Contents

List of Figures

List of Tables

Parallelization of a FE code using OpenMP and MPI

Na Young Ahn

German Research School for Simulation Sciences GmbH
Chair for Computational Analysis of Technical Systems
na.ahn@rwth-aachen.de

1 Introduction

In this final project, we implemented and compared various parallelization methods and settings to retrieve the most optimal result for our finite element solver for the 2D unsteady heat diffusion equation problem. First, we optimize at compile time by running various compiler optimization methods on the serial case. Then we apply OpenMP for shared memory parallelization and explore how runtime improves for the reduction parallel contruct better than the critical synchronization construct when run on thread amounts: 1,2, and 4; and how runtime of different scheduling methods change for different chunk sizes. After having optimized for parallel construct and chunk size, we finally run on various mesh sizes and threads, which we use the result to plot the speedup and efficiency of our solver. We then compare the speedup and efficiency with solver run with MPI, distributed memory parallelization.

2 Theory

Contents in this section was largely referred from lecture slides, hpc primer 8.4, Intel compiler and OpenMP reference guides.

2.1 Heat equation and finite element method

2.1.1 Heat equation

In this project, we are given the model problem: 2D unsteady heat diffusion problem on a unit square domain. The temperature is obtained by solving the following heat equation.

$$\frac{\partial T}{\partial t} - \kappa \nabla^2 T = f \quad \text{in } \Omega \tag{1}$$

$$\begin{aligned} T(\mathbf{x}, t) &= T_D \quad \text{on } \Gamma_D \\ T(\mathbf{x}, 0) &= T_0 \quad \text{on } \Omega \end{aligned} \tag{2}$$

where T is the temperature, κ, the thermal diffusivity and f, the thermal heat source. Ω is the domain where the equation is solved and at boundary Γ a dirichlet boundary condition is imposed with temperature T_D. The problem is solved for time $t \in (0, t_f)$ using initial condition T_0 and final solution time T_f.
We are given an analytic solution $T(x, y, z) = 1 + x^2 + \alpha y^2 + \beta$ for $t \to \infty$ where α and β are user defined parameters.

Parameter	Variable	Value	Unit
Width	L_x	1.0	[m]
Height	L_y	1.0	[m]
Heat source on the area	$f(x, y, z)$	$\beta - 2 - 2\alpha$	[W/m]
Dirichlet BC temperature	$T_D(x, y, z)$	$1 + x^2 + \alpha y^2 + \beta$	[K]
Initial temperature	$T_0(x, y, z)$	500	[K]
1st coefficient	α	2.0	[-]
2nd coefficient	β	1.0	[-]
Thermal diffusivity	κ	1.0	[m^2/s]

Table 1: Problem parameters for the heated disk problem

2.1.2 Finite element method

Finite element method is a way of constructing discrete test space V_N based on given triangulation of Ω Given our strong form with second order differential equation for heat diffusion, we solve by following the procedure:

1. reformulate differential equation as a variational problem: find solution $u \in S$ s.t. for all test function $w \in T$, bilinear form = linear functional.
2. select finite-dimensional interpolation function spaces, which we discretize with Ritz-Galerkin.
3. solve the discrete problem.

Our spatial discretization is performed on three different structured triangular mesh with varying number of elements i.e. coarse, medium, and fine.
We are left with the global representation of matrix form of equations $[M]\{\dot{T}\} + [K]\{T\} = \{F\}$, where K is the stiffness matrix, M, mass matrix and f, the residual vector.

2.2 HPC performance metrics and compiler optimization

2.2.1 Speedup, theoretical and actual

- **Theoretical speedup** := $S_p(N) = \frac{T_1}{T_p(N)}$, where T_1 is the program execution time of one processor and $T_p(N)$ is the program execution time required by N processors.
  This definition, however, understates performance on one processor as the parallel algorithm is often not the best serial algorithm. Thus we define:
- **Actual speedup** := $S_p(N) = \frac{\text{time of best serial algorithm}}{T_p(N)}$.

2.2.2 Efficiency and scalability

- **Efficiency** := $E_p(N) = \frac{S_p(N)}{N}$ i.e. the ratio of speedup with N processors to N number of processors. At 100% efficiency, the speedup is the number of processors.
- **Scalability** means efficiency remains close to 1. [Behr, 2021]

2.2.3 Amdahl's law

Amdahl's Law provides an upper limit to speedup. Assume the program is comprised of serial fraction s, which cannot be parallelized and perfectly parallelizable fraction p with 100% efficiency and N processors. Then s+p = 1. Then the time on N processors is $T\_p = s * T\_1 + \frac{p}{N} * T\_1$. The speedup $S_(N) = \frac{1}{s + \frac{1-s}{N}}$, with $N \to \infty$ becomes $\frac{1}{s}$. The efficiency then goes to 0 as $N \to \infty$. A better speedup will be obtained if serial part s is reduced e.g. by increasing problem size.

