Project Report

Final project of the lecture
Parallel Computing for Computational Mechanics [22ss-41.07360]
at RWTH Aachen University

Author
Oliver Ahrend, 432734

Summer Semester 2022

Aachen
July 2022

Abstract

The following document is a report that documents the processing of the tasks given by the final project of the lecture *Parallel Computing for Computational Mechanics [22ss-41.07360]* at RWTH Aachen University, in summer semester 2022.

The main goal of the project was to give a better understanding of how parallel computing can be used for reducing simulation times in scientific or engineering problems, and what opportunities but also challenges come with that.

The case under study was the transient simulation of a temperature distribution in a 2D, circular domain, using Finite-Element method, short FEM.
During the project many simulations have been performed using serial and parallelized codes. All the codes, which have already been written, use the programming language C++. Parallelization had been realized using OpenMP and MPI specifications. The runs were executed on RWTH compute cluster CLAIX-2018.

It could be shown that using more computational power, in terms of more CPU cores, does not necessarily reduce the amount of wall clock time needed to solve a certain simulation problem. Especially using low numbers of nodes and elements to discretize the domain of interest, can even lead to an increase in computation time, when distributing the workload onto multiple cores. The most prominent reason for that is assumed to be the overhead that is caused by the runtime when splitting workload. If that overhead is significant with respect to the actual time needed to solve the distributed work package, the overall runtime can be larger than compared to a serial execution. While Using OpenMP the overhead is mainly introduced by splitting the workload, while in the MPI implementation it could be shown that an unfortunate partitioning of the domain occurred which is assumed to introduce significant communication overhead.
Furthermore, it could be shown that also the implementation of parallelization as well as the compile options used for generating the code can have significant influence on the computation time.

Table of content

Abbreviations

FEM	Finite-Element method
OpenMP	Open Multi-Processing, programming standard for shared memory systems
MPI	Message-Passing-Interface, programming standard for distributed memory systems

Symbols

Ø	Average

1 Introduction

Simulation is seen as the third pillar of scientific information acquisition. One reason for that is the steadily increasing computation power. But because of different reasons that prevent single-core systems from increasing their throughput, utilizing multiple cores of a compute unit has become the method of choice for the last decades already. This is only possible, if the simulation code is able to divide the computation into smaller work packages, that can then be processed efficiently by different compute units at the same time, which is then often called parallel execution. The whole process is then called parallel computation, employing parallelized code.

This report compares different ways of parallelization using the OpenMP and MPI standard. The effects of different implementations are shown employing a simple transient 2D temperature problem.

2 Theory & Methods

2.1 Heat equation and domain of interest

In the following the heat equation is solved on a 2D disk Ω with boundary Γ. A visual representation is shown in Figure 1.

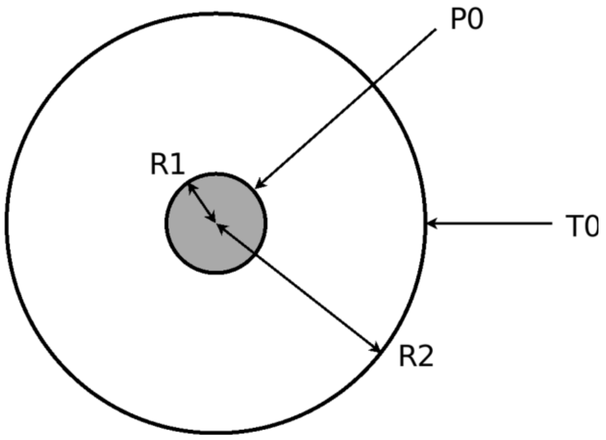


Figure 1 Problem domain, copied from project assignment

The different parameters are sown in Table 1.

The heat equation reads:

$$\frac{\partial T}{\partial t} - \kappa \nabla T = f \qquad \text{on } \Omega\ \forall t \in (0, t_f), \qquad [1]$$

$$T(x,t) = T_O \qquad \text{on } \Gamma\ \forall t \in (0, t_f), \qquad [2]$$

$$T(x,0) = T_0 \qquad \text{on } \Omega \qquad [3]$$

with T being temperature as function of space x and time t, κ being the thermal diffusivity, and f being the thermal heat source. A time invariant Dirichlet boundary condition T_O (capital letter O, not numeral 0) is specified for the boundary Γ. A space invariant initial condition is given as T_0 (numeral 0, not letter O). The heat equation is solved until the final time t_f. The heat source is time invariant, but space variant with:

$$f(r) = \begin{cases} \frac{Q}{\pi R_1^2} & \forall r \in (0, R_1), \\ 0 \quad \forall r \in (R_1, R_2), \end{cases} \qquad [4]$$

With r being the radial coordinate of the domain, since there is rotational symmetry, and Q being the heat source, assuming a thickness of the disk of d_z. Table 1 shows the values of the different parameter.

Table 1 Parameters for heated disk problem

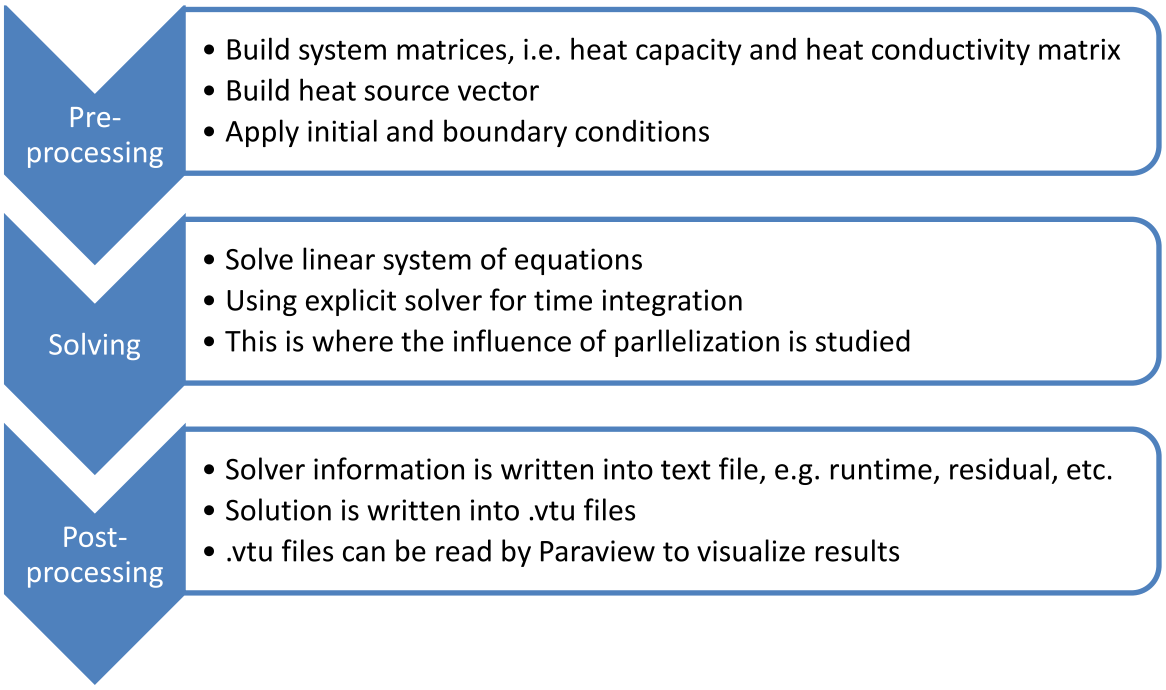
Parameter	Variable	Value	Unit
Inner circle radius	R1	0.01	m
Outer circle radius	R2	0.1	m
Heat source	Q	100	W
Dirichlet boundary	T_O	500	K
Initial condition	T_0	0	K
Thermal diffusivity	κ	1.0	m²/s
Disk thickness	dz	0.1	m

