

OTH Regensburg

A parallel solution of the two-dimensional heat expansion

Student:

Markus Heimerl

Professor:

Dr. rer. nat., Dipl.-Inf.
HACKENBERG

24.07.2021

Content

INTRODUCTION	3
Problem statement	3
Objective of the work	3
Solution strategy	3
Super MUC NG	4
Structure	4
Parallelism	4
Cooling system.....	5
EXPERIMENT.....	7
Presentation of the solution	7
Program entry and matrix initialization	7
Iterative calculation with the Jacobi method	8
Exchange of results between processes: MPI_Allgather	9
Evaluation	11
Tests with different compiler options	11
Automated testing through Bash scripts	16
IMPROVEMENT IDEA AND CONCLUSION	18

Introduction

Problem statement

The task was to simulate the two-dimensional heat expansion in a plate. The heat equation is a partial differential equation and reads as follows

$$\frac{\partial}{\partial t}u(\vec{x}, t) - a\Delta u(\vec{x}, t) = 0,$$

Where $u(\vec{x}, t)$ is the temperature at the point in \vec{x} time t , Δ the Laplace operator with respect to \vec{x} and the constant $a > 0$ is the thermal diffusivity of the medium ¹.

In addition, this expansion should be parallelized on the *Super MUC NG* in Munich with the help of the MPI Library.

Objective of the work

My intention was to first complete an iterative, sequential implementation in C++ and then parallelize it.

The Gauss-Seidl and Jacobian methods are two ways to solve linear systems of equations. The latter is easier to parallelize, and therefore seems to be a good approach here.

The goal should be to create a parallelization that is faster than a non-parallelized offline single-process solution optimized for speed.

Solution strategy

I had already parallelized a matrix multiplication during the course of the semester as part of an exercise sheet. Here, my approach was to distribute matrix A line-by-line, evenly across the available threads. I wanted to reuse as much of my code implemented there as possible.

¹ <https://de.wikipedia.org/wiki/W%C3%A4rmeleitungsgleichung> (Retrieved 05.06.21)

Super MUC NG

Structure

It is a supercomputer and currently the fifth fastest computer in Europe and 15th in the world rankings.² It consists of 6336 thin compute nodes, each with 48 cores and 96GB of RAM, and 144 thick compute nodes, each with 48 cores and 768GB of RAM. In total, the *Super MUC NG* has 311040 compute cores with 719TB of RAM and achieves a peak performance of 26.9 PetaFlop/s. All nodes are equipped with Intel Xeon Skylak processors. The internal interconnect is a fast OmniPath network with 100 Gbit/s.

The computing nodes are bundled in 8 domains, so-called islands. Within an island, the OmniPath network topology is a so-called *thick tree*, which ensures particularly fast communication. Between the islands, the OmniPath network is trimmed in speed by a factor of 1 to 4.³

In addition, there are 115 compute nodes in the *Compute Cloud*⁴, 32 of which are equipped with two Nvidia Tesla V100 GPUs each, and one node is a huge compute unit with 6TB of memory and 192 cores.

Parallelism

The parallelism is made available to the programmer via the MPI Library. The so-called *Message Passing Interface* is a standard of which there are many implementations. It describes the message exchange in parallel computations on distributed computer systems. It specifies a collection of operations and their semantics, i.e. a programming interface, but not a concrete protocol and implementation. However, the code remains the same for each thread. Rather, there is an MPI call in the standard that returns the thread number.⁵

By using these MPI calls, all threads execute their, via *ifelse check of* the thread number, part of the code and communicate with each other.

A node has the capability for actual parallelism using the cores of its processor. The processor itself and the operating system take care of the distribution of processes to these cores. The OmniPath network of the *Super MUC NG*, however, makes it possible to communicate between nodes and thus between processes of different processors in the same way as with processes within a processor. Thus, many more actual (instead of *scheduled*) parallel processes can be used than would be possible on a PC.

How many nodes and processes (tasks) are used is defined via the job configuration file. Here there are the arguments *nodes* and *ntasks*.⁶ It is also important to select the correct partition. The maximum and minimum number of cores available for a job depends on this.⁷

² <https://top500.org/lists/top500/list/2020/11/?page=1> (Retrieved 05.06.21)

³ <https://doku.lrz.de/display/PUBLIC/Hardware+of+SuperMUC-NG> (Retrieved 05/06/21)

⁴ <https://doku.lrz.de/display/PUBLIC/Compute+Cloud+of+SuperMUC-NG> (Retrieved 05/06/2021)

⁵ https://de.wikipedia.org/wiki/Message_Passing_Interface (Retrieved 05/06/2021)

⁶ <https://hpc-support.lboro.ac.uk/slurm-nodes-cpus-tasks.html> (Retrieved 05/06/2021)

⁷ <https://doku.lrz.de/display/PUBLIC/Job+Processing+with+SLURM+on+SuperMUC-NG> (Retrieved 05/06/2021)

Cooling system

The *Super MUC NG* is water-cooled and avoids the emissions and costs of an air-cooling system. Specifically, a hot water-cooling system developed by IBM is used, which cools the cooling water with the help of the outside temperature.⁸ Thus, although the water can reach up to 40° Celsius, this is still sufficient to cool the processors to a safe operating temperature. The LRZ expects to save several million euros in cooling costs with the help of these systems.⁹

IBM's *iDataPlex® Direct Water Cooled dx360 M4 cluster* makes all this possible. The heat can then be used to heat adjacent rooms.¹⁰

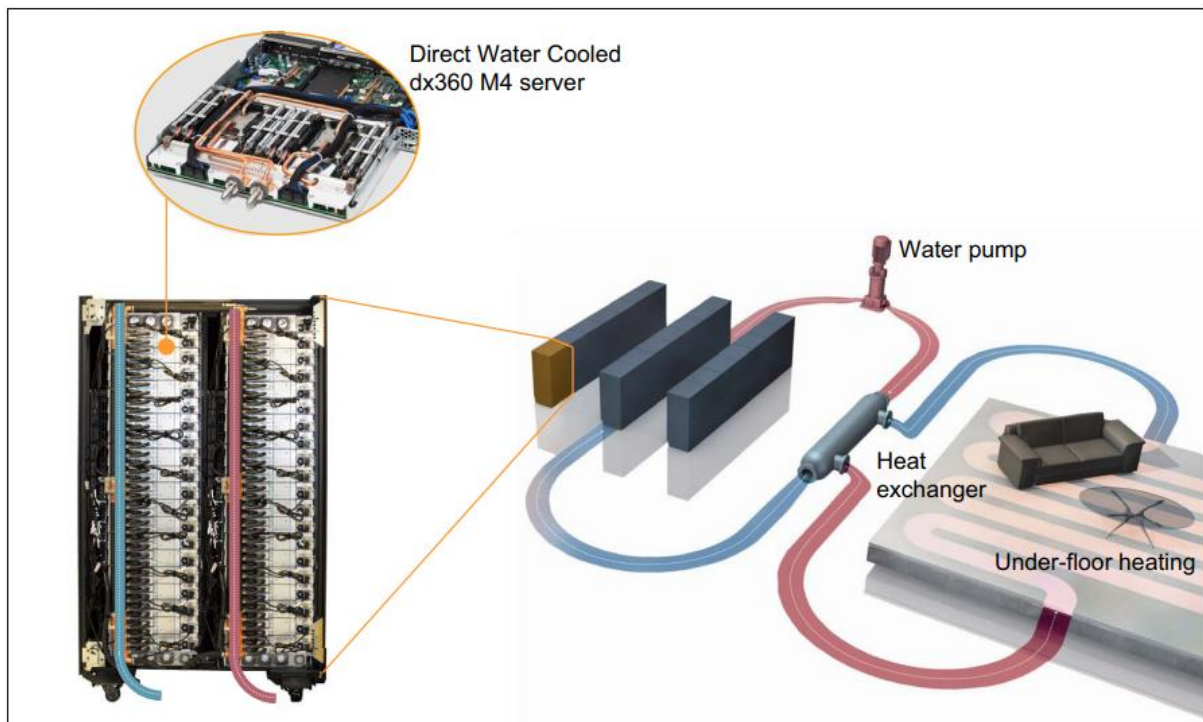


Figure 1 Direct water-cooling technology; Direct heating utilization of waste heat

Due to the efficient heat transfer to water, cooling elements become smaller and therefore cheaper and more resource efficient.

