GEORG-AUGUST-UNIVERSITÄT GÖTTINGEN IN PUBLICA COMMODA SEIT 1737

HPS

Seminar Report

Implementing and Analysing a scalable numerical Laplace solver

Jakob Hördt, Philipp Müller

MatrNr: 21565573, 21874297

Supervisor: Jonathan Decker

Georg-August-Universität Göttingen
Institute of Computer Science

September 30, 2022

Abstract

Laplace's equation describes many real world phenomena. For complex boundary conditions, finding an analytic solution becomes infeasible. We design and implement a proof-of-concept numeric Laplace solver using MPI. We show how it exhibits strong and weak scaling behaviour and scales to large problem sizes on an HPC cluster using comprehensive benchmarking. The software is evaluated further using Vampir, LIKWID and basic performance modeling. Two of the identified optimization opportunities are implemented and shown to improve performance. One of them, avoiding computations with denormalized floating point numbers, had an especially large impact. Although the software comes close to a basic peak performance model, it is currently memory bound and further optimization opportunities are identified and discussed.

Contents

List of Tables

List of Figures

Listings

List of Abbreviations

GWDG Gesellschaft für wissenschaftliche Datenverarbeitung mbH Göttingen

HPC High-Performance Computing

MPI Message Passing Interface

PGM Portable GrayMap

SCC Scientific Compute Cluster

SIMD Single instruction, multiple data

LIKWID Like I Knew What I'm Doing

1 Introduction

Laplace's equation, $\Delta f = 0$, is a partial differential equation named after Pierre-Simon Laplace. In physics many phenomena, like the electric potential, the gravitational potential, or the stationary heat equation, are described by it. Finding analytic solutions to Laplace's equation is only feasible for simple boundary conditions. For more complex boundary conditions, as they appear in models of the real world, a numeric approach offers an approximate solution. These numeric solutions typically employ a finite element method, which means discretizing the solution domain into a grid. Numeric solvers quickly take significant amounts of time to converge when scaling up the grid size. Users may wish for faster wall clock convergence on typical size grids or to solve the Laplace equation for grid sizes too large to fit into the memory of a single machine. To satisfy this need we develop a numeric Laplace solver that scales to large problem sizes on a High-Performance Computing (HPC) cluster.

Firstly, background information regarding Laplace's equation and numerical solution approaches are covered in sections 2 and 3 respectively. We implement a sequential version and then proceed to parallelize it. Their design and implementation details are discussed in sections 5.1 and 5.2 respectively.

The software will be evaluated with respect to the above use cases in section 6. The goal is to identify and possibly fix potential performance issues. A series of benchmarks is conducted to assess the scaling behaviour of the software in sections 6.1 and 6.2. We present and evaluate two changes in section 6.4, one of which drastically raised the scaling performance of the software by reducing calculations with denormalized floating point numbers. An estimate for the efficiency of our software considering the capabilities of our test hardware is discussed in section 6.5. We build a basic roofline model of our software in section 6.6. With the roofline analysis, we learn that the software is memory bound and thus that optimizations in this area would be advisable.

2 Laplace's equation

For $f : \mathbb{R}^n \to \mathbb{R}$ a possibly multidimensional scalar valued function that is twice continuously differentiable, eq. (1) is Laplace's equation[FLS13, chapter 7.1]:

$$\Delta f = 0 \tag{1}$$

Here, Δ is the Laplace Operator which can be thought of as the second order derivative for multidimensional functions. It is defined as $\Delta f = \nabla^2 f$ where the ∇ (spoken Nabla) operator is defined as a vector of partial derivatives: $\nabla = \left(\frac{\partial}{\partial x_1}, \cdots \frac{\partial}{\partial x_n}\right)$. When applied to a scalar function via scalar multiplication the ∇ operator yields the gradient of f:

$$\nabla f = \text{grad} f = \left(\frac{\partial f}{\partial x_1}, \cdots \frac{\partial f}{\partial x_n}\right) \tag{2}$$

The gradient of a scalar function is a vector field where each vector indicates the rate and direction of fastest increase. It it a generalization of the first order derivative for multivariable functions. An example of a gradient can be seen in fig. 2. When the ∇

operator is applied to a vector field $\vec{g} : \mathbb{R}^n \to \mathbb{R}^n$ via the dot product we get its divergence:

$$\nabla \vec{g} = \text{div}\vec{g} = \frac{\partial \vec{g}}{\partial x_1} + \cdots + \frac{\partial \vec{g}}{\partial x_n} \tag{3}$$

The divergence of any vector field is a scalar field that represents how much of a source the vector field is at any given point. When the ∇ operator is applied to the gradient ∇f of f we get the divergence of the gradient, which is the Laplacian, of f:

$$\nabla \cdot \nabla f = \nabla \cdot \left(\frac{\partial f}{\partial x_1}, \cdots \frac{\partial f}{\partial x_n} \right) = \frac{\partial^2 f}{\partial x_1^2} + \cdots + \frac{\partial^2 f}{\partial x_n^2} \tag{4}$$

As an example, consider a scalar field that has a local minimum. At the minimum, the gradient diverges away from it, thus it, and any other function that reaches local minima or maxima, cannot satisfy Laplace's equation. Equation (4) leads to an alternative notation for the Laplacian:

$$\Delta f = \sum_{i=1}^{n} \frac{\partial^2 f}{\partial x_i^2} \tag{5}$$

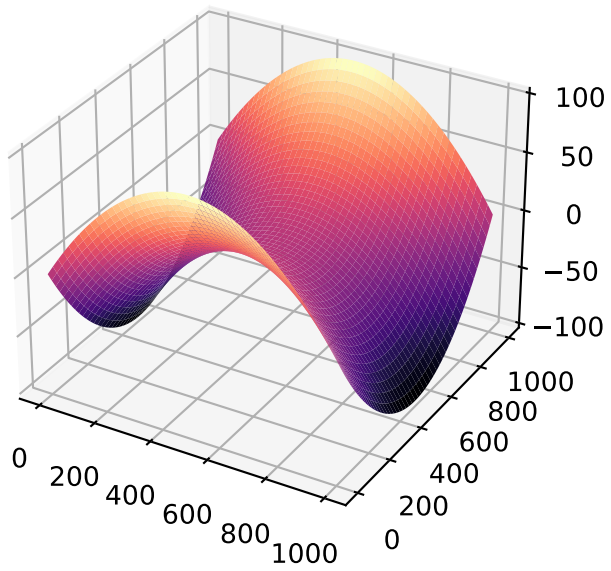

Figure 1: A plot of the function $f(x, y) = \left(\frac{y}{50} - 10\right)^2 - \left(\frac{x}{50} - 10\right)^2$. The Laplacian is not plotted, it would be the plane at $z = 0$.

Figure 1 shows an example of a function that satisfies Laplace's equation, since its Laplacian is zero on the entire domain as shown here:

$$\begin{aligned} \Delta f = \frac{\partial^2 f}{\partial x^2} + \frac{\partial^2 f}{\partial y^2} &= \frac{\partial}{\partial x}\left(-2\left(\frac{x}{50} - 10\right)\frac{1}{50}\right) + \frac{\partial}{\partial y}\left(2\left(\frac{y}{50} - 10\right)\frac{1}{50}\right) \\ &= \frac{\partial}{\partial x}\left(-\frac{x}{1250} + \frac{2}{5}\right) + \frac{\partial}{\partial y}\left(\frac{y}{1250} - \frac{2}{5}\right) = -\frac{1}{1250} + \frac{1}{1250} = 0 \end{aligned} \tag{6}$$

Figure 2 shows the same function with the gradient overlayed on top. Imagine the gradient depiction as a flow of water and try to find a spot *within* the graph where water would accumulate or disappear. There is no such point, because the gradient does not diverge anywhere.

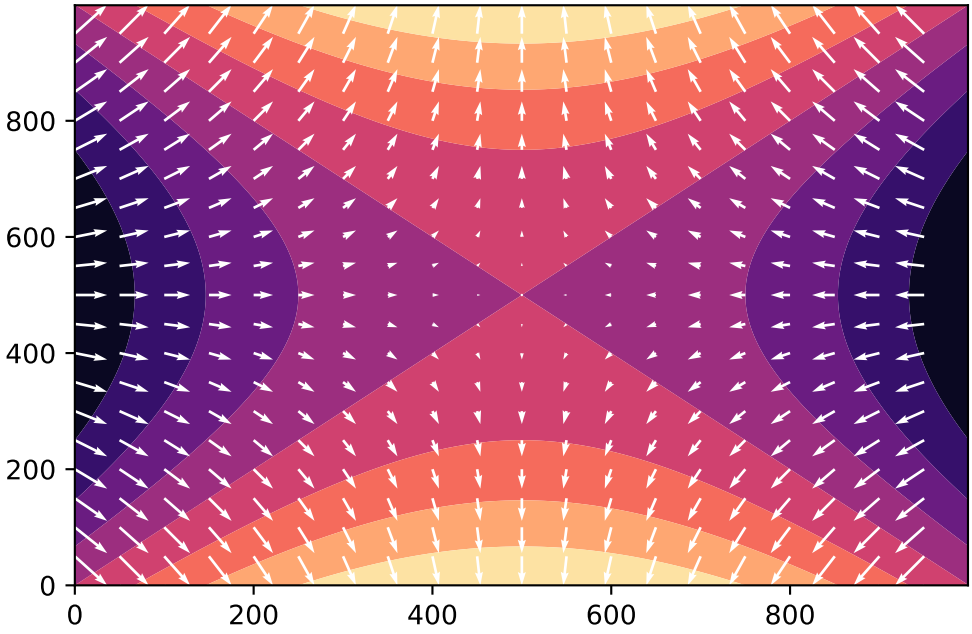

Figure 2: A contour plot of the function $f(x, y) = \left(\frac{y}{50} - 10\right)^2 - \left(\frac{x}{50} - 10\right)^2$. The white arrows depict the gradient ∇f of f.

2.1 Applications

Informally, Laplace's equation describes many physical phenomena because often, the gradient of a physical quantity corresponds to some kind of flow. Heat for example flows from hot regions to cold regions. The divergence of the gradient is the total flow out of a small volume around a given point. If that is zero, that means the flow is in equilibrium. The **heat** equation for example is the following where u is the temperature at position x and time t and a is the heat conduction coefficient[Dem18, p.281]:

$$\frac{\partial}{\partial t} u(x, t) = a \Delta_x u(x, t) \tag{7}$$

When $\frac{\partial}{\partial t} u(x, t) = 0$, u satisfies Laplace's equation.

