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

Parallelization with MPI

4 Results

4.1 Serial Solver

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 it's global maximum. Accounting for large changes in the temperature gradient at the center of the disk.
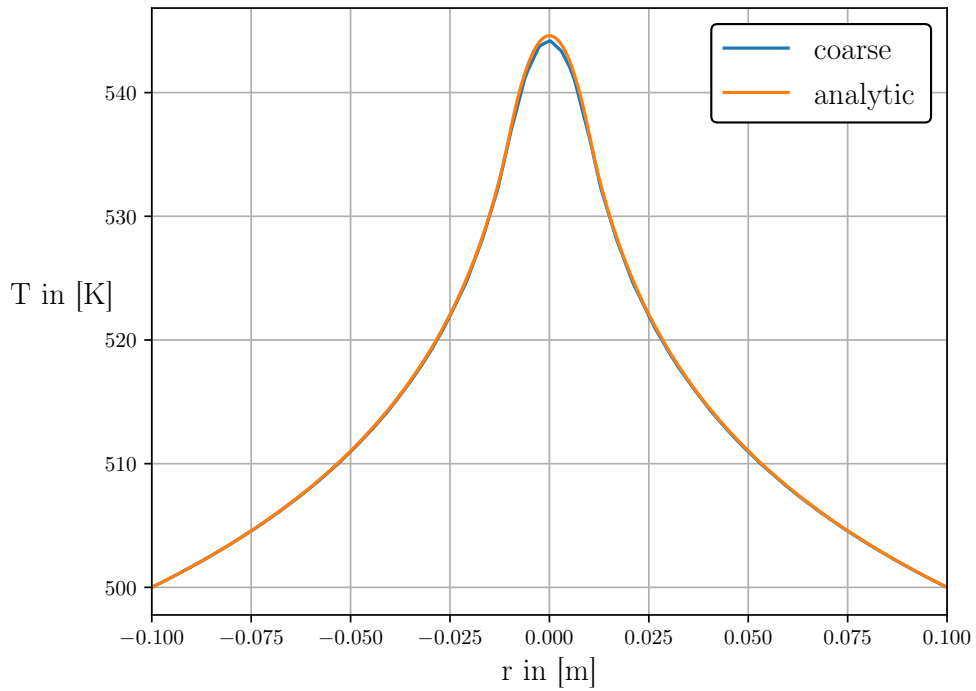
Figure 1: Comparison of the steady-state temperature distribution of the analytical solution (refEQ) and the numerical solution on the coarse mesh

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.
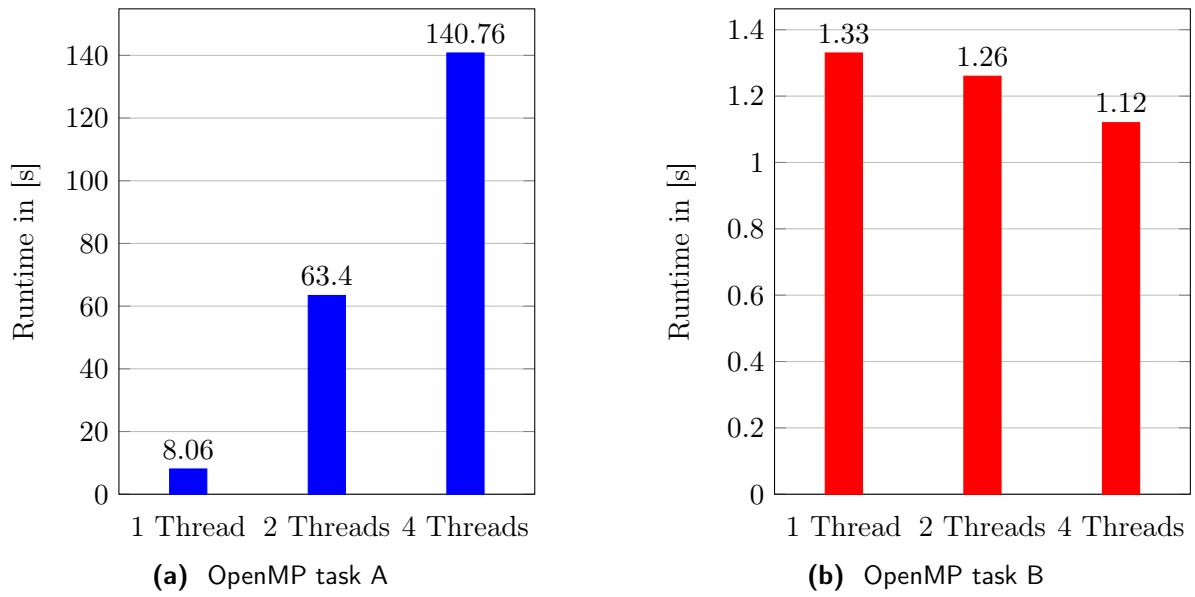
(a) OpenMP task A