2.2.4 Compiler flags

From class participation 2, we categorized compiler option groups into the following:

General	Architechtural / CPU	Floating point	Code trashing	Interprocedural
- O0 - O1 - O2 - O3 - g - fast	- axSSE2 - xSSE2	- prec-div - pc32 - fp-model	- unroll - inline-functions	- ipo - ip - pro-gen - pro-use

Table 2: Compiler option groups

Below are explanations for compiler flags used for this project.

- `-O0` No optimization. Disables any optimization. This option accelerate the compilations during the development/debugging stages. [an Mey et al., 2016]
- `-O1` Optimize for size. This omits optimizations that tend to increase object size.
- `-O2` (Default) Optimize for maximum speed. Enables optimization including vectorization and intrafile interprocedural optimization. Other optimizations include global common expression elimination, algebraic simplification, copy propagation, constant propagation, loopinvariant optimization, global register allocation, basic block merging, loop unrolling, software pipelining, tail recursion elimination, tail call elimination.[int]
- `-O3` Enables `-O2` optimizations with more aggressive loop and memory access optimizations, such as scalar replacement, loop unrolling, loop blocking to allow more efficient use of cache and additional data prefetching. This option implies `-align, -prefetch, -scalar_rep, -unroll`.
- `-fast` Maximizes speed across the entire program. A simple, but less portable way to get good performance. The -fast option turns on -O3, -ipo, -static and -no-prec-div.[an Mey et al., 2016] Should beware of possible incompatibility of binaries, especially with older hardware.
- `-g` produces symbolic debug information in object file.[an Mey et al., 2016]
- `-Os` GNU compiler option. Aims to provide high performance without a significant increase in code size. Depending on your application, the performance provided by -Os might be similar to -O2 or -O3. -Os provides code size reduction compared to -O3. It also degrades the debug experience compared to -O1. The differences when using -Os as compared to -O3 are: The threshold at which the compiler believes it is profitable to inline a call site is lowered. The amount of loop unrolling that is performed is significantly lowered.
- `-Ofast` GNU compiler option. Performs optimizations from level -O3, including those optimizations performed with the -ffast-math armclang option. This level also performs other aggressive optimizations that might violate strict compliance with language standards. This level degrades the debug experience, and might result in increased code size compared to -O3.
- `-axSKYLAKE-AVX512` intel target option, this option turns on the automatic vectorizer60 of the compiler and enables code generation for processors which employ the vector operations contained in the AVX512 and CPU `SKYLAKE`.
  - `-ax` does Automatic Processor Dispatch. Generates specialized code for the corresponding Intel processors while also generating a default code path. Multiple values, separated by commas, may be used to tune for additional processors in the same executable[int]

  - `AVX512` advanced vector extension, AVX-512 are 512-bit extensions to the 256-bit Advanced Vector Extensions SIMD instructions for x86 instruction set architecture (ISA) proposed by Intel in July 2013, and implemented in Intel's Xeon Phi x200 (Knights Landing) and Skylake-X CPUs

- `-ipo` intel advanced option, interprocedural optimization across source files.[an Mey et al., 2016]
- `-no-prec-div` intel floating point option, enables optimizations that give slightly less precise results than full IEEE division [int]
- `-static` library linking, the libraries are embedded in the executable at compile time.

2.3 OpenMP

2.3.1 Data race

The data usage pattern in an OpenMP parallel region creates a data race. A data race occurs when two threads access the same memory without proper synchronization. This can cause the program to produce non-deterministic results in parallel mode. In OpenMP, loops are parallelized by assigning different loop iterations to different threads. This strategy requires the loop iterations to be independent so they can run in any order. [Board, 2021]

2.3.2 Reduction vs. Critical

- `Reduction` is a parallelization clause. It combines every element in a collection into a single result uses an associative combiner function i.e. different ordering is possible. The reduction clauses are data-sharing attribute clauses that can be used to perform some forms of recurrence calculations in parallel.
- `Critical` is a synchronization construct. It is executed by all threads, but by only one thread simultaneously. We use critical to coordinate writing accesses to shared variables. The critical construct restricts execution of the associated structured block to a single thread at a time. [Board, 2021]

2.3.3 Scheduling and chunk size

Scheduling allows to fine-tune work distribution among threads. The threads execute contiguous sections of the loop.