Another application of Laplace's equation is the gravitational potential. Gauß's law for **gravity** describes the gravitational acceleration g with ρ being the mass density[1]:

$$\nabla g = -4\pi G \rho \tag{8}$$

With g expressed in terms of the gravitational potential

$$g = -\nabla \phi \tag{9}$$

and eq. (8) we get Poisson's equation for gravitational fields:

$$\Delta \phi = 4\pi G \rho \tag{10}$$

[1]Gauss's law and gravity, accessed Sept 27, 2022: `https://physicscourses.colorado.edu/phys2210/phys2210_fa20/lecture/lec29-gauss-law-gravity/`

Where $\rho = 0$, ϕ satisfies Laplace's equation for gravitational fields:

$$\Delta\phi = 0 \tag{11}$$

A third example application, that closely resembles the gravitational potential, comes from Electrostatics[Dem13, p.10-11]. Gauß's law for **electricity** describes the electric field $\mathbf{E}$ with ρ being the charge density:

$$\nabla\mathbf{E} = \frac{\rho}{\varepsilon_0} \tag{12}$$

With $\mathbf{E}$ expressed in terms of the electric potential

$$\mathbf{E} = -\nabla V \tag{13}$$

and eq. (12) we get Poisson's equation for electricity:

$$\Delta V = -\frac{\rho}{\varepsilon_0} \tag{14}$$

Where $\rho = 0$, V satisfies Laplace's equation for the electric potential:

$$\Delta V = 0 \tag{15}$$

2.2 Boundary Conditions

Laplace's equation by itself has an uncountably infinite set of solutions. To model a scenario you define boundary conditions on the boundaries of your desired domain. Klemens Burg explains the distinction between Dirichlet, Neumann, and mixed boundary conditions[Bur04, chapter 5.3]. Let $f : D \to \mathbb{R}$. D is the Domain of f. Now ∂D denotes the boundary of D which has its intuitive meaning. With Dirichlet boundary conditions, the value of f is given directly on the boundary by the continuous function g. In formal terms:

$$f = g \quad \text{on } \partial D \tag{16}$$

Dirichlet boundary conditions make the solution to Laplace's equation unique.

With Neumann boundary conditions, the gradient of f, ∇f, in the direction of the normal vector $\vec{n}$ of the boundary is given by the scalar function g on the boundary. Formally:

$$\nabla f \cdot \vec{n} = g \quad \text{on } \partial D \tag{17}$$

The solution to a Neumann boundary value problem is unique apart from an additive constant. The third case is a mix between a Neumann and Dirichlet boundary conditions. For the purpose of this course however we focus exclusively on Dirichlet boundary conditions, that is with fixed initial values on the boundary.

3 Numerical Solutions to Laplace's equation

For simple boundary conditions, you can find a solution to Laplace's equation analytically. Proving that a function satisfies Laplace's equation can be done by calculating the

Laplacian as shown in eq. (6). Finding such a function is much more complex. Feynman et al. give an example: "Even such a simple problem as that of a charged cylindrical metal can closed at both ends [...] can be solved only approximately, using numerical methods."[FLS13, chapter 7.1]. This demonstrates the need for numerical solvers like the one implemented in this work.

In their excellent tutorial, Per Brinch Hansen derives and discusses numerical solutions to Laplace's equation[Han92]. The author focuses on a two dimensional steady state heat flow scenario but the concepts apply to any Laplace equation. To solve Laplace's equation numerically the first step is to discretize the domain of f into grid cells u with spacing h. Here, this is shown for a two dimensional scenario:

$$\begin{array}{ccc} & u_{i,j+1} = f(x, y+h) & \\ u_{i-1,j} = f(x-h, y) & u_{i,j} = f(x,y) & u_{i+1,j} = f(x+h, y) \\ & u_{i,j-1} = f(x, y-h) & \end{array}$$

This practice of discretizing a continuous domain into a finite number of elements is known as a finite element method. Now, the values of the grid cells introduced above can be approximated by a second order Taylor polynomial centered on (x, y), if h is small enough. In x-direction this results in:

$$u_{i+1,j} \approx u_{i,j} + h\frac{\partial f}{\partial x}(x,y) + \frac{1}{2}h^2\frac{\partial^2 f}{\partial x^2}(x,y) \tag{18}$$

$$u_{i-1,j} \approx u_{i,j} - h\frac{\partial f}{\partial x}(x,y) + \frac{1}{2}h^2\frac{\partial^2 f}{\partial x^2}(x,y) \tag{19}$$

Adding both together yields the following where the first order term is canceled out:

$$u_{i+1,j} + u_{i-1,j} \approx 2u_{i,j} + h^2\frac{\partial^2 f}{\partial x^2}(x,y) \tag{20}$$

Similarly, for the y-dimension we get:

$$u_{i,j+1} + u_{i,j-1} \approx 2u_{i,j} + h^2\frac{\partial^2 f}{\partial y^2}(x,y) \tag{21}$$

Adding eq. (20) and eq. (21) gives an approximation for the Laplacian:

$$u_{i+1,j} + u_{i-1,j} + u_{i,j+1} + u_{i,j-1} \approx 4u_{i,j} + h^2\left(\frac{\partial^2 f}{\partial x^2} + \frac{\partial^2 f}{\partial y^2}\right)(x,y) \tag{22}$$

$$(u_{i,j+1} + u_{i,j-1} + u_{i+1,j} + u_{i-1,j} - 4u_{i,j})/h^2 \approx \left(\frac{\partial^2 f}{\partial x^2} + \frac{\partial^2 f}{\partial y^2}\right)(x,y) = \Delta f(x,y) = 0 \tag{23}$$

Since Δf must be zero for a solution to Laplace's equation we can derive the following approximation for the value of each grid point in terms of its four direct neighbors, also called the Von Neumann neighborhood:

$$u_{i,j+1} + u_{i,j-1} + u_{i+1,j} + u_{i-1,j} - 4u_{i,j} \approx 0 \tag{24}$$

$$u_{i,j+1} + u_{i,j-1} + u_{i+1,j} + u_{i-1,j} \approx 4u_{i,j} \tag{25}$$

$$(u_{i,j+1} + u_{i,j-1} + u_{i+1,j} + u_{i-1,j})/4 \approx u_{i,j} \tag{26}$$

As you can see in eq. (26), for a solution to Laplace's equation each grid point is simply, approximately the average of its neighbors! We can use this to replace our partial differential equation by $n \cdot n$ linear equations for an $n \times n$ grid.

This system of linear equations can be solved with any method but according to Hansen such a sparse system where each equation depends only on a few variables is best solved by an iterative method. Hansen refers to this as relaxation. The basic idea of the Jacobi relaxation method is to replace each non boundary grid point by the average of its neighbor values (eq. (26)) from the previous step, in each relaxation step. The Gauß-Seidel method is similar, except that the grid points are updated in-place. Every update uses old values and updated values. The unknown values are initialized with a guess. The relaxation step is repeated a fixed number of times or until some break condition. Both Jacobi and Gauß-Seidel take $\mathcal{O}(n^2)$ relaxation steps to converge for a runtime of $\mathcal{O}(n^4)$, because of the n^2 grid cells. Gauß-Seidel converges twice as fast as Jacobi, because it uses on average two out of four updated values in each new approximation.

Hansen introduces the technique of overrelaxation. Instead of updating grid cells with their neighbor averages directly, it is desirable to over correct and weight the difference between the current value and the average and add that to the current value. This is expressed here where u is the old grid cell value, w is the so called relaxation factor, and "average" is the average from eq. (26).

$$\text{next} = u + w * (\text{average} - u) \tag{27}$$

For this method to converge, w must be picked in accordance with the updating scheme. From our observations, Jacobi is unstable with any amount of overrelaxation. According to Hansen, the Gauß-Seidel updating scheme works with overrelaxation. If the overrelaxation is applied correctly, this method requires only $\mathcal{O}(n)$ relaxation steps and $\mathcal{O}(n^3)$ runtime.

4 Methodology

In this work we implement a numeric Laplace solver based on the Gauß-Seidel overrelaxation method. It should be able to take an arbitrary scenario defined by a grid that is partially filled with fixed values, which represent Dirichlet boundary conditions, as input. The program should calculate an approximate solution to Laplace's equation in an appropriate time. The program should produce output in a format that is easy to import or use with other software. We validate the program by comparing its output for a specific scenario to a known analytic solution. Our solver should support two-dimensional scenarios only. During the computation, the solver should not become numerically unstable. It should feature automatic termination depending on the convergence progress.

Firstly, we implement a completely sequential version satisfying the above requirements. Then we will proceed to parallelize the program using a distributed memory paradigm for it to be able to run on a HPC cluster. For this we will use the Message Passing Interface (MPI) library. MPI is as standardized API specification that enables process communication through message passing[2]. This approach does not require the processes to reside on the same machine, making it is suitable for computer clusters. Note that implementations *can* use shared memory for faster communication if available.

Our parallel implementation should be able to reduce the wall clock time it takes to solve a problem over the sequential version. While doing this it should exhibit strong scaling, which means the speedup should be close to proportional to the provided parallelism. The parallel version with two tasks should solve the same problem as the single threaded

[2]MPI home page: `https://www.mcs.anl.gov/research/projects/mpi/index.htm`

version in almost half the time. Additionally, the parallel version's memory usage of each sub task should be anti proportional to the number of tasks to make it possible for the software to solve problems too large to fit in the memory of a single machine. Our software should also show good weak scaling behaviour, meaning the speedup should not degrade much when increasing the problem size and parallelism proportionally together.

To validate these scaling properties we conduct a series of benchmarks and compare the results to the ideal behaviour. We also make use of profiling tools to analyze the runtime performance of our parallel and sequential program and potentially identify performance problems in our implementation.

5 Implementation

Before coding the Sequential version, there are some facts to be clarified. When iterating through the field, for overrelaxation to work the iteration can not be randomized, but every grid point needs two old and two new values next to them. So the two possibilities are to calculate row-wise or column-wise. We decided to calculate the grid row wise, as it brings cache locality and the values that are calculated one after another are standing next to each other in storage and we where hoping for performance improvement.

Also the border of the grid, meaning the points that only have two or three neighbours, must be given by the input. Looking at chapter 3, it can be seen that in order for the calculation of a grid point to work properly, it must have four neighbours. If a point only had three neighbours, the taylor approximation of the point would be wrong. In order to avoid making theoretical mistakes, we just make it impossible for the border to not be initialized by the user.

5.1 Sequential version

In the sequential version we decided to use a one dimensional vector instead of a two-dimensional field to improve performance. In our datastruct we store the width and the height of the field, and the vector containing our data is localized here. Also there is an index function which returns the correct place in the vector for a certain x and y pair.

```
struct Data {
    std::ptrdiff_t width;
    std::ptrdiff_t height;
    std::vector<scalar_t> data;

    scalar_t& idx(std::ptrdiff_t x, std::ptrdiff_t y) {
        return data[x + width * y];
    }
};
```

Listing 1: Datastruct

The next function is the get_input function. It sets the general conditions for the experiment. In this case the heat is set to 100. It could also be electric potential or a unit from another field where the laplace equation can be applied. Also this function sets the

size of the grid. It is to be noted, that the grid's size does not match the experiments size in real life, but the accuracy of the numerical solution. A bigger width and height while containing the same experiment size does mean the accuracy is bigger and the calculated numerical solution is closer to the analytic solution.

