Final Report

Parallel Computing for Computational Mechanics

Johannes Grafen 380149
johannes.grafen@rwth-aachen.de

Abstract: Lorem ipsum dolor sit amet, consectetuer sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua. At vero eos et accusam et justo duo dolores et ea rebum. Stet clita kasd gubergren, no sea takimata sanctus est Lorem ipsum dolor sit amet. Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut lre et dolore magna aliquyam erat, sed diam voluptua. At vero eos et accusam et justo duo dolores et ea rebum. Stet clita kasd gubergren, no sea takimata sanctus est Lorem ipsum dolor sit amet. Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua. At vero eos et accusam et justo duo dolores et ea rebum. Stet clita kasd gubergren, no sea takimata sanctus est Lorem ipsum dolor sit amet.

1 Introduction

The parallelization of code for the simulation of complex mechanical systems is a key-topic in modern computational mechanics. Constantly increasing computing power of high-performance clusters allow for the simulation of large and complex systems. This requires an efficient parallelization approach, where different parallelization paradigms can be of use.

In this report the parallelization of a finite element code for the simulation of the stationary temperature distribution on a two-dimensional disk is investigated. For the shared memory approach the OpenMP library is used, where as the distributed memory approach is assessed by using a one-sided parallelization approach with MPI. The different parallelization approaches are discussed and compared for different number of CPUs. This report is structured as follows. In the second chapter *Theory and Methods* are presented. The *Implementation and Validation* of the presented problem is discussed in chapter 3. The results of the optimization of the runtime of the code is illustrated in chapter 4 *Results* and the assessment of the different optimization techniques is discussed in chapter 5 *Discussion*. Finally, this report is closed with a *Conclusion* in chapter 6.

serial optimization is performed by

2 Theory and Methods

- describe different meshes

3 Implementation and Validation

Parallelization with OpenMP

A critical region is to be executed by only one thread at a time. This is important to prevent data races. A data race means that multiple threads try to access and manipulate the same variable / memory address at the same time. This might lead to unpredictable behaviour, which can result in incorrect computations.

Listing 1: Parallelization of two most time consuming loops with implementation A and B

```
1     for (int iter=0; iter<nIter; iter++)
2     {
3         // clear RHS MTnew
4         A) #pragma omp parallel for
5         for(i=0; i<nn; i++){
6             MTnew[i] = 0;
7         }
8         A) #pragma omp parallel firstprivate(MTnew)
9         B) #pragma omp parallel
10        {
11            // Evaluate right hand side at element level
12            A) #pragma omp for private(elem, M, F, K, TL, i, MTnewL)
13            B) #pragma omp for private(elem, M, F, K, TL, i, MTnewL)
14            B) reduction(+: MTnew[0:nn]) schedule(dynamic,512)
15            for(int e=0; e<ne; e++)
16            {
17                elem = mesh->getElem(e);
18                M = elem->getMptr();
19                F = elem->getFptr();
20                K = elem->getKptr();
21                for(i=0; i<nen; i++)
22                {
23                    TL[i] = T[elem->getConn(i)];
24                }
25
26                MTnewL[0] = M[0]*TL[0] + dT*(F[0]-(K[0]*TL[0]+...));
27                MTnewL[1] = M[1]*TL[1] + dT*(F[1]-(K[3]*TL[0]+...));
28                MTnewL[2] = M[2]*TL[2] + dT*(F[2]-(K[6]*TL[0]+...));
29
30
31                // RHS is accumulated at local nodes
32                A) #pragma omp critical
33                MTnew[elem->getConn(0)] += MTnewL[0];
34                A) #pragma omp critical
35                MTnew[elem->getConn(1)] += MTnewL[1];
36                A) #pragma omp critical
37                MTnew[elem->getConn(2)] += MTnewL[2];
38            }
39            // Evaluate the new temperature on each node on partition level
40            partialL2error = 0.0;
41            globalL2error = 0.0;
42            A) #pragma omp for private(pNode, massTmp, MT, Tnew)
43            B) #pragma omp for private(pNode, massTmp, MT, Tnew)
44            B) reduction(+:partialL2error) schedule(dynamic,512)
45            for(int i=0; i<nn; i++)
46            {
47                pNode = mesh->getNode(i);
48                if(pNode->getBCtype() != 1)
49                {
50                    massTmp = massG[i];
51                    MT = MTnew[i];
52                    Tnew = MT/massTmp;
53                    A) #pragma omp critical
54                    partialL2error += pow(T[i]-Tnew,2);
55                    T[i] = Tnew;
56                }
57            }
```

```
58          }
59
60          globalL2error = sqrt(partialL2error/nn);
```

Parallelization with MPI

4 Results

4.1 Serial Solver

To run the finite element solver as efficient and fast a possible in parallel, it is important that the serial performance is optimized. Therefore, the effect on the runtime of multiple compiler flags is compared in this section.