- `static` chunk size is preset and fixed. Interleaved scheduling. Threads execute interleaved sections of preset sizes of the loop. There is a compromise between load balancing and cache reuse. When no chunk-size is specified, OpenMP divides iterations into chunks that are approximately equal in size and it distributes at most one chunk to each thread. The static scheduling type is appropriate when all iterations have the same computational cost.
- `dynamic` chunks are assigned to threads to balance the load. Threads execute sections of the loops as they become available. It is good for loops with randomly varying load. If we do not specify chunk-size, it defaults to one. The dynamic scheduling type is appropriate when the iterations require different computational costs. This means that the iterations are poorly balanced between each other. The dynamic scheduling type has higher overhead then the static scheduling type because it dynamically distributes the iterations during the runtime.
- `guided` similar to dynamic but with decreasing chunk size. The size of a chunk is proportional to the number of unassigned iterations divided by the number of the threads. Therefore the size of the chunks decreases. The guided scheduling type is appropriate when the iterations are poorly balanced between each other. The initial chunks are larger, because they reduce overhead. The

smaller chunks fills the schedule towards the end of the computation and improve load balancing. This scheduling type is especially appropriate when poor load balancing occurs toward the end of the computation.

- `auto` The auto scheduling type delegates the decision of the scheduling to the compiler and/or runtime system. [Speh, 2016]

2.4 MPI

2.4.1 Deadlock

status in which no process can move forward because it waits on another process. cf) `MPI_Sendrecv`

2.4.2 One sided vs. Two sided communication

- Two sided communication requires excessive synchronization, tedious matching of send and receive calls, excessive copying, and poses risk of deadlock
- One sided communication replaces send-receive pairs with put or get, separates communication and synchronization, but requires knowledge of memory layout on other processing elements, and area of memory affected must be framed by a collective window.

3 Methods

We use our finite element solver from the course to solve the temperature distribution on a unit square domain. Then we compile and execute the solver serially and in parallel using MPI and OpenMP to seek optimal performance.

3.1 Optimization at Compile Time

Before any parallelization, we need to ensure the efficiency of the solver's serial performance. We do so by making full use of the compiler's ability to improve code efficiency. Thus, we tested several compiler options and their optimization flags on the model problem using the medium mesh to see the impact on runtime.

First, it is important to know which batch system (SLURM, Simple Linus Utility for Resource Management) we use in our hpc cluster. We use the partition `c18m` with Intel hardware and CPU `Skylake`. There are three compiler options for C/C++ available in our cluster: Intel(icc), Oracle and GNU(gcc). According to the RWTH HPC-Cluster User's Guide Version 8.4, it is recommended to use the Intel compilers for best performance. Note `#SBATCH -exclusive` was added to the `run{Serial, MPI, OMP}.j` files to make sure the node is not shared with other jobs and all cases were tested several times each and the respective best runtimes were taken.

1. Hence, we start the investigation with Intel's general optimization options: `-O0, -O1, -O2, -O3`.
2. Then we try to improve the runtime by adding architectural level compiler optimization option using `-axSKYLAKE-AVX512`.
3. To confirm that Intel compiler gives the best results, general compiler flags of similar optimization degree from Oracle and GNU compilers were measured.
4. Lastly, full combination of compiler optimization group was tested with the theoretically most aggressive general optimization option `-O3`.

3.2 Shared-Memory Parallelization using OpenMP

Having tuned for the most optimal compiler flag for our serial solver, we keep this setting and now investigate the efficiency improvement of the parallelized solver using the OpenMP shared-memory interface. Note, due to the author's mistake the results from the following section were made with the compiler optimization flag `-O1 -fast` instead of `-O2 -axSKYLAKE-AVX512`.

3.2.1 OpenMP_Task_A vs. OpenMP_Task_B, on coarse mesh for thread amount(s) 1, 2 and 4

For homework 3, we identified the two most time consuming loops using `omp_get_wtime()`: *MT-New/RHS assembly*, where we evaluate the right hand side at the element level and accumulate at local nodes and *partialL2error calculation*, where we evaluate the new temperature on each node on partition level. To reduce the execution time, we attempt to parallelize the loops. However, the data usage pattern in an OpenMP parallel region creates a data race as two threads access the same memory without proper synchronization. To combat such race conditions, we can manually change the scopes of the variables or apply: `reduction` or `critical` Board [2021]

`OpenMP_Task_A` applies the `critical` synchronization construct and `OpenMP_Task_B` applies the `reduction` clause. We compile both OpenMP solvers using each task build selectors with CMake.

3.2.2 OpenMP_Task_B with different scheduling methods and chunk sizes on medium mesh on 4 threads

As we have found the optimal compiler flag and the better OpenMP construct that improves the two most time consuming loops, we investigate the OpenMP scheduling methods: `static, dynamic, guided, auto` and respective optimal chunk sizes.