(b) OpenMP task B

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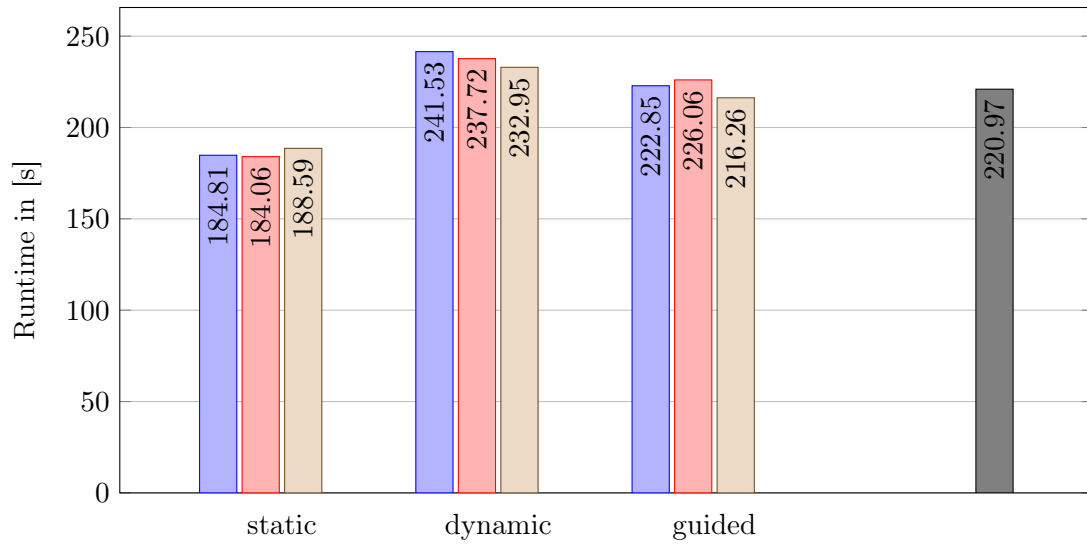

Figure 3: Comparison of different scheduling options (static, dynamic, guided, auto) for different chunk sizes(128, 256, 512) in first parallelised loop for 4 threads using the finest mesh