After that, the field is initialized with the set width and height and it's filled with NaN values. In the next step, the boundary conditions for the experiment are set. Those come from the user, depending on the scenario that's being calculated. In this case, the upper border is initialized with the set heat value, while all other borders are set to zero.

```
Data get_input() {
    const int heat = 100;

    Data input;
    input.width = 1000;
    input.height = 1000;
    input.data.resize(input.width*input.height,
    std::numeric_limits<scalar_t>::quiet_NaN());

    for (auto y = 0z; y < input.height; ++y) {
    for (auto x = 0z; x < input.width; ++x) {
        if (y == 0) {
            input.idx(x,y) = heat;
        } else if (x == 0 || x == input.width-1 || y == input.height-1) {
            input.idx(x,y) = 0;
        }
    }
}
return input;
}
```

Listing 2: Function: get_input

Because of the setting of the experiment, the values setting the boundary condition do not change, they are not being calculated. To make sure those are not touched by the program, we initialize another field containing the positions of calculable grid points.

```
auto get_variable_coordinates{
    std::vector<Coordinate> variable_coordinates;

    for (auto y = 0z; y < data.height; ++y) {
       for (auto x = 0z; x < data.width; ++x) {
          if (std::isnan(data.idx(x,y))) {
              if (is_border(x,y)) {
                    throw std::runtime_error{"Error"};
                }
                variable_coordinates.push_back({x,y});
            }
        }
    }
    return variable_coordinates;
}
```

Listing 3: Function: get_variable_coordinates

The NaN values make it possible to see, which value has been initialized and which value is still available for the computation. This fact is used by the function get_variable_coordinates which fills a field as mentioned before. Also, while the function iterates through the field, it checks whether a value is located on the border. As we know, values on the border can't be calculated, so the program gives an error if it detects a border value that has not been initialized and is still NaN.

After the boundary conditions are set and the mutable grid points are known, the function make_first_guess goes through the field and initializes every NaN value with 0. That is needed, as the program would not work with NaN values because they cant be calculated. In order to save performance, the function is not iterating through the whole field, but just through the variable_coordinates as they contain every grid point with a NaN value.

```
void make_first_guess(Data& data,
const std::vector<Coordinate>& variable_coordinates) {
    for (auto [x, y] : variable_coordinates) {
        data.idx(x,y) = 0;
    }
}
```

Listing 4: Function: make_first_guess

In theory, the value used to initialize the field does not need to be zero but could be every other value. We choose zero to be a good value for our calculation as its the lowest value that can be reached with our initialisation. Other values might be better to converge faster but zero was most intuitive for the beginning. Also one could imagine to make the value changeable by the user, as they maybe know best where the values might converge. After the field is initialized with it's boundary conditions and filled with zeros, the computation process can begin. The main iteration loop goes through the field in every iteration and calculates every grid point from its neighbouring points.

```
1  const auto max_iterations = 10000;
2  auto i = 0;
3  for (; i < max_iterations; ++i) {
4      scalar_t residual = 0;
5      for (auto [x,y] : variable_coordinates) {
6          const auto old_value = data.idx(x,y);
7          const auto average = (data.idx(x,y-1) + data.idx(x-1,y)
8          + data.idx(x+1,y) + data.idx(x,y+1)) / 4;
9
10         data.idx(x,y) = data.idx(x,y) + relaxation_factor
11         * (average - data.idx(x,y));
12
13         residual += std::pow(data.idx(x,y) - old_value, 2);
14     }
15     if (residual < precision) {
16         break;
17     }
18 }
```

Listing 5: main iteration loop

The function goes through the field of variable_coordinates in line 5. For every point, the new value is calculated with the formula for over relaxation we already know in line 7-8 and 10-11. The relaxation factor we use is 1.9. After that, in line 13 the difference of the old and the new value is being added to the residual value. Line 15-17 makes sure the code breaks if the residual is smaller than a certain, user set, precision.

The last function of the sequential version is generating the output image. For our test purposes we used the Portable GrayMap (PGM) image format which encodes grayscale images and makes them easy to manipulate.[3] We picked it because it is easy to write without a software dependency.

```
1      const auto max_value = *std::max_element(data.data.begin(), data.data.end());
2
3      std::cout << "P2\n" << data.width << ' ' << data.height << "\n" <<
4      max_value << "\n" << "#" << max_iterations << ' ' << i << '\n';
5
6
7      for (auto y = 0z; y < data.height; ++y) {
8          for (auto x = 0z; x < data.width; ++x) {
9              std::cout << static_cast<int>(data.idx(x,y)) << ' ';
10         }
11         std::cout << '\n';
12     }
```

Listing 6: Function: print_output

Line 1 calculates the maximum value used, as the pgm format needs a maximum value

[3] Official Netpbm Website, accessed Sept 29, 2022: `https://netpbm.sourceforge.net/`

to work. Lines 3-4 print the pgm header and lines 7-12 print the calculated field in fitting lines.

The sequential version of the code almost satisfies all of our needs. For a certain input it gives a fitting output in a reasonable amount of time. To make the program more comfortable to use, the input function should be able to take data input and read from it. That being said its still not really needed for the program to work. The biggest problem of the sequential version is, that it needs a lot of time for grids that are bigger than 1000x1000. As shown in the performance tests later, a grid of 2500x2500 already needs around 80 minutes of time, which is too much when considering that, depending on the problem, even bigger grids with more complex scenarios might be needed. It is important to mention that a bigger grid does not just scale linear to its size because a bigger grid also needs more iterations to converge. A 2000x2000 points field calculates over four times longer than a 1000x1000 grid does. So in order to use the program properly on real problems, it needs to run on more threads than one.

5.2 Parallel version

In addition to making the program parallel, the parallel version of the program contains some other improvements we found out to be helping with performance while we where testing the program.

While trying to parallelize the sequential version we where thinking about two ways of splitting the work among the threads. While the only time consuming calculations are happening in the main iteration loop. The first method is to split the iterations up on different threads. The problem with splitting the iterations is, that every iteration depends on the data calculated in the previous iteration, so the iterations would have to wait for the previous calculation to finish. In fact, that does not speed up the program at all, so this method of parallelizing does simply not work.

The second method is to split up the grid. Splitting up the grid means, that the iteration itself is faster, but the number of iterations calculated by each thread does not change. Also, a lot of communication is needed as every thread needs to know the calculations of its neighboring thread in order to start its next iteration. Also the residual needs to be communicated, so that the processes all know when to stop the calculation. Lastly we needed to find a way to split the grid upon our threads. We found two ways of splitting the grid, one being easier to initialize(fig. 4) and one being the more effective one (fig. 3).

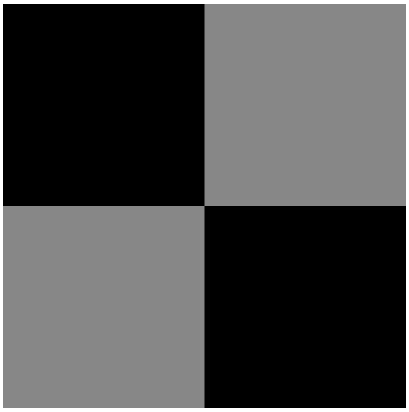

Figure 3: Split in 2 dimensions

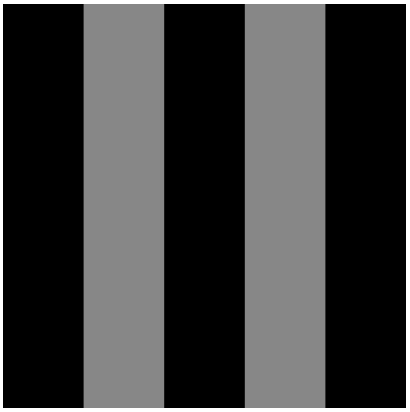

Figure 4: Split in 1 dimension

When splitting the grid in two dimensions (in fact n dimensions when working in n dimensions), the added border of all the parts is a lot smaller than when only splitting in one dimension. When talking about border, only borders next to another threads part are relevant, as they need to be communicated. Outer borders are irrelevant for the calculation. A 100x100 square split in two dimensions has 100 points of border for each small square, adding up to 400 points of border. Splitting the same Square of 100x100 in just one dimension means every thread has 200 border points if they are not on the edges. The threads on the edges have a total of 100 points. In the example, splitting on one dimension would lead to 600 points of border. So in this case splitting one dimension needs 1.5x more border communication. Still we decided to choose the one dimensional approach for our program. To us it was more important to make the program easy scalable, which would not have been possible with the 2 dimensional split. The program could have only be used with an even number of threads, ideally with a square number. So we choose to use the one dimensional method. Now when we want to use more threads, we simply make more slices. For performance it is important to split the grid in the largest dimension when not working with a square. This minimizes the border. Choosing the largest dimension as the dimension that's being split is up to the user and not implemented in the code.

The first thing we needed to do was to write a function to split the points among the threads we where using. The get_input function is extended by a part, that sets the size of the tasks local grid.

```
1 input.global_width = size;
2 const auto lower_local_width = input.global_width / world.size();
3 const auto tasks_with_lower_width_count =
4         world.size() - input.global_width % world.size();
5
6 input.width = world.rank() < tasks_with_lower_width_count
7                   ? lower_local_width
8                   : lower_local_width + 1;
9 input.height = size;
```

Listing 7: Function: addition to get_input

We chose to part the tasks into two groups: the ones with more grid points and the ones with less grid points. In line two we calculate the width of the smaller tasks and in line three we find out how many of them we have. In line 6-8 we make each task check whether it has a lower local width or a higher local width. Summarized, the first n (n = tasks_with_lower_width_count) tasks get a lower local width and every remaining task gets a higher local width until every task knows its size.

```
1  const auto offset_from_lower_width_tasks =
2          std::min(std::ptrdiff_t{world.rank()}, tasks_with_lower_width_count) *
3          lower_local_width;
4
5  const auto offset_from_higher_width_tasks =
6          std::max(world.rank() - tasks_with_lower_width_count, 0l) *
7          (lower_local_width + 1);
8
9  input.x_offset =
10         offset_from_lower_width_tasks + offset_from_higher_width_tasks;
```

Listing 8: Calculation of the offset

In the second part of the function, the offset is calculated. We need a way to calculate the global value from each tasks local value. In lines 2-4, if the task has a lower local width, the offset from lines 2-4 is already the right offset, and in lines 6-8 its offset would come out as zero because $\max(\text{world.rank}() - \text{tasks\_with\_lower\_width\_count}, 0l)$ would be zero. If the task has a higher local width, lines 2-4 would come out of the added offset of all the lower local width tasks and lines 6-8 would be its offset in the higher local width tasks. With the two offset values being added in lines 10-11, every task knows its position in the whole grid.