An analytical solution can be found in terms of radial coordinates as:

$$T(r) = \begin{cases} T_O - \dfrac{Q}{2\pi\kappa}\left(\dfrac{1}{2}\left(\dfrac{r^2}{{R_1}^2} - 1\right) + ln\left(\dfrac{R_1}{R_2}\right)\right) & \forall r \in (0, R_1), \\ T_O - \dfrac{Q}{2\pi\kappa}\left(ln\left(\dfrac{r}{R_2}\right)\right) & \forall r \in (R_1, R_2). \end{cases} \qquad [5]$$

2.2 Pre-, postprocessing and solver

To solve the heat equation given in section 2.1 the method of finite-elements (FEM) was used. For comparison three different FE meshes had been used, that vary in element size and therefore also number of elements and nodes, and finally computational effort to compute the solution. The qualitative workflow looks as follows:



- Build system matrices, i.e. heat capacity and heat conductivity matrix
- Build heat source vector
- Apply initial and boundary conditions

- Solve linear system of equations
- Using explicit solver for time integration
- This is where the influence of parllelization is studied

- Solver information is written into text file, e.g. runtime, residual, etc.
- Solution is written into .vtu files
- .vtu files can be read by Paraview to visualize results

The explicit solver uses the forward Euler method. The iteration scheme can be written as follows:

$$T_{i+1} = M^{-1} * \left(M * T_i + \Delta t(f - K * T_i)\right), \quad [6]$$

$$T_{i+1} = M^{-1} * IterationVector_i, \quad [7]$$

with indices $._i$ and $._{i+1}$ denoting consecutive time steps that are Δt apart, M denoting the heat capacity matrix, and K denoting the heat conductivity matrix. $IterationVector$ basically is what generates the computational effort for the time integration because it must be evaluated in every time step. This is stressed here specifically because this is where the parallelization will be applied.

2.3 Parallel programming

In the following a short summary of the methods used for parallelization should be given. This section does not have the intent of lecturing in any way and assumes the reader´s familiarity with the named parallelization concepts and standards.

All code was given, and therefore it was not necessary to write code during this project.

All code was written using the programming language C++. Parallelization was realized using the renowned standards of OpenMP and MPI.
Overall, four different codes had been provided. One that employs purely serial execution, two that used different implementations of OpenMP for parallelization, and one that uses an implementation of MPI.

During the project the influence of different compile options should be analyzed. Therefore, it was necessary to recompile each of the solvers, mentioned above, several times. For this the standard Intel icpc compiler of version 19.1 was used, for the solver variant using MPI, additionally IntelMPI2018 was used.

However, the basic solver steps are the same amongst all the solver versions. All go through the consecutive time steps updating the temperature solution, which is divided into two major steps. First, the $IterationVector$ (see eqn. [7]) is computed on element level, and then assembled globally, in the following called step I.I and I.II, secondly, the latter is divided by the global nodal mass to obtain the solution at the new time step, and the residual is computed, in the following called step II.I and II.II. The further splitting in substeps .I and .II are only important for the parallelization using MPI (see section 2.3.2).

2.3.1 **Shared memory parallelization using OpenMP**

As mentioned before there are two solver versions using OpenMP. To distinguish both methods they are called (a) and (b).

Both variants make use of a parallel region that encloses both major steps I and II. Both variants also make use of two *#pragma omp for* directives, one for each major step I and II.
In step I there is a data race on the memory of the globally assembled $IterationVector$, because

nodes can belong to multiple elements, and so the conversion from local to global data results in a reduction on node level.
Variant (a) tackles that by applying additional *#pragma omp critical* directives to ensure every entry of said vector is touched by only one thread at a time.
Variant (b) adds a *reduction* clause to the for directive.

In step II again a data race occurs for computing the global L2 residual from the nodal residuals. Variant (a) again employs a critical region to take care of that, while variant (b) again uses an additional *reduction* clause in the for directive.

2.3.2 Distributed memory parallelization using MPI

As mentioned before the splitting in substeps .I and .II now becomes important. In contrast to OpenMP, MPI found to be used on shared memory systems. Since steps I.I, and II.I, (see end of section 2.3) both only require local data, the shared memory implementation allows for having only this local data stored at each MPI *Rank*. Since in the end the global solution is of interest, and on the interfaces between neighboring *Ranks* local data belongs to more than one *Rank*, a global data exchange cannot be avoided. For this the entries of the $IterationVector$ as well as the temperatures on the interfaces, must be communicated in the steps I.II and II.II. In step I.II the global $IterationVector$ is assembled by adding up the contributions of elements that are sharing nodes. For this the *MPI_accumulate* function is used. The same holds for computing the residual in step II.II where again *MPI_accumulate* is used. The given implementation additionally requires updating the local temperatures among the *Ranks*. In contrast to the previously mentioned communications this is only an information transfer, and not an algebraic operation, thus the accumulate function cannot be used here. Therefore, a different method is employed, the *MPI_Get* function. Since this is a one-sided communication, manual synchronization is required, and implemented using the *MPI_Win_Fence* function.

3 Project tasks

This section is meant to answer the given tasks. It includes a quick summary of the tasks themselves, as well as their results and discussion. A significant part of the project focuses on employing optimization at compile time, i.e., using compiler options, to allow the compiler to come up with fast executing code.
Section 3.1 focuses on finding good compile options that allow for a fast execution of the serial version of the solver. Using found compile options will then be used in sections 3.2 and 3.3, where different parallelization strategies and implementations will be compared.