4.2 Parallel OpenMP Solver

R2 a

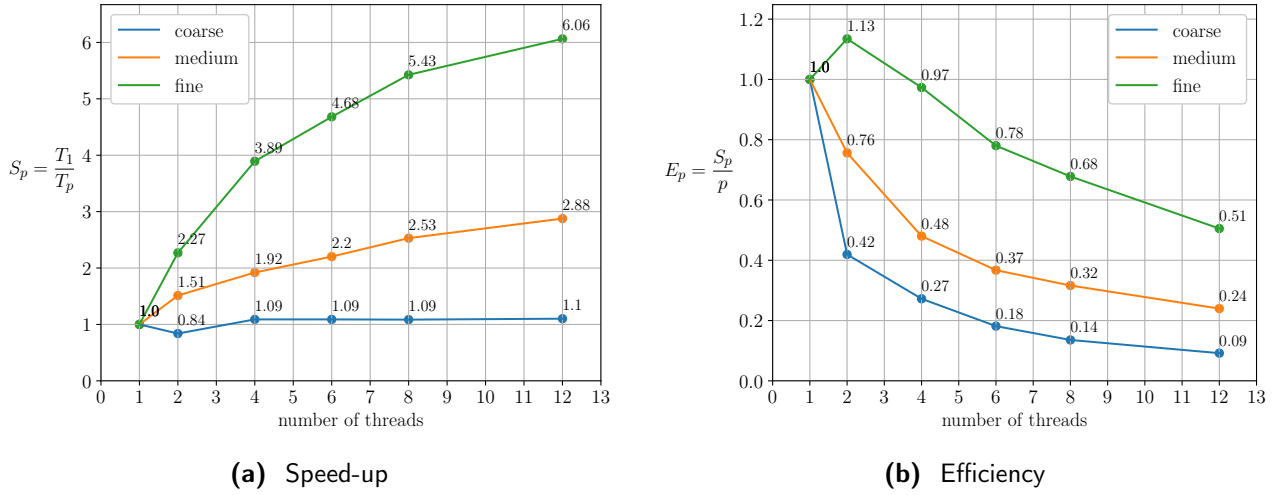**R2 b**

R2 c

R2 d

- look into presentation

(a) Speed-up

(b) Efficiency

Figure 4: Speed-up and efficiencies for coarse, medium and fine mesh for different numbers of threads