⁸ https://www.lrz.de/presse/ereignisse/2020-11-25_Strom-sparen-mit-warmem-Wasser-und-Daten/ (Retrieved 06/06/2021)

⁹ <https://www.lrz.de/services/compute/museum/supermuc/systemdescription/> (Retrieved 06/06/2021)

¹⁰ <https://lenovopress.com/sg247629.pdf> (page 36; accessed 06.06.2021)

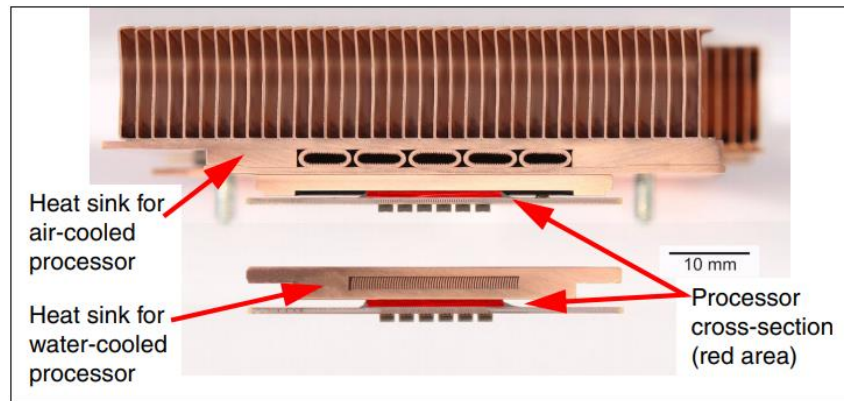


Figure 2 Section through an air-cooled system (top) and a water-cooled system (bottom)

A single server rack can get by with as little as 70 ml to cool all the processors, voltage regulators, and *Intel Platform Controller Hub* mounted on it.

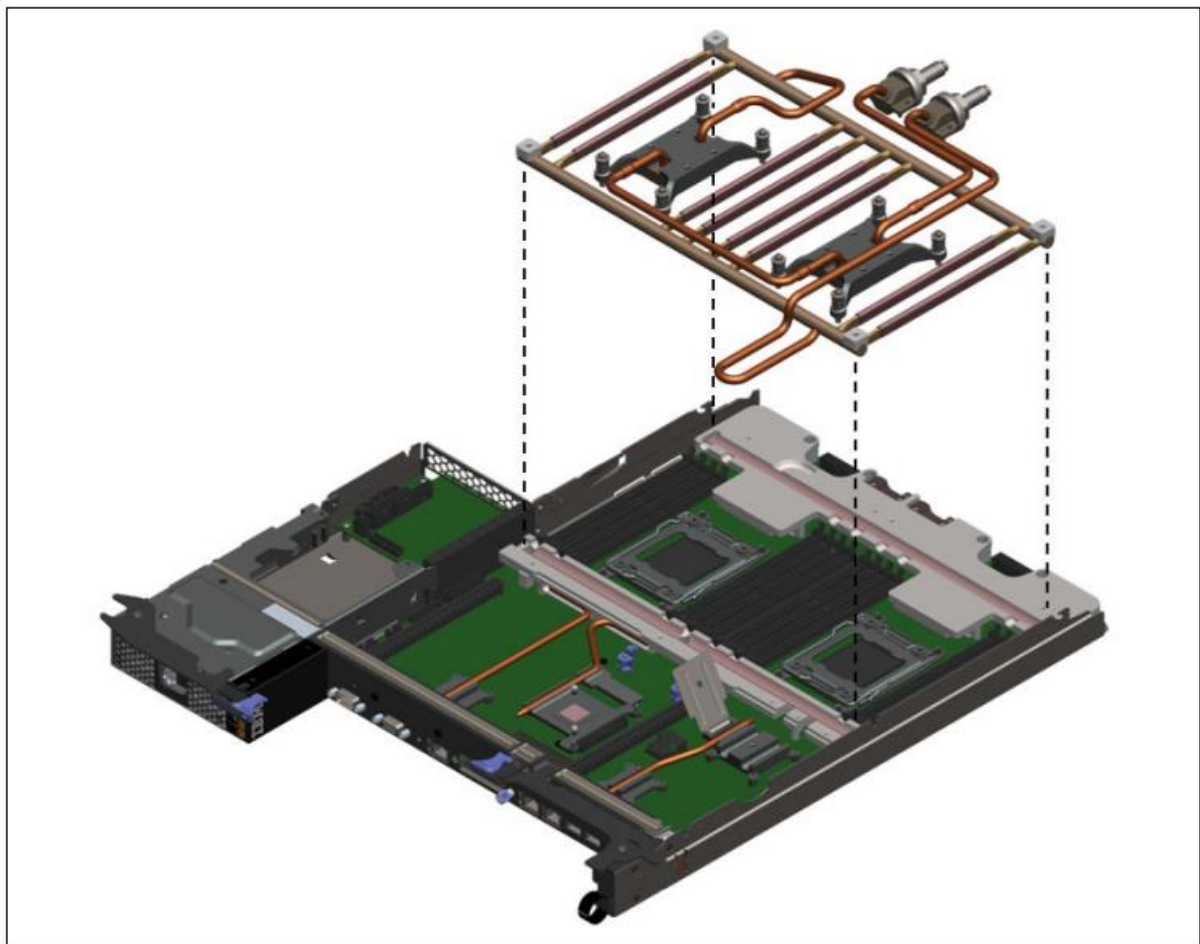


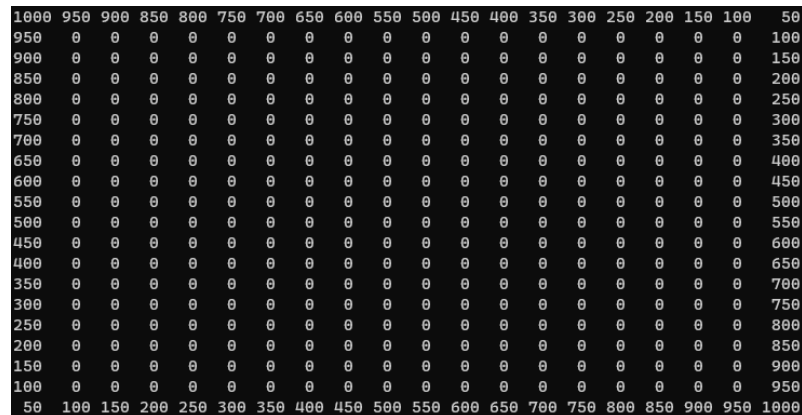
Figure 3 Water circuit of a Direct Water Cooled dx360 M4 server

Experiment

Presentation of the solution

Program entry and matrix initialization

In order to solve the heat equation, boundary conditions must apply. I have given, according to the specification of the study work, to the left upper and the right lower corner in each case a temperature of 1000. Then I filled the edges linearly descending from 1000 to 1000/n.