3.2.3 OpenMP_Task_B with `schedule(static,2048)` on all meshes for thread amount(s) 1, 2, 4, 6, 8, and 12

Keeping the optimal compiler flag, OpenMP parallel solver with reduction clause and scheduling i.e. static with 2048 chunks, we document the runtime for the meshes: coarse, medium, and fine for thread amount(s) 1, 2, 4, 8, and 12.Using the obtained results, we compute and plot the speedup and efficiency.

3.3 Distributed Memory Parallelization using MPI

In this section, we implement distributed memory parallelization using MPI on our 2D unsteady heat equation solver. Most importantly, the entire domain/mesh should be cut evenly into chunks and be distributed among the processors. Therefore, the mesh has to be divided into nodes/elements per processor `nnc,nec` and associated nodes per processor `nnl` i.e. nodes of the elements on the processor that might not necessarily be the same as the nodes of the processor. The connectivity is thus reorganized on a local (partition/processor) level rather than global.

In homework 4, we have thus determined the `nnc,nec,nnl` in `readMeshFiles()` in file, tri.cpp, localized the coordinates in `localizeNodeCoordinates()`, gathered entries for the mass matrix in `accumulateMass()` for the local copy on each processor, localized temperature values in `localizeTemperature()`, and finally, gathered entries for the MTNew vector in `accumulateMTNew()`.

3.3.1 MPI solver on all meshes for thread amount(s) 1, 2, 4, 6, 8, 12, and 16

For this project, we apply our optimal compiler flag from prior sections and compile the MPI solver. Then, we run the solver for all three mesh types on core amount(s): 1, 2, 4, 6, 8, 12, 16. Using the

obtained results, we compute and plot the speedup and efficiency as well as the runtime over different numbers of cores.

3.3.2 Paraview visualization of the partitions on medium mesh with 8 cores

We load the numbered `medium-mesh_{0 - 7}.vtu` files and individually select a non overlapping solid color to distinguish each partition. When imported all at once, paraview automatically takes the numbers as timestamps. Hence, each numbered file were imported individually to apply different color to each partition.

4 Results

4.1 Serial Solver

Compiler flags	-O0	-O1	-O2	-O3
Runtime (s)	558.56000	238.83000	185.84000	158.11000
Runtime (s) with -axSKYLAKE-AVX512	561.09000	215.79000	**150.58000**	167.04000

Table 3: Runtime for Intel general and general + CPU compiler optimization flags

Compiler flags	-g -fast	-fast	-Os	-Ofast
Runtime (s)	170.14000	159.69000	213.97000	168.89000

Table 4: Runtime for Oracle and GNU general compiler optimization flags

Compiler flags	Runtime (s)
-O3	158.11000
-O3 -axSKYLAKE-AVX512	167.04000
-O3 -ipo -no-prec-div -xSKYLAKE-AVX512	156.96000
-O3 -ipo -no-prec-div -static -xSKYLAKE-AVX512	162.78000

Table 5: Runtime for full combination of intel compiler optimization flags with -O3

Best runtime was received from the combination of `-O2 -axSKYLAKE-AVX512`.

4.2 Parallel OpenMP Solver

Thread amount		1	2	4
Runtime per task (s)	OpenMP_Task_A	142.82135	689.16519	2051.79147
	OpenMP_Task_B	20.12222	14.61010	**11.51243**

Table 6: Runtime for various thread amounts per task

	Chunk size				
OpenMP scheduling	1	64	128	1024	2048
static	754.16573	473.02634	453.34887	410.30458	**408.84921**
dynamic	> 1800	666.17699	625.26187	578.40369	558.55202
guided	555.30369	550.55152	550.05138	546.81175	539.00951
auto	551.86914				

Table 7: Comparison of OpenMP scheduling methods on various chunk sizes

	Thread amount	1	2	4	8	12
Runtime per mesh (s)	coarse	20.14386	13.10963	**8.53396**	9.23879	10.79010
	medium	175.16414	81.51134	61.06851	**45.68339**	48.22316
	fine	1350.55766	655.54430	348.25882	278.56309	**243.77623**