All given simulation runtimes are averages of three simulation runs, in an effort of minimizing variations, and make the results more comparable.

The three different meshes all use triangular elements with linear shape functions. This results in mesh data summarized by Table 2.

Table 2 Mesh data for the three different FE meshes

Mesh	Nodes	Elements
Coarse	2,041	3,952
Medium	21,650	21,332
Fine	85,290	84,656

****

Insert figures of meshes

****

3.1 Serial solver

To determine the compile options that allow a fast execution of the serial solver, multiple solvers have been compiled and tested.

3.1.1 Comparison of analytical and numerical solution

First the comparison of the analytical and numerical solution of the heat equation should be shown, using the coarse mesh for the simulation.

Figure 2 shows the temperature profile of the analytical and numerical solution of the heat equation. Since the analytical solution is time invariant, the simulation has been evaluated at the final time step. 1000 sample point are used. Since both lines in the figure are very close to each other, visualizing deviation is implemented by using different marker sizes, and the sample points. The markers of the simulation data are approximately 2.5 times the size of the analytical solution. This way it can be seen, that close to the center of the disk, there is a slight deviation between both solutions. Investigating the discrete values, it can be shown that the maximum deviation is less than 0.64K. By increasing distance from the center this difference becomes even smaller.

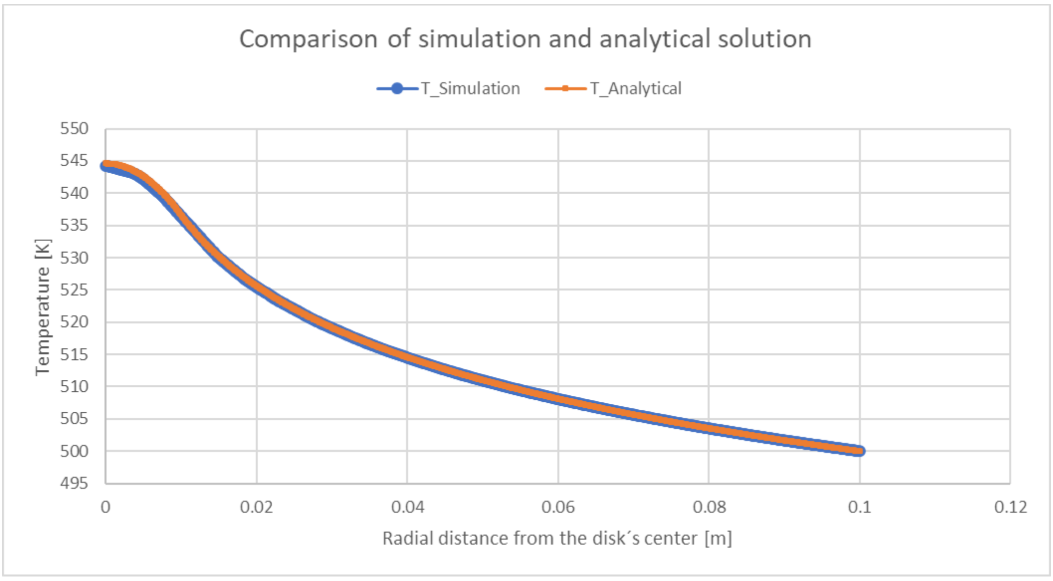


Figure 2 Comparison of analytical and numerical (simulation) solution of the heat equation over the radial distance from the disk´s center, evaluated using 1000 sample points

3.1.2 Comparison of runtime using different compiler options

Secondly, multiple solvers should be compiled using different (combinations of) compile options. The grading of which (combination of) compile options are good four our purposes, is the resulting runtime of the solver, solving the heat equation using the medium mesh.

Table 3 shows the comparison of runtimes for the serial solver using different compiler options. As requested, the first run was performed using the coarse mesh. All other runs then used the medium mesh.

The following gives a short summary of each of the compile options according to the official Intel reference guide (Intel, 2022):

- -O0, -O1, -O2, -O3 are called general optimization flags. Apart from -O0, using these the compiler has the freedom to use a combination of many other compile options, with the objective to speed up the code. The higher the number the more aggressive the optimization is. However, -O3 is known to produce slower code than -O2 in many cases. -O3 is especially meant to be used with code that processes large amounts of data. As can be seen in Table 3, the case under study is such a case where the solver compiled using -O3 is slower than the one compiled using -O2. The exception in this group of options is -O0 which disables all optimization. For more information on why this might be desirable please see the end of this section.
- -vec allows vectorization of data processing. That speeds up the processing of SIMD constructs, e.g., adding two vectors. It also allows to vectorize for loops.

- -xSKYLAKE-AVX512 combines multiple compile options in one. The -x part tells the compiler to generate code that can make use of the hardware architecture specified after x. In this case the Intel SKYLAKE architecture that is available on CLAIX-2018. Codes compiled with such an option are probable to not be executable on different architectures. The extension AVX512 specifies vectorization. 512 stands for vector registers with 512bit, so that SIMD instructions can be executed with 512bit of data. Additionally, the AVX standard comes with feature detection that allows to identify sections of the code that can be vectorized accordingly.
- -m64 can be used to instruct the compiler to make use of the Intel 64 architecture. Unfortunately, the documentation does not give any more information on that.
- -unroll allows the compiler to unroll loops to a maximum of n times in case n is specified. If n is not specified, the compiler will determine the maximum number a loop can be unrolled.
- -fp-model==fast=={1|2} allows to speed up floating point calculations in a trade-off with precision. Option 2 is more aggressive than option 1.
- -cXORE-AVX2 I quite similar to the previously mentioned-xSKYLAKE-AVX512 compile option. However, it only allows usage of 256bit registers and is less aggressive in terms of vectorizing functions and other computations. AVX2 was released before AVX512 and is therefore applicable on older architectures as well.
- -inline-level=2 as the name says, specifies how the compiler is allowed to perform function inlining. Level 0 disables inlining of all user defined functions. Level 1 allows inlining where a keyword is used within the code. Level 2 gives the compiler the freedom to choose for every function.

The best combination of compile options in this specific case, was found to be -O2 -xSKYLAKE-AVX512 -inline-level=2 -unroll, as it can be seen in Table 3, highlighted in green.