R1 a)

For the validation of the FEM solver, the solution of the coarse mesh is compared to the analytical solution in Fig. 1. The numeric solution is in good agreement with the analytical solution. Slight deviations are observed at the disk's center where the temperature distribution has it's global maximum. Accounting for large changes in the temperature gradient at the center of the disk.

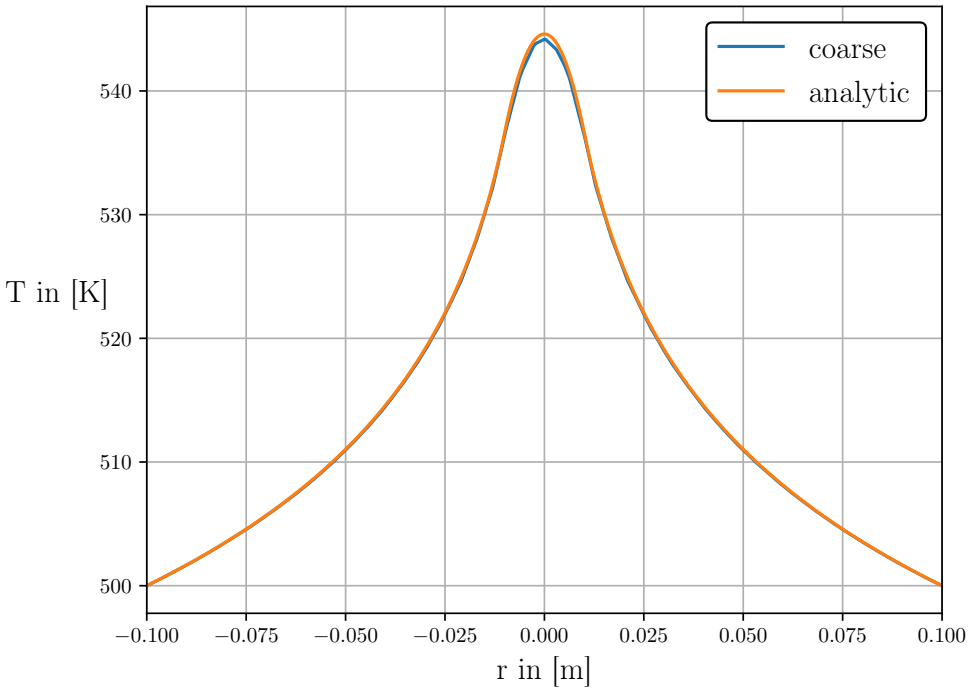


Figure 1: Comparison of the steady-state temperature distribution of the analytical solution (refEQ) and the numerical solution on the coarse mesh

In order to decrease the serial runtime of the implemented solver, different compiler flags were tested and assessed by comparing their influence on the solver runtime. The code is compiled with the Intel C++ Compiler (v19.1). The compiled code is tested on a Intel Platinum 8160 which is based on Intel's Skylake architecture using the "medium" unstructured mesh with 21650 nodes resulting in 21332 triangular elements.

The investigated compiler flags are contained in Tab. 1. In the following subsection the individual compiler flags are briefly presented and explained:

-O0

The capital number behind the capital "O" refers to the optimization level and performs general optimizations by bundling several individual compiler flags. Using the base Level (0), the compiler will disable all optimizations. This is useful for debugging (section R1 b).

-O1

Optimization Level 1 includes the optimization for speed, but disables all optimizations that increase code size and affect the codes speed. It includes the analysis of data-flow, code motion, strength reduction and further performance enhancing options.

-O2

The second optimization level includes vectorization, which allows for concurrent execution of the separate steps that are necessary to perform different basic mathematical operations on array-like data structures. Such as an element wise addition of an array. A scalar or "Von Neumann" processor would simply perform all four necessary steps for each index step by step. Where as a vector processor allows for concurrent operations. This prevents that instruction units are idle, while waiting on a task to be finished.

The option also enables the inlining of intrinsics and intra-file inlining of functions. Which means the replacement of the function call with the instructions inside of a certain function itself. This can cause significant performance gain, especially if functions are called multiple times inside the body of a loop.

-O3

Performs O2 optimizations with higher thresholds and enables more aggressive loop transformations such as Fusion, Block-Unroll-and-Jam, and collapsing IF-statements. It can be used with the option "-ax" to force the compiler to do a more aggressive data dependency analysis and optimize for a certain CPU architecture. Where "x" refers to the CPU's instruction set.