Table 8: Runtime for different mesh sizes and thread amount

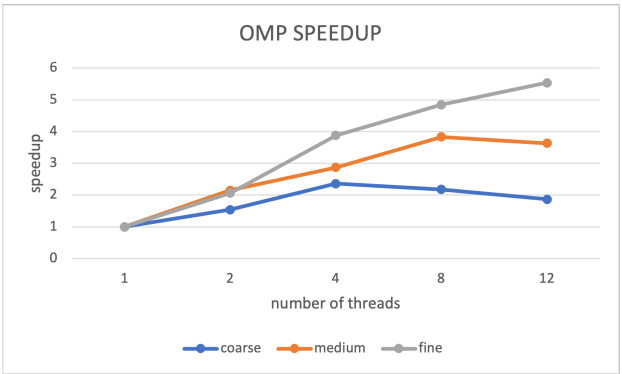

Figure 1: `OpenMP_Task_B` Speedup comparison

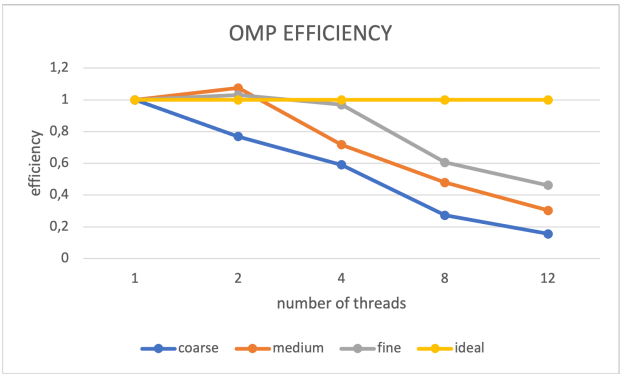

Figure 2: `OpenMP_Task_B` Efficiency comparison

4.3 Parallel MPI Solver

	Thread amount	1	2	4	6	8	12	16
Runtime per mesh (s)	coarse	26,14323	**19,27098**	23,41094	34,16945	50,10879	91,18225	117,67713
	medium	255,23032	114,87193	**109,02231**	115,29804	138,20103	199,34417	238,70077
	fine	1.513,32044	827,51567	**466,34947**	510,85450	531,54311	622,72001	768,38616

