(file, "%.6f ", 0.0)) {perror("Couldn't write to output file");}
        for (int j = 0; j < n; j++) {
                if (0 > fprintf(file, "%.6f ", 0.0)) {perror("Couldn't write to output
                    file");}
        }
        if (0 > fprintf(file, "%.6f ", 0.0)) {perror("Couldn't write to output file");}

        if (0 != fclose(file)) {
                perror("Warning: couldn't close output file");
        }
        return 0;
}

void print_local(double* l,  double* d_left, double* d_right, double* d_up, double*
    d_down, int h, int v, int rank){
    /*
        For debuging purpose a print function for local data.
        To avoid collision of different processes it uses sleep(rank)
    */
    sleep(rank);
    printf("         ");
        for (int j = 0; j < h; j++){
            printf("%lf ",d_up[j]);
        }
```

```
65      printf("\n");
66      for (int i = 0; i < v; i++){
67          printf("%lf␣",d_left[i]);
68          for (int j = 0; j < h; j++){
69              printf("%lf␣",l[i*h+j]);
70          }
71          printf("%lf␣",d_right[i]);
72          printf("\n");
73      }
74      printf("␣␣␣␣␣␣␣␣␣");
75          for (int j = 0; j < h; j++){
76              printf("%lf␣",d_down[j]);
77          }
78      printf("\n\n");
79  }
80
81  void fill_u(double* glob_u,double* loc_u, int x, int y, int h, int v,int n){
82      /*
83          A helperfunction for gather_u
84          This function fills up the global u from local u data using the positioning
                variables
85      */
86      for (int i = 0; i < v; i++) {
87          for (int j = 0; j < h; j++) {
88              glob_u[(x+i)*n+y+j] = loc_u[i*h+j];
89          }
90      }
91  }
92
93  void gather_u(double** glob_u, double* u, int n, int hsize, int vsize, int x_id, int
        y_id, int rank, int size){
94      /*
95          Gathreing the u values in the root process using MPI communication
96      */
97
98      if(rank==root) {
99          *glob_u=(double*)malloc(n*n*sizeof(double));
100         double* loc_u;
101         for (int i = 0; i < size; i++) {
102             if(i!=root){
103                 // communication
104                 int data[4];
105                 MPI_Recv(data, 4, MPI_INT, i, i, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
106                 int h = data[0]; int v = data[1]; int x = data[2]; int y = data[3];
107                 loc_u=(double*)malloc(h*v*sizeof(double));
108                 MPI_Recv(loc_u, h*v, MPI_DOUBLE, i, i, MPI_COMM_WORLD, MPI_STATUS_IGNORE
                        );
109
110                 // fill in the local u values in the global u
111                 fill_u(*glob_u, loc_u, x-1, y-1, h, v, n);
112                 free(loc_u);
113             }
114             else{
115                 // fill in the root process own data
116                 fill_u(*glob_u, u, x_id-1, y_id-1, hsize, vsize, n);
117             }
118         }
119     }
120     else{
121         int data[4]={hsize,vsize,x_id,y_id};
122         MPI_Send(data, 4, MPI_INT, root, rank, MPI_COMM_WORLD);
123         MPI_Send(u, hsize*vsize, MPI_DOUBLE, root, rank, MPI_COMM_WORLD);
124     }
125
126 }
127
128 double vector_update1(double * __restrict u, double * __restrict d, double * __restrict
        g, double * __restrict q, double tau, int size) {
129     /*
130         Implementation for line 6,7,8 vector update and local dot product
131     */
132     double dot=0;
133     for(int i=0; i<size; i++) {
134         u[i] += tau*d[i];
135         g[i] += tau*q[i];
136         dot += g[i]*g[i];
137     }
```