4.3 Parallel MPI Solver

R3 a

R3 b

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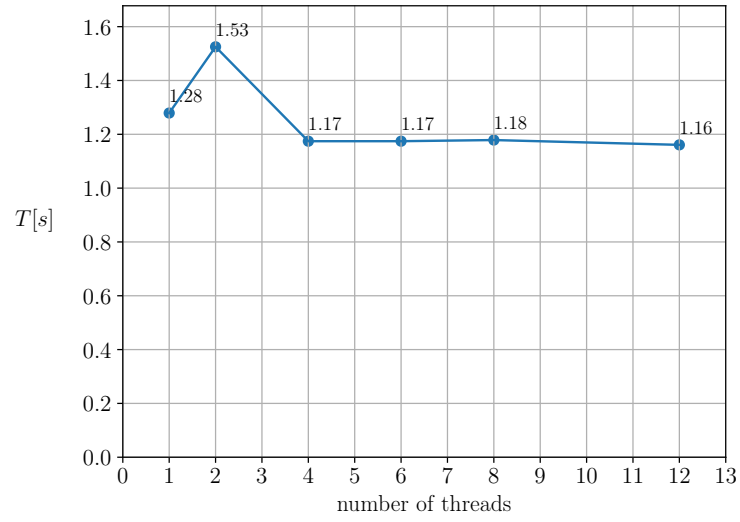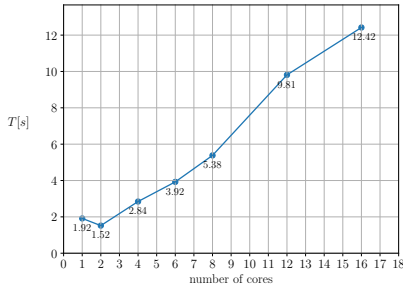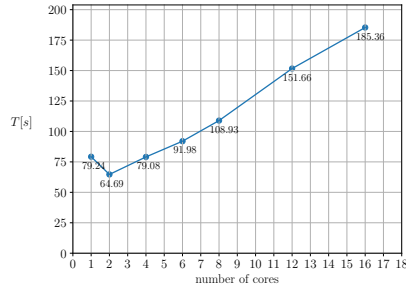
(a) coarse mesh
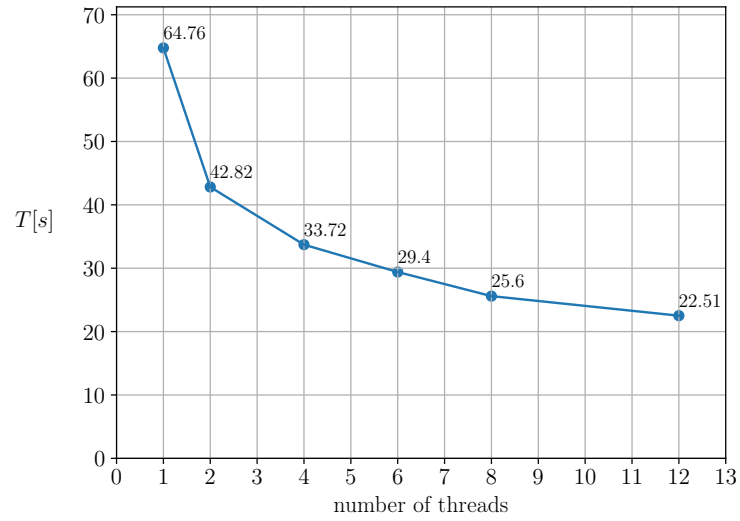
(b) medium mesh
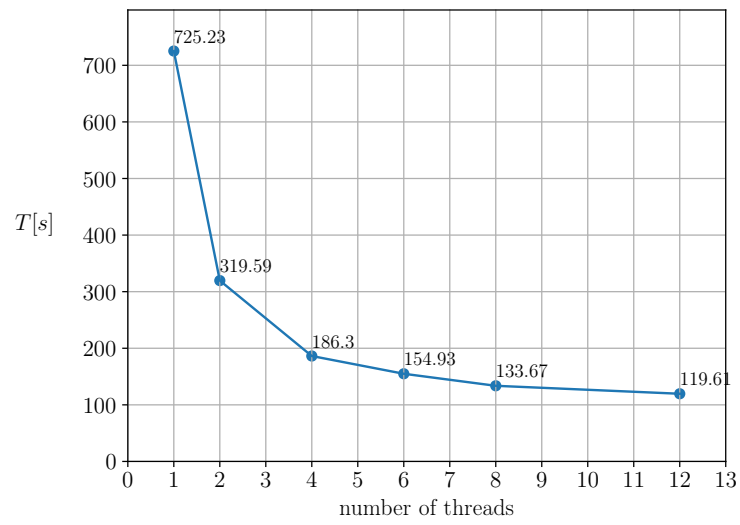
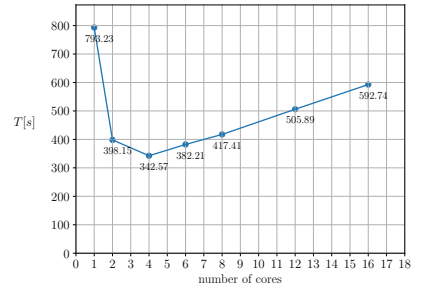**(c)** fine mesh

Figure 5: Runtime for coarse, medium and fine mesh for different number of threads
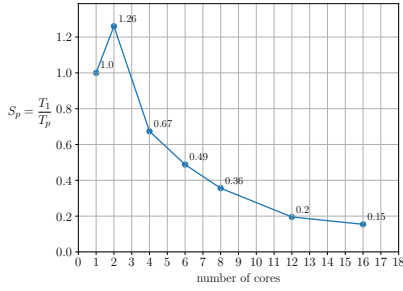
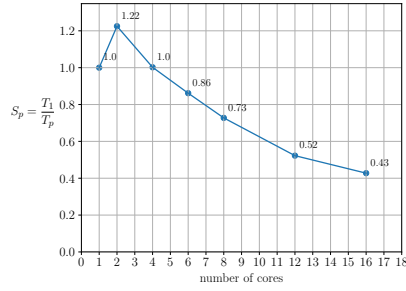**(a)** coarse mesh   **(b)** medium mesh   **(c)** fine mesh