Table 9: Runtime for different mesh sizes and core amount

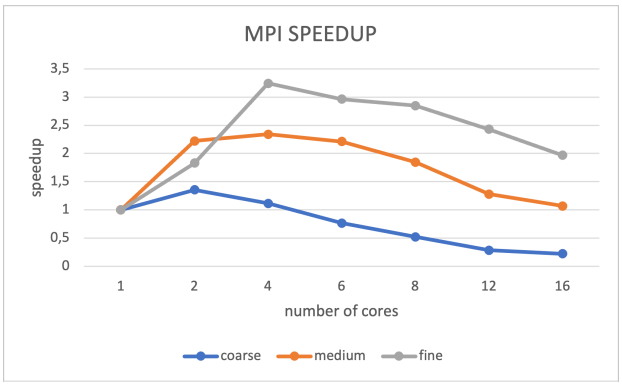

Figure 3: `MPI` speedup comparison

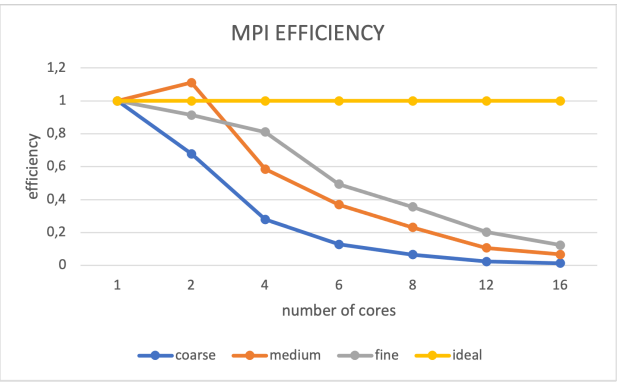

Figure 4: `MPI` efficiency comparison

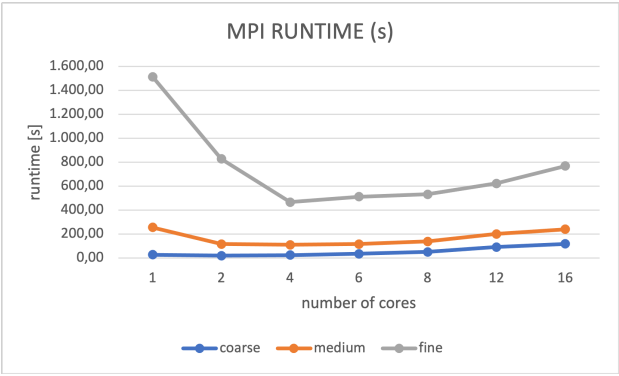

Figure 5: `MPI` runtime comparison

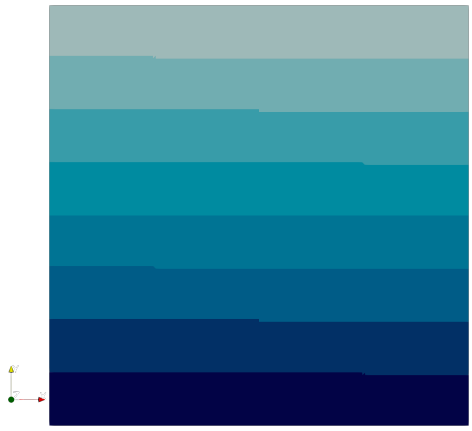

Figure 6: `MPI` partitions using medium mesh on 8 cores

4.4 Additional: OpenMP and MPI performance plots using actual speedup

	Serial runtime (s)
Coarse	18.68
Medium	174.39
Fine	1212.64

Table 10: Runtime of "best" serial algorithm on all meshes

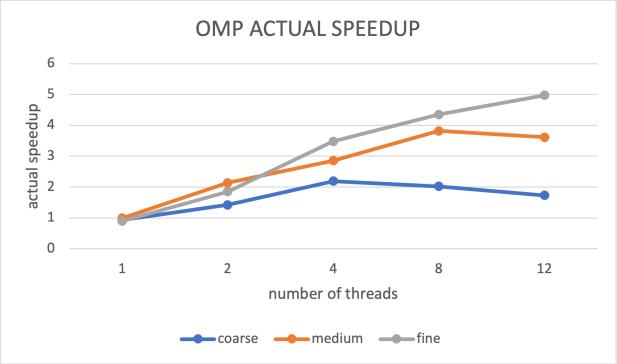

Figure 7: `OpenMP_Task_B` Actual speedup comparison

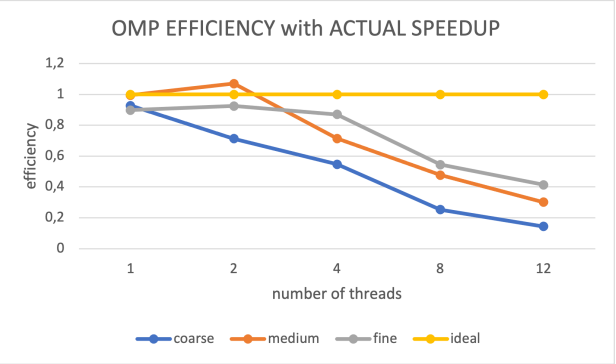

Figure 8: `OpenMP_Task_B` Efficiency comparison with actual speedup

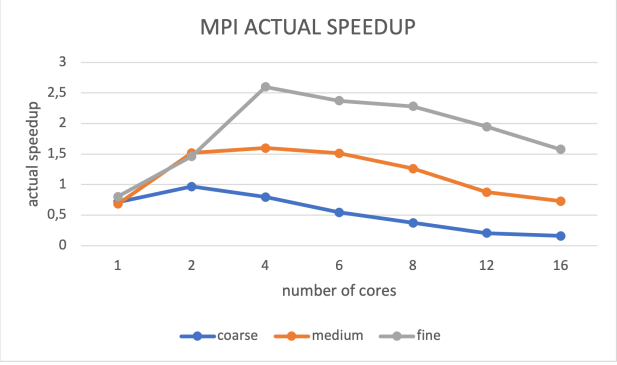

Figure 9: `MPI` Speedup comparison

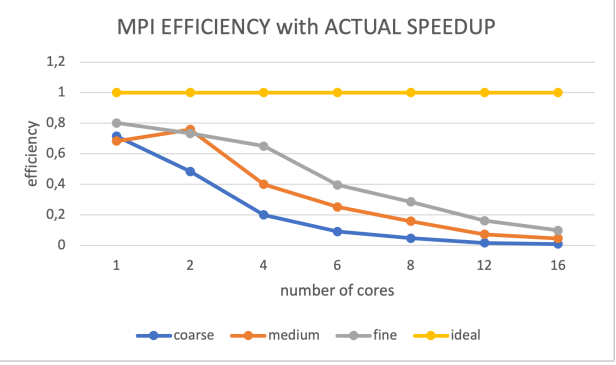

Figure 10: `MPI` Efficiency comparison with actual speedup

5 Discussion

5.1 Serial Solver

The best runtime for the serial solver using medium mesh was given with the combination of `-O2 -axSKYLAKE-AVX512`. This is interesting because, it is generally expected to have better runtime with the more aggressive `-O3` and combinations.

R1a) Provide a table with timings for the various compiler flags you have tried. Describe what each flag does. State the best combination of flags that you have found.

See relevant section in Theory 2.2.4 and Results 4.1.

R1b) 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?