1000	950	900	850	800	750	700	650	600	550	500	450	400	350	300	250	200	150	100	50
950	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	100
900	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	150
850	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	200
800	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	250
750	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	300
700	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	350
650	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	400
600	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	450
550	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	500
500	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	550
450	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	600
400	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	650
350	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	700
300	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	750
250	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	800
200	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	850
150	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	900
100	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	950
50	100	150	200	250	300	350	400	450	500	550	600	650	700	750	800	850	900	950	1000

Figure 4 Initialization of a 20x20 matrix according to specified boundary conditions

```
46 int main(int argc, char **argv){
47
48     int rank = 0, size = 0;
49     MPI_Init(&argc, &argv);
50     MPI_Comm_rank(MPI_COMM_WORLD, &rank);
51     MPI_Comm_size(MPI_COMM_WORLD, &size);
52
53     // --- MATRIX INITIALISATION ---
54     int dimT[] = {atoi(argv[1]), atoi(argv[2])};
55
56     double* T = new double[dimT[0]*dimT[1]];
57     for(int i = 0; i < dimT[0]; i++){
58         for(int j = 0; j < dimT[1]; j++){
59             T[dimT[1]*i + j] = 0.0;
60         }
61     }
62
63     double scale = 1000.0;
64     T[dimT[1]*0 + 0] = scale;
65     T[dimT[1]*(dimT[0]-1) + (dimT[1]-1)] = scale;
66
67     // horizontal
68     for(int i = 1; i < dimT[1]; i++){
69         double val = scale/dimT[1] * (double)((dimT[1]-1)-i);
70
71         T[dimT[1]*0 + (i-1)] = val;
72         T[dimT[1]*(dimT[0]-1) + (dimT[1]-1-(i-1))] = val;
73     }
74
75     // vertical
76     for(int i = 1; i < dimT[0]; i++){
77         double val = scale/dimT[0] * (double)((dimT[0]-1)-i);
78
79         T[(i-1) + 0] = val;
80         T[(dimT[0]-1-(i-1)) + (dimT[1]-1)] = val;
81     }
82
83     double* T_old = new double[dimT[0]*dimT[1]];
84     memcpy(T_old, T, dimT[0]*dimT[1]*sizeof(double));
85
86     double* T_new = new double[dimT[0]*dimT[1]];
87     memcpy(T_new, T, dimT[0]*dimT[1]*sizeof(double));
88     // ---
```

Figure 5 Program entry and matrix initialization

This initialization is also the start of my code. All MPI processes execute it for themselves.

First *MPI_Init*¹¹ is called to prepare the library for further calls and then the "rank" or the number of the MPI process is¹² determined via *MPI_Comm_rank* and stored in the variable "rank". The number of all started MPI processes is then still determined with *MPI_Comm_size*¹³ and stored in the variable "size".

The dimensions are defined via the parameters passed to the program.

A matrix of this size is created at the heap and initialized with 0. The corners are then assigned 1000 and the edges are calculated first horizontally and then vertically for the given matrix in the same way as in Figure 4.

¹¹ https://www.mpich.org/static/docs/v3.2.1/www3/MPI_Init.html (Retrieved 07/07/2021)

¹² https://www.mpich.org/static/docs/v3.3.x/www3/MPI_Comm_rank.html (Retrieved 07/07/2021)

¹³ https://www.mpich.org/static/docs/latest/www3/MPI_Comm_size.html (Retrieved 07/07/2021)

Finally, "T_old" and "T_new" are created with the same dimensions and content as "T".

Iterative calculation with the Jacobi method

```

123 // --- SETZE BERECHNUNGSVARIABLEN ---
124 double error = 1.0;
125 int n_iteration = 1;
126 double c = 0.1;
127 double delta_s = 1.0 / float(dimT[0]-1);
128 double delta_t = ((delta_s*delta_s)/(4*c)); // delta_t muss kleiner gleich (delta_s^2)/(2*c) sein
129 // --- ---
130
131 double starttime = MPI_Wtime(); // Startpunkt der Zeitmessung
132
133 while(error > TOLERANCE && n_iteration < MAX_ITERATIONS){
134
135     DEBUGCODE(std::cout << "--- Iteration: " << n_iteration << " ---" << std::endl);
136
137     // --- BERECHNE MATRIXZEILEN ---
138     for(int j = 1; j < dimT[1]-1; j++){
139         for(int i = rowsperthread*rank; i < rowsperthread+(rowsperthread*rank); i++){
140             if(i == 0 || i == (dimT[0]-1)) continue; // "Continue" bei den Rändern. Die sind fest.
141             // jakobi
142             T[dimT[1] * i + j] = T_old[dimT[1] * i + j] + c * (delta_t/(delta_s*delta_s)) *
143             (T_old[dimT[1] * (i-1) + j] + T_old[dimT[1] * (i+1) + j] - 4*T_old[dimT[1] * i + j]
144             + T_old[dimT[1] * i + (j-1)] + T_old[dimT[1] * i + (j+1)]);
145         }
146     }
147     // --- ---
148
149     // --- TAUSCHE ERGEBNISSE MIT ANDEREN THREADS AUS ---
150     MPI_Allgather(T+rowsperthread*rank*dimT[1], rowsperthread*(dimT[1]), MPI_DOUBLE,
151     T_new, rowsperthread*(dimT[1]), MPI_DOUBLE, comm_useable_threads);
152     // --- ---
153
154     // --- ERMITTLE DEN FEHLER ---
155     double maximum = 0.0;
156     for(int i = 0; i < dimT[0]; i++){
157         for(int j = 0; j < dimT[1]; j++){
158             double sub = fabs(T_new[dimT[1] * i + j] - T_old[dimT[1] * i + j]);
159             if(sub > maximum) maximum = sub;
160         }
161     }
162     error = maximum;
163     // --- ---
164
165     memcpy(T_old, T_new, dimT[0]*dimT[1]*sizeof(double));
166     n_iteration++;
167
168     DEBUGCODE(std::cout << "--- ---\n" << std::endl);
169 }
170
171 double endtime = MPI_Wtime(); // Ende der Zeitmessung

```

Figure 6 Main loop, matrix value calculation, interprocess communication and time measurement

Calculation variables are set here initially. The first variable, "error", holds the largest error of the last iteration. It holds one of the two termination conditions for the while loop in line 133. The second, "n_iteration", holds the number of iterations that have already gone into the calculation. It forms the second termination condition. TOLRANCE and MAX_ITERATIONS are macros that are given the values 0.0001 and 1000. Especially to perform comparable tests, it was important to set an iteration count limit, instead of an error tolerance limit. In comparing different matrix sizes with different numbers of processors, all tests ran into the low 1000 iteration limit. Reaching the tolerance limit requires significantly more iterations for all matrix sizes tested.

The variables "c", "delta_s" and "delta_t" are variables for the Jacobi method.¹⁴

In line 131 the time measurement starts. This measurement was used for all tests.

Inside the while loop, the nested for loops begin. The first one with the count variable "j". This represents the current column of the matrix and starts at 1, because the left column with index 0 is

¹⁴ <https://www.cosy.sbg.ac.at/events/parnum05/book/horak1.pdf> (page 48; accessed 07/22/2021)

fixed. It runs as long as it is smaller than "dimT[1]-1". I.e., it runs up to the second last column, because the right column of the matrix may not be changed again. The second for-loop with the count variable "i" iterates the rows of the matrix. Here one of the two central characteristics of parallelization can be found. Those rows that are calculated depend on the "rank" variable. Thus, of it, which number the executing process possesses. The variable "rowspertthread" is declared before with the division between number of rows of the matrix and number of available processes. Here it is to be assumed at first that this division is always to be accomplished without remainder. So, the start index is calculated from the multiplication between "rank" and "rowspertthread". The first process number is 0, so process 0 starts at the very first row. This and the last one is protected from change with the "continue" in the for loop. The index, which represents the last row to be calculated, is just the index of the first calculated row plus "rowspertthread". Afterwards the Jacobi calculation follows:

$$u_{i,j}^{k+1} = u_{i,j}^k + c \cdot \frac{\Delta t}{(\Delta s)^2} \left(u_{i+1,j}^k + u_{i-1,j}^k - 4u_{i,j}^k + u_{i,j+1}^k + u_{i,j-1}^k \right)$$

Figure 7 Formula for the Jacobi method (source: footnote 15)

The matrix "T" describes the part of the new matrix $u_{i,j}^{k+1}$ which is calculated by the process "rank". The matrix "T_old" contains the total state of the last iteration, i.e. $u_{i,j}^k$. You can see here the special difference between the Jacobi and the Gauss-Seidel process: For the calculation of a new matrix cell value, only surrounding cell values of the past matrix are necessary.