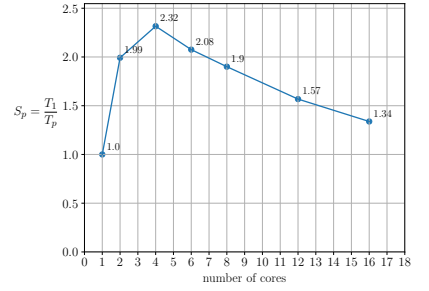**Figure 6:** Runtime for coarse, medium and fine mesh for different number of cores

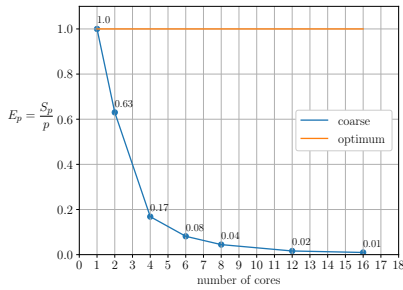**(a)** coarse mesh   **(b)** medium mesh   **(c)** fine mesh
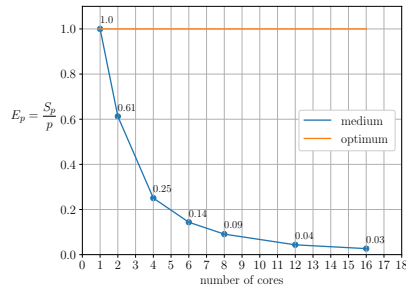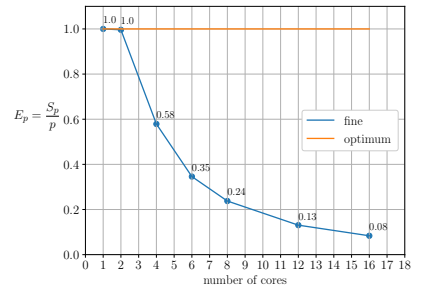
Figure 7: Speed-up for coarse, medium and fine mesh for different number of cores

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

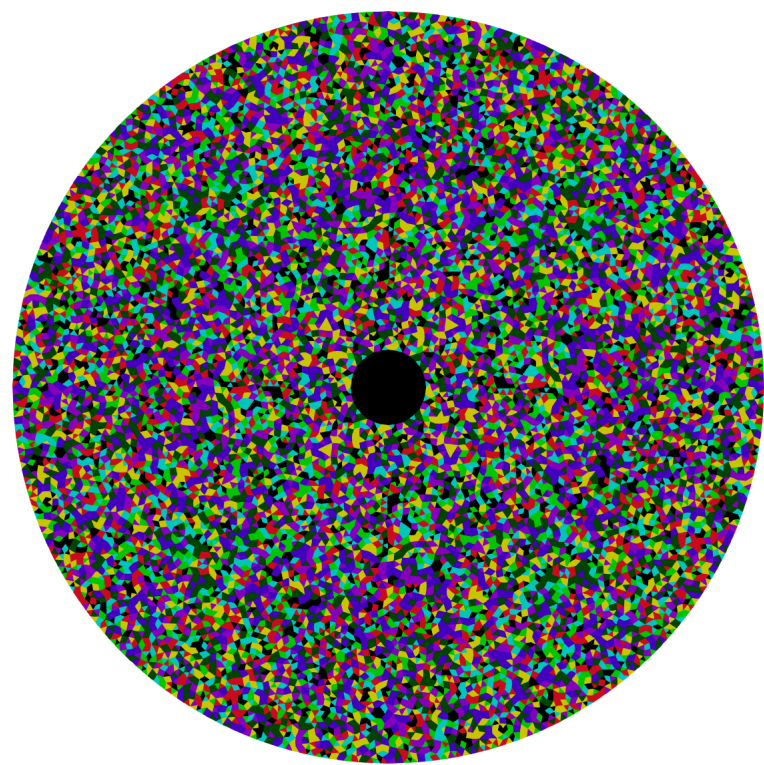**Figure 8:** Efficiency for coarse, medium and fine mesh for different number of cores

Figure 9: Partitioning of medium mesh using 8 cores