When talking about parallelization, border exchange was mentioned. After every iteration, every thread needs to know the calculation results of its neighbouring threads. As the threads can't calculate the next iteration without this information, we use MPI_Sendrecv for the communication. When every thread gets its size, all the threads except the border threads get two rows more than they store data. Those two rows are then used to write the data of the neighboring threads directly into the data field of the thread. This process saves a lot of memory when its compared to writing the values into a different variable for the receiving thread.

```
1  auto cart = [] {
2          mpi::communicator world;
3          mpi::cartesian_dimension dims[] = {{world.size(), false}};
4          return mpi::cartesian_communicator(
5                  world, mpi::cartesian_topology(dims), true
6          );
7  }();
8
9  const auto [left_neighbor, right_neighbor] = cart.shifted_ranks(0, 1);
```

Listing 9: MPI Cartesian Communicator

For the Sendrecv call to work, every thread needs to know its neighbours. That's what happens in line nine after initializing a cartesian communicator in lines 1-7. Now that every thread knows it's neighbours, they can communicate their borders.

```
1  if (const auto result = MPI_Sendrecv(
2                  &data.idx(0, 0), data.height, mpi_type_scalar, left_neighbor, 0,
3                  &data.idx(data.width, 0), data.height, mpi_type_scalar,
4                  right_neighbor, 0, cart, MPI_STATUS_IGNORE
5          );
6      result != MPI_SUCCESS) {
7          throw mpi::exception{"MPI_Sendrecv", result};
8  }
9
10 if (const auto result = MPI_Sendrecv(
11                 &data.idx(data.width - 1, 0), data.height, mpi_type_scalar,
12                 right_neighbor, 0, &data.idx(-1, 0), data.height, mpi_type_scalar,
13                 left_neighbor, 0, cart, MPI_STATUS_IGNORE
14         );
15     result != MPI_SUCCESS) {
16         throw mpi::exception{"MPI_Sendrecv", result};
17 }
```

Listing 10: MPI_Sendrecv calls

In lines 1-8 and 10-17 the two sendrecv calls can be seen, one for the left and one for the right side. The two threads on the edge just have one partner per sendrecv call, once they send to MPI_PROC_NULL and in the other call they receive from MPI_PROC_NULL. In order for the threads to make communication easier, we chose to change from row major to column major, because it enables the threads to write their data directly into the memory section of their neighbouring threads.

When comparing to the sequential version, now the residual needs to be communicated. In order to save resources, every thread calculates its own residual but in the end they still need to be communicated. As the threads calculate their own residual, they might not all have the same residual. That means, in order for the threads to stop at the same time, the residual also needs to be communicated. In order to reduce communication, we decided to communicate the residual only every 100 calculations. That means that in the worst case, the program makes 100 calculations extra even though the precision was reached, but that still saves a lot of time. Every 100 iterations, the residual of all the threads is added with an MPI_all_reduce call. Now every thread has the same residual, checks whether the precision is bigger and terminates if so.

```
1  if constexpr (with_residual) {
2          residual = mpi::all_reduce(cart, residual, std::plus<>{});
3          return residual;
4  }
```

Listing 11: Communicating the residual

While most of the main iteration loop works as in the sequential version, the relaxation factor used in the parallel version is 1.7. This value came from us trying to find the largest value while the program was still stable. As the method we are using contains elements from jacobi and gauß-seidel, there was no right formula for the factor. 1.7 came out to be

the biggest, factor we didn't have any destabilization problems with. When the threads quit from the main iteration loop, every thread has a part of the field with the converged data. Now the threads all generate their own output image, still in the pgm format.

Figure 5: Output Thread 1 Figure 6: Output Thread 2 Figure 7: Output Thread 3 Figure 8: Output Thread 4

As seen in figures fig. 5-fig. 8, every thread outputs a part of the whole image, where the first thread has the first part of the image and the last thread outputs the last. The images can be put together in the console with the command
`convert +append out_0.pgm out_1.pgm out_2.pgm out_3.pgm out.png`.
After being put together, the image looks like it was generated from one thread fig. 9.

Also, the thread with rank zero calculates the time throughout the process, so that the benchmarks are easier to compare.

6 Performance analysis

In this section we evaluate the performance of our program. This includes assessing the strong and weak scaling behaviour. A realistic scenario with boundary conditions that include two squares in the middle is used in our benchmarks. This scenario is visualized in fig. 9. Our benchmarks are performed on the Scientific Compute Cluster (SCC)[4], which is a modern HPC cluster operated by the Gesellschaft für wissenschaftliche Datenverarbeitung mbH Göttingen (GWDG). To compile our program we used the command `mpicxx -lboost_mpi -std=c++17 main.cpp -O2 -march=native`. `-O2` enables most optimizations and produced a faster program than `-O3` in a small benchmark we

[4]SCC: `https://www.gwdg.de/web/guest/hpc-on-campus/scc`

Figure 9: Composed result of 4 threads

performed. From experience, this is often the case. The `-march=native` flag instructs the compiler to tune the generated code to, and utilize the full instruction set of, the specific architecture the compiler is executed on. Unfortunately, we compiled the code on the login node of the SCC and realized too late that the processor used on the worker nodes is slightly different. In appendix B.1 we show the exact compiler flags `-march=native` translates to. Luckily, the only difference is the `l2-cache-size` and since this applies to all benchmarks, our scaling analysis conclusions are unaffected.

We used boost version 1.77.0[5] and Open MPI 4.1.1[6]. Our compiler is GCC 9.3.0[7]. The benchmarks were executed on the amp* nodes of the SCC which have two Cascade Lake Intel Platinum 9242[8] processors each.

As mentioned in section 5.2, the program measures the relaxation time internally and outputs the result. None of the initialization or output is included in the measurement. We decided to focus on the relaxation itself because including multiple program stages would have made the results much harder to interpret, as different stages, like initialization and output, might have different scaling behaviour than the main calculation. We also determined that the calculation dominates the total runtime and that the other stages would be negligible to users of our software. Even for our benchmark with the largest output (run=0, ntasks=128 in table 9) the total runtime with 183s is only about 7s more than the relaxation time.

We repeat every benchmark configuration three times to get some statistical confidence. In addition to the benchmark based analysis, we use tools and basic performance modeling below.

[5]Boost 1.77.0: `https://www.boost.org/users/history/version_1_77_0.html`

[6]Open MPI 4.1: `https://www.open-mpi.org/software/ompi/v4.1/`

[7]GCC 9.3.0: `https://gcc.gnu.org/onlinedocs/9.3.0/`

[8]Cascade Lake Intel Platinum 9242: `https://ark.intel.com/content/www/us/en/ark/products/194145/intel-xeon-platinum-9242-processor-71-5m-cache-2-30-ghz.html`

6.1 Strong scaling

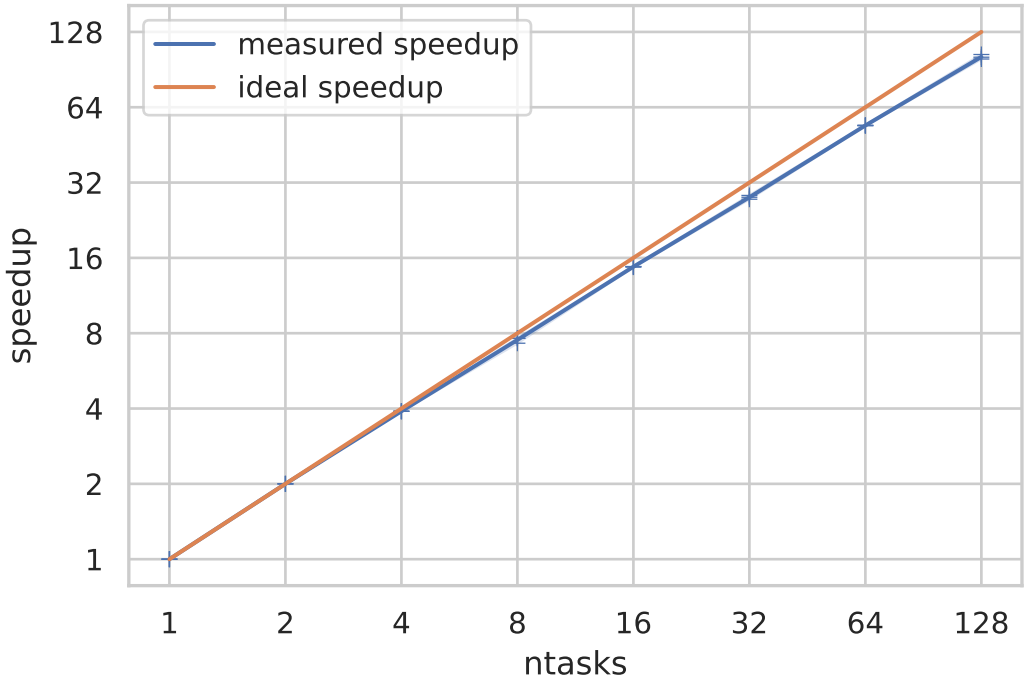

Figure 10: Strong scaling performance of our solver. The ideal speedup, shown in orange would be exactly the parallelism. The blue line shows the actually measured speedup. The thin bands around the line shows the standard deviation, calculated from the three samples of each task count.

In the strong scaling we use a fixed grid of 2500x2500 points. We run the test on 1, 2, 4, 8, 16, 32, 64 and 128 tasks. As we where using the amp nodes of the SCC Göttingen, which are containing 96 CPU cores, only the 128 thread test was run on more than one node, in fact we ran it on 4 nodes with 32 tasks each. If running the program on more than one node impacts performance, it should become clear by that test. The speedup used in fig. 10 is the calculation time of the used tasks $t(N)$ divided by the time used by one thread $t(1)$.

$$\text{speedup(N)} = \frac{t(N)}{t(1)} \tag{28}$$

The commit used for strong scaling was "strong v3". As it can be seen in fig. 10, the scaling of the program is quite good and pretty close to the ideal speedup. In the beginning the speedup is almost perfect, when reaching 16 tasks it slowly begins to fall of the ideal speedup. That can also be read from the average of the three trials average values in table 1. The full table can be found in the appendix.