Exchange of results between processes: MPI_Allgather

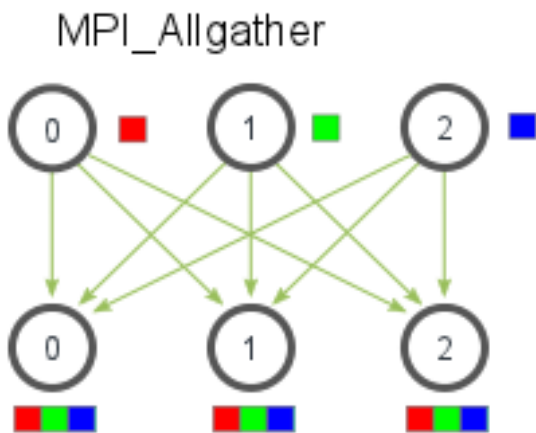


Figure 8 Visualization of MPI_Allgather ¹⁶

Central to my parallelization solution is the MPI function *MPI_Allgather*¹⁵. This can best be understood as an answer to the following problem: Each process has unique data. All processes should receive the data of all other processes.

But this is not the problem of the Jacobi parallelization. Figure 7 shows that not every process needs every line. Mainly processes calculate with their

¹⁵ https://www.mpich.org/static/docs/v3.3.x/www3/MPI_Allgather.html (Retrieved 07/22/2021)

¹⁶ <https://mpitutorial.com/tutorials/mpi-scatter-gather-and-allgather/> (Retrieved 07/22/2021)

self-determined data from the last iteration. Only the first and last row of each process part matrix must be sent with the processes that calculate the rows before and after said process part matrix. Figure 9 illustrates this. Process 2 needs the upper pink cell from the memory of process 1 to calculate the turquoise cell. Thus, to fully compute the first blue row, process 2 needs only the last row from process 1, nothing more. Process 1 also does not need any values from process 0 at all.

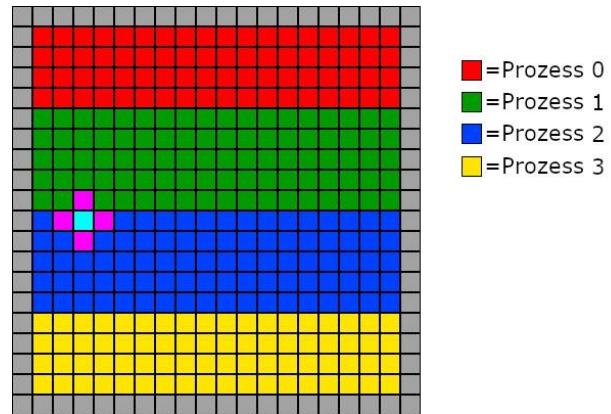


Figure 9 Visualization of required data for a cell

However, there is no MPI function that specifically solves this problem. My first solution therefore used the primitive functions *MPI_Send*¹⁷ and *MPI_Recv*¹⁸. However, it is not trivial to coordinate these commands efficiently in time. Before a send is made, a process must already be expecting this message with a receive. In addition, an implementation with primitive functions involves more lines and is thus error-prone. The solution with *MPI_Allgather* is not optimal, due to the large amount of communications not needed, but it is short, simple, clean and handles any synchronization between processes itself.

The call itself uses the matrices "T" and "T_new". There are three matrices in total:

1. Matrix "T": Contains the results of the row calculations of process "rank" of the current iteration.
2. Matrix "T_old": Contains the entire matrix state of the last iteration or, in case of the first iteration, the initialization state.
3. Matrix "T_new": Contains the entire matrix state of the current iteration **after the MPI_Allgather call**.

```

149 // --- TAUSCHE ERGEBNISSE MIT ANDEREN THREADS AUS ---
150 MPI_Allgather(T+rowspertthread*rank*dimT[1], rowspertthread*(dimT[1]), MPI_DOUBLE,
151               T_new, rowspertthread*(dimT[1]), MPI_DOUBLE, comm_useable_threads);
152 // ---

```

Figure 10 MPI_Allgather call for communication between processes

The first parameter is the start memory address of the data to be sent. It is calculated here for each process by pointer arithmetic. The second parameter is the number of values to be sent. The third parameter describes the type of the values. Parameter number four is the memory address where all data should be joined. Parameter number five is the number of expected values per process. Each process then has the entire matrix with all current values itself in "T_new".

Since each process owns all current cell values, each process can also calculate the error, without further communication via a *MPI_Reduce*¹⁹ for example. In line 165ff. the whole state is copied from

¹⁷ https://www.mpich.org/static/docs/latest/www3/MPI_Send.html (Retrieved 23/07/2021)

¹⁸ https://www.mpich.org/static/docs/latest/www3/MPI_Recv.html (Retrieved 23/07/2021)

¹⁹ https://www.mpich.org/static/docs/v3.2.1/www3/MPI_Reduce.html (Retrieved 23/07/2021)

"T_new" to "T_old", the number of iterations is counted up and the next iteration is entered or the while loop is acknowledged.

```

--- t = 850 ---
1000 950 900 850 800 750 700 650 600 550 500 450 400 350 300 250 200 150 100 50
950 985.263 860.526 815.789 771.052 726.315 681.578 636.841 592.104 547.367 502.63 457.894 413.157 368.42 323.683 278.947 234.21 189.473 144.737 100
900 860.526 821.052 781.578 742.104 702.63 663.156 623.682 584.208 544.735 505.261 465.787 426.314 386.84 347.367 307.893 268.42 228.947 189.473 150
850 815.789 781.578 747.367 713.156 678.945 644.734 610.523 576.312 542.102 507.891 473.681 439.471 405.26 371.05 336.84 302.63 268.42 234.21 200
800 771.052 742.104 713.156 684.208 655.26 626.312 597.364 568.417 539.469 510.522 481.575 452.628 423.68 394.734 365.787 336.84 307.893 278.947 250
750 726.315 702.63 678.945 655.26 631.575 607.89 584.206 560.521 536.837 513.153 489.469 465.785 442.101 418.417 394.734 371.05 347.367 323.683 300
700 681.578 663.156 644.734 626.312 607.89 589.469 571.047 552.626 534.205 515.783 497.363 478.942 460.521 442.101 423.68 405.26 386.84 368.42 350
650 636.841 623.682 610.523 597.364 584.206 571.047 557.889 544.73 531.572 518.414 505.257 492.099 478.942 465.785 452.628 439.471 426.314 413.157 400
600 592.104 584.208 576.312 568.417 560.521 552.626 544.73 536.835 528.94 521.046 513.151 505.257 497.363 489.469 481.575 473.681 465.787 457.894 450
550 547.367 544.735 542.102 539.469 536.837 534.205 531.572 528.94 526.309 523.677 521.046 518.414 515.783 513.153 510.522 507.891 505.261 502.63 500
500 502.63 505.261 507.891 510.522 513.153 515.783 518.414 521.046 523.677 526.309 528.94 531.572 534.205 536.837 539.469 542.102 544.735 547.367 550
450 457.894 465.787 473.681 481.575 489.469 497.363 505.257 513.151 521.046 528.94 536.835 544.73 552.626 560.521 568.417 576.312 584.208 592.104 600
400 413.157 426.314 439.471 452.628 465.785 478.942 492.099 505.257 518.414 531.572 544.73 557.889 571.047 584.206 597.364 610.523 623.682 636.841 650
350 368.42 386.84 405.26 423.68 442.101 460.521 478.942 497.363 515.783 534.205 552.626 571.047 589.469 607.89 626.312 644.734 663.156 681.578 700
300 323.683 347.367 371.05 394.734 418.417 442.101 465.785 489.469 513.153 536.837 560.521 584.206 607.89 631.575 655.26 678.945 702.63 726.315 750
250 278.947 307.893 336.84 365.787 394.734 423.68 452.628 481.575 510.522 539.469 568.417 597.364 626.312 655.26 684.208 713.156 742.104 771.052 800
200 234.21 268.42 302.63 336.84 371.05 405.26 439.471 473.681 507.891 542.102 576.312 610.523 644.734 678.945 713.156 747.367 781.578 815.789 850
150 189.473 228.947 268.42 307.893 347.367 386.84 426.314 465.787 505.261 544.735 584.208 623.682 663.156 702.63 742.104 781.578 821.052 860.526 900
100 144.737 189.473 234.21 278.947 323.683 368.42 413.157 457.894 502.63 547.367 592.104 636.841 681.578 726.315 771.052 815.789 860.526 905.263 950
50 100 150 200 250 300 350 400 450 500 550 600 650 700 750 800 850 900 950 1000
---

```

Figure 11 Result of a 20x20 matrix with tolerance of 0.001

Evaluation

Tests with different compiler options

The tests of the program included five matrix sizes, each of which was run with ten processors of different sizes. The focus of my evaluation was the difference between the results with the "trace" compiler flag, which supplements the program to write trace analyzer data to disk during runtime, and those with the "ofast" compiler flag, which optimizes the program for speed. All tests run for 1000 iterations of the while loop (see Figure 6). All matrices are square, so matrix size n is both the number of rows and the number of columns. The runtime t is always measured in seconds.