Table 3 Comparison of runtime of serial solver using different compile options

Code	Compiler Flags	Mesh	Runtime Ø [s]	Speed-Up
Serial	-O0	coarse	4.45	-
	-O0	medium	227.43	1.00
	-O1	medium	61.97	3.67
	-O2	medium	54.20	4.20
	-O3	medium	57.18	3.98
	-O0, -vec	medium	224.43	1.01
	-xSKYLAKE-AVX512	medium	53.17	4.28
	-O0, -m64	medium	228.09	1.00
	-m64	medium	54.48	4.17
	-unroll	medium	56.45	4.03
	-fp-model=fast=2	medium	56.07	4.06
	-fp-model=fast=1	medium	54.79	4.15
	-xCORE-AVX2	medium	54.34	4.19
	-inline-level=2	medium	55.35	4.11
	-O2 -xSKYLAKE-AVX512	medium	52.31	4.35
	-O2 -xSKYLAKE-AVX512 -inline-level=2 -unroll	medium	51.99	4.37
	-O3 -xSKYLAKE-AVX512 -inline-level=2 -unroll	medium	55.23	4.12
	-O2 -xSKYLAKE-AVX512 -inline-level=2	medium	52.58	4.33
	-O3 -xSKYLAKE-AVX512 -inline-level=2	medium	56.05	4.06

An additional question of the project assignment was whether one might choose a non-optimized code, and what potential drawbacks of compiler optimization are.
One reason to use non-optimized code is debugging. Since compiler optimization can change the code to a specific extend, debugging can be more difficult. Also, non-optimized code can be used to gather profiling data, to find functions or parts in the code in general that take the most computation time. This information can then be used to optimize the code on the programmer side. Additionally, some compiler optimizations can have impact on the accuracy of the calculation, e.g, see option *-fp-model=fast* above. And finally relying on the compiler to optimize the code might lead to programmers not fully understand how the code works. Optimization by the programmer requires a deep understanding of the code itself.

3.2 Parallel solver – OpenMP

Table 4 shows the comparison of both OpenMP parallelized solver variants with respect to the simulation runtime on the coarse mesh, using varying number of threads, using the compile options that were found to generate the shortest runtime in section 3.1.2. It can be seen that variant (a) in general has significantly longer runtimes than variant (b). Also, the runtime of variant (a) increases exponentially with respect to the number of threads. The runtime of variant (b) seems to be almost not influenced by the number of threads. However, for variant (b) the longest runtime was measured using two threads, such that one and four threads both lead to shorter runtimes. The red highlighted row of variant (b) using four threads lead to an early exit of the time iteration loop in one of the three runs to average the results, thus the given results are given on a basis of two successful runs, instead of three.

Table 4 Comparison of the two variants of the OpenMP solvers in terms of simulation runtime on the coarse mesh using 1,2 and 4 threads

Code	Compiler Flags	Mesh	Processes	Runtime Ø [s]	Speed-Up
OpenMP_A	-O2 -xSKYLAKE-AVX512 -inline-level=2 -unroll	coarse	1	8.23	0.54
			2	77.62	0.06
			4	137.73	0.03
OpenMP_B	-O2 -xSKYLAKE-AVX512 -inline-level=2 -unroll	coarse	1	1.10	4.06
			2	1.49	2.99
			4	1.22	3.65

The next question asked whether this behavior of scaling with respect to the number of threads, was expected. As explained in section 2.3.1 the different scaling can be explained by the implementation of parallelization. Variant (a) uses *critical* compiler directives to handle the data race in updating the $IterationVector$ which causes significant overhead, that scales stronger than linear since every iteration all threads need to be synchronized with one another.
Variant (b) using the *reduction* clause can make efficient use of the algebraic summation of the local data. Therefore, it was expected that variant (b) makes way more efficient use of multiple threads.

Based on this knowledge of the inefficient parallelization of variant (a), the next task was to have a closer look on variant (b). More specifically the task was to analyze the influence of different scheduling approaches in the parallelized sections as described in section 2.3.1, using a constant number of four threads, and the fine mesh.
Initially it was planned to compare three scheduling approaches *static, dynamic, guided*. Since there is a bug, either in the code or the compiler, that leads to indeterministic behavior when using scheduling approaches dynamic and guided, the focus was set on the influence of the chunk size using *static* scheduling. Table 5 shows said results using a varying chunk size of 128 up to 16384. The first row with chunk size “-“ does not specify a chunk size which results in the runtime computing the chunk size to $ceiling(iterations/numberofthreads)$. It can be seen that the runtimes decrease with increasing chunk size. The only exception from this is the chunk size of 4096. The red highlighted last row using a chunk size of 16384 also lead to an early exit of the time iteration loop. In this case for all three simulation runs. Therefore, no timing could be provided.

Table 5 Comparison of runtime of OpenMP solver variant (b) using different chunk sizes for static scheduling

Code	Compiler Flags	Mesh	Chunk Size	Runtime Ø [s]	Speed-Up
OpenMP_B	-O2 -xSKYLAKE-AVX512 -inline-level=2 -unroll	fine	-	183.67	1.00
			128	196.36	0.94
			256	185.93	0.99
			512	189.00	0.97
			1024	188.91	0.97
			2048	178.35	1.03
			4096	180.85	1.02
			8192	176.80	1.04
			16384	-	-

The next task was to use the latest findings and compare runtime, speed up and efficiency of the best combination of compile options, scheduling, and chunk size for solver variant (b). For this runtime was measured for running simulations on all three meshes – coarse, medium, fine – using different number of threads, varying from one to twelve. Since Table 5 already shows complications for large chunk size with respect to number of nodes/elements in the mesh, it was decided to not use a chunk size of 8192, but rather 2048, because both lead to quite similar runtimes.

Table 6 shows said findings. It can be seen that in general runtime scales with mesh granularity.

Scaling of runtime with respect to number of threads seems to be mesh dependent, since on the coarse mesh there is no clear tendency to observe. However, for medium and fine mesh runtime decreases with number of threads.

Figure 3 and Figure 4 show the speed up and parallel efficiency of the given runtimes. As observed already in Table 6 there is no real speed up to observe on the coarse mesh. Then on medium and fine mesh there is visible speed up, and fine mesh has the highest speed up.
The same is observed in terms of parallel efficiency. Since there is no significant speed up on the coarse mesh, also efficiency is the smallest. Then with increasing mesh granularity speed up as well as efficiency increase, with highest efficiency for the fine mesh. Noteworthy is the parallel efficiency >1 on the fine mesh using two threads. Here doubling the number of threads from one to two reduced the runtime by slightly more than 50%. Since a theoretically perfect parallel efficiency is one, this is assumed to be a result of runtime variance, and the relatively small number of three runs of which results are averaged.
Furthermore, it can be seen (on the medium and fine mesh) that with an increasing number of threads speed up approaches a saturation. This is expected according to *Amdahl´s law* since not the complete code can be parallelized and so there will always be parts that need to be executed in serial. This is also observable in terms of the parallel efficiency that decreases with increasing number of threads.