```
138     return dot;
139 }
140
141 void vector_update2(double * __restrict d, double * __restrict g, double beta, int size)
        {
142     /*
143         Implementation for line 10 vector update
144     */
145     for(int i=0; i<size; i++) {
146         d[i] = beta * d[i] - g[i];
147     }
148 }
149
150 void get_vhxy_size(int* vsize, int* hsize, int*x, int*y, int n, int size, int ver, int
        hor, int vgrid, int hgrid){
151     /*
152         set up the process coorinat variables based on the 2D Cartesian grid
153             hsize, vsize - horizontal and vertical size of the locally calculated
                    variable
154             x,y the vertical and horizontal coordinates of the top left coorner of the
                    local data in the global grid
155     */
156     int vdiv = n/vgrid;
157     int vrem = n%vgrid;
158
159     int hdiv = n/hgrid;
160     int hrem = n%hgrid;
161     if(ver<vrem)
162         (*vsize)=vdiv+1;
163     else
164         (*vsize)=vdiv;
165     if(hor<hrem)
166         (*hsize)=hdiv+1;
167     else
168         (*hsize)=hdiv;
169
170     *x=1;
171     *y=1;
172     for (int i = 0; i < ver; i++){
173         if(i<vrem)
174             *x+=vdiv+1;
175         else
176             *x+=vdiv;
177     }
178     for (int i = 0; i < hor; i++){
179         if(i<hrem)
180             *y+=hdiv+1;
181         else
182             *y+=hdiv;
183     }
184 }
185
186 void set_up_gd(double* g,double* d,int hsize,int  vsize,int x_id,int y_id,int n){
187     /*
188         Based in the process position variables set up the local starting values defined
                by a given function
189     */
190     double bij; double h = 1/(double)(n+1);
191     double xi, yi;
192     for (int i = 0; i < vsize; i++){
193         for (int j = 0; j < hsize; j++){
194             xi=(i+x_id)/(double)(n+1);
195             yi=(j+y_id)/(double)(n+1);
196             bij = 2*h*h*(xi*(1-xi)+yi*(1-yi));
197             g[i*hsize+j]=-bij;
198             d[i*hsize+j]=bij;
199         }
200     }
201 }
202
203 void border_sync(int hsize, int  vsize, double* d, double* d_left, double* d_right,
        double* d_up, double* d_down, int left, int right, int up, int down, MPI_Comm*
        comm2D, MPI_Datatype* col_type){
204     /*
205         Simple implementation of the border data exchange with the neighbours
206             Note: The Cartesian grid is non periodic, so at the border the send/recv are
                    ignored
```

```
207       */
208       // rows | top - bottom
209       MPI_Sendrecv(d, hsize, MPI_DOUBLE, up, 1, d_up, hsize, MPI_DOUBLE, up, 1, *comm2D,
              MPI_STATUS_IGNORE);
210       MPI_Sendrecv(&(d[hsize*(vsize-1)]), hsize, MPI_DOUBLE, down, 1, d_down, hsize,
              MPI_DOUBLE, down, 1, *comm2D, MPI_STATUS_IGNORE);
211
212       //cols | left - right
213       MPI_Sendrecv(d, 1, *col_type, left, 1, d_left, vsize, MPI_DOUBLE, left, 1, *comm2D,
              MPI_STATUS_IGNORE);
214       MPI_Sendrecv(&(d[hsize-1]), 1, *col_type, right, 1, d_right, vsize, MPI_DOUBLE,
              right, 1, *comm2D, MPI_STATUS_IGNORE);
215   }
216
217   void inner_stencil(double* stencil,double* d,int hsize,int vsize){
218       /*
219           Stencil calculation except the border
220       */
221       for (int i = 1; i < vsize-1; i++){
222           for (int j = 1; j < hsize-1; j++){
223               stencil[i*hsize+j] = 4*d[i*hsize+j] - d[(i-1)*hsize+j] - d[(i+1)*hsize+j] -
                      d[i*hsize+(j-1)] - d[i*hsize+(j+1)];
224           }
225       }
226   }
227
228   void outer_stencil(double* stencil, double* d, double* d_left, double* d_right, double*
          d_up, double* d_down, int hsize, int vsize){
229       /*
230           Stencil calculation at the border
231       */
232       // corner
233       stencil[0]=4*d[0]-d_up[0]-d_left[0]-d[1]-d[hsize];
234       stencil[hsize-1]=4*d[hsize-1]-d_up[hsize-1]-d_right[0]-d[hsize-2]-d[2*hsize-1];
235       stencil[(vsize-1)*hsize]=4*d[(vsize-1)*hsize]-d_down[0]-d_left[vsize-1]-d[(vsize-1)*
              hsize+1]-d[(vsize-2)*hsize];
236       stencil[vsize*hsize-1]=4*d[vsize*hsize-1]-d_down[hsize-1]-d_right[vsize-1]-d[vsize*
              hsize-2]-d[(vsize-1)*hsize-1];
237
238       // border | left -right
239       for (int i = 1; i < vsize-1; i++){
240           stencil[i*hsize] = 4*d[i*hsize] - d[(i-1)*hsize] - d[(i+1)*hsize] - d_left[i] -
                  d[i*hsize+(+1)];
241           stencil[i*hsize+hsize-1] = 4*d[i*hsize+hsize-1] - d[(i-1)*hsize+hsize-1] - d[(i
                  +1)*hsize+hsize-1] - d[i*hsize+-1+hsize-1] - d_right[i];
242       }
243       // border | top - bottom
244       for (int j = 1; j < hsize-1; j++){
245           stencil[j] = 4*d[j] - d_up[j] - d[hsize+j] - d[j-1] - d[j+1];
246           stencil[(vsize-1)*hsize+j] = 4*d[(vsize-1)*hsize+j] - d[(vsize-2)*hsize+j] -
                  d_down[j] - d[(vsize-1)*hsize+(j-1)] - d[(vsize-1)*hsize+(j+1)];
247       }
248   }
249
250
251   int main(int argc, char **argv) {
252           if (2 != argc) {
253                   printf("Usage: cg n\n Here n denotes the number of nodes on one side\n")
                          ;
254                   return 1;
255           }
256       // problem variable
257           int n = atoi(argv[1]);
258
259       //Communication variables
260       int reorder; int ndims; int periods[2]; int coord[2]; int dims[2];
261       MPI_Comm comm2D;
262       MPI_Datatype col_type;
263       int up, down, left, right;
264       int rank, size;
265
266
267       //local variables
268                                   // note x_id is vertical and y_id is horizontal
269       int vsize, hsize, loc_size, x_id, y_id;                                          //
              local rectangle size and position
270       double* g; double* d; double* u; double* q; double* glob_u;                      //
```