As we have seen from the significant improvement of runtime for the serial solver on medium mesh using various compiler optimization flags, automatic compiler optimization allows for optimal results on each architecture and leaves code readability unaffected. However, there are potential drawbacks:

- debugging can be difficult as code can be changed during optimization. Mankarse [2012]
- optimization for speed can come at the cost of accuracy.
- compiler has limited knowledge of data dependencies across function boundaries
- optimization is difficult in C if unrestricted pointer arithmetic is used
- there can be an increase in compile time

Thus, we can deduce that one could prefer to use a non-optimized code over an optimized one especially for debug purposes and to have better understanding and control of the code. Also important is to note that there is a trade off between using more aggressive optimization flags for speedup and accuracy. Of course there are compiler flags such as `-g`, that generates debug information in `-O0`, and `-prec-div` to improve precision of floating-point divides, it is crucial to keep in mind what the limitations are before implementation.

5.2 Parallel OpenMP Solver

R2a) Document the timings of the solvers with the OpenMP_Task_A and OpenMP_Task_B options on the coarsest mesh for 1,2 and 4 threads. Did you expect those results and why do they differ?

See relevant section in Results table 6.
`OpenMP_Task_A` applies the `critical` synchronization construct and `OpenMP_Task_B` applies the `reduction` clause for the most time consuming loops of our solver.
We achieve better runtime with OpenMP_Task_B as the `reduction` performs recurrence calculations in parallel, unlike `critical` that executes the loop with all threads but single thread at a time i.e. serially.

R2b) Document the timings when using the solver with the OpenMP_Task_B option with different types of scheduling and chunk sizes. Use the medium mesh and a constant number of threads of 4. How do the different scheduling types compare? Which chunk size works best for which scheduling?

See relevant section in Results table 7.
The author obtained best runtime for the larger chunk sizes for all methods: `static, dynamic,`

`guided`, of which `static` performed best. Note, runtime for `guided` does not seem too affected by the chunk size.

R2c) Document the timings when using the best performing solver from above with respect to scheduling and chunk size. Document the timings for 1,2,4,6,8 and 12 threads for the coarse, medium and fine mesh. From the timings obtained, calculate the speed-up and parallel efficiency. Provide the timings in a table and the speed-up and efficiencies in plots. Discuss the results.

See relevant section in Results table 8.
The author obtained best runtime at:

- 4 threads with the coarse mesh,
- 8 threads with the medium mesh, and
- 12 threads with the fine mesh.

The two performance metrics help decide on how many threads to run. If we want to get the best runtime compared to the serial time, we look into speedup plot and see we have best speedup for thread amounts (4, 8, 12) for meshes (coarse, medium, fine). Note we observe the best speedup for fine mesh for most of the measured thread amounts. We check the efficiency plot to know at which thread amount we can most efficiency use the core-hours. Result shows, at thread amount(s) (1, 2, 2 and/or 4) for meshes (coarse, medium, fine) we can use the core-hours efficiently.

R2d) What are the default settings for scheduling in OpenMP? What are the default chunk sizes for static, dynamic and guided?

For more detailed explanation, see relevant section in Theory 2.3.3.
With Intel, we get the default OpenMP scheduling, `static`. The default chunk sizes are:

- `static`: $\frac{\text{loop count}}{\text{number of threads}}$, When no chunk-size is specified, OpenMP divides iterations into chunks that are approximately equal in size and it distributes at most one chunk to each thread. int
- `dynamic`: 1 If we do not specify chunk-size, it defaults to one. int
- `guided`: $\approx \frac{\text{loop count}}{\text{number of threads}}$ int

5.3 Parallel MPI solver

R3a) Provide the timing, speed-up and parallel efficiency plots for all 3 meshes. For the efficiency plot, include the optimum efficiency that a parallelized code could achieve. Describe the plots in the text. Include a discussion of the results. For the discussion, think about what you expected from a parallel code. What are expected values for speed-up and efficiency? What are the differences between the meshes? What are maximum/minimum values of the speed-up and efficiency? Give your opinion on which number of cores to use for which mesh and why.

See relevant section in Results table 9 and figures 3,4,5.
The author expected to see a decreasing runtime and efficiency, and increasing speedup with the increase of number of cores. However, what we observe is gradually increasing runtime after a short decrease at the lower number of cores (2, 4, 4) for meshes (coarse, medium, fine), and inversely, gradually decreasing speedup after short increase at the lower number of cores (2, 4, 4) for meshes (coarse, medium, fine). Efficiency shows decreasing trend with increasing number of cores as expected. With the obtained result and plots, following number of cores are recommended for each mesh types.