Table 6 Comparison of runtime of OpenMP solver variant (b) using varying number of threads on coarse, medium and fine mesh

Code	Compiler Flags	Mesh	Processes	Chunk Size	Runtime Ø [s]
OpenMP_B	-O2 -xSKYLAKE-AVX512 -inline-level=2 -unroll	1_coarse	1	2048	1.09
		1_coarse	2	2048	1.27
		1_coarse	4	2048	1.02
		1_coarse	6	2048	1.12
		1_coarse	8	2048	1.11
		1_coarse	12	2048	1.03
		2_medium	1	2048	59.64
		2_medium	2	2048	47.09
		2_medium	4	2048	32.92
		2_medium	6	2048	30.16
		2_medium	8	2048	29.01
		2_medium	12	2048	23.14
		3_fine	1	2048	681.09
		3_fine	2	2048	320.79
		3_fine	4	2048	188.19
		3_fine	6	2048	150.46
		3_fine	8	2048	130.10
		3_fine	12	2048	113.62

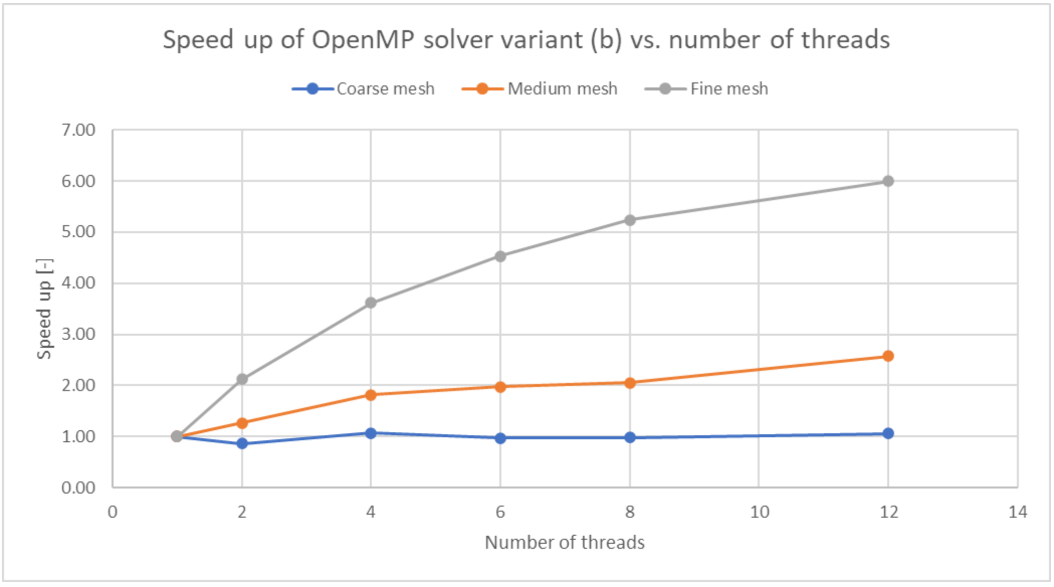


Figure 3Speed up of OpenMP solver variant (b) over number of threads using different meshe

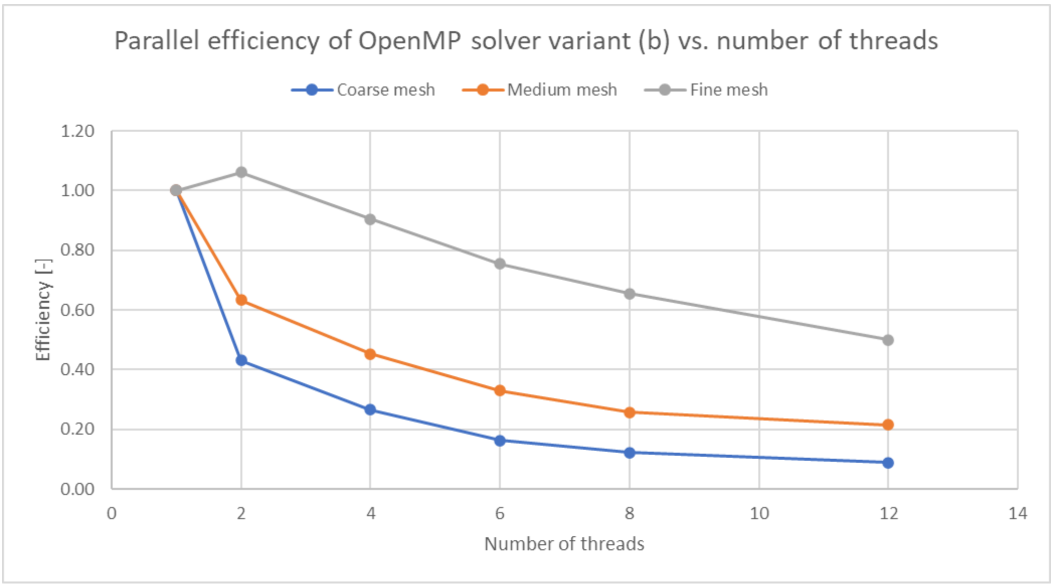


Figure 4 Parallel efficiency of OpenMP solver variant (b) over number of threads using different meshes

As mentioned before it was not possible to reliably generate results using different scheduling approaches. Nevertheless, the general idea of each approach, as well as the respective default behaviors can be stated here.
The three major scheduling approaches are, as given in the official OpenMP documentation (OpenMP Architecture Review Board, 2021):

- Static: divides loop iterations upon the available threads, according to the chunk size. Each thread is assigned its loop iterations, and only once all threads have finished the runtime can continue. For balanced loops this is supposed to be the most efficient approach since it generates the least overhead compared to other scheduling approaches. If no chunk size is specified, the number of chunks is equal to the number of threads.
- Dynamic: also divides the loops into chunks of same size. In contrast to *static* the number of chunks is usually larger than the number of threads, and the chunks are not fixed to be executed by a predefined thread. This allows the chunks to be worked on in a first come, first serve manner which can improve handling load imbalance between the loop iterations. The default chunk size is one, which has the strongest balancing effect on unbalanced loops. However, handling chunks creates overhead, and so there is a trade off between balancing load imbalance, and overhead.
- Guided: is very similar to dynamic scheduling. The main difference is that guided does use chunks of different sizes. The first chunks are the largest, then the chunk size is reduced. The chunks are worked on by whichever thread has finished a previous one. Due to the small chunk sizes towards the end of the loop iterations, this scheduling approach might also be used on unbalanced loops. In contrast to *dynamic* scheduling larger chunk sizes at the beginning reduce the overhead.

