Final Project 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

2.1 Heat Equation and Problem Domain

From the Final Project Assignment: The heat equation is solved on a disk Ω with the boundary Γ to obtain the radial temperature distribution (Fig. 1). The Problem is given by the following heat equation, using Dirichlet boundary conditions at the disk outer boundary and an initial homogeneous Temperature T_0 on the whole disk:

$$\frac{\partial T}{\partial t} - \kappa \nabla^2 T = f \qquad\qquad \text{on } \Omega \ \forall t \in (0, t_f), \tag{1}$$

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

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

where T is the temperature, κ the thermal diffusivity, and f the thermal heat source. The problem is solved for the time interval $t \in (0, t_f)$, with the final solution time t_f. The heat source is a function of the radius of the plate with:

$$f(r) = \begin{cases} \frac{Q}{\pi R_1^2} & \text{if } r < R_1, \\ 0 & \text{if } r \geq R_1 \end{cases}. \tag{4}$$

where $Q = P \cdot d_z$ is the volumetric heat source and $d_z = 0.1m$ the out-of-plane thickness. For validation purposes an analytical solution for $t \to \infty$ is given by:

$$T(r) = \begin{cases} T_0 - \frac{Q}{2\pi\alpha}\left(\frac{1}{2}\left(\frac{r^2}{R_1^2} - 1\right)\right) + ln\left(\frac{R_1}{R_2}\right) & \text{if } r < R_1, \\ T_0 - \frac{Q}{2\pi\alpha}ln\left(\frac{r}{R_2}\right) & \text{if } r \geq R_1 \end{cases}. \tag{5}$$

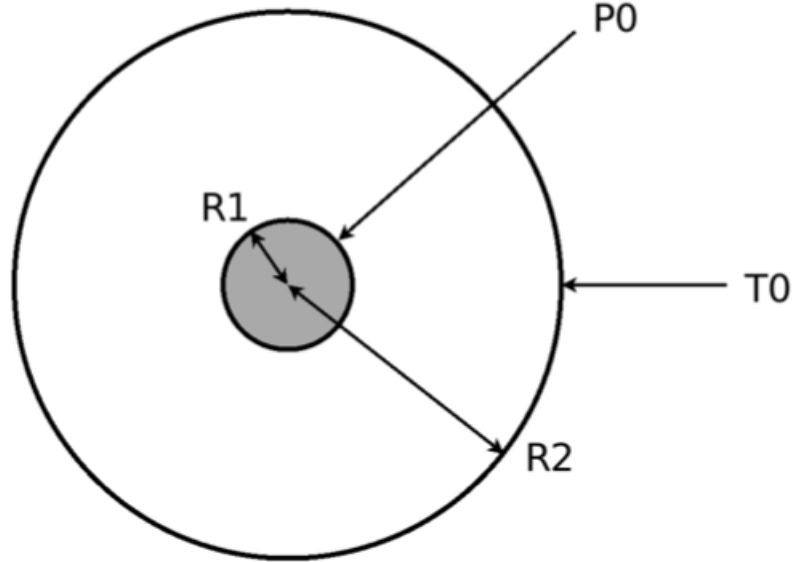where α is a user-specified parameter. The parameters for the specific heated disk problem are given in Tab. 1.

Figure 1: Problem domain

Parameter	Variable	Value	Unit
Inner circle radius	R_1	0.01	$[m]$
Outer circle radius	R_2	0.1	$[m]$
Heat source on the area	Q	100	$[W]$
Dirichlet BC temperature	$T_0(x, y)$	500	$[K]$
Initial temperature	$T_s(x, y, 0)$	0	$[K]$
Thermal diffusivity	κ	1.0	$[m^2/s]$

Table 1: Problem parameters for the heated disk problem

2.2 Weak Form

To obtain a discrete formulation in space and time for a finite element discretization for the unsteady heat equation, a weak or variational form of Eq. (1) has to be defined. Two sets of function spaces have to be defined in the context of the standard Galerkin formulation [1]:

$$\mathcal{V} = \{w \in \mathcal{H}^1(\Omega) | w = 0 \text{ on } \Gamma_D\} \tag{6}$$

$$\mathcal{S} = \{T \in \mathcal{H}^1(\Omega) | T = T_D \text{ on } \Gamma_D\} \tag{7}$$

where the space $\mathcal{V}$ is composed of test functions that are square integrable and have square integrable first derivatives over the computational domain Ω. They vanish on the Dirichlet boundary Γ_D. The set of functions $\mathcal{S}$ are the trial solutions. This collection of functions has similar properties to the test functions, except that those functions have to satisfy the Dirichlet boundary condition on the domain boundary Γ_D. $\mathcal{S}$ and $\mathcal{V}$ are both continuous function spaces. For the finite element method these function spaces are approximated by finite dimensional subsets $\mathcal{S}^h$ and $\mathcal{V}^h$.