-Ofast

On Linux systems this sets "-O3 -no-prec-div -fp-model fast=2". This causes the compiler to enhance the code with alle possible optimizations of "-O3" and reduce the precision of floating-point divides.

-ax *arch*

Optimization of instruction set for a specific hardware architecure of the CPU. As the simulations were conducted on a two-socket node with two Intel Platinum 8160 with each 24 cores, based on the Skylake architecture. The shortest runtime was achieved with the flag "-axSKYLAKE-AVX512". Where AVX512 is the abbreviation for *Advanced Vector Extensions* with 512-bit instructions and is an extension of the SIMD instruction set.

-fp-model fast

This flag controls the precision of floating-point operations. Using Level 2, enables a more aggressive optimization of floating-point calculations. This might decrease accuracy.

-ipo

Enables the interprocedural optimization between files functions from different files may be inlined.

-unroll

Performs loop unrolling which helps to exploit multiple instruction units, by increasing the stride length of the loop variable and performing the same operation on multiple array elements in one stride.
[1, 2, 3, 4]

The runtimes that were measured for the different compiler flags are shown in Tab. 1. For each flag, three measurements on a single core with a single thread where taken and the average time was determined. The shortest runtime is achieved by using the compiler flag combination "-O3 -axSKYLAKE-AVX512" resulting in an average runtime of 54.77 seconds which is 4.37 times faster that the not optimised code (-O0). For further investigations, assessing the parallel optimisation with OpenMP and MPI, the "-O3 -axSKYLAKE-AVX512" flag combination was used.

compiler flag	time 1	time 2	time 3	average time
-O0 (no optimisation)	233.71	241.5	243.02	239.41
-O1	62.41	63.71	62.76	62.96
-O2	56.96	55.39	57.11	56.49
-O3	55.49	55.23	57.8	56.17
-O3 -axSSE4.2, SSSE3, SSE2 -fp-model fast=2	56.17	56.21	56.46	56.18
-O3 -fp-model fast=2	57.69	56.19	57.89	57.26
-O3 -axSKYLAKE-AVX512	53.99	54.06	56.25	**54.77**
-O3 -ipo	54.57	57.26	60.38	57.40
-unroll	55.17	57.01	55.94	56.04
-Ofast	55.43	56.17	56.27	55.96

Table 1: Timings of the various compiler flags for serial optimisation, the shortest achieved runtime is printed in bold font

R1 b)

A higher level of optimization can cause a decrease in the accuracy of the floating-point operations. For the validation of a certain code or high fidelity simulations, like the implemented FEM Solver, one is advised to minimize the numerical error, by using the highest level of accuracy. So the optimizations should not inflict with the desired results. Also for debugging purposes, optimization is not helpful as the compiler changes the code structure, when using more aggressive optimizations. To name a few effects: functions are inlined, loops are unrolled or fused together. This makes is very hard to search for bugs, when using debugging software.