The default scheduling approach is *static* as stated in [insert source]

3.3 Parallel solver -MPI

This section focuses on parallelization using the MPI standard. To generate results that stick to the context of the previous tasks, compile options and meshes are the same as in section 2.3.1 which focused on parallelization using OpenMP.

The first task was to compare runtime, speed up and parallel efficiency with respect to the number of processes, on all three of the previously introduced meshes. The number of threads was varied between one and 16. Figure 5, Figure 6 and Figure 7 show the comparison of runtime, speed up and parallel efficiency.
From the obtained runtimes it can be seen that there is similar behavior as in the OpenMP parallelized code. On the coarse mesh there is no significant change in runtime by varying number of processes. On medium and fine mesh there is a stronger dependency of runtime with respect to number of processes. Measured the absolute runtime differences again the finest mesh has the strongest scaling. First the runtime decreases with increasing the number of processes, then it increases with all meshes, once a certain number of processes is exceeded. This number of threads however varies with the mesh granularity and the corresponding computational effort. For the coarse and medium mesh the shortest runtime was achieved using two processes. For the fine mesh using four processes lead to the shortest runtime.

This behavior is again reflected in the speed up. The strongest scaling of speed up with respect to the number of processes is achieved with the fine mesh, then the medium and the coarse mesh has the weakest scaling. This again is due to the fact that on the coarse mesh the workload of each process is relatively small, compared to medium and fine mesh, and therefore the overhead of communication between the processes is more relevant. Nevertheless, on all meshes there is a speed up by going from one to two processes, while only on the fine mesh there still is a speed up >1 for higher number of processes, while the runtime on the medium and coarse mesh would increase again, as soon as using four or more processes, the longest runtime on the fine mesh remains at one process.

The comparison of parallel efficiency again reflects these findings, however it allows for even more detailed insight. Since it is not known how much of the code can be parallelized, it is hard to come up with an estimation of what efficiency this code could achieve, neglecting all overhead. According to Amdahl´s law it is possible to make assumptions based on the ratio of serial (S) and parallel (P) part of the code, and then plot the maximum efficiency that such a code would allow (using S+P=1, where S is the fraction of code that cannot be parallelized at all, and P is the fraction of code which can be parallelized perfectly, meaning without overhead). In addition to the measurements Figure 7 also shows these theoretical efficiencies according to *Amdahl´s law*. Here the curves with markers display the simulation results while the curves without markers display the theoretical results.

Based on the observed runtimes, speed up and efficiency it is possible to come up with a recommendation for how many processes/cores to use for which mesh. This way one might want to use two processes for coarse and medium mesh, because that choice provided the only runtime shorter than using only one process. On the fine mesh the shortest runtime was achieved using 4 processes. However also all other number of processes lead to runtimes smaller than using only one process. Since more than four processes lead to an increasing runtime again it might not be advisable to use more. However, in case hardware is a limiting factor in any way, also two processes would give reasonable speed up, that is only little smaller than using four processes.

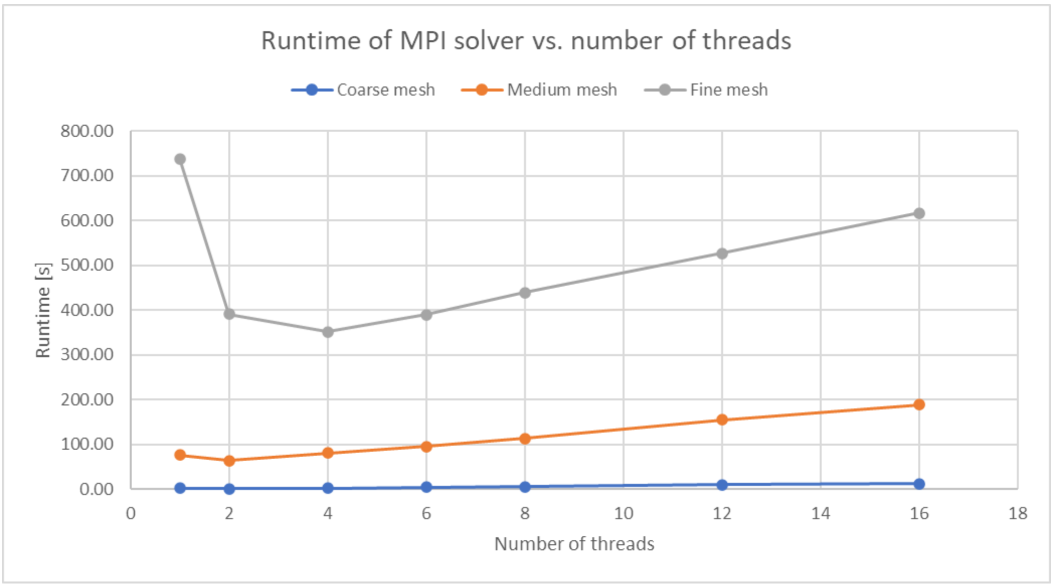


Figure 5 Comparison of runtime of MPI parallelized solver over number of threads for different meshes

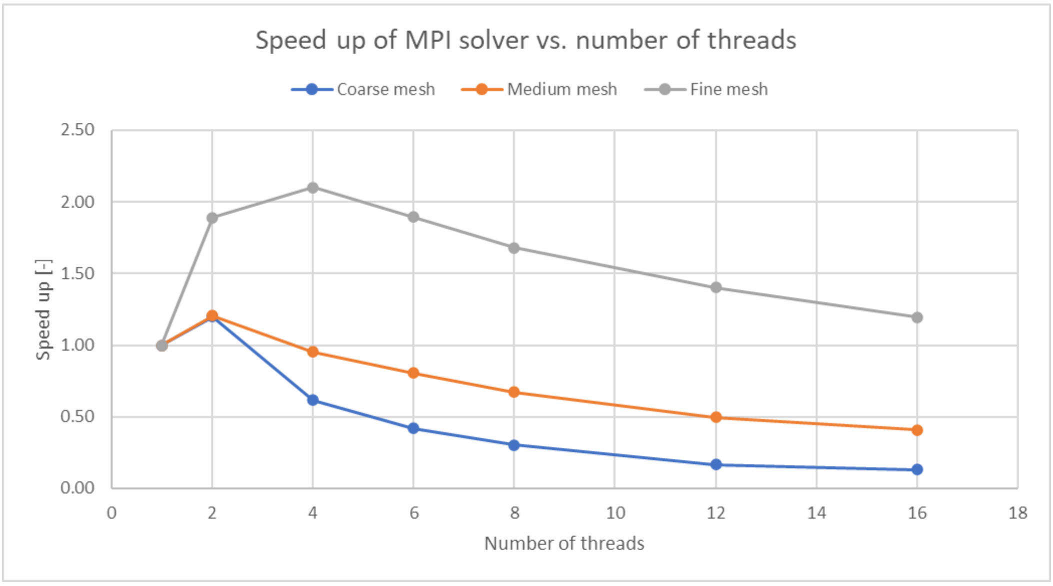


Figure 6 Comparison of speed up of MPI parallelized solver over number of threads for different meshes