```
            vectors
271         double* d_left; double* d_right; double* d_up; double* d_down;                  // d
                vector local border
272         double q0,q1,loc_q1, beta, dTq, tau, loc_dTq;                                   //
                scalar values
273         double s_time, r_time, time;
274
275         MPI_Init(&argc, &argv);
276         MPI_Comm_size(MPI_COMM_WORLD, &size);
277         MPI_Comm_rank(MPI_COMM_WORLD, &rank);
278
279         // Createing 2D structure
280         periods[0]=0; periods[1]=0;
281         dims[0]=0; dims[1]=0;
282         MPI_Dims_create(size, 2, dims);
283         reorder=1, ndims=2;
284         MPI_Cart_create(MPI_COMM_WORLD,ndims,dims,periods,reorder,&comm2D);
285         MPI_Cart_shift(comm2D,1,1,&left,&right);
286         MPI_Cart_shift(comm2D,0,1,&up,&down);
287
288         // Defining indexes of the local variables based on rank
289         MPI_Cart_coords(comm2D, rank, 2, coord);
290         get_vhxy_size(&vsize, &hsize, &x_id, &y_id, n, size ,coord[0], coord[1], dims[0],
                dims[1]);
291
292         //allocating local variables
293         loc_size=hsize*vsize;
294         g = (double*)malloc(loc_size*sizeof(double));
295         d = (double*)malloc(loc_size*sizeof(double));
296         u = (double*)calloc(loc_size,sizeof(double));
297         q = (double*)malloc(loc_size*sizeof(double));
298         d_left = (double*)calloc(vsize,sizeof(double));
299         d_right = (double*)calloc(vsize,sizeof(double));
300         d_up = (double*)calloc(hsize,sizeof(double));
301         d_down = (double*)calloc(hsize,sizeof(double));
302
303         // set up local part of vectors starting value
304         set_up_gd(g,d,hsize, vsize, x_id, y_id, n);
305
306         // set up MPI type for column
307         MPI_Type_vector(vsize, 1, hsize, MPI_DOUBLE, &col_type);
308         MPI_Type_commit(&col_type);
309
310     // Start timer - the begining of the algorithm
311         s_time= MPI_Wtime();
312
313         // q0
314         loc_q1=0;
315         for (int i = 0; i < loc_size; i++)
316                 loc_q1+=d[i]*d[i];
317         MPI_Allreduce(&loc_q1, &q0, 1, MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD);
318
319
320
321     #if is_precise
322         // line 3/1
323         for (int i = 0; i<MAX_ITER; i++) {
324     #else
325         // for 200 iter - instead of line 3
326         for (int i = 0; i < 200; i++) {
327     #endif
328             //sync boundaries -line 4/1
329             border_sync(hsize, vsize, d, d_left, d_right, d_up, d_down, left, right, up,
                    down, &comm2D, &col_type);
330
331             // 2D stencil application - line 4/2
332                 // inner stencil
333                 inner_stencil(q, d, hsize, vsize);
334                 //outer stencil
335                 outer_stencil(q, d, d_left, d_right, d_up, d_down, hsize, vsize);
336
337             //dq dot product
338                 //line 5/1
339             loc_dTq=0;
340             for (int i = 0; i < loc_size; i++){
341                 loc_dTq+=d[i]*q[i];
342             }
```