The formulation of Eq. (1) is given as:

$$\int_\Omega w \frac{\partial T}{\partial t} d\Omega - \kappa \int_\Omega w \nabla^2 T d\Omega = \int_\Omega w f d\Omega \quad \forall w. \tag{8}$$

Applying the Green-Gauss divergence theorem on the left hand side of Equation Eq. (8) the order of the spatial derivatives can be reduced. Together with a zero homogeneous Neumann boundary condition ($n \cdot \nabla T = 0$ on Γ_h), we obtain the following equation:

$$\int_\Omega w \frac{\partial T}{\partial t} d\Omega + \kappa \int_\Omega \nabla w \cdot \nabla T d\Omega = \int_\Omega w f d\Omega \tag{9}$$

2.3 Finite Element Method

The discretized variational formulation of the continous PDE in weak form (Eq. (9)) that is used for the finite element method is formulated as: find $T^h \in \mathcal{S}^h$, such that $\forall w^h \in \mathcal{V}^h$

$$\sum_{B \in \eta \setminus \eta_D} \left[(N_A, N_B) \frac{\partial T_b}{\partial t} + a(N_A, N_B) T_B \right] = (N_A, f) - \sum_{B \in \eta_D} a(N_A, N_B) T_D(\mathbf{x}_B), \forall A \in \eta \setminus \eta_D. \tag{10}$$

with

$$T^h(\mathbf{x}) = \sum_{A \in \eta} N_A(\mathbf{x})T_A, \tag{11}$$

where N_A are the shape functions used for the interpolation of the continuous weighting function $w(x,y)$, N_B are the shape functions for the interpolation of $T(x,y)$, $a(N_A, N_B)$ represents the discretized thermal diffusive part of the equation (second term on left-hand-side in Eq. (9)) and (N_A, f) denotes the discretized version of the right-hand-side. η represents the set of all nodes and η_D is a subset of η containing all nodes on the boundary, where the Dirichlet boundary condition is applied.

This variational formulation leads to a matrix form of equations:

$$[M]\{\dot{T}\} + [K]\{T\} = \{F\} \tag{12}$$

where M refers to the mass matrix, K to the stiffness matrix and F to the source vector. After applying a first order explicit time integration scheme and isolating the LHS by inverting the mass matrix M the explicit equation to obtain the temperature at the nodes for a timestep n can be written as:

$$\{T\}^{n+1} = ([M]^n)^{-1}([M]^n\{T\}^n + \Delta t(\{F\}^n - [K]^n\{T\}^n) \tag{13}$$

In a first step the mass matrices are computed on element level and then assembled globally. To solve the integrals efficiently in combination with the an unstructured grid an isoparametric approach is chosen. Therefore, the integrals are evaluated on a triangular reference element. To map the results back to the physical domain, the transformation theorem is applied:

$$\int_{\Omega^e} \Phi(\mathbf{x})d\mathbf{x} = \int_{\Omega^{ref}} \Phi(\xi, \eta))|\mathbf{J}(\xi, \eta)|d\xi d\eta \tag{14}$$

Φ denotes an arbitrary function in $\mathcal{S}$, $\mathbf{J}$ the jacobian for the transformation between global coordinates (x, y) and the reference coordinates (ξ, η), Ω^e the physical element domain and Ω^{ref} the reference domain of the element. The integrals are solved numerically by using Gaussian quadrature:

$$\int_0^1 \int_0^{1-\eta} \Phi(\xi, \eta)d\xi d\eta \approx \sum_{i=1}^{n_{quad}} w_i \Phi(\xi_i, \eta_i) \tag{15}$$

with weights w_i and ξ_i, η_i the quadrature points at which $\Phi(\xi, \eta)$ is determined. The individual entries of the matrices M, K and F are then calculated as follows:

$$M_{i,j}^e = \sum_{m=1}^{n_{quad}} w(m)S_i(\xi_m \eta_m)S_j(\xi_m, \eta_m)|\mathbf{J}_m|, \tag{16}$$

$$K_{i,j}^e = \kappa \sum_{m=1}^{n_{quad}} w(m)(\partial_x S_i(\xi_m, \eta_m))\partial_x S_j(\xi_m, \eta_m) + \partial_y S_i(\xi_m, \eta_m)\partial_y S_j(\xi_m, \eta_m)|\mathbf{J}_m|, \tag{17}$$

$$F_i^e = \sum_{m=1}^{n_{quad}} w(m)fS_i(\xi_m, \eta_m)|\mathbf{J}_m|. \tag{18}$$

For an efficient inversion of the mass matrix which is necessary to obtain the spacial temperature distribution for the next time step (Eq. (13)) mass lumping is used. This technique is an approximation of the original system, transforming the mass matrix into a diagonal matrix, which can be stored as a vector.

2.4 Meshes

For the FEM Solver, three different unstructured meshes are used with different refinement levels (coarse, medium and fine). The number of nodes and elements of the three different unstructured triangular meshes (coarse, medium and fine) are given in Tab. 2.

mesh	number nodes (nn)	number elements (ne)
coarse	2041	3952
medium	21650	21332
fine	85290	84656

Table 2: Number of nodes and elements for the three different meshes

3 Implementation and Validation

3.1 Convergence Criterion

The unsteady solution of the heat equation converges to a stead-state for $t \to \infty$. In order to break out of the time loop a convergence criterion has to be defined, which states the change of the temperature between two subsequent timesteps:

$$\epsilon_{RMS} = \sqrt{\frac{1}{nn} \sum_{i=1}^{nn} \left(T_i^{n+1} - T_i^n \right)^2} \leq 10^{-7} \tag{19}$$

with the root-mean-square error ϵ_{RMS} and the number of nodes $nn \in \eta \backslash \eta_D$. If ϵ_{RMS} is lower than a predefined error (10^{-7}), the simulation is declared as converged and is finished.

3.2 Serial Code

The calculation of the element matrices is performed as presented in Eq. (16) to Eq. (18). The corresponding code is illustrated in Code excerpt 1. The values of the shape functions and their derivatives, as well as the determinate of the jacobian is precomputed for each element node. The matrices M, K and F are computed at element level. After performing the mass lumping procedure the element level matrices must be hard copied to the corresponding triangular element. Then the Dirichlet boundary condition Eq. (2) is applied at the domain boundary nodes. The global mass for each node is accumulated.

Listing 1: Calculation of Element Matrices

```
1  // First, fill M, K, F matrices with zero for the current element
2  (...)
3
4  // Now, calculate the M, K, F matrices
5  for(int p=0; p<nGQP; p++)
6  {
7      (...)
8      for(int i=0; i<nen; i++)
9      {
10         for(int j=0; j<nen; j++)
11         {
12             // Consistent mass matrix
13             M[i][j] = M[i][j] +
14             mesh->getME(p)->getS(i) * mesh->getME(p)->getS(j) *
15             mesh->getElem(e)->getDetJ(p) * mesh->getME(p)->getWeight();
16             // Stiffness matrix
17             K[i][j] = K[i][j] +
18             D * mesh->getElem(e)->getDetJ(p) * mesh->getME(p)->getWeight() *
19             (mesh->getElem(e)->getDSdX(p,i) * mesh->getElem(e)->getDSdX(p,j) +
20             mesh->getElem(e)->getDSdY(p,i) * mesh->getElem(e)->getDSdY(p,j));
21         }
22         // Forcing matrix
23         F[i] = F[i] + factor_F * mesh->getME(p)->getS(i);
24
25     }
26  }
27
28  //Calculation of total mass and the total diagonal mass and perform mass lumping
29  (...)
30
31  //Total mass at each node is accumulated on local node structure:
```

```
32  for(int i=0; i<nen; i++)
33  {
34      node = mesh->getElem(e)->getConn(i);
35      mesh->getNode(node)->addMass(M[i][i]);
36  }
37
38  // At this point we have the necessary K, M, F matrices as a member of femSolver
39  // object.
40  // They must be hard copied to the corresponding triElement variables.
41  for(int i=0; i<nen; i++)
42  {
43      node = mesh->getElem(e)->getConn(i);
44      mesh->getElem(e)->setF