As a benchmark I wrote an implementation without the MPI library and compiled it with gcc 9.3.0. Also here I used the "ofast" flag. The runtimes are from an Intel Core i5-9600K CPU @ 3.70GHz.

Matrix size n	512	1024	2048	4096	8192
Running time t [s]	1,287	10,281	65,592	326,615	1460,012

The goal of MPI parallelization should be to run 1000 iterations faster than this single-processor application, starting from a certain number of processors p.

Trace - Values

	p=1	p=2	p=4	p=8	p=16	p=32	p=64	p=128	p=256	p=512
n=512	2,9273	6,2143	4,0691	2,9057	2,5897	2,6612	3,1324	3,4992	3,9295	4,7425
n=1024	12,164	24,766	21,421	14,719	13,001	12,475	14,132	15,517	15,856	30,00
n=2048	102,21	208,93	89,768	71,764	58,242	57,551	53,459	61,398	63,244	68,205
n=4096	456,35	1144,2	664,45	419,32	244,57	232,56	221,76	273,03	257,91	253,27
n=8192	1880,1	4557,0	2649,9	1700,9	1218,4	1087,4	923,35	1113,0	1156,4	1289,5

With the "Trace" flag, the MPI implementation could only hold its own from n=2048. It finds its fastest executions between 16 and 64 processes. The reason for increasing runtimes after this is the

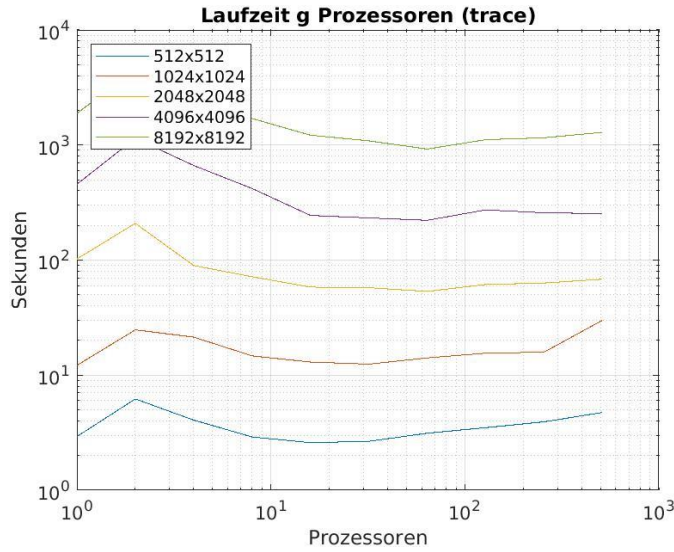


Figure 12 Logarithmic graph of MPI-trace values

large amount of unnecessarily transmitted matrix rows and increasing amount of basic MPI function calls (e.g., *MPI_Send*) in *MPI_Allgather*. Larger matrices have their fastest executions with more processes (64), than smaller matrices, because the amount of computation per process increases strongly with increasing matrix size and only with these higher processor numbers the optimal relationship between *MPI_Allgather* and Jacobi computation is found. The jump with two processes has to do with the fact that with one processor no communication between processes must be logged and thus no time is lost.

Compared to the execution of one process on the supercomputer, the parallelization is faster when its speedup is less than one. This happened for every matrix size except 1024. Here, the logging overhead of interprocess communication was too high to give a speedup advantage over a process with parallelization. Test i has a speedup s, for a processor count p, like:

$$s_i = \frac{time_p^i}{time_1^i}$$

The efficiency e of a test is calculated as follows:

$$e_i = \frac{s_i}{p}$$

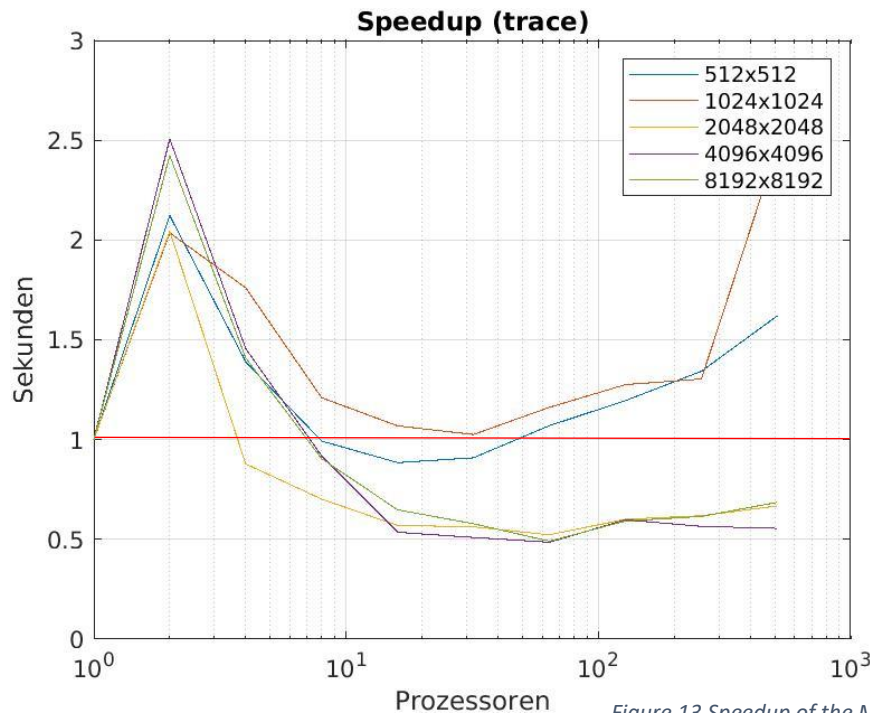
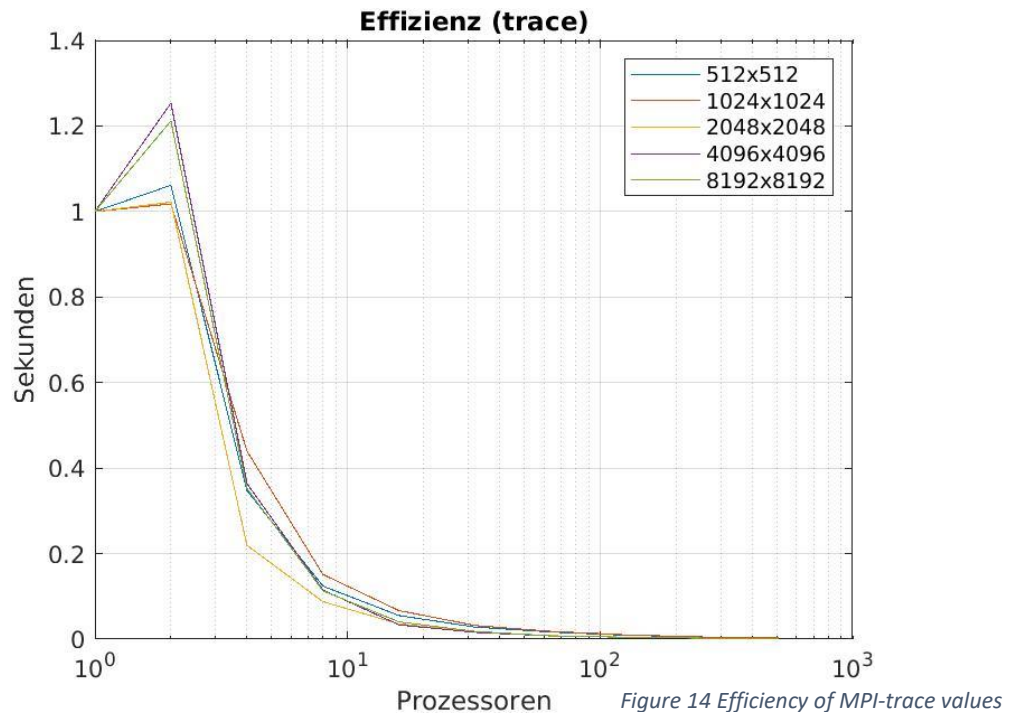


Figure 13 Speedup of the MPI-trace values



With the help of the Intel Trace Analyzer²⁰, analysis data about the execution can be obtained.