```
343             //dot product sharing qTq
344                 //line 5/2
345             MPI_Allreduce(&loc_dTq, &dTq, 1, MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD);
346             tau=q0/dTq;
347 
348             //u, g update + gTg dot product
349                 // line 6,7,8/1
350             loc_q1=vector_update1(u, d, g, q, tau, loc_size);
351 
352             //dot product sharing q1
353                 //line 8/2
354             MPI_Allreduce(&loc_q1, &q1, 1, MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD);
355 
356             //line 9
357             beta=q1/q0;
358 
359             // d update
360                 //line 10
361             vector_update2(d, g, beta, loc_size);
362 
363             //line 11
364             q0=q1;
365 
366         #if is_precise
367             //line 3/2
368             if(sqrt(q0)<eps){
369                 break;
370             }
371         #endif
372         }
373 
374 
375         // ||q||^2
376             loc_q1=0;
377             for (int i = 0; i < loc_size; i++)
378                     loc_q1+=g[i]*g[i];
379         MPI_Allreduce(&loc_q1, &q0, 1, MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD);
380 
381     // Stop timer - end of the given task
382     r_time = MPI_Wtime();
383 
384         MPI_Reduce(&r_time, &time, 1, MPI_DOUBLE, MPI_MAX, root, MPI_COMM_WORLD);
385 
386         // printing data from root
387         if(rank==root){
388             #if print_norm
389             printf("%.10f\n", sqrt(q0));
390             #endif
391             #if print_time
392             printf("%lf\n", time);
393             #endif
394         }
395 
396     #if save
397         //gatharing and saving of the resulting vector
398          gather_u(&glob_u, u, n, hsize, vsize, x_id, y_id, rank, size);
399          if(rank==root){
400             char out[80];
401             sprintf(out, "out_%d.txt", n);
402             write_output(out,glob_u,n);
403             free(glob_u);
404          }
405     #endif
406 
407 
408         free(d);
409         free(u);
410         free(g);
411         free(q);
412         free(d_left);free(d_right);free(d_up);free(d_down);
413         MPI_Comm_free(&comm2D);
414         MPI_Finalize();
415             return 0;
416     }
```

Makefile

```
1  ################################################################################
2  # Makefile for assignment 2, Parallel and Distributed Computing 2023.
3  ################################################################################
4
5  CC = mpicc
6  CFLAGS = -std=c99 -g -O3
7  LIBS = -lm
8
9  BIN = cg
10
11 all: $(BIN)
12
13 cg: cg.c
14         $(CC) $(CFLAGS) -o $@ $< $(LIBS)
15
16 clean:
17         $(RM) $(BIN)
```

Serial implementation of CG - used for serial runtime reference