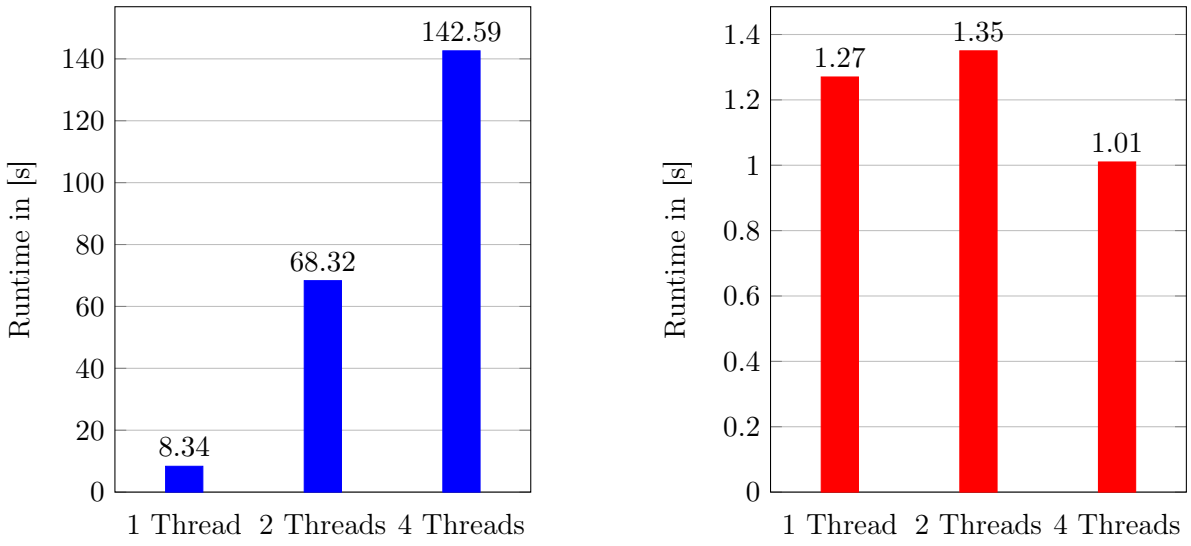
4.2 Parallel OpenMP Solver

In this section the performance gain by parallelization with OpenMP of the two most time consuming loops is investigated (Listing 1).

R2 a

First, two different OpenMP parallelization approaches are compared an discussed. The Approaches are denoted with "A)" and "B)" in the code extract in Listing 1. Approach A) uses critical regions, which are executed by each thread serially for the computation of the RHS at element-level and the partial L2 errors. Where as approach B) is performing a reduction operation for the determination of MTnew and the partial L2 error, using a dynamic scheduling with a chunk size of 512 for both loops.

To compare both approaches, the runtimes for the simulation of the temperature distribution on the coarse mesh is determined for 1, 2 and 4 threads for each parallelization approach. Three runtimes were measured per thread count and averaged. As presented in Fig. 2, the runtimes of the approach A) (Fig. 2a) for different number of threads is in general significantly higher than the runtimes of approach B) (Fig. 2b), with a top runtime of 142.59 seconds for a run with 4 threads for approach A and a runtime of 1.01 seconds for approach B. The scaling behaviour of approach B was found to be as theoretically expected, as the runtime decreases with an increasing number of threads. Due to the very short runtimes the run with 2 threads was measured to be slower than a run with only a single thread. This requires further investigations as possible fluctuations in the runtime, for instance due to memory access, are in the order of the runtime itself. Approach A shows the opposite dependency of thread number and runtime. The runtime increases with an increasing number of threads.



(a) OpenMP task A **(b)** OpenMP task B

Figure 2: Timings for solver with different OpenMP optimisations using the coarse mesh

As the critical regions of approach A) are to be executed by only one thread at a time, the runtime will increase with an increasing number of threads, as observed in Fig. 2a. A large overhead is created due to the management of lock, to ensure that only one thread at a time is running through a critical region. This overhead increases with a higher number of threads. While one thread is executing a critical region the other threads have to wait for this specific thread to finish this section. Only if this thread has completely finished to computations inside a critical region the next thread is allowed to execute this region for its assigned loop indices. To prevent the overhead of multiple locks and to allow threads to work in parallel the reduction keyword for openMP is used. The reduction operation for the computation of the RHS at element level for each element node, allows each thread to first compute

local sums of the each MTnew entry in parallel and then performs a serial summation of all local sums to obtain the value of an MTnew entry. But this serial summation is only computed once, compared to the critical region. The same procedure applies also for the computation of the partial L2 error.

R2 b

To further optimize the parallelization with OpenMP, different scheduling options and chunk sizes are investigated for approach B) (Listing 1). Therefore simulations on the fine mesh with a constant number of 4 threads were conducted. Some unexplained behaviour was observed, when using scheduling options other than static for the second loop in which the partial L2 error are computed, where the L2 error gets initialised again with 0, which leads to an untimely convergence of the simulation. Therefore different scheduling options and chunk sizes were first only applied for the first loop (line 15 Listing 1) for the computation of the RHS on element level. For the second loop (line 45 Listing 1) a static scheduling with default chunk size was used. The results are presented in Fig. 3. In general the static scheduling performs best with a shortest runtime of 184.06 seconds for a chunk size of 256. In general larger runtimes are observed for the dynamic and guided scheduling options. For dynamic scheduling a larger chunk size of 512 iterations leads to shorter runtimes, where as for static and guided scheduling the medium chunk size of 256 iterations performs best. The runtime of the auto scheduling is at 220.97 seconds, which is marked in grey in Fig. 3.