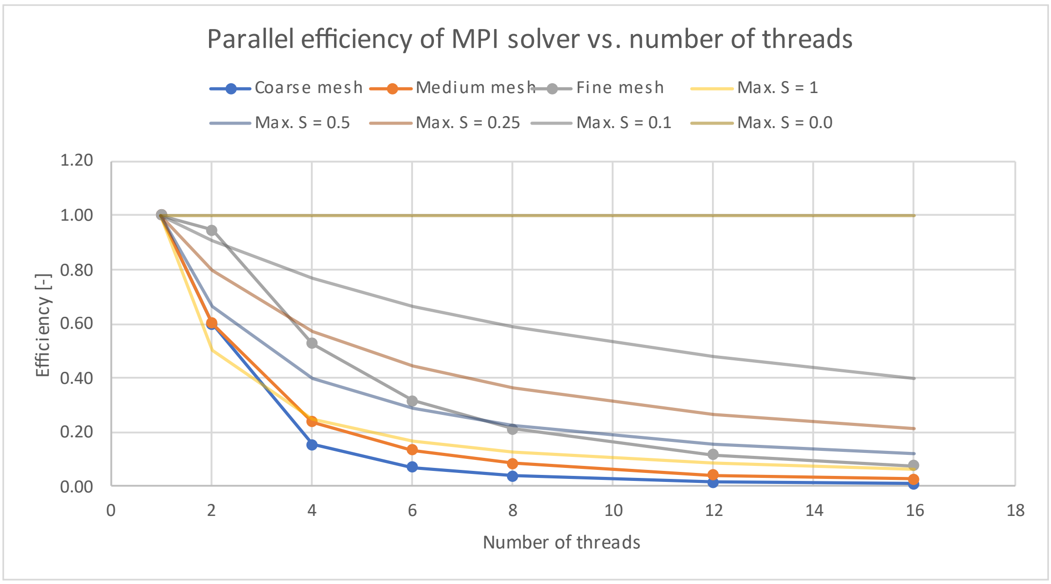


Figure 7 Comparison of parallel efficiency of MPI parallelized solver over number of threads for different meshes and theoretical results according to Amdahl´s law with S being fraction of the code that is inherently serial, and 1-S being the fraction that is perfectly parallelizable, curves with markers are simulation results, curves without markers are theoretical results

For the next task an image of the partitioning of the medium mesh was requested using eight processes. This is provided in Figure 8. Each color represents one partition, so there are in total eight colors. From this it can be seen that the partitions are very much spread out over the whole domain, having only one exception of the inner circle that seems to be located in a single partition. Keeping in mind that information on the boundaries between partitions has to be communicated between processes, this is a very unfortunate partitioning, because there are only very few neighboring elements belonging to the same partition, and so the communication effort is expected to be very high. From only the information about the partitioning, the code is most probable not able to make efficient use of multiple processes/cores. Furthermore, since MPI is a distributed memory approach, it is favorable used for parallelization over very large numbers of processes, for example on a compute cluster. Then the communication between the different processes takes even more time, than in the case of this report, where all cores were located on the same compute node.

This finding is supported by the plot of parallel efficiency in Figure 7, where the efficiency drops rather quickly with increasing number of threads.

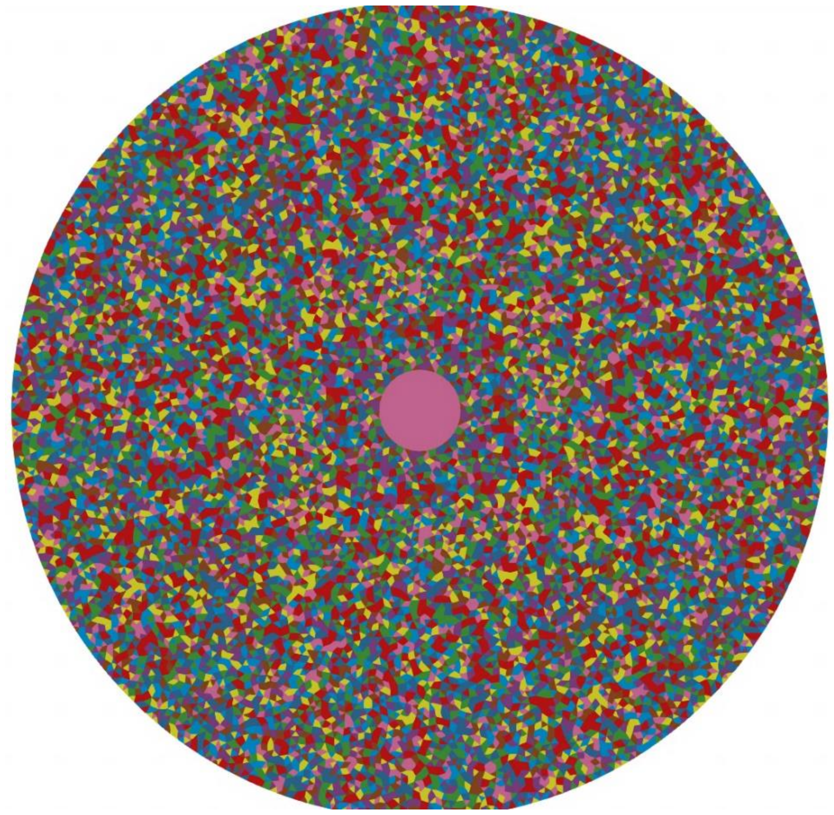


Figure 8 Visualization of the partitioning of the medium mesh using 8 processes with the MPI solver

The performance of the MPI solver can be discussed further.
For this it might be helpful to have a look at Figure 7 again. Here the relatively good compliance of the measured efficiency curves of the coarse and medium mesh with the yellow curve, displaying the theoretical maximum efficiency for a serial fraction of S=1, meaning that all the code is inherently serial, can be observed. This means that even though there has been applied a significant effort to write parallelized code, there is basically no return of investment, because the code "scales" almost equal as a code that was fully serial.
Furthermore, the simulation was performed using the same compile options and meshes as in section 2.3.1 that focused on the parallelization using OpenMP. This allows now to compare runtimes, speed up and efficiency here as well. From comparing results of the OpenMP solver variant (b) in Table 6, Figure 3 and Figure 4 with the latest findings using the MPI solver, it can be seen that the MPI solver lead to longer runtimes, in any of the comparable cases.
All this gives rise that the performance of the MPI solver is not optimal.