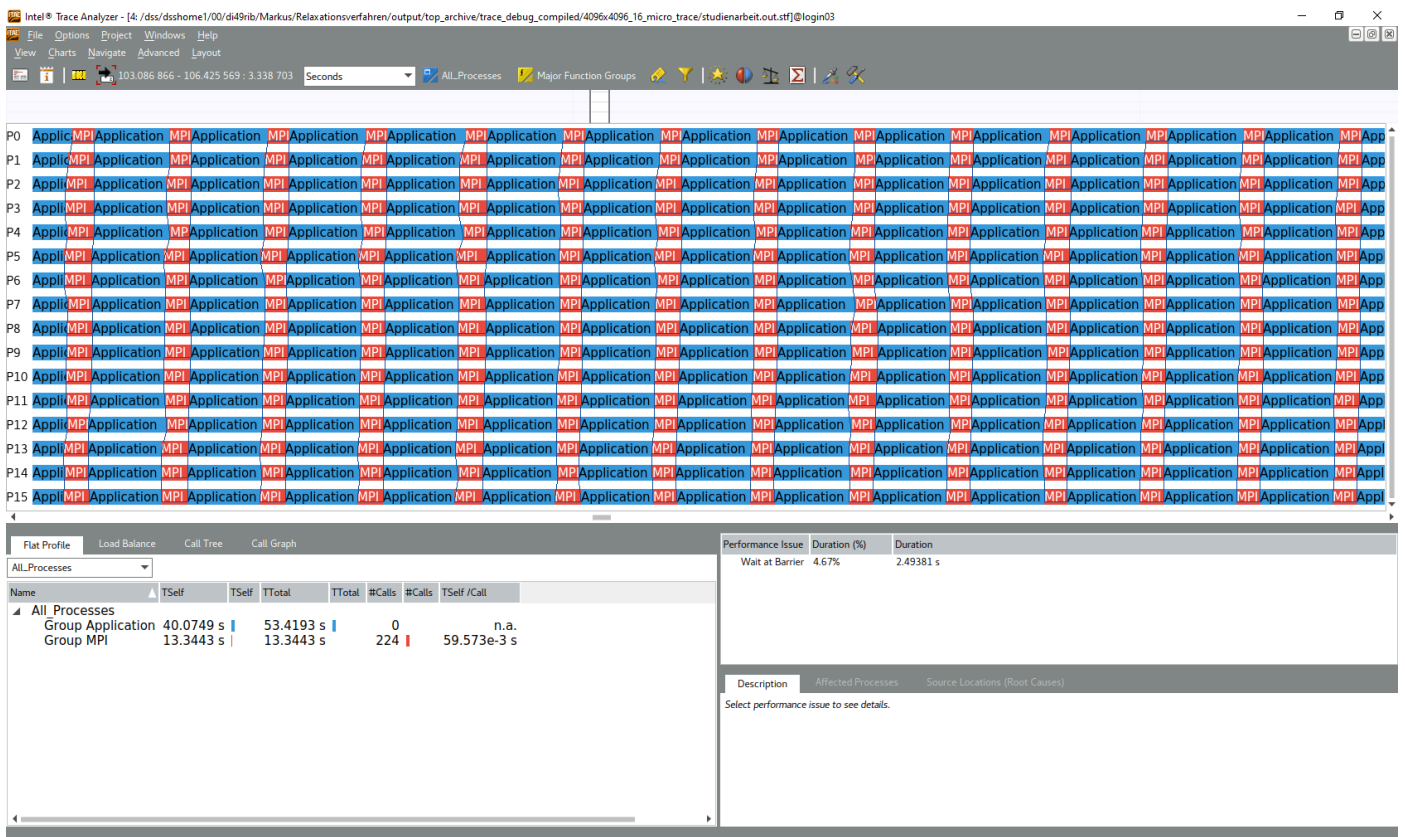


Figure 15 Trace file of the 4096x4096 matrix with 16 processes

²⁰ <https://software.intel.com/content/www/us/en/develop/tools/oneapi/components/trace-analyzer.html>
(Retrieved 24/07/2021)

Of particular interest is the time share of the total execution time consumed by MPI functions and the time share that the processes had to wait for each other ("*Wait at Barrier*"). Here ideas for runtime optimization can be found.

As can be seen in Figure 15, the time share of MPI functions increases with a larger number of processors. This is to be expected, since larger matrices are exchanged (see chapter *Exchange of results between processes: MPI_Allgather*). The fraction of time wasted in waiting between processes increases with the number of processors for most matrix sizes. This is to be expected since there is a higher probability per *MPI_Allgather* that a process is out of synchronization. I attribute the almost inverted behavior of the matrix size 4096 to a measurement error.