One explanation for the falling values in tries with more threads can be, that different numbers of threads needed different numbers of iterations to converge. For example with one thread the program made 91707 iterations, with 16 threads it made 92515 and with 128 threads the program iterated 96252 times. This difference in iterations can explain the slightly worse scaling in higher task amount tests. The reason for that is related to

ntasks	seconds	speedup
1	4778.103	1.000
2	2388.700	2.000
4	1224.847	3.901
8	635.521	7.522
16	323.990	14.748
32	171.017	27.945
64	88.290	54.118
128	46.983	101.729

Table 1: Average values of the three strong scaling tests

in- and out-of-place calculation. Inplace calculation is almost twice as fast as outplace calculation. when using only one task, the whole grid is calculated inplace, but when the program starts running on more than one task, the borders of the tasks don't get communicated after every step in the grid calculation but after every whole iteration through the grid. That means, that the borders of the tasks are calculated as if the calculation was outplace as the border values are from the last iteration. The result is, the more threads we use, the more outplace calculation happens, leading to slightly more iterations the more threads are used. Resolving this issue is just not worth it, as the solution to this would be to communicate the border points after every row and not just after every whole iteration. That would lead to much more communication and would cost more time than it would save.

6.2 Weak scaling

Weak scaling is the ability of a parallel software to maintain efficiency while jointly increasing the problem size and parallelism. This is the definition of the weak scaling speedup or efficiency where $t(N)$ is the time it takes to complete a problem of size N with N units of parallelism[9].

$$\text{speedup(N)} = \frac{t(1)}{t(N)} \tag{29}$$

What the problem size is depends on the software in question. In our case, the smallest unit of work is one grid cell update. Thus, we need to keep the amount of grid cell updates proportional to N. We do this by setting the grid width to $n = 1000 \cdot \sqrt{N}$ and keeping the iteration count fixed since the amount of cell updates also depends on the number of iterations. We set the iteration count to 20000 manually. This is necessary because larger grids need more iterations to converge to the same precision as smaller grids. The iteration count is normally between $\mathcal{O}(n^2)$ and $\mathcal{O}(n)$ as explained in section 3. The total amount of updates performed with 20000 iterations is $n^2 \cdot 20000 = 1000^2 \cdot \sqrt{N}^2 \cdot 20000 = N \cdot 2\text{e}10$ and is therefore proportional to N, which is the parallelism *and* problem size in our benchmarks. For benchmarking, we implemented taking N as a command line parameter so that the program can adjust the grid size accordingly[10].

Apart from the above modifications, we use the same scenario as in section 6.1, also with 1, 2, 4, 8, 16, 32, 64 and 128 tasks. The results are shown in table 2. The speedup

[9] `https://hpc-wiki.info/hpc/Scaling#Weak_Scaling`

[10] Problem size as command line argument: `https://gitlab.gwdg.de/j.hoerdt/parallaplacation/-/blob/963540bc7fc00ed3135de6e67fee49334f230237/src/main.cpp#L230`

as well as the time is averaged over the three repetitions. The speedup is calculated with the average time for $N = 1$ as $t(1)$. The results are also displayed in the lower blue line in fig. 11. Please disregard the two lines in the middle for now.

N	seconds	speedup
1	165.462	1.000
2	172.536	0.959
4	189.733	0.872
8	224.550	0.737
16	325.351	0.509
32	631.740	0.262
64	1271.256	0.130
128	1669.418	0.099

Table 2: Weak scaling results. This table is produced by taking the average time over three repetitions from table 8. The speedup is calculated with the average time for $N = 1$ as $t(1)$ and averaging over the three repetitions again.

As you can see, our speed speedup falls off pretty quickly. Ideally, it should stay close to one, the green line in fig. 11. The drop in performance from $N = 16$ to $N = 32$ is especially bad. In table 2 you can see that $2 \cdot t(16) \approx t(32)$, meaning that you can almost run problem size 16 twice sequentially in the time it takes to run problem size 32. This also means that in this case, we gain close to no wall clock time improvement from completing the same work load with double the parallelism.

With Gustafson's law we can estimate the sequential portion of our software.

$$S = N + (1 - N) \cdot s \tag{30}$$

Here, S is the scaled speedup and s the estimated sequential portion. For $N = 2$ for example we get $0.959 * 2 = 2 + (1 - 2)s$. Transformed to s we get $s = 0.082$. This could potentially be explained by communication or load imbalance. For $N = 32$ however we get $s = 0.762$ which cannot be justified easily.

It is expected that the weak scaling performance falls off for high N but here it clearly falls off sooner and faster than we expected. We do not have large sequential parts in our software that could justify this falloff. Unexpectedly, to the right in fig. 11, you can see that the speedup falls off slower than before. We will determine the cause for some of these issues in the following sections.

6.3 Vampir

To verify our expectations regarding the parallel runtime behaviour of our software, identify performance problems, and potentially determine a focus for optimization efforts, we use Vampir. Vampir[11] is an application that can visualize the runtime behaviour of MPI based software with profiling and tracing data. We generate this data using Score-P[12], which is a code instrumentation tool. The instrumented code then produces profiling data during execution. After instrumenting we performed an initial profiling run without tracing. We used the same scenario as in our benchmarks, with a grid width of 2000 and 32 tasks. The resulting profile can be inspected with `scorep-score`[13]. We do not have a

[11] Vampir website: `https://vampir.eu/`

[12] Score-P: `https://www.vi-hps.org/projects/score-p/`

[13] Score-P Cheat Sheet: `https://vampir.eu/public/files/pdf/spcheatsheet_a4.pdf`

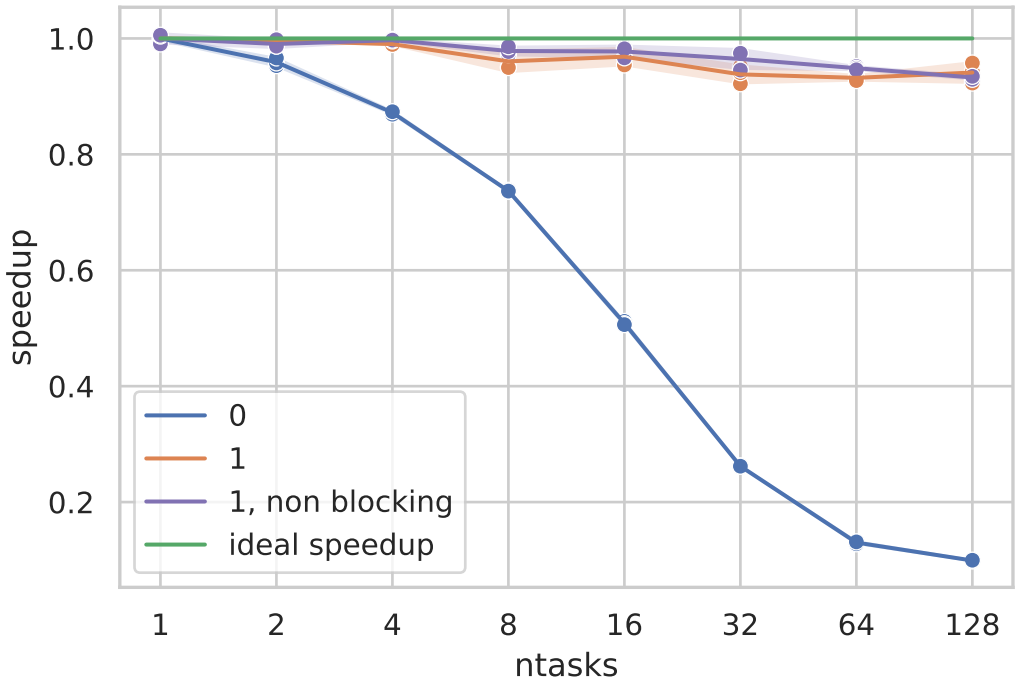

Figure 11: Weak scaling performance of our solver. The ideal speedup, shown in green would be one at any problem size / task count. The line at the bottom shows the speedup with an initial guess of zero. The orange line in the middle shows the improved speedup with an initial guess of one. The purple line mostly above, shows the further improved speedup with non-blocking communication. The thin bands around the lines show the standard deviation, calculated from the three samples of each problem size.

complex call hierarchy so the main takeaway for us is estimated size of the event trace, which is 131MB. We decided that we did not need any filtering for the tracing information and simply ran the program again, this time with tracing enabled. Now we can open the trace with Vampir and look at various visualizations. The timeline displayed in fig. 12 caught our eye immediately. In the section after `MPI_Init`, the relaxation appears to be dominated by communication overhead for a few seconds. We zoom into this section in fig. 13. As you can see, thread 19 takes about 6.7ms for one step, while the other threads are only taking around 1ms, and are directly or indirectly waiting for its results. As it turns out, the reason for the large load imbalance were denormalized floating point numbers[Gol91][14]. Denormalized floating points can represent numbers extremely close to zero, but are notorious for being slow to calculate with on modern hardware[15]. This hypothesis was plausible because we initialize the unknown values with zero, leading to lots of denormalized floating point numbers arising in calculations until they dissipate.

To quantify this effect on the SCC hardware, we constructed a small micro benchmark using Google Benchmark[16] that is shown in full in listing 12. The output is shown in

[14]754-2019 IEEE Standard for Floating-Point Arithmetic: `https://doi.org/10.1109%2FIEEESTD.2019.8766229`

[15]Why does changing 0.1f to 0 slow down performance by 10x?: `https://stackoverflow.com/a/9314926/6859487`

[16]Google Benchmark: `https://github.com/google/benchmark`

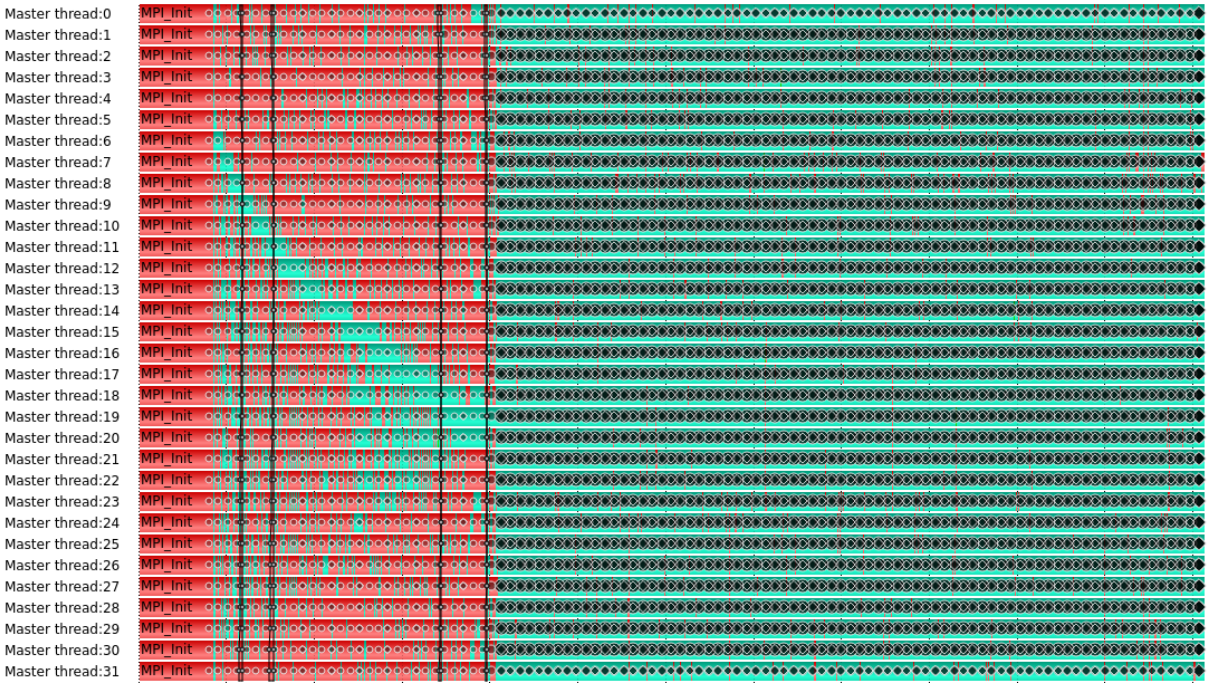

Figure 12: Vampir timeline with initial value=0. Each row is the timeline for one task. MPI communication is displayed in red. Relaxation in green.