One possible part of the code that is assumed to greatly increase performance is the partitioning. As seen in Figure 8 and described previously the partitioning under use generates enormous need for communication which can contribute greatly to the solver runtime. Also, the overhead is increased with increasing number of processes.
Another part for improvement might be the synchronization of the different processes/ranks after each iteration. Since the solver works with distributed memory, the solution is computed locally. However, to continue and compute the $IterationVector$ of the next iteration, which is done locally

as well, it is necessary to communicate the newly found solution values from other processes, since not all nodal solutions are available at every process. In the provided implementation, every iteration the full, global, computed solution is communicated to all processes. This would in general not be necessary, since not all processes need all nodal information. However, keeping in mind the partitioning shown in Figure 7, improving the communication itself between the processes might not give the desired speed up because of the previously mentioned obstacles of the unfortunate partitioning.

The last task of the project asks several questions, that are cited again for providing answers (Schuster, 2022):

1. *Why do we use MPI_Accumulate in the code instead of MPI_Put?*
2. *What are advantages of using one-sided MPI communication over two-sided MPI communication?*
3. *What advantage does MPI offer over OpenMP, what is a main obstacle of MPI?*

The answers are as follows:

1. In the lines *MPI_Accumulate* is used, the underlying task is to sum up contributions from different processes. *MPI_Put* just copies data from a memory location of one process, to the memory location of other process(es). The important thing here is the algebraic operation of summing up, which can be done (very much) more efficiently than just copying data to the process where the summation should take place, e.g., round-robin, tree communication, etc.
   In terms of the global L2-Error of the calculation *MPI_Put* would require transferring all the local residuals to (at least) one process, that could then sum up all terms. *MPI_Accumulate* could send around only one double value from process to process, and at each receiving process add the incoming to the local value. This reduces the amount of communicated data greatly
2. Two-sided MPI communication starts by initializing a communication by providing a message that states that one process would like to start a communication. Then another process must send a message back, that it is ready to communicate. Only then the communication can start. Depending on what communication pattern was chosen it could, but does not have to, happen that again two small messages must be sent and received to verify successful completion of the communication.
   One sided communication just assumes that the other process is able to communicate, and so does not need to verify this. In the end this reduces the overhead of communication. One can now already see that the larger the message to communicate the less significant the overhead is.
3. The answer to this question lays in the fundamental approach of each of the standards. OpenMP being a shared memory approach, and MPI being a distributed memory approach, the use cases can differ greatly. One advantage of MPI is, that it is possible to execute parallelized code over systems that do not share the same memory space, e.g.,

it is possible to orchestrate a large number of compute nodes with even larger number of cores, as it would be found on a cluster. In contrast to that OpenMP only allows to orchestrate cores that share a single memory space. However, MPI can also be used to parallelize code on shared memory systems, as can be seen for example in this report. A disadvantage of the latter is that MPI induces more overhead communicating on shared memory systems, than OpenMP does.
A disadvantage of MPI is the more complex programming structure. While OpenMP allows to write code as it would be executed in a serial run, it just requires adding compiler directives that tell the compiler and runtime to split certain computation packages amongst multiple compute units. In contrast MPI requires to manually parallelize the code and take care of data handling by the programmer.
A final advantage would be that when coming up with a new code, it might seem sufficient to use shared memory compute units. If then some time later it would be desirable to also make use of distributed systems, OpenMP code would require to be build again, almost from scratch. If the code would have been parallelized using MPI already, this step would not be a major obstacle.
However, using shared memory systems only, chances are high that OpenMP code can make use of the given computation units more efficiently.

4 Conclusion

Following the given project tasks this report presented a brief overview about the differences as well as similarities of parallelizing scientific or engineering code using the OpenMP or MPI standard. For more feasible understanding a use case was given to calculate the transient temperature behavior of a 2D disk, that was heated from the center.

For comparison one serial solver, two solvers that use different implementations of OpenMP, and one solver that uses MPI, were compared. Since both standards have a different approach of parallelizing code execution in terms of shared and distributed memory, the comparisons still required using shared memory systems only.
As expected OpenMP could make more efficient use of the compute resources than MPI. Important to mention here is, that also one variant of the OpenMP implementation still performed worse than the MPI implementation. So, it can be concluded for sure that choosing OpenMP over MPI is <u>not</u> guaranteed to generate faster code on shared memory systems.
At the same time, it could be shown that the provided implementation of MPI uses a very unfortunate partitioning of the mesh, which is assumed to generate immense communication overhead, that could be avoided, and then again a comparison between OpenMP and MPI would be required.
Nevertheless, the serial solver was used for a base line, as well as to determine compile options that make the serial code execute the fastest. For that it could be shown that compile options can greatly influence the code execution time. Whether these found compile options are really the most suitable for the parallelized solvers was then not tested, but rather assumed.
Furthermore, it could be shown, that increasing the number of threads (OpenMP) or processes (MPI) does not always decrease the runtime of a solver. It was shown that for OpenMP (scheduling approaches could not be evaluated due to technical issues,) chunk sizes and especially handling of potential data races greatly influence the runtime.
Finally, it could be shown that also the amount of computation is influencing the parallel efficiency of a code. As expected, the larger the number of computations the more efficient the parallelization can potentially scale. This behavior could be observed for the OpenMP as well as the MPI solvers.

From this point in time, it would be possible, maybe even necessary, to improve at least the implementation of the MPI solver. Especially the implementation of the partitioning shows significant potential for improvement. After that a new evaluation of the runtimes might be more reliable.

5 References

Intel. (2022, 07 21). *C++ Compiler Developer Guide and Reference*. Retrieved from C++ Compiler Developer Guide and Reference: https://www.intel.com/content/www/us/en/develop/documentation/cpp-compiler-developer-guide-and-reference/top.html

OpenMP Architecture Review Board. (2021, November). *OpenMP Application Programming Interface*. Retrieved from https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5-2.pdf

Schuster, M. (2022, Summer Semester). Final Project - Parallelization of a FE code using OpenMP and MPI.