```
1
2  #include <stdio.h>
3  #include <stdlib.h>
4  #include <unistd.h>
5  #include <sys/time.h>
6
7
8    static double get_wall_seconds() {
9      struct timeval tv;
10     gettimeofday(&tv, NULL);
11     double seconds = tv.tv_sec + (double)tv.tv_usec / 1000000;
12     return seconds;
13   }
14
15 int write_output(char *file_name, int n, double matrix[n+2][n+2]) {
16         /*
17         A function to output the gathered global u value for plotting purposes
18         This function can use to plot any 2d array with a 0 valued border
19     */
20     FILE *file;
21         if (NULL == (file = fopen(file_name, "w"))) {
22                 perror("Couldn't open output file");
23                 return -1;
24         }
25     //matrix body
26         for (int i = 0; i < n+2; i++) {
27         for (int j = 0; j < n+2; j++) {
28                 if (0 > fprintf(file, "%.6f ", matrix[i][j])) {perror("Couldn't write to
                       output file");}
29         }
30         if (0 > fprintf(file, "\n ")) {perror("Couldn't write to output file");}
31         }
32
33         if (0 != fclose(file)) {
34                 perror("Warning: couldn't close output file");
35         }
36         return 0;
37 }
38
39
40
41
42
43 int main(int argc, char **argv) {
44         if (2 != argc) {
45                 printf("Usage: cg n\n Here n denotes the number of nodes on one side\n")
                       ;
46                 return 1;
47         }
48     // problem variable
49         int n = atoi(argv[1]);
50
51
52     //local variables
53     int loc_size=(n+2)*(n+2);
54     double g[n+2][n+2]; double d[n+2][n+2]; double u[n+2][n+2]; double q[n+2][n+2];
           double glob_u[n+2][n+2];
55     double q0,q1,beta, dTq, tau;
56     double s_time, r_time;
57
58
59     //allocating local variables to 0
60     for (int i = 0; i < n+2;i++)
61     {
62         for (int j = 0; j < n+2; j++)
63         {
64             g[i][j]=0;
65             d[i][j]=0;
66             u[i][j]=0;
67             q[i][j]=0;
68         }
69
```

```
70      }
71
72      // set up local part of vectors starting value
73      double bij; double h = 1/(double)(n+1);
74     double xi, yi;
75      for (int i = 1; i < n+1; i++){
76          for (int j = 1; j < n+1; j++){
77              yi=(i)/(double)(n+1);
78              xi=(j)/(double)(n+1);
79              bij = 2*h*h*(xi*(1-xi)+yi*(1-yi));
80              g[i][j]=-bij;
81              d[i][j]=bij;
82          }
83      }
84
85
86
87  // Start timer - the begining of the algorithm
88      s_time= get_wall_seconds();
89
90      // q0 - line 2
91      q0=0;
92      for (int i = 1; i < n+1; i++)
93          for (int j = 1; j < n+1; j++)
94              q0 += d[i][j]*d[i][j];
95
96
97
98
99      // for 200 iter - instead of line 3
100     for (int i = 0; i < 200; i++) {
101
102
103
104
105         // 2D stencil application - line 4
106         for (int i = 1; i < n+1; i++)
107             for (int j = 1; j < n+1; j++)
108                 q[i][j]=4*d[i][j]-d[i][j+1]-d[i][j-1]-d[i+1][j]-d[i-1][j];
109
110
111         //dq dot product
112             //line 5
113         dTq=0;
114         for (int i = 1; i < n+1; i++)
115             for (int j = 1; j < n+1; j++)
116                 dTq+=d[i][j]*q[i][j];
117
118         tau=q0/dTq;
119
120         //u, g update + gTg dot product
121             // line 6,7,8
122             q1=0;
123         for (int i = 1; i < n+1; i++)
124             for (int j = 1; j < n+1; j++){
125                 u[i][j]+=tau*d[i][j];
126                 g[i][j]+=tau*q[i][j];
127                 q1+=g[i][j]*g[i][j];
128             }
129
130         //line 9
131         beta=q1/q0;
132
133         // d update
134             //line 10
135          for (int i = 1; i < n+1; i++)
136             for (int j = 0; j < n+2; j++)
137                 d[i][j]=beta*d[i][j]-g[i][j];
138
139         //line 11
140         q0=q1;
141
142     }
143
144
145     // ||q||^2
146         q0=0;
```

```
147             for (int i = 1; i < n+1; i++)
148                 for (int j = 1; j < n+1; j++)
149                     q0+=d[i][j]*d[i][j];
150
151
152     // Stop timer - end of the given task
153     r_time = get_wall_seconds();
154
155
156
157     // printing data
158
159     printf("%lf\n", r_time-s_time);
160
161     char out[80];
162     sprintf(out, "out_%d.txt", n);
163     write_output(out,n,u);
164
165     return 0;
166     }
```