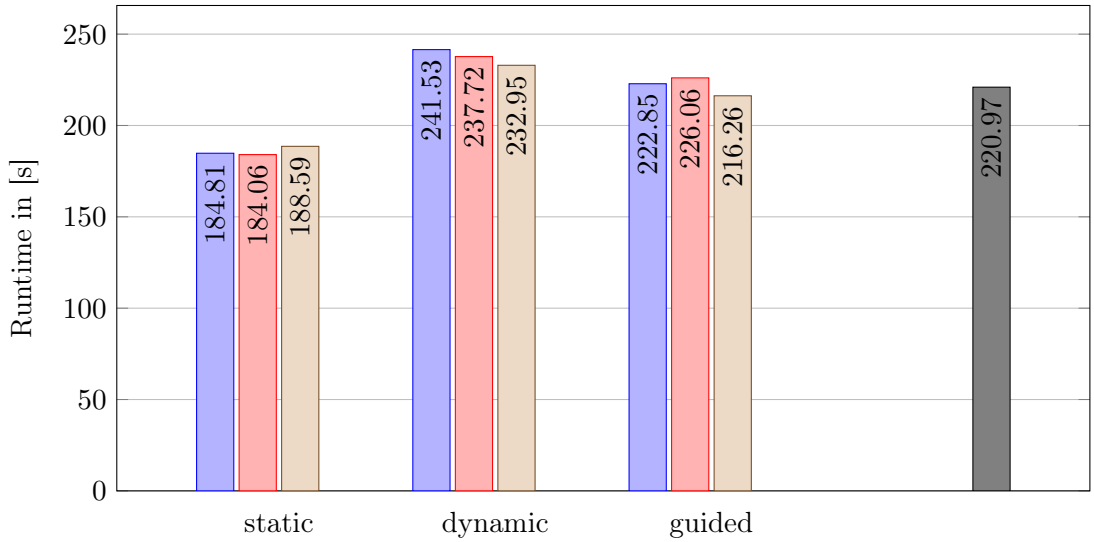


Figure 3: Comparison of different scheduling options (static, dynamic, guided, auto) for different chunk sizes(128, 256, 512) in first parallelised loop for 4 threads using the finest mesh

In addition to the introductory investigations regarding the different scheduling options, static scheduling was inspected further. Therefore the reduction operations for the parallelisation of the first and second loop are both equipped with static scheduling and the effects of different chunk sizes are compared in Fig. 4. The runtime is decreasing with an increasing chunk size. The shortest runtime is observed for the default option in scheduling with 177.21 seconds on average, which divides the number of loop iterations by the number of threads. This results in a chunk size of 21164 iterations for the first loop and in a chunk size of 21322.5 iterations on average for the second loop for the fine mesh with 85290 nodes and 84656 elements. Thus, the static scheduling with the default chunk size is used for the calculation of speed-up and efficiency for the different meshes in the following section.

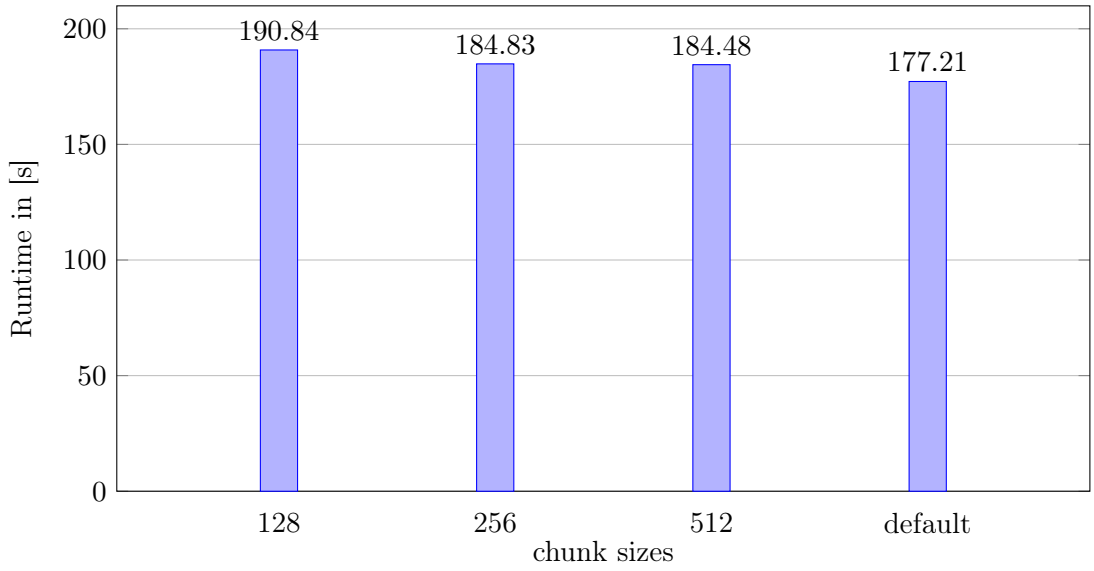
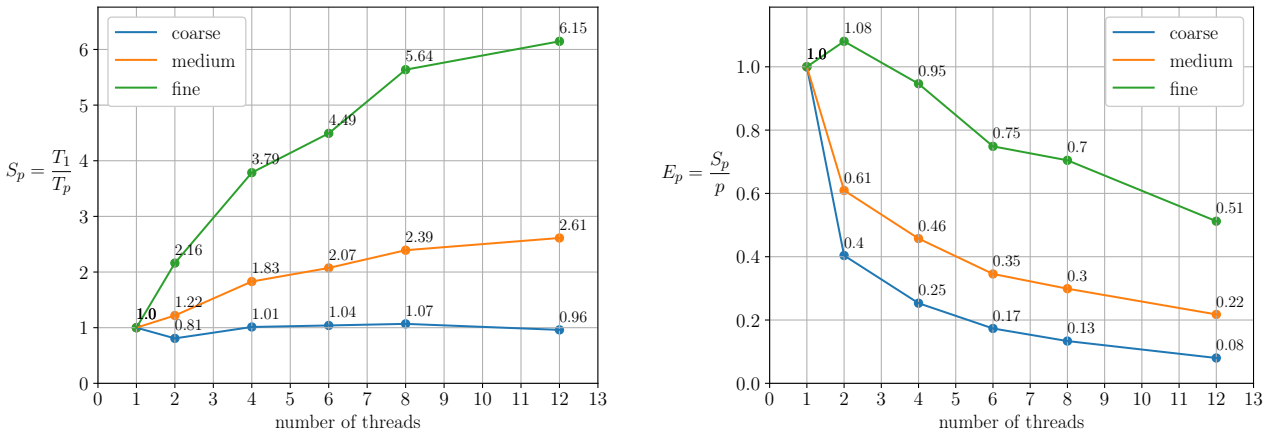