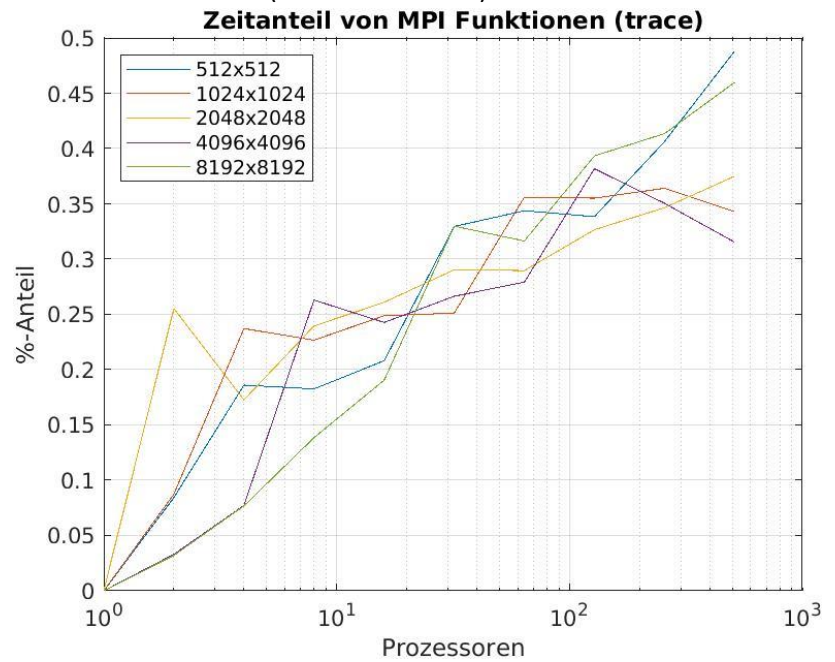


Figure 16 Matlab graph of MPI time fraction

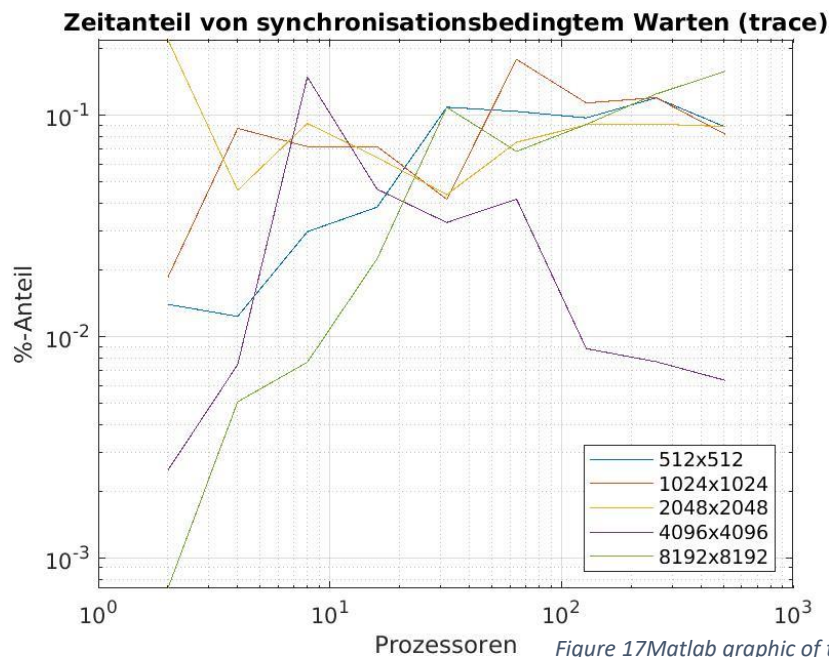


Figure 17 Matlab graphic of the Busy Waits

Ofast - values

	p=1	p=2	p=4	p=8	p=16	p=32	p=64	p=128	p=256	p=512
n=512	2,9092	1,9150	1,4054	1,0593	1,0500	1,5454	1,9371	2,7810	3,4166	3,8355
n=1024	11,831	8,4897	7,1661	6,9390	6,6986	8,6747	9,1859	13,046	13,264	13,953
n=2048	103,79	47,610	37,639	32,431	30,434	39,268	37,312	50,663	53,523	61,562
n=4096	458,83	274,64	192,31	143,93	125,23	156,13	153,52	232,60	219,32	206,29
n=8192	2778,2	1138,5	810,87	671,08	587,96	673,24	636,53	992,47	1044,3	1146,1

Parallelization has beaten the single-processor solution for every matrix size: **success!** On average, it is twice as fast for the ideal number of processors:

$$\frac{\sum_{i=1}^5 \frac{time_i^{gcc}}{mintime_i^{mpi}}}{5} = 2,0014$$

It has always been fastest with 16 processes. But it would be expected that with larger matrix dimensions, the number of processors for the fastest execution would increase. It does here, but it would have needed larger matrices to see it in a discrete processor scale. On the SuperMUC NG, however, these were always stopped by a scheduler. I suspect they exceeded a memory limit per job that I don't know about.

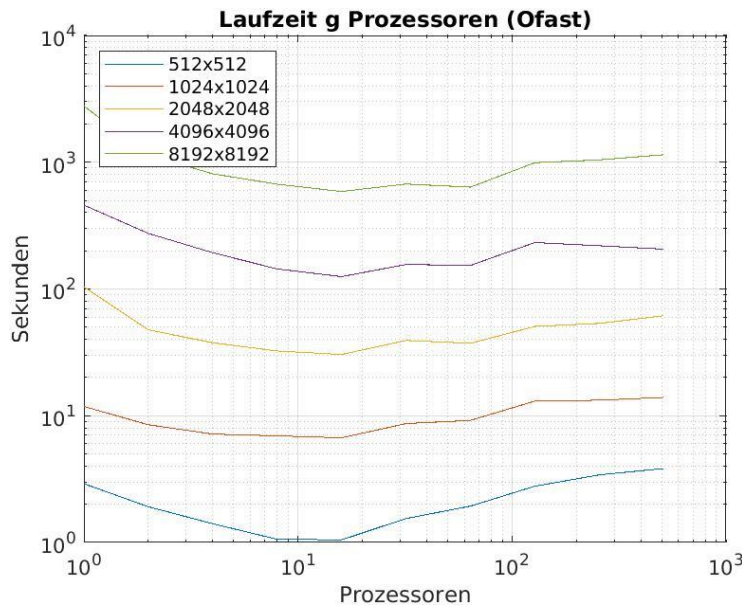


Figure 18 Logarithmic graph of MPI Fast values

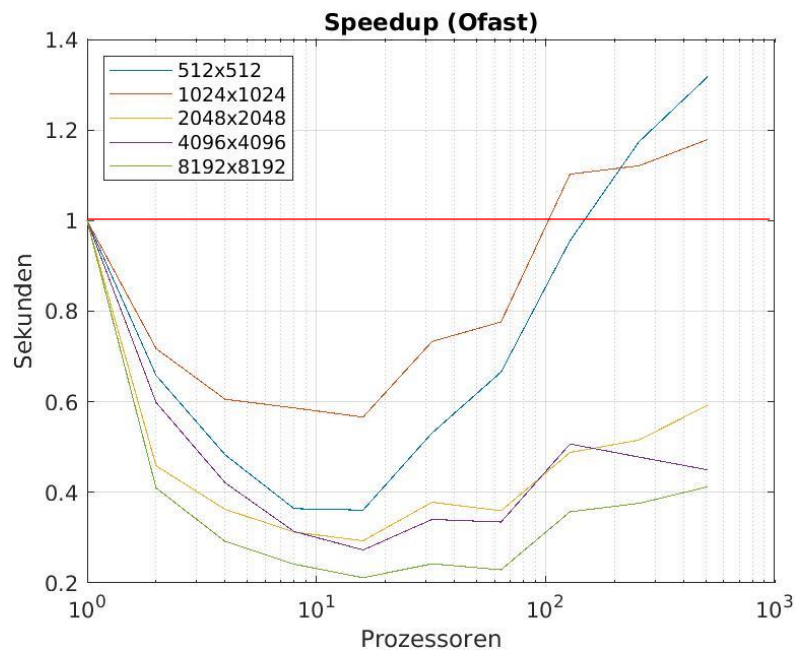


Figure 19 Matlab Visualization of Ofast Speedup

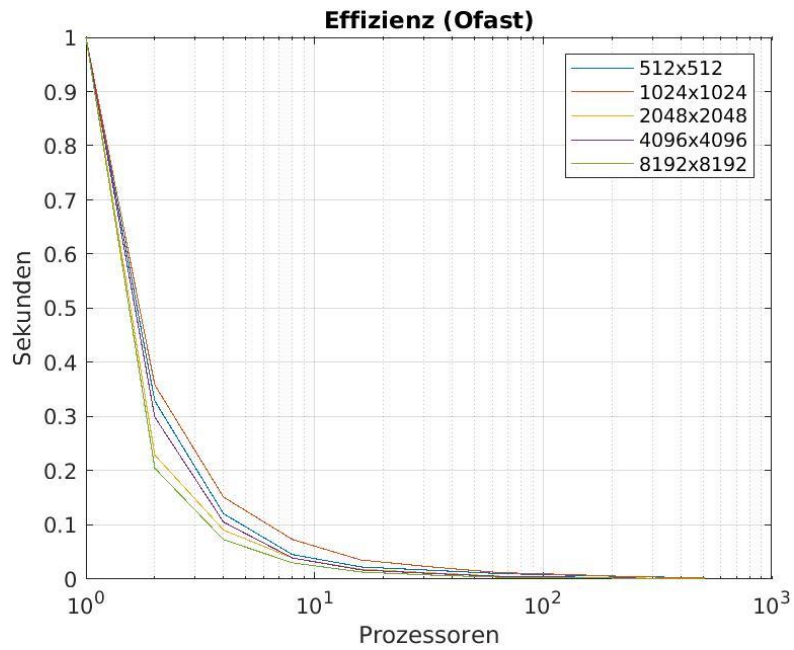


Figure 20 Matlab visualization of ofast efficiency

Automated testing through Bash scripts

```

.
|--cancel.sh
|--check.sh
|--exec_big_benchmark.sh
|--output
|--scripts
|   |--compile.sh
|   |--run.sh
|--studienarbeit.cpp
|--trace_analyzer.sh
di49rib@login01:~/Markus/Relaxationsverfahren>