Benchmark	Time	CPU	Iterations
normal	1.95 ns	1.94 ns	360534115
prev_denormal	1.95 ns	1.94 ns	360584168
neighbor_denormal	1.95 ns	1.94 ns	360684522
neighbors_denormal	50.9 ns	50.8 ns	13790702

Table 3: Denormalized floating point benchmark results from listing 12. Executed on an SCC amp* node.

table 3. As you can see, certain calculations involving denormalized floating point numbers are up to 26 times slower than regular calculations on the amp* node's architecture.

Looking at the Vampir visualisations we also realized that our two border exchanges run sequentially with respect to each other. This did not necessarily manifest in a performance issue, but it is an unnecessary limitation. Instead both send receive calls could run asynchronously using `MPI_Isendrecv` as a simple optimisation improving scaling behaviour in certain settings.

6.4 Improvements

6.4.1 Denormalized Floating Points

The aforementioned issue with denormalized floating point numbers was easily fixed. The initial value of the unknown values has no influence on the correctness of our algorithm. Only the convergence speed will be different from the initial value of zero depending on the scenario. Thus, we can simply change our initial guess in listing 4 to one[17], to avoid

[17]Changing the initial value in our repository: `https://gitlab.gwdg.de/j.hoerdt/parallaplacation/-/commit/0afe0ff88e24623692b3477a88348a35f87a814c`

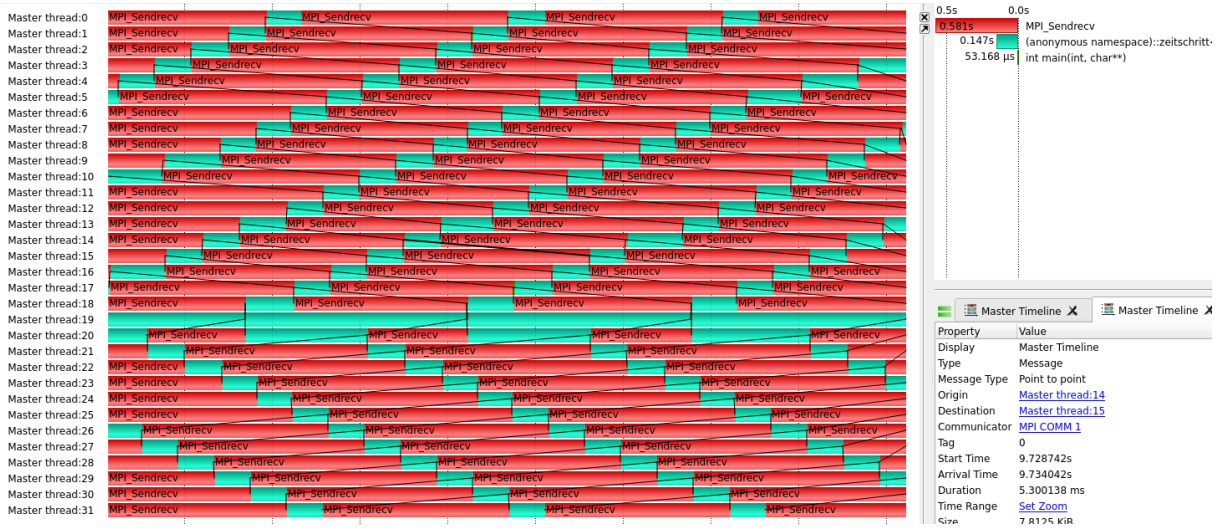

Figure 13: Zoomed Vampir timeline with initial value=0. Zoomed into the suspicious section. Each row is the timeline for one task. MPI communication is displayed in red. Relaxation in green.

the formation of denormalized floating point numbers. We repeated our weak scaling benchmarks to validate the improvement. The results are displayed in table 4 and in fig. 11, as the orange line in the middle. The weak scaling graph now looks much better

N	seconds	speedup
1	162.442	1.000
2	163.059	0.996
4	164.049	0.990
8	169.170	0.960
16	167.748	0.969
32	173.210	0.938
64	174.302	0.932
128	172.630	0.941

Table 4: Weak scaling results with an initial value of 1. This table is produced by taking the average time over three repetitions from table 9. The speedup is calculated with the average time for $N = 1$ as $t(1)$ and averaging over the three repetitions again.

and closer to our expectations, that is, it is closer to the ideal scaling for longer. We also repeated the Vampir analysis with the updated program. The resulting timeline is shown in fig. 14. Compared to fig. 12, the stage of denormalized floating point propagation is missing. After `MPI_Init` only 3.5% of time is spent in communication. The scaling graph for an initial guess of one in fig. 11 now shows a significant variation in speedup from one problem size to the next. This is likely the result of a load imbalance due to our scenario not offering equal load for equal slab widths. Depending on N, different tasks get different workloads independent of the repetition. When comparing table 4 and table 2, you may notice that the time for $N = 1$ was only improved insignificantly. This is because the problem size is small here and the denormalized calculations do not take much of the total time. For larger N, the slow calculations in some tasks result in lots of tasks idling, waiting for the results, while the problem size is much larger as well. For $N = 128$ for

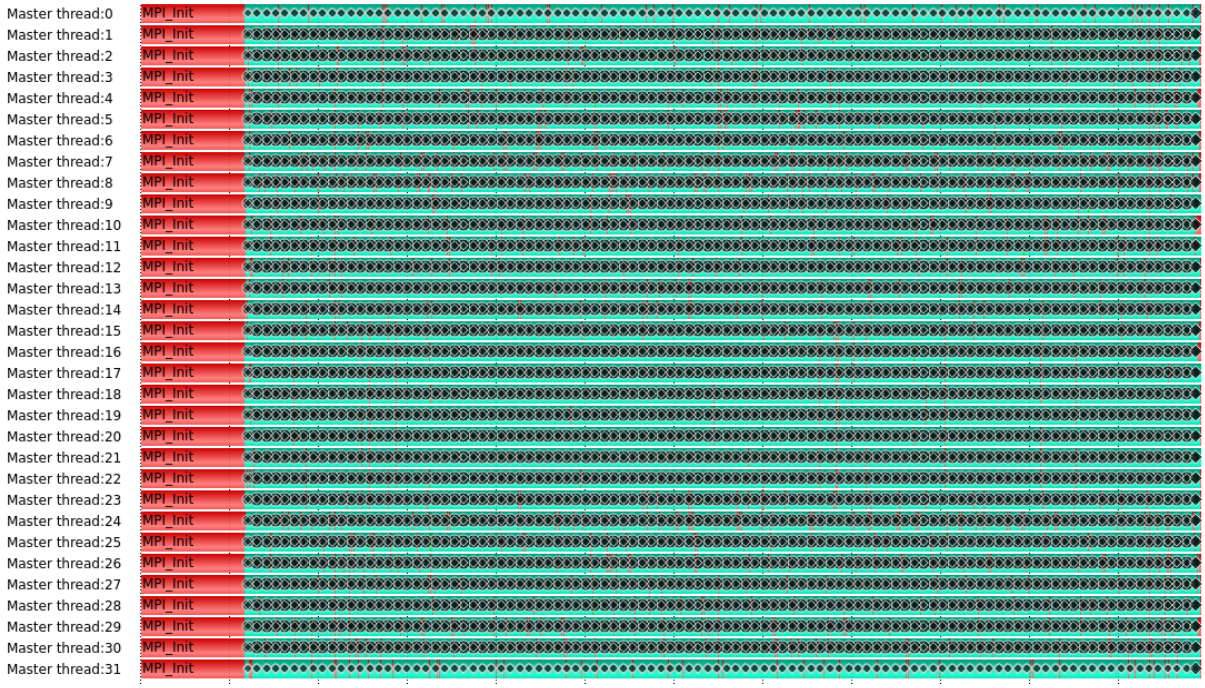

Figure 14: Vampir timeline with initial value=1. Each row is the timeline for one task. MPI communication is displayed in red. Relaxation in green.

example, the program is now 9.67 times as fast as before the change.

6.4.2 Non Blocking Communication

As mentioned in section 6.3, the boundary exchange is not performed completely asynchronously. We can fix this by using `MPI_Isend` and `MPI_Irecv`. `MPI_Isendrecv` is unfortunately not yet available on the SCC. `MPI_Waitall` is used to make sure the communication is finished before the relaxation continues. To test whether this change is an improvement, the weak scaling test was repeated again. The results are also displayed in the purple line in fig. 11. And in table 5. The change slightly improves the scaling behaviour, except for $N = 128$, where the speedup is worse than before. To determine if the change really is an improvement in most cases, more benchmarks would be needed

N	seconds	speedup
1	163.758	1.000
2	165.347	0.990
4	164.245	0.997
8	167.388	0.978
16	167.535	0.978
32	169.810	0.965
64	172.612	0.949
128	175.595	0.933

Table 5: Weak scaling results with an initial value of 1 and non blocking communication. This table is produced by taking the average time over three repetitions from table 10. The speedup is calculated with the average time for $N = 1$ as $t(1)$ and averaging over the three repetitions again.

with even higher N.