Figure 4: Runtimes for parallelization of loop in line 15 and loop in line 45 in Listing 1 with static scheduling and the same option for chunk size



(a) Speed-up **(b)** Efficiency

Figure 5: Speed-up and efficiencies for coarse, medium and fine mesh for different numbers of threads

R2 c

R2 d

- look into presentation

4.3 Parallel MPI Solver

R3 a

R3 b

- Plots größer

R3 c

R3 d

5 Discussion

6 Conclusion

Lorem ipsum dolor sit amet, consectetuer sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua. At vero eos et accusam et justo duo dolores et ea rebum. Stet clita kasd gubergren, no sea takimata sanctus est Lorem ipsum dolor sit amet. Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut lre et dolore magna aliquyam erat, sed diam voluptua. At vero eos et accusam et justo duo dolores et ea rebum. Stet clita kasd gubergren, no sea takimata sanctus est Lorem ipsum dolor sit amet. Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua. At vero eos et accusam et justo duo dolores et ea rebum. Stet clita kasd gubergren, no sea takimata sanctus est Lorem ipsum dolor sit amet.

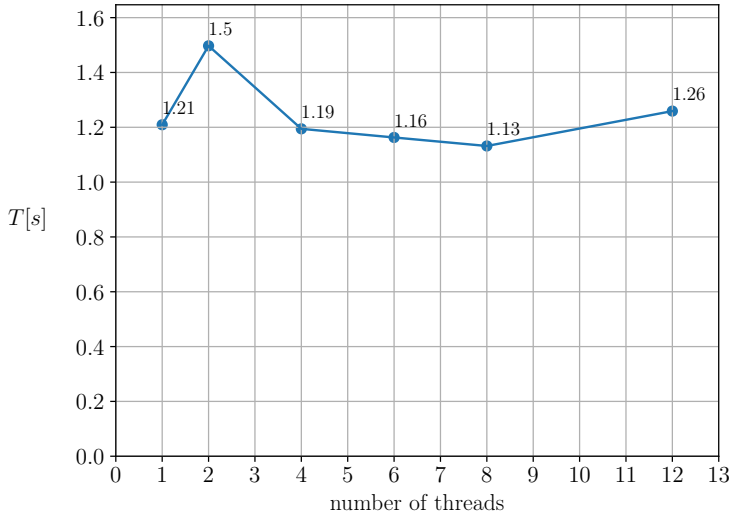
References

[1] Dieter an Mey, Christian Terboven, Paul Kapinos, Dirk Schmidl, Sandra Wienke, and Tim Cramer. The RWTH HPC-Cluster User's Guide Version 8.4.0.