```

I was only able to accumulate the large amount of data by automating testing through bash scripts. The script "check.sh" and "cancel.sh" would respectively output or cancel all active and queued jobs.

```

di49rib@login01:~/Markus/Relaxationsverfahren> cat cancel.sh
#!/bin/bash
scancel -u di49rib
di49rib@login01:~/Markus/Relaxationsverfahren> cat check.sh
#!/bin/bash
squeue --user di49rib

```

Figure 22 Contents of Check and Cancel

I created the job file inside the "run.sh" script depending on command line parameters. The only script I had to call from the command line was "exec_big_benchmark.sh".


```

di49rib@login01:~/Markus/Relaxationsverfahren> cat exec_big_benchmark.sh
#!/bin/bash

module unload intel-mkl
module unload intel-mpi
module unload intel
module load intel-parallel-studio
module load slurm_setup

compilation_mode="trace"
#compilation_mode="ofast"

batchrun() {
    /dss/dsshome1/00/di49rib/Markus/Relaxationsverfahren/scripts/run.sh 1 micro $1 $2 $3 $compilation_mode
    /dss/dsshome1/00/di49rib/Markus/Relaxationsverfahren/scripts/run.sh 2 micro $1 $2 $3 $compilation_mode
    /dss/dsshome1/00/di49rib/Markus/Relaxationsverfahren/scripts/run.sh 4 micro $1 $2 $3 $compilation_mode
    /dss/dsshome1/00/di49rib/Markus/Relaxationsverfahren/scripts/run.sh 8 micro $1 $2 $3 $compilation_mode
    /dss/dsshome1/00/di49rib/Markus/Relaxationsverfahren/scripts/run.sh 16 micro $1 $2 $3 $compilation_mode
    /dss/dsshome1/00/di49rib/Markus/Relaxationsverfahren/scripts/run.sh 32 micro $1 $2 $3 $compilation_mode
    /dss/dsshome1/00/di49rib/Markus/Relaxationsverfahren/scripts/run.sh 64 micro $1 $2 $3 $compilation_mode
    /dss/dsshome1/00/di49rib/Markus/Relaxationsverfahren/scripts/run.sh 128 micro $1 $2 $3 $compilation_mode
    /dss/dsshome1/00/di49rib/Markus/Relaxationsverfahren/scripts/run.sh 256 micro $1 $2 $3 $compilation_mode
    /dss/dsshome1/00/di49rib/Markus/Relaxationsverfahren/scripts/run.sh 512 micro $1 $2 $3 $compilation_mode
}

/dss/dsshome1/00/di49rib/Markus/Relaxationsverfahren/scripts/compile.sh $compilation_mode

# --- RUNNABLE ---
batchrun 512 512 00:30:00
batchrun 1024 1024 00:30:00
batchrun 2048 2048 00:30:00
batchrun 4096 4096 00:15:00
batchrun 8192 8192 00:15:00
# --- ---

```

Figure 23 Content of execbigbenchmark.sh

```

di49rib@login01:~/Markus/Relaxationsverfahren> cat scripts/run.sh
#!/bin/bash

# rm -r /dss/dsshome1/00/di49rib/Markus/Relaxationsverfahren/output/$3x$4_$1_$2_$6
mkdir /dss/dsshome1/00/di49rib/Markus/Relaxationsverfahren/output/$3x$4_$1_$2_$6

sbatch <<EOT
#!/bin/bash
#SBATCH -J $3_$1_$6
#SBATCH -o ./output.out
#SBATCH -e ./error.err
#SBATCH -D /dss/dsshome1/00/di49rib/Markus/Relaxationsverfahren/output/$3x$4_$1_$2_$6
#SBATCH --mail-type=NONE
#SBATCH --time=$5
#SBATCH --no-queue
#SBATCH --export=NONE
#SBATCH --get-user-env
#SBATCH --account=pr92ki
#SBATCH --partition=$2
#SBATCH --ntasks=$1

mpirun -n $1 /dss/dsshome1/00/di49rib/Markus/Relaxationsverfahren/output/compiled/studienarbeit.out $3 $4
EOT

```

Figure 24 Content of run.sh

```

di49rib@login01:~/Markus/Relaxationsverfahren> cat scripts/compile.sh
#!/bin/bash

if [[ $1 == "trace" ]]
then
    mpiCC -trace -g -tcollect -o ~/Markus/Relaxationsverfahren/output/compiled/studienarbeit.out ~/Markus/Relaxationsverfahren/studienarbeit.cpp
elif [[ $1 == "ofast" ]]
then
    mpiCC -Ofast -o ~/Markus/Relaxationsverfahren/output/compiled/studienarbeit.out ~/Markus/Relaxationsverfahren/studienarbeit.cpp
else
    echo "Invalid compilation mode"
    exit 1;
fi

chmod -R 777 ~/Markus/Relaxationsverfahren/output/compiled

```

Figure 25 Content of compile.sh

```

di49rib@login02:~/Markus/Relaxationsverfahren> ./exec_big_benchmark.sh
intel-parallel-studio: using intel wrappers for mpicc, mpif77, etc
Submitted batch job 1343125
Submitted batch job 1343126
Submitted batch job 1343127
Submitted batch job 1343128
Submitted batch job 1343129
Submitted batch job 1343130
Submitted batch job 1343131
Submitted batch job 1343132
Submitted batch job 1343133
Submitted batch job 1343134
Submitted batch job 1343135
Submitted batch job 1343136
Submitted batch job 1343137
Submitted batch job 1343138
Submitted batch job 1343139
Submitted batch job 1343140
Submitted batch job 1343141
Submitted batch job 1343142
Submitted batch job 1343143
Submitted batch job 1343144
Submitted batch job 1343145
Submitted batch job 1343146
Submitted batch job 1343147
Submitted batch job 1343148
Submitted batch job 1343149
Submitted batch job 1343150
Submitted batch job 1343151
Submitted batch job 1343152
Submitted batch job 1343153
Submitted batch job 1343154

```

Figure 27 Output of `exec_big_benchmark.sh`

```

di49rib@login01:~/Markus/Relaxationsverfahren/output> ls
1024x1024_128_micro_trace  1024x1024_4_micro_trace    2048x2048_1_micro_trace    2048x2048_64_micro_trace    512x512_2_micro_trace      clear.sh
1024x1024_16_micro_trace   1024x1024_512_micro_trace  2048x2048_256_micro_trace  2048x2048_8_micro_trace    512x512_32_micro_trace    compiled
1024x1024_1_micro_trace    1024x1024_64_micro_trace   2048x2048_2_micro_trace    512x512_128_micro_trace    512x512_4_micro_trace     extractalloutput.sh
1024x1024_256_micro_trace  1024x1024_8_micro_trace    2048x2048_32_micro_trace   512x512_16_micro_trace     512x512_512_micro_trace   top_archive
1024x1024_2_micro_trace    2048x2048_128_micro_trace  2048x2048_4_micro_trace    512x512_1_micro_trace     512x512_64_micro_trace
1024x1024_32_micro_trace   2048x2048_16_micro_trace   2048x2048_512_micro_trace  512x512_256_micro_trace    512x512_8_micro_trace

```

Figure 26 Result after `exec_big_benchmark.sh`

Improvement idea and conclusion

The program is not the fastest solution, because of the `MPI_Allgather`. While writing this thesis, I came across the whole catalog of MPI functions²¹. I have by far not checked all functions for their usefulness to solve the problem of this thesis, but `MPI_Get`²² caught my eye. This function reads a certain memory area of another MPI - process. Here a process could read values for the Jacobi - calculation directly from the memory of the responsible process for the row in question, without further synchronization. The actuality of the data would happen by reading out the "n_iterations" - variable also over `MPI_Get`.

In conclusion, the project can be considered a success. My MPI implementation outperformed a single-processor application by a factor of two for all matrix sizes tested.

²¹ <https://www.mpich.org/static/docs/v3.2/www3/index.htm> (Retrieved 24/07/2021)

²² https://www.mpich.org/static/docs/v3.2/www3/MPI_Get.html (Retrieved 24/07/2021)