- For coarse mesh, **1 core** is recommended as there is minimal difference in runtime, and smaller speedup benefit compared to the bigger loss in efficiency by running on one more core.

	Maximum (mesh type, no. of cores)	Minimum (mesh type, no. of cores)
Speedup	3.2450352 (fine, 4)	0.2221607 (coarse, 16)
Efficiency	1.1109343 (medium, 2)	0.013885 (coarse, 16)

Table 11: Maximum/minimum values of speedup and efficiency for MPI solver

- For medium mesh, **2 cores** are recommended as we observe close to minimal runtime, and close to maximal runtime for medium mesh while maintaining scalability.
- For fine mesh, **4 cores** are recommended as it has minimal runtime, maximal speedup while not losing so much efficiency compared to efficiency of 2 cores and risking significant decrease of efficiency at 6 cores.

R3b) Provide a image of the partitions on the medium mesh, when using 8 cores. Comment on whether this is a good partitioning or not. What effect does this have on the code?

See relevant section in Results table 9.
We observe a partitioning, in which the quadrailaterals are horizontally divided. It is noteworthy to have a partitioning with minimum bordering nodes and elements as it reduces the number of necessary communications between processing elements.

R3c) Discuss the performance of the code. Do you see room for improvement? If so, how would you improve the code? Make suggestions to the developers.

With maximal speedup barely over 3.2, which is half of what we achieved with OpenMP and quickly decreasing efficiency, the overall performance is poor for this code.
Better performance can be expected with improved communication, especially between with other processing elements. Instead of looping over all processing elements, which increases the loop size with increasing number of processing elements, we can have a static lookup table with the 'location' of the target processors and its elements, such that asking or sending data to all other processing elements is no longer necessary.

R3d) Answer the following MPI related questions:

- **Why do we use `MPI_Accumulate` in the code instead of `MPI_Put`?**
  - We want to perform an additive reduction via `MPI_Accumulate`. `MPI_Put`, on the other hand, overwrites the values.
- **What are advantages of using one-sided MPI communication over two-sided MPI communication?**
  - One sided communication is easier to implement. There is no risk of deadlocks but there exist higher risk for errors if we do not pay attention to the communication patterns.
  - Two sided communication is harder to implement as it requires more attention to making the correct communication between sender and receiver. Blocking can increase waiting time.
- **What advantage does MPI offer over OpenMP, what is a main obstacle of MPI?**
  - Compared to OpenMP that has hardware limit i.e. limited the memory on each CPU, MPI has no limitation on number of processing elements given cluster with infinite availability of number of cores.
  - OpenMP, however, is easier to implement in an existing code.

6 Conclusion

In this project, we investigated the effects of automatic optimization and shared/distributed memory parallelization of the given FEM solver for the 2D unsteady heat diffusion problem. When optimizing for the serial solver, we saw a significant improvement of runtime with even the most basic compiler flags, of which the author found -O2 with additional CPU compiler option to give the best result. When implementing shared memory parallelization with OpenMP, we parallelized and optimized for performance first in the code, with the reduction clause for the two most time consuming loops then with the scheduling method and chunk size `static,2048`, and lastly, on the thread by measuring for increasing thread amounts on all mesh types. Increasing speedup and decreasing efficiency trend was shown with increase of threads used for parallelization. For distributed memory parallelization with MPI, however, we no longer see an increasing speedup trend as core amount increases. We conjecture this phenomenon to occur due to communication burden between the processing elements as number of cores increases. Hence propose minimal bordering between quadrilateral elements in the partition and a more targeted call potentially using a static lookup table with the location information of the target processors and elements.

References

Marek Behr. Lecture slides for parallel computing for computational mechanics_ss2021, 2021. URL https://moodle.rwth-aachen.de/course/view.php?id=15196&section=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, Sep 2016. URL https://capi.sabio.itc.rwth-aachen.de/documents/0962a984657bb2e901657bc9e2810004/1jifm42ljvvhw/32da639f446a4cdba868e91dbdbf9041/primer-8.4.0.pdf.

Quick reference guide to optimization with intel c++ and fortran compilers v19.1. URL https://software.intel.com/content/www/us/en/develop/download/quick-reference-guide-to-optimization-with-intel-compilers-v19.html.

OpenMP Architecture Review Board. Reference guides, May 2021. URL https://www.openmp.org/spec-html/5.0/openmp.html.

Jaka Speh. Openmp scheduling, Jun 2016. URL http://jakascorner.com/blog/2016/06/omp-for-scheduling.html.

Mankarse. Why not always use compiler optimization?, Nov 2012. URL https://stackoverflow.com/questions/7857601/why-not-always-use-compiler-optimization.