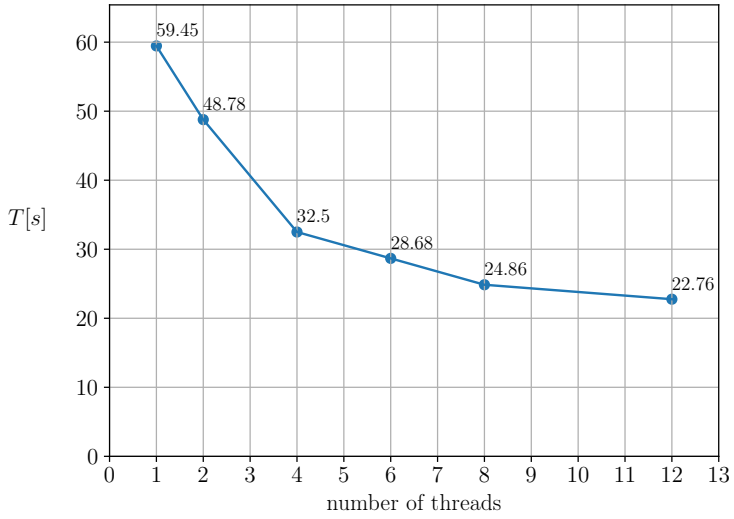
[2] Marek Behr. Lecture: Parallel Computing for Computational Mechanics.

[3] Quick Reference Guide to Optimization with Intel® C++ and Fortran Compilers v19.

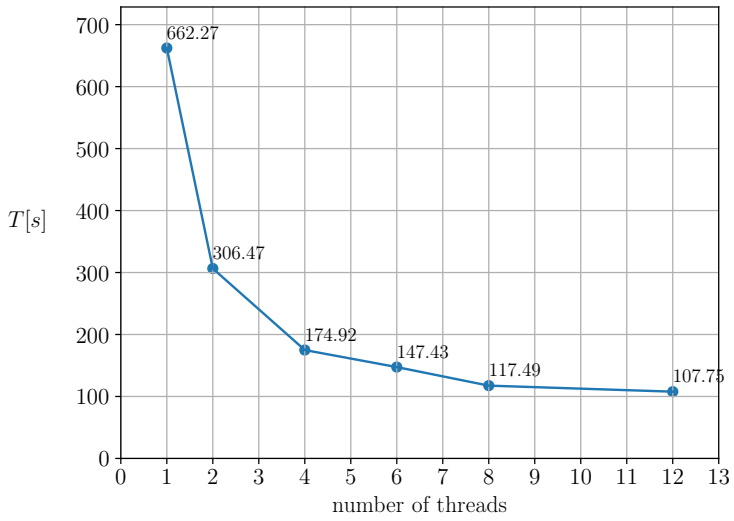
[4] Intel® C++ Compiler Classic Developer Guide and Reference, April 2022.