6.5 Efficiency estimate

Here we try to make a basic estimate on the efficiency of our program. To establish a baseline, according to our measurements in table 4, our software can perform 2.56e12 grid cell updates in 172.630s on 128 tasks. To estimate the ideal throughput, we look at the cell update from listing 5. Calculating the average takes three additions and a division by four. The over relaxed update takes an addition, a substraction, and a multiplication. The residual is not calculated in each step and can be ignored in this estimate. We can lookup the rough cycle counts of each of those operations for the Cascade Lake architecture used on the amp* nodes[18]. The relevant data is shown in table 6. The sum of the total cycles is

Instruction	Latency	Required count	Total Cycles
FADD	3	4	12
FSUB	3	1	3
FMUL	5	1	5
FDIV	14-16	1	14-16

Table 6: Latencies of relevant instructions on the Cascade Lake architecture taken from `https://www.agner.org/optimize/instruction_tables.pdf`. Required count is to update one grid cell. The Total cycles is the product of the Latency and the required count.

34-36. The processor used on the amp* nodes has a base frequency of 2.3GHz[19], meaning it can perform 2.3e9 cycles/s. Since we have 128 tasks, we should be able to perform $2.3e9 \cdot 128 = 2.944e11$ cycles/s. Since one grid cell update takes about 35 cycles, we should get 2.944e11/35 = 8.411e9 updates/s. The baseline benchmark ran for 172.630s, giving us an estimated capacity of $172.63 \cdot 8.411e9 = 1.452e12$ updates in total, for the baseline scenario. That means the estimated efficiency of our software is

$$\frac{2.56e12}{1.452e12} = 1.763 \tag{31}$$

How is it possible for our software to achieve an efficiency greater than one? Our estimate for the ideal throughput is a bit naïve and therefore too low. A compiler can, for example, replace the expensive division, as the divisor is known at compile time to be four, by a cheaper multiplication by 0.25[20]. If we adapt our estimate with this optimization we get an efficiency of 1.26, which is still larger than one. There are also many more instructions available to implement the grid point update more efficiently. A fused multiply add instruction for example is commonly used to perform an addition and a multiplication together and can be used here as well. Another example are Single instruction, multiple data (SIMD) instructions that can perform the same instruction for multiple values. Another reason for our low estimate could be that in practice, the processor uses a higher dynamic clock rate than the base clock rate. The processor specification lists a "Turbo" frequency of 3.8GHz. Using this in our estimate gives the efficiency 0.762. The estimate

[18]Lists of instruction latencies: `https://www.agner.org/optimize/instruction_tables.pdf`

[19]Cascade Lake Intel Platinum 9242: `https://ark.intel.com/content/www/us/en/ark/products/194145/intel-xeon-platinum-9242-processor-71-5m-cache-2-30-ghz.html`

[20]How a grid point step may be compiled: `https://godbolt.org/z/vaG4eGK3o`

could be refined more, but assuming that it is not completely off, it is reassuring to see that our measured throughput is close to what would correspond to a basic assembly implementation and does not obviously leave a large chunk of performance on the table.

6.6 Roofline Model with LIKWID

Like I Knew What I'm Doing (LIKWID) is a performance tool suite[21] for Linux. We will use it to construct a basic roofline model of our software as described in a tutorial on the LIKWID wiki[22].

First we determine the peak floating point performance on an amp* node with the command `likwid-bench -t peakflops_sp -W N:32kB:1`. We get 6143.69 MFlops/s. The L1 cache size was determined with `likwid-topology`. Then the bandwith limit is determined with the command `likwid-bench -t stream_sp -W N:2GB:`. The result is 11399.34 MB/s. We use the stream kernel, because it is close to the read/write ratio used in our application, as is recommended in the tutorial. Next, we determine the arithmetic intensity of our application. Normally, you would do this with the `MEM_SP` group, but this is not supported on the SCC. Instead the MFlops/s of our application is optained with the command `likwid-perfctr -C 0 -g FLOPS_SP main 1`. The result is 848.2630 MFlop/s. This is divided by 2959.2310 MB/s, which is the L2 memory bandwidth of our application obtained with the command `likwid-perfctr -C 0 -g L2 main 1`, to get the arithmetic intensity of 0.286 Flops/B. These findings are plotted in fig. 15.

[21] LIKWID: `https://hpc.fau.de/research/tools/likwid/`

[22] Empirical Roofline with LIKWID: `https://github.com/RRZE-HPC/likwid/wiki/Tutorial:-Empirical-Roofline-Model`

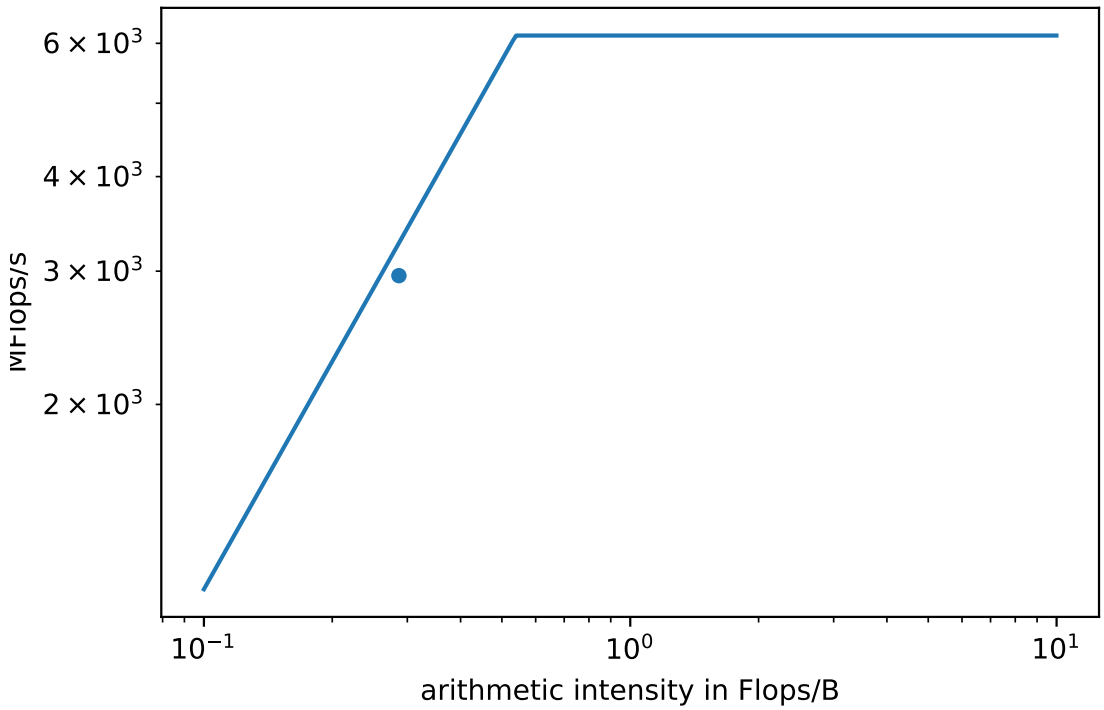

Figure 15: Basic roofline model of our application. The line is the roof consisting of the memory bound and the CPU bound. The dot represents our application.

As you can see, the program is well in the memory bound area, that is to the left of the roof intersection. This means, to optimize the program, we should firstly look into reducing the amount of memory touched by the application during the relaxation. The first target could be listing 3, where we store, in the worst case, two more 64bit indices for every grid cell. This could easily be halved by converting it to one index into the flattened data. There are also alternative approaches to keeping track of what cells are mutable. It would for example be possible to store the mutability of grid cells in a dynamic bit set, reducing the storage for mutability to 1/128.

7 Discussion

While coding the sequential version, not a lot of problems occurred. The prototype was working quite fast and also unexpectedly well. While in the beginning we where still very convinced by the idea of writing the program n-dimensional, this idea moved into distant future after thinking about it and realising all the problems it would bring.

Before starting to parallelize, we invested a lot of time to find our way of parallelizing the problem. While two options of dividing the grid where shown in the report, we had some more ideas that where even more complex and not feasible at all. We where thinking about all kinds of settings the program could work with, how we would handle input data with circles and how that would impact the grid division. An even bigger challenge was, to find a way of exchanging the borders of the threads. We where thinking about ways to only make the exchange every five iterations by having a border area of 10 grid points

that would have not been calculated in every iteration. Also we where thinking about calculating the border points with only three neighboring values. After some thinking the ideas came out to be unpractical or just wrong in logical terms, as many others we had.

After we decided to make the parallel version as easy as possible with exchanging the borders after every iteration and dividing the grid in one dimension, the parallel version of the program also was already quite fast. While finding the right size to make performance tests was a little chaotic, the strong scaling tests and their results where quite satisfying. However the weak scaling tests where kind of sobering as our program seemed to scale quite bad. Finding the reason for that was quite time consuming and took us until we made performance analysis with vampir. After overcoming the denormalised floating point problem, the performance increase was even bigger than expected, making the program as performant as we expected it to be.

Of course, dividing the grid in a different way would maybe solve the problem in a more effective way. Also, we could have done more research on the relaxation-factor, as the one we have maybe is not the ideal one but rather the save option. With a higher relaxation factor, the grid converges faster, but the risk for destabilisation increases. We did not invest a lot of time finding the perfect factor or even making the factor dependent on the scenario and the grid size as we would have to just try a lot of different factors and scenarios. Another method would be to detect destabilisation in the calculation and restart it with adjusted relaxation factor, this method only would be effective for very large problem sizes. Last, we didn't use openMP in our project. Maybe by diving into the possibilities of openMP there would be ways to solve the problem in another, more effective way.

Overall, the code works as intended. The version we wrote is only suitable for experimental purposes and not for usage in science. The code does what we intended, in acceptable time, but the handling is quite bad. It's not intuitive to input a certain scenario without the code accepting a file and the user having to choose their longest dimension as x is also a lot to expect. The fact that the code outputs numerous files instead of one makes the results quite hard readable. When talking about scientific usage, it's very important to mention that there is another, more effective method of calculating the grid. With the multigrid[Han92] technique, a program would first start with a very coarse grid and converge it. Then every grid point would be split into four points and the program would iterate until the grid is converged. Repeating this process until the desired accuracy is reached solves a problem of n^2 grid points in $O(n^2)$ time. A scientifically useful program would have to use this method in order to be effective. Apart from that, the program is a very good result for students trying to parallelize a problem, as the parallelization works quite good and also fast as shown.

8 Conclusion

Summarized, we wanted to program a numerical solver to the laplace equation and parallelize it. It was very important to us to find and solve the denormalized floating point issue, as the solve brought a huge performance rise and it showed the influence of good analysis on a project. In our opinion, the project was definitely successful and we learned a lot about the usage of MPI and different performance measurement methods. The achievement of the project is a, in our opinion, quite well parallelized program that works totally as intended. For the future one could imagine to make the program work with

n-dimensional problems. Also the output format could be a little better as the pgm format generates quite big files. The program also could take data from a file and read it, to make it more usable and intuitive. And lastly, a more adapted relaxation factor would improve the program.

