Project Report

Final project of the lecture
Parallel Computing for Computational Mechanics [23ss-41.07360]
At RWTH Aachen University

Author

Prashant Sharma, 427436

Summer Semester 2023

Abstract

This document is a report to documents the results of the tasks assigned in the final project of the lecture **Parallel Computing for Computational Mechanics [23ss-41.07360]** at RWTH Aachen University, in summer semester 2023.

The main objective behind the project was to gain an understanding of how parallel computing can be used for reducing simulation time for various scientific and engineering problems, and what are the advantages and challenges that come with that.

The problem considered for this study is the transient simulation of a temperature distribution in a 2D circular domain, using Finite-element method (FEM).

For comparing the simulation time, the problem is solved by using serial and parallelized codes. All the required codes are provided, and these codes have already been written using C++ as the programming language. Parallelization had been realized using OpenMP and MPI specifications.

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. 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 various compiler options used for generating the code can have significant influence on the computation time.

Keywords: Parallel computing, FEM, OpenMP, MPI, CPU cores, computational time.

Table of Contents

Abstract 2

Abbreviations 4

1 Introduction 5

2 Theory and Methods 5

2.1 Heat equation and domain of interest 5

2.2 pre-, postprocessing and solver 6

2.3 Parallel programming 7

3 Project Tasks 7

3.1 Serial solver 8

3.1.1 Comparison of analytical and numerical solution 8

3.1.2 Comparison of runtime using different compiler options 8

3.2 Parallel solver-OpenMP 10

3.3 Parallel solver -MPI 14

4 Conclusion 20

5 References 21

Abbreviations

2D	Two-dimensional
FEM	Finite element method
OpenMP	Open Multi-processing, programming standard for shard memory system
MPI	Message-Passing-Interface, programming standard for distributed memory systems
PC	Personal computers
SIMD	Single instruction multiple data

1 Introduction

Simulation plays a very important role in scientific research and model development. Simulation involves the study and analyzing the model under different test conditions. However, as the geometry of model and the test condition become more and more complex, it become nearly impossible to analyze the problem on standard personal computers (PC) because of the computing power constraints which directly influence computational time. Utilizing multiple cores of a computer unit has become the state of the art over the last few decades. This is only possible if the simulation code can 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 and Methods

2.1 Heat equation and domain of interest

In the following section the heat equation is solved on a 2D disk over domain Ω with boundary Γ. (see Figure 1 for details).

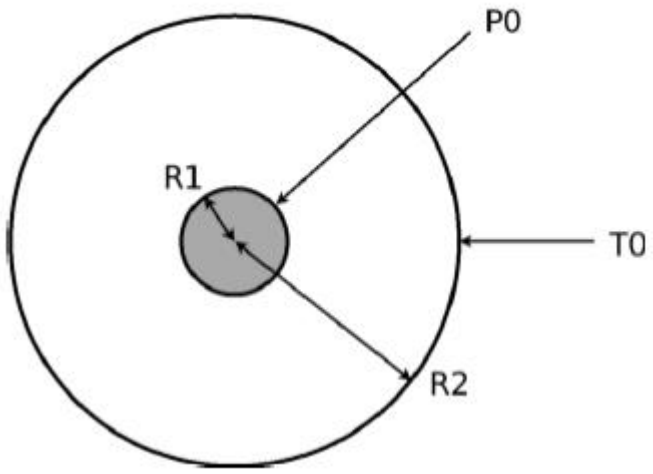


Figure 1 Problem domain

The different parameters for the problem are shown in Table 1.
The standard heat equation for such problem is as follow:

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

$$T(x,t) = T_0 \qquad \text{on } \Gamma \quad \forall t \in (0, t_f), \qquad (2)$$

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

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_0 (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:

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

with r being the radial coordinate of the domain, since there is rotational symmetry, and Q being the heat source. Table 1 shows the values of the different parameters.

Table 1 Parameters for heated disk problem

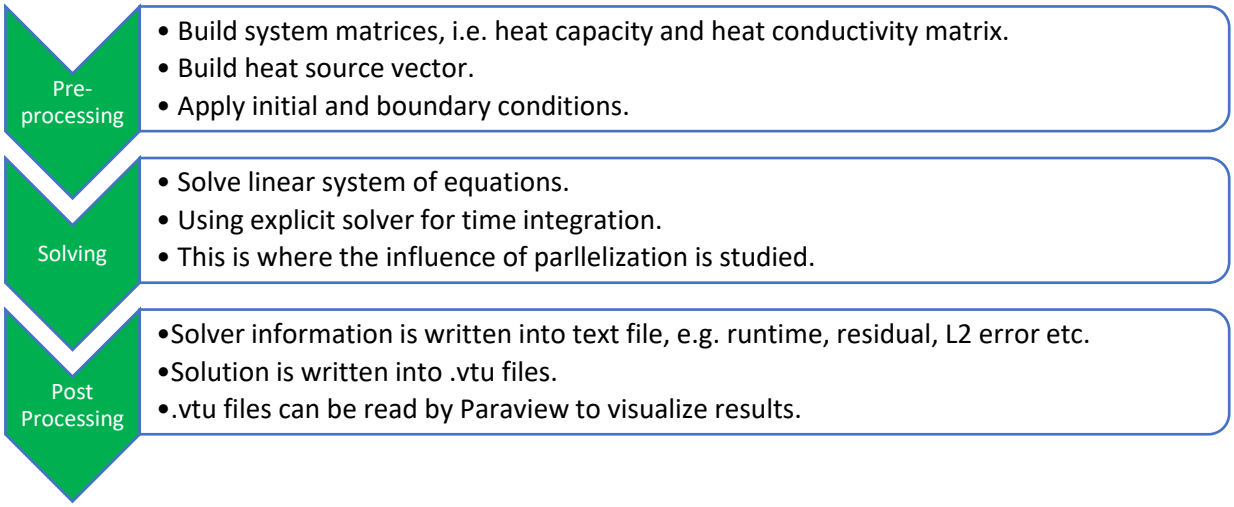
Parameter	Variable	Value	Unit
Inner circle radius	R1	0.01	m
Outer circle radius	R2	0.1	m
Heat source	Q	100	W
Dirichlet boundary	T_O	500	K
Initial condition	T_0	0	K
Thermal diffusivity	κ	1.0	m^2/s
Disk thickness	dz	0.1	m

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

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

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 and accuracy also get effected because of this variability. The qualitative workflow looks as follows:



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

$$T_{i+1} = M^{-1}(M * T_i + \Delta t\,(f - K * T_i) \tag{6}$$

$$T_{i+1} = M^{-1} * Iteration\,Vector_i \tag{7}$$

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 is given. All the required code are provided 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 were 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 compiler options should be analyzed. Therefore, it was necessary to recompile each of the solvers, mentioned above, several times.
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 sub steps .I and .II are only important for the parallelization.

3 Project Tasks

This section and sub-section are meant to answer the given tasks for the project. It includes a quick review of the task, as well as their corresponding results. A significant part of the project focuses on employing optimization of compile time, i.e., using different compiler options, to allow the compiler to come up with fast execution 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 section 3.3, where different parallelization strategies and implementations will be compared.

The three different meshes all use triangular elements with linear shape functions. The following table gives the number of nodes and elements corresponding to each type of meshes.

Table 2 Mesh data for the three different FE meshes.

Mesh Type	Number of nodes	Number of elements
Coarse	2,041	3,952
Medium	21,650	21,332
Fine	85,290	87,656

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. Investigating the discrete values, it is observed that the maximum deviation is less than 0.38K. By increasing distance from the center this difference becomes even smaller.

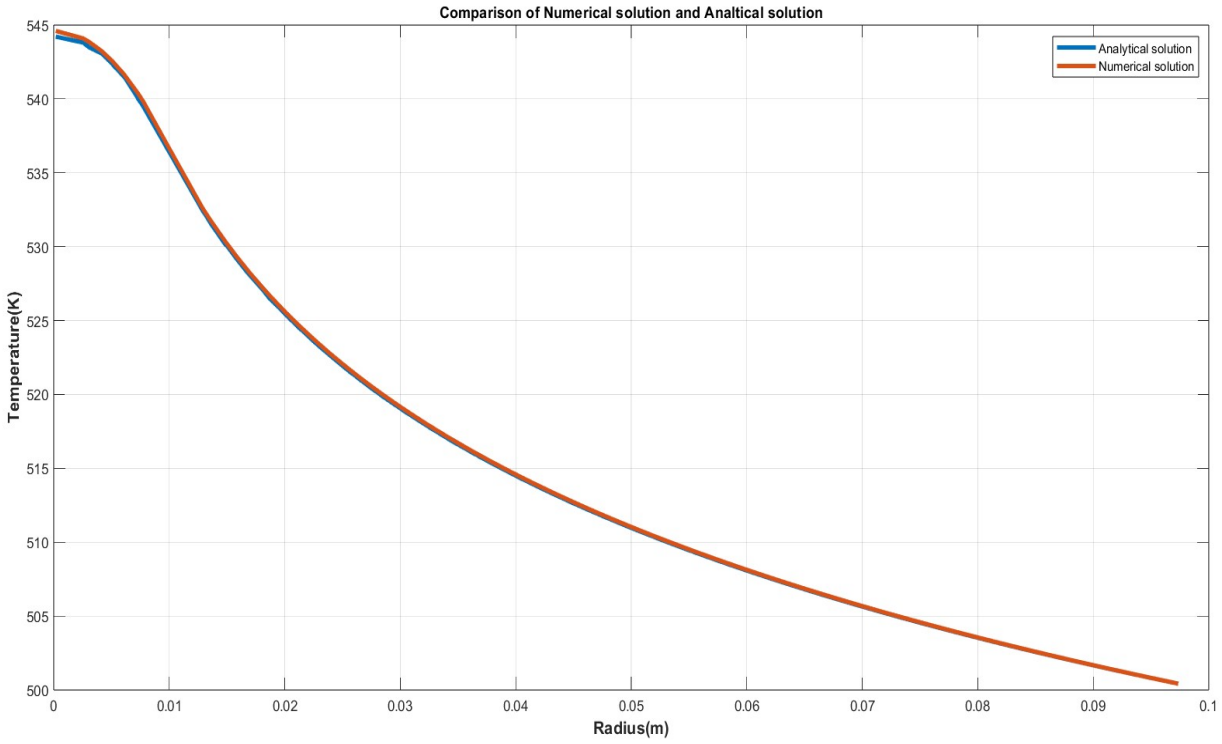


Figure 2 Comparison of analytical and numerical (simulation) solution of the heat equation over the radial distance from the disk´s center.

3.1.2 Comparison of runtime using different compiler options

Secondly, multiple solvers should be compiled using single or combinations of multiple compiler options. The best combination of compiler options is the one which has the least run-time 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 Single instruction multiple data (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. The extension AVX512 specifies vectorization. 512 stands for vector registers with 512bit, so that SIMD instructions can be executed with 512 bit 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.
- -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.
- -inline-level=2 as the name says, specifies how the compiler is allowed to perform function inlining. Level 0 disables the 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 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	Runtime	Speed-Up	Mesh
Serial	-O0	2.23769		Coarse
	-O0	349.23586	1	Medium
	-m64	54.05784	6.460411	Medium
	-O0,-m64	364.848551	0.957208	Medium
	-O0,-vec	344.63977	1.013336	Medium
	-O1	147.21573	2.372273	Medium
	-O2	54.16532	6.447592	Medium
	-O3	55.27819	6.317788	Medium
	-fp-model=fast=1	55.68518	6.271612	Medium
	-fp-model=fast=2	54.20284	6.443128	Medium
	-inline-level=2	55.84134	6.254074	Medium
	-O2, -xSKYLAKE-AVX512	51.82664	6.738539	Medium
	-O2, -xSKYLAKE-AVX512, -inline-level=2	60.78497	5.745431	Medium
	-O2, -xSKYLAKE-AVX512, -inline-level=2,-unroll	50.8081	6.873626	Medium
	-O3, -xSKYLAKE-AVX512 -inline-level=2	55.33656	6.311123	Medium
	-O3, -xSKYLAKE-AVX512 -inline-level=2, -unroll	52.49014	6.653361	Medium
	-unroll	54.11633	6.453428	Medium
	-xSKYLAKE-AVX512	51.60565	6.767396	Medium
	-O3, -xCORE -AVX512 -fast (build IT Rec)	54.09868	6.455534	Medium
	-O3 -ip -xSSE4.2 -axCORE-AVX2, AVX-fp-model fast =2 (build Intel Rec)	56.51721	6.357675	Medium

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 one thread and shortest being observed with 2 threads.

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	Threads	Runtime (s)	Speed-up
OpenMP_A	-O2, -xSKYLAKE-AVX512, -inline-level=2, -unroll	Coarse	1	8.86988	0.25228
	-O2, -xSKYLAKE-AVX512, -inline-level=2, -unroll	Coarse	2	37.81932	0.059168
	-O2, -xSKYLAKE-AVX512, -inline-level=2, -unroll	Coarse	4	77.98042	0.028696
OpenMP_B	-O2, -xSKYLAKE-AVX512, -inline-level=2, -unroll	Coarse	1	1.14506	1.954212
	-O2, -xSKYLAKE-AVX512, -inline-level=2, -unroll	Coarse	2	0.65686	3.406647
	-O2, -xSKYLAKE-AVX512, -inline-level=2, -unroll	Coarse	4	0.71443	3.132133

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.

As explained 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.

For the next task, it is asked to run the simulation with solver with the OpenMP_Task_B option with different types of scheduling and chunk sizes. Use the finest mesh and a constant number of threads of 4. Table 5 represent the recorded Runtime for different type of meshes using different chunk size (i.e., 128, 256, 512, 1024 and 2048). Minimum runtime is observed with chunk size of 2048 for static scheduling. (Marked with green)

Table 5 Comparison of runtime of OpenMP solver variant (b) using different chunk sizes for different scheduling with constant number of threads on fine mesh.

Code	Compiler Flags	Mesh	Chunk size	Scheduling Type	Runtime (s)	Speed-up
OpenMP_B	-O2 -xSKYLAKE-VX512, -inline-level=2, -unroll	Fine	-	Static	212,42571	0,959617082
	-O2 -xSKYLAKE-VX512, -inline-level=2,-unroll	Fine	128	Static	197,86161	1,030252104
	-O2 -xSKYLAKE-VX512, -inline-level=2,-unroll	Fine	256	Static	192,21166	1,060535766
	-O2 -xSKYLAKE-VX512, -inline-level=2,-unroll	Fine	512	Static	205,09971	0,993893848
	-O2 -xSKYLAKE-VX512, -inline-level=2,-unroll	Fine	1024	Static	201,03788	1,013974779
	-O2 -xSKYLAKE-VX512, -inline-level=2,-unroll	Fine	2048	Static	186,45525	1,093277556
	-O2 -xSKYLAKE-VX512, -inline-level=2,-unroll	Fine	4096	Static	198,07043	1,029165939
	-O2 -xSKYLAKE-VX512, -inline-level=2,-unroll	Fine		Auto	190,3293	1,071024482
OpenMP_B	-O2 -xSKYLAKE-VX512, -inline-level=2,-unroll	Fine	128	Dynamic	194,1855	1,049755723
	-O2 -xSKYLAKE-VX512, -inline-level=2,-unroll	Fine	256	Dynamic	203,7698	1,000380527
	-O2 -xSKYLAKE-VX512, -inline-level=2,-unroll	Fine	512	Dynamic	196,39251	1,037958831
	-O2 -xSKYLAKE-VX512, -inline-level=2,-unroll	Fine	1024	Dynamic	200,80608	1,015145259
	-O2 -xSKYLAKE- VX512, -inline-level=2,-unroll	Fine	2048	Dynamic	198,74285	1,025683893
	-O2 -xSKYLAKE-VX512, -inline-level=2,-unroll	Fine	4096	Dynamic	203,45525	1,001927156
OpenMP_B	-O2 -xSKYLAKE-VX512, -inline-level=2,-unroll	Fine	128	Guided	219,00636	0,930782741
	-O2 -xSKYLAKE- VX512, -inline-level=2,-unroll	Fine	256	Guided	228,61911	0,89164611
	-O2 -xSKYLAKE- VX512, -inline-level=2,-unroll	Fine	512	Guided	220,09009	0,926199539
	-O2 -xSKYLAKE- VX512, -inline-level=2,-unroll	Fine	1024	Guided	233,32734	0,873653898
	-O2 -xSKYLAKE- VX512, -inline-level=2,-unroll	Fine	2048	Guided	214,99634	0,948143303
	-O2 -xSKYLAKE- VX512, -inline-level=2,-unroll	Fine	4096	Guided	225,59794	0,903586886

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. Table 6 shows said findings. 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 drastically with the number of threads.

Figure 3 and Figure 4 show the speed up and parallel efficiency of the given runtimes. 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, efficiency is also the smallest. Then with increasing mesh granularity speed up as well as efficiency increase, with highest efficiency for the fine mesh.

Table 6 Comparison of runtime of OpenMP solver variant (b) using different chunk size 2048 for static scheduling with different number of threads.

Code	Compiler Flags	Mesh	Chunk size	Scheduling Type	Threads	Run-time	Speed-up	Efficiency
OpenMP_B	-O2 -SKYLAKE-AVX512, -inline-level=2,-unroll	Fine	2048	Static	1	661,0312	1	1
	-O2 -xSKYLAKE-VX512, -inline-level=2,-unroll	Fine	2048	Static	2	338,67504	1,951816	0,975908
	-O2 -xSKYLAKE-VX512, -inline-level=2,-unroll	Fine	2048	Static	4	201,47541	3,280952	0,487954
	-O2 -xSKYLAKE-VX512, -inline-level=2,-unroll	Fine	2048	Static	6	135,45139	4,88021	0,546825
	-O2 -xSKYLAKE-VX512, -inline-level=2,-unroll	Fine	2048	Static	8	122,5129	5,395605	0,610026
	-O2 -xSKYLAKE-VX512, -inline-level=2,-unroll	Fine	2048	Static	12	105,22431	6,282115	0,449634
OpenMP_B	-O2 -xSKYLAKE-VX512, -inline-level=2,-unroll	Medium	2048	Static	1	58,49533	1	1
	-O2 -xSKYLAKE-VX512, -inline-level=2,-unroll	Medium	2048	Static	2	37,97434	1,540391	0,770195
	-O2 -xSKYLAKE-VX512, -inline-level=2,-unroll	Medium	2048	Static	4	33,29632	1,756811	0,439203
	-O2 -xSKYLAKE-VX512, -inline-level=2,-unroll	Medium	2048	Static	6	29,35752	1,992516	0,332086
	-O2 -xSKYLAKE-VX512, -inline-level=2,-unroll	Medium	2048	Static	8	25,654	2,280164	0,285021
	-O2 -xSKYLAKE-VX512, -inline-level=2,-unroll	Medium	2048	Static	12	22,02613	2,655724	0,22131
OpenMP_B	-O2 -xSKYLAKE-VX512, -inline-level=2,-unroll	Coarse	2048	Static	1	1,28074	1	1
	-O2 -xSKYLAKE-VX512, -inline-level=2,-unroll	Coarse	2048	Static	2	1,695910	0,755193	0,377597
	-O2 -xSKYLAKE-VX512, -inline-level=2,-unroll	Coarse	2048	Static	4	0,87116	1,470155	0,367539
	-O2 -xSKYLAKE-VX512, -inline-level=2,-unroll	Coarse	2048	Static	6	1,09314	1,171616	0,195269
	-O2 -xSKYLAKE-VX512, -inline-level=2,-unroll	Coarse	2048	Static	8	1,51608	0,844771	0,105596
	-O2 -xSKYLAKE-VX512, -inline-level=2,-unroll	Coarse	2048	Static	12	1,16212	1,102072	0,091839

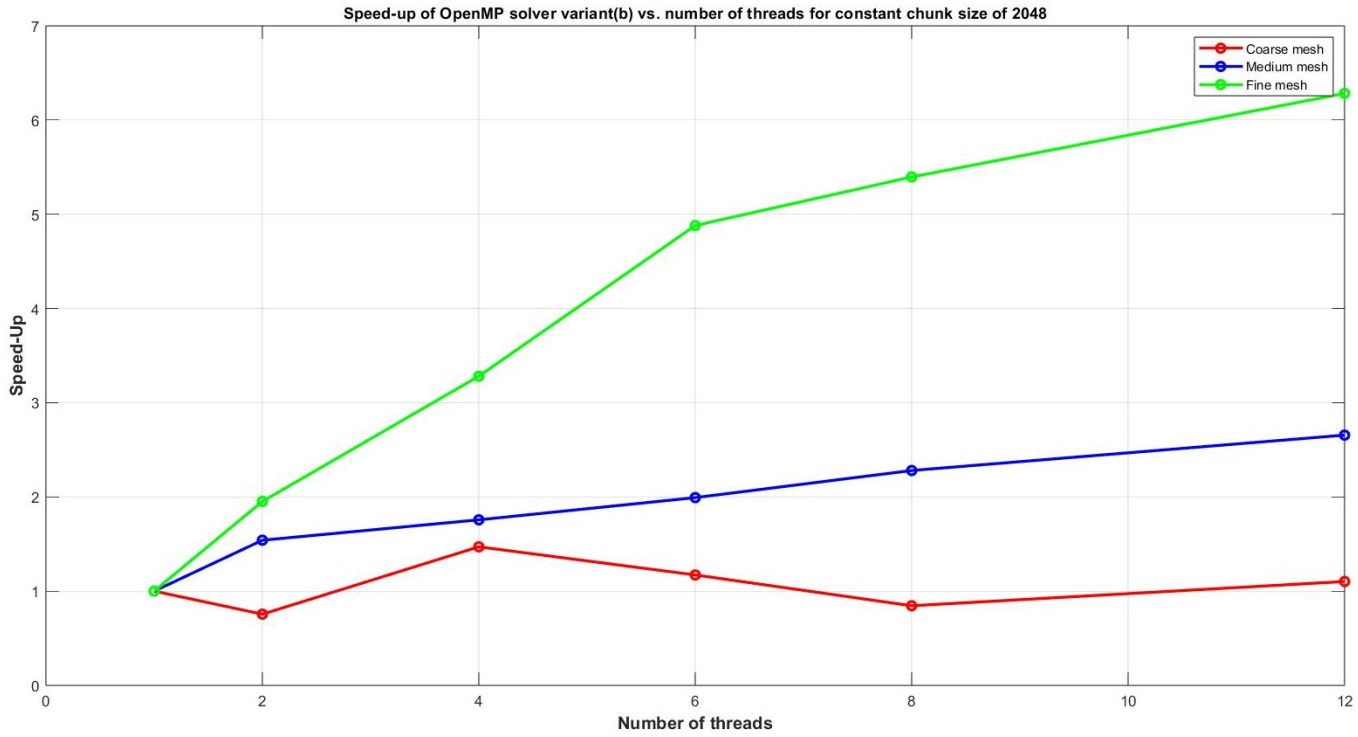


Figure 3 Comparison of speed-up for different types meshes of OpenMP solver variant B vs number of threads with constant chunk size of 2048 for Static scheduling.

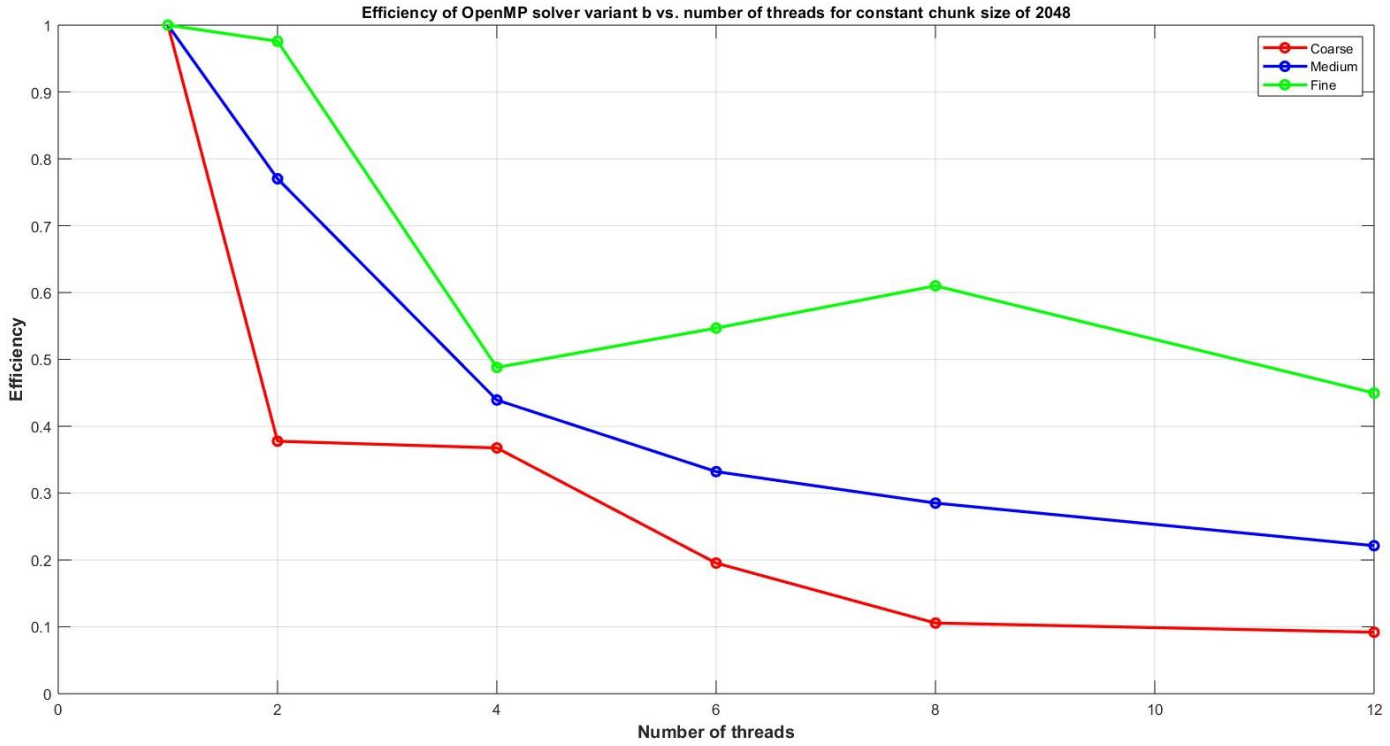


Figure 4 Comparison of efficiency for different types meshes of OpenMP solver variant B vs number of threads with constant chunk size of 2048 for static scheduling.

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.

3.3 Parallel solver -MPI

The next task of the project is focused on parallelization using the MPI standard. The results are generated using the most optimized complier flag from section 3.1.2 and meshes are the same as used in previous section.

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 CPU cores varied between 1 and 16. Figure 5, Figure 6 and Figure 7 show the comparison of runtime, speed up and parallel efficiency. From the obtained runtimes, 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 the number of cores. 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 six 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 because 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).

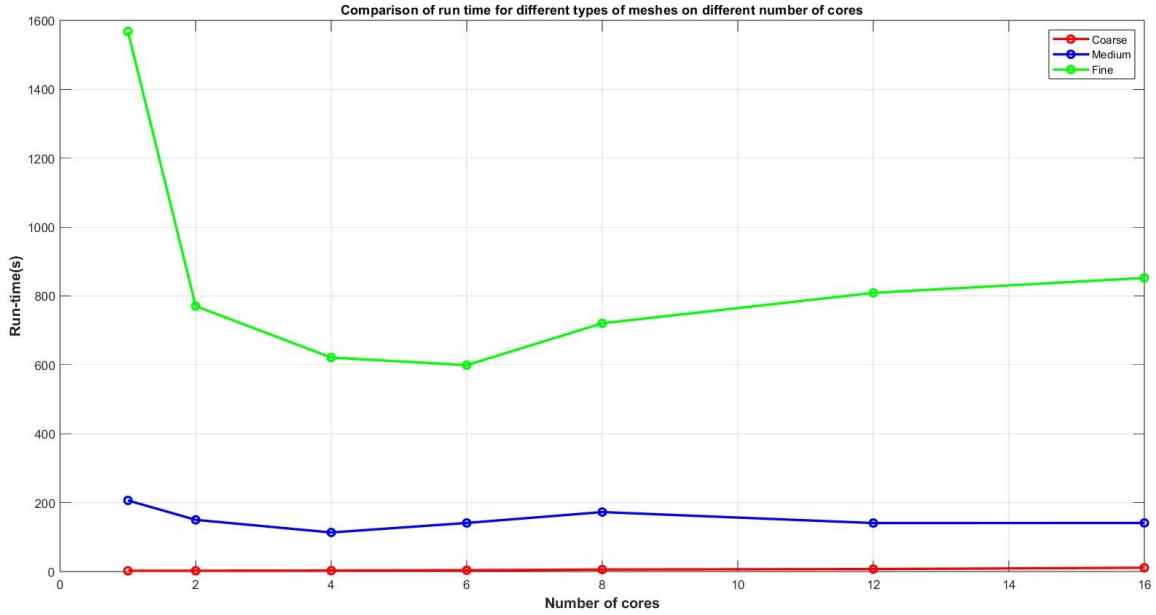


Figure 5 Comparison of speed up of MPI parallelized solver over number of threads for different meshes

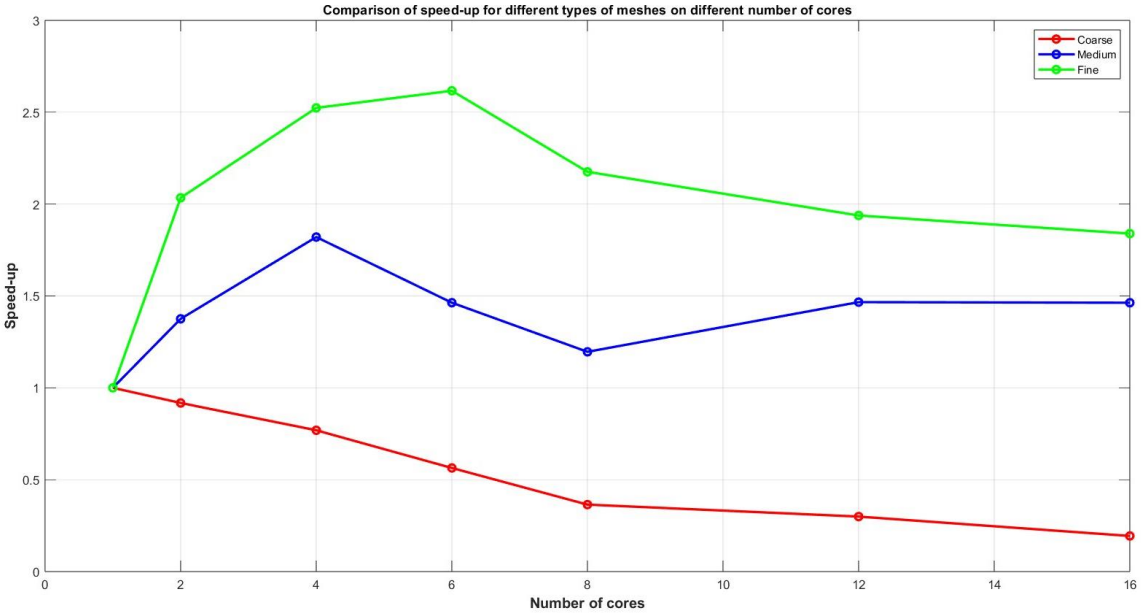


Figure 6 Comparison of speed up of MPI parallelized solver over number of threads for different meshes.

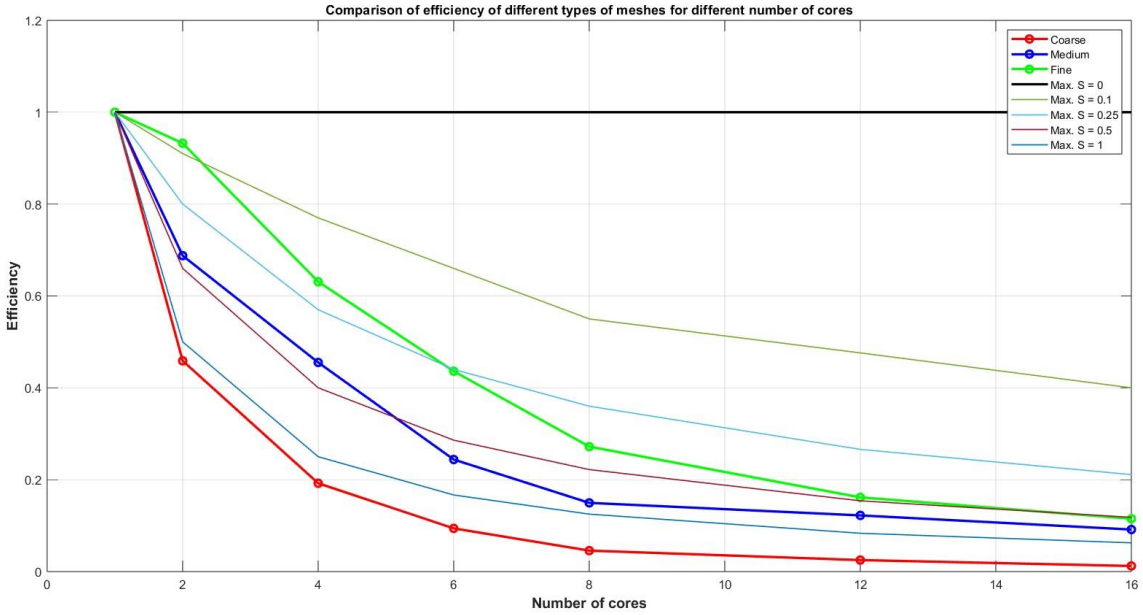


Figure 7 Comparison of parallel efficiency of MPI parallelized solver over number of threads for different meshes and theoretical results according to Amdahl´s law with S being fraction of the code that is inherently serial, and 1-S being the fraction that is perfectly parallelizable, curves with markers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

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 6 processes. However, all other number of processes lead to runtimes smaller than using only one process. Since more than six 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, two or four processes would give reasonable speed up, that is only little smaller than using six processes.

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 black 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 3.1.2 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, the MPI solver led to longer runtimes, in any of the comparable cases. All this gives rise to the fact 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 8, improving the communication itself between the processes might not give the desired speed up because of the previously mentioned obstacles of the unfortunate partitioning.

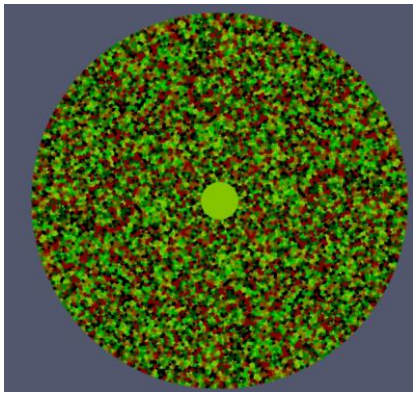


Figure 8 Visualization of the partitioning of the medium mesh using 8 processes with the MPI solver.

Additional Questions:

Q1: What is the content of the mien file in the mesh format we used in Homework 1 and why do we need it?
Ans.
The mien file contains the connectivity information for the mesh. It provides information about nodes networks (or how node is connected to other nodes, how the solution at one node will affect the solution at one other connected node).

Q2: Why do we need to use the loop over for (int p = 0; p < nGQP; p + +) in the routine calculateElementMatrices in solver.cpp, starting around line 139?
Ans.
The loop is used to perform the numerical integration for all elements of M, K and F matrices, where nGQP is the number of Gauss quadrature points. Numerical integration is a standard technique to approximate the complex analytical integration using very few Gauss quadrature points. The order of accuracy for nGQP is 2*nGQP -1.

Q3: What happens if we do not specify any compiler flag in CMakelLists.text for the heat equation solver when compiling with Intel compiler? Do you expect the code to exhibit optimized or non-optimized behaviour?
Ans.
When no compiler flag is specified in CMakeLists.text for this problem while compiling with intel compiler, the Runtime for this case is 54.93138 sec with medium meshes size. When compared this with runtime for different compiler from Table 3. This runtime is very close to -O3 compiler runtime (55.27819), which is far less than non-optimized runtime.

Q4: Why do we need the variables nec, mec, mnc, nnc in the MPI context? What do they do?
Ans.
nec contains the information about the number of elements on the current processor.
mec contains information about the number of elements across all the processors.
nnc contains the information about the number of nodes per processor.
mnc contains the information maximum number of nodes across all processors.
These variables are quite crucial for Load Balancing, communication, Data Exchange, Memory allocation and Performance optimization.

Q5: Answer the following MPI related questions in written form: I) Why do we use MPI_Accumulate in the code instead of MPI_Put? II) What are advantages of using one-sided MPI communication over two-sided MPI communication? III) What advantage does MPI offer over OpenMP, what is a main obstacle of MPI?

Ans.

I) 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.

II) 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 can 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.

III) The answer to this question lies 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 many 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 manually parallelizing the code and taking 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 computer units. If then sometime later it would be desirable to also make use of distributed systems, OpenMP code would require to be built again, almost from scratch. If the code had 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.

Q6: What are the default settings for scheduling in OpenMP? What is the default chunk sizes for static, dynamic and guided?

Ans.

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 tradeoff 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.

Q7: Can you think of a case where you would prefer the non-optimized code over the optimized one? What are potential drawbacks of using optimization flags?

Ans.
One reason to use non-optimized code is debugging. Since compiler optimization can change the code to a specific extent, 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 an impact on the accuracy of the calculation. And finally relying on the compiler to optimize the code might lead to programmers not fully understanding how the code works. Optimization by the programmer requires a deep understanding of the code itself.

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 computing 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 not 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 executes 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 the amount of computation is also 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. (2023, Summer Semester). Final Project - Parallelization of a FE code using OpenMP and MPI.