(a) coarse mesh

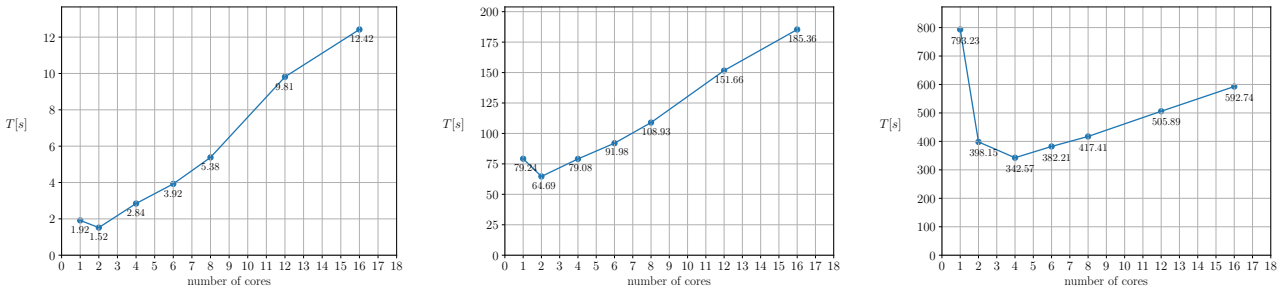


(b) medium mesh



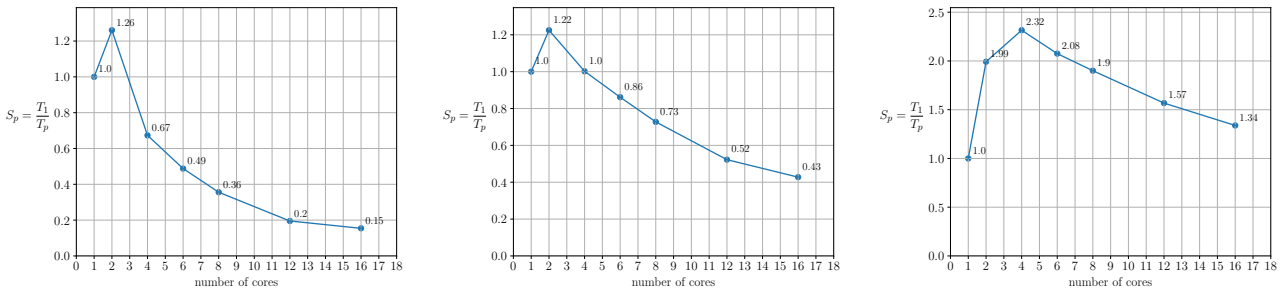
(c) fine mesh

Figure 6: Runtime for coarse, medium and fine mesh for different number of threads



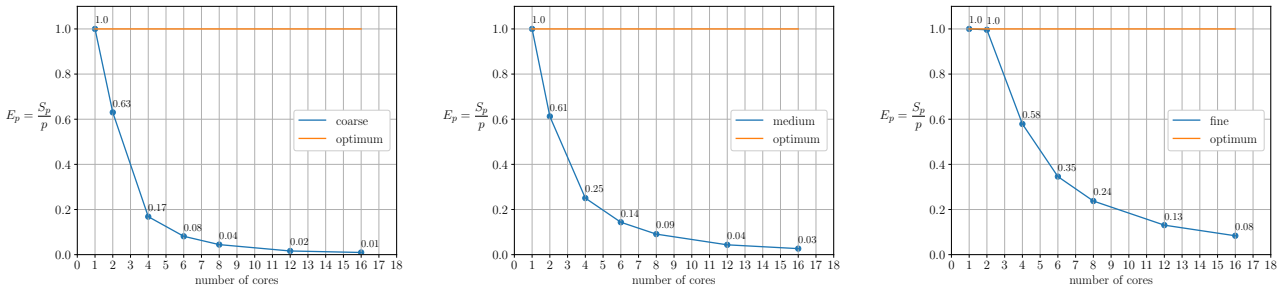
(a) coarse mesh **(b)** medium mesh **(c)** fine mesh

Figure 7: Runtime for coarse, medium and fine mesh for different number of cores



(a) coarse mesh **(b)** medium mesh **(c)** fine mesh

Figure 8: Speed-up for coarse, medium and fine mesh for different number of cores



(a) coarse mesh **(b)** medium mesh **(c)** fine mesh

Figure 9: Efficiency for coarse, medium and fine mesh for different number of cores

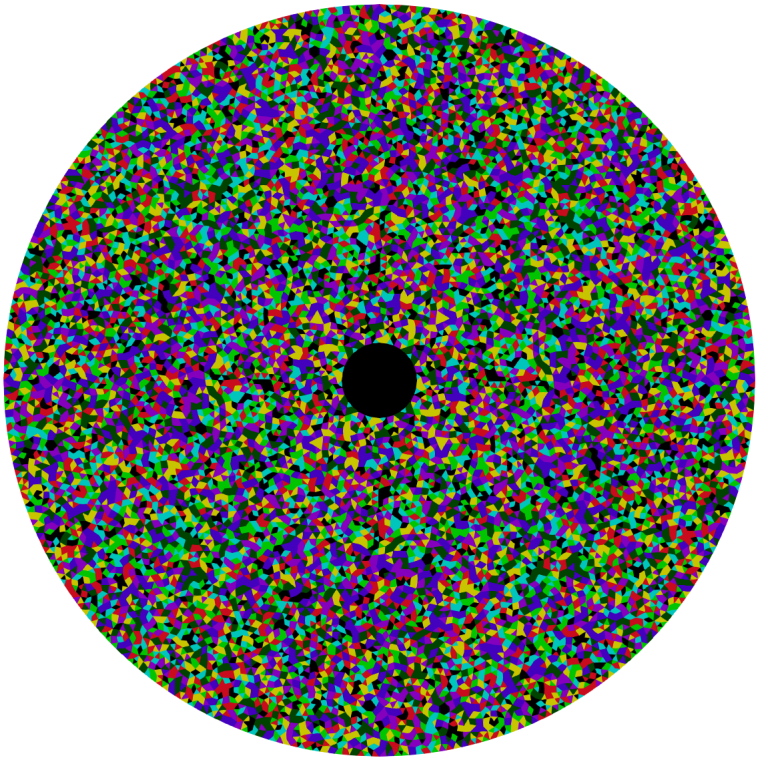


Figure 10: Partitioning of medium mesh using 8 cores