References

[Bur04] Klemens Burg. *Partielle Differentialgleichungen*. ger. Springer eBook Collection | Life Science and Basic Disciplines. 1 Online-Ressource (XIV, 419 S. 234 Abb). Wiesbaden: Vieweg+Teubner Verlag, 2004. ISBN: 978-3-322-96788-6. DOI: https://doi.org/10.1007/978-3-322-96788-6.

[Dem13] Wolfgang Demtröder. *Experimentalphysik 2 : Elektrizität und Optik*. ger. Springer-Lehrbuch. Online-Ressource (XVI, 482 S.), Ill., graph. Darst. Berlin: Springer, 2013. ISBN: 978-3-642-29944-5. DOI: https://doi.org/10.1007/978-3-642-29944-5.

[Dem18] Wolfgang Demtröder. "Experimentalphysik 1 : Mechanik und Wärme". In: 2018.

[FLS13] Richard P. Feynman, Robert B. Leighton, and Matthew L. Sands. *The Feynman lectures on physics : Volume II ; mainly electromagnetism and matter*. eng. Online-Ressource. Pasadena, Calif.: Calif. Inst. of Techn., 2013. URL: http://feynmanlectures.caltech.edu/II_toc.html.

[Gol91] David Goldberg. "What Every Computer Scientist Should Know about Floating-Point Arithmetic". In: *ACM Comput. Surv.* 23.1 (Mar. 1991), pp. 5–48. ISSN: 0360-0300. DOI: 10.1145/103162.103163. URL: https://doi.org/10.1145/103162.103163.

[Han92] Per Brinch Hansen. "Numerical Solution of Laplace's Equation". In: School of Computer and Information Science, Syracuse University, 1992. URL: https://surface.syr.edu/cgi/viewcontent.cgi?article=1160&context=eecs_techreports.

A Work sharing

In general, we divided the workload so that Jakob Hördt, for who this is a 6C module, did $\frac{6}{11}$ of the work and Philipp Müller, for who this is a 5C module, did $\frac{5}{11}$ of the work. The code was created almost completely in a pair-programming fashion. We decided that, for the purpose of this course, it would be beneficial for both of us to have a good understanding of every part of the code. Furthermore, our project is not easily split into loosely coupled modules that could be implemented in parallel.

A.1 Jakob Hördt (21565573)

In the presentation, I am solely responsible for the sections 1, 2, and 6. In chapter 5 Performance Analysis, I made the Weak Scaling, and Vampir Slides. In this report, I am responsible for sections 1, 2, 3, 4, and 6 except 6.1.

A.2 Philipp Müller (21874297)

I was responsible for showing and explaining functions we used in the sequential version of the Program. Also I talked through the parallelization process and how we made the program suitable for parallel computing. In the performance analysis chapter I presented the strong scaling setup, what we found out when running the tests and the graph. In the report, i mainly made the implementation chapter, also i made the strong scaling and Discussion/Conclusion. It is to mention, that Jakob made the Graphs as i made the tables.

B Extras

B.1 march=native

The following is the list of flags activated by `-march=native` on the SCC's gwdu101 login node[23]:

```
-march=cascadelake -mmmx -mno-3dnow -msse -msse2 -msse3 -mssse3 -mno-
   sse4a -mcx16 -msahf -mmovbe -maes -mno-sha -mpclmul -mpopcnt -mabm -
   mno-lwp -mfma -mno-fma4 -mno-xop -mbmi -mno-sgx -mbmi2 -mno-pconfig -
   mno-wbnoinvd -mno-tbm -mavx -mavx2 -msse4.2 -msse4.1 -mlzcnt -mrtm -
   mhle -mrdrnd -mf16c -mfsgsbase -mrdseed -mprfchw -madx -mfxsr -mxsave
    -mxsaveopt -mavx512f -mno-avx512er -mavx512cd -mno-avx512pf -mno-
   prefetchwt1 -mclflushopt -mxsavec -mxsaves -mavx512dq -mavx512bw -
   mavx512vl -mno-avx512ifma -mno-avx512vbmi -mno-avx5124fmaps -mno-
   avx5124vnniw -mclwb -mno-mwaitx -mno-clzero -mpku -mno-rdpid -mno-
   gfni -mno-shstk -mno-avx512vbmi2 -mavx512vnni -mno-vaes -mno-
   vpclmulqdq -mno-avx512bitalg -mno-movdiri -mno-movdir64b -mno-waitpkg
    -mno-cldemote -mno-ptwrite --param l1-cache-size=32 --param l1-cache
   -line-size=64 --param l2-cache-size=16896 -mtune=cascadelake
```

For the amp* nodes of the SCC, `-march=native` means:

[23] How to see which flags -march=native will activate? https://stackoverflow.com/a/9355840/6859487

```
-march=cascadelake -mmmx -mno-3dnow -msse -msse2 -msse3 -mssse3 -mno-
   sse4a -mcx16 -msahf -mmovbe -maes -mno-sha -mpclmul -mpopcnt -mabm -
   mno-lwp -mfma -mno-fma4 -mno-xop -mbmi -mno-sgx -mbmi2 -mno-pconfig -
   mno-wbnoinvd -mno-tbm -mavx -mavx2 -msse4.2 -msse4.1 -mlzcnt -mrtm -
   mhle -mrdrnd -mf16c -mfsgsbase -mrdseed -mprfchw -madx -mfxsr -mxsave
    -mxsaveopt -mavx512f -mno-avx512er -mavx512cd -mno-avx512pf -mno-
   prefetchwt1 -mclflushopt -mxsavec -mxsaves -mavx512dq -mavx512bw -
   mavx512vl -mno-avx512ifma -mno-avx512vbmi -mno-avx5124fmaps -mno-
   avx5124vnniw -mclwb -mno-mwaitx -mno-clzero -mpku -mno-rdpid -mno-
   gfni -mno-shstk -mno-avx512vbmi2 -mavx512vnni -mno-vaes -mno-
   vpclmulqdq -mno-avx512bitalg -mno-movdiri -mno-movdir64b -mno-waitpkg
    -mno-cldemote -mno-ptwrite --param l1-cache-size=32 --param l1-cache
   -line-size=64 --param l2-cache-size=36608 -mtune=cascadelake
```

The only difference is the `l2-cache-size`

B.2 tables

run	ntasks	seconds
0	1	4782.52
0	2	2389.87
0	4	1224.67
0	8	654.902
0	16	325.027
0	32	170.921
0	64	88.4413
0	128	47.8932
1	1	4771.96
1	2	2387.66
1	4	1224.91
1	8	625.569
1	16	323.568
1	32	168.084
1	64	88.364
1	128	47.1057
2	1	4779.83
2	2	2388.57
2	4	1224.96
2	8	626.091
2	16	323.374
2	32	174.047
2	64	88.0648
2	128	45.949

Table 7: Strong scaling raw

run	ntasks	seconds
0	1	165.762344
0	2	172.762192
0	4	190.349182
0	8	224.815445
0	16	325.866577
0	32	630.836548
0	64	1292.10791
0	128	1666.6947
1	1	164.955917
1	2	171.230942
1	4	189.405746
1	8	224.637131
1	16	326.867523
1	32	631.773438
1	64	1260.13025
1	128	1659.80957
2	1	165.669189
2	2	173.613983
2	4	189.443008
2	8	224.197281
2	16	323.318146
2	32	632.610168
2	64	1261.52869
2	128	1681.75012

Table 8: Weak scaling raw, init=0

run	ntasks	seconds
0	1	162.65715
0	2	163.306564
0	4	163.852249
0	8	165.561234
0	16	170.229568
0	32	176.22464
0	64	174.078827
0	128	175.994644
1	1	162.430923
1	2	163.197266
1	4	164.15062
1	8	170.919006
1	16	165.231659
1	32	170.761032
1	64	173.647476
1	128	172.252151
2	1	162.236786
2	2	162.236786
2	4	164.143143
2	8	171.029953
2	16	167.783752
2	32	172.644867
2	64	175.180984
2	128	169.644684

Table 9: Weak scaling raw, init=1

run	ntasks	seconds
0	1	163.0914
0	2	164.115677
0	4	164.474503
0	8	167.559402
0	16	169.437469
0	32	168.041367
0	64	172.641815
0	128	176.177872
1	1	162.860611
1	2	165.973907
1	4	164.236237
1	8	166.138412
1	16	166.73439
1	32	173.218842
1	64	173.119232
1	128	175.214767
2	1	165.321793
2	2	165.952759
2	4	164.023834
2	8	168.466171
2	16	166.432159
2	32	168.169037
2	64	172.074234
2	128	175.391861

Table 10: Weak scaling raw, init=1, non-blocking communication

C Code samples

```cpp
#include <limits>
#include <benchmark/benchmark.h>
auto grid_point_step(float a, float b, float c, float d, float prev_result) {
        const auto average = (a + b + c + d) / 4;
        const auto result = prev_result + 1.7 * (average - prev_result);
        return result;
}
static void normal(benchmark::State& state) {
        volatile float a = 1, b = 1, c = 1, d = 1;
        volatile float prev_result = 1;
        for (auto _ : state) {
                const auto result = grid_point_step(a, b, c, d, prev_result);
                benchmark::DoNotOptimize(result);
        }
}
BENCHMARK(normal);
static void prev_denormal(benchmark::State& state) {
        volatile float a = 1, b = 1, c = 1, d = 1;
        volatile float prev_result = std::numeric_limits<float>::denorm_min();
        for (auto _ : state) {
                const auto result = grid_point_step(a, b, c, d, prev_result);
                benchmark::DoNotOptimize(result);
        }
}
BENCHMARK(prev_denormal);
static void neighbor_denormal(benchmark::State& state) {
        volatile float a = 1, b = 1, c = 1;
        volatile float d = std::numeric_limits<float>::denorm_min();
        volatile float prev_result = 1;
        for (auto _ : state) {
                const auto result = grid_point_step(a, b, c, d, prev_result);
                benchmark::DoNotOptimize(result);
        }
}
BENCHMARK(neighbor_denormal);
static void neighbors_denormal(benchmark::State& state) {
        volatile float a = std::numeric_limits<float>::denorm_min();
        volatile float b = std::numeric_limits<float>::denorm_min();
        volatile float c = std::numeric_limits<float>::denorm_min();
        volatile float d = std::numeric_limits<float>::denorm_min();
        volatile float prev_result = 1;
        for (auto _ : state) {
                const auto result = grid_point_step(a, b, c, d, prev_result);
                benchmark::DoNotOptimize(result);
        }
}
BENCHMARK(neighbors_denormal);
```

Listing 12: Denormalized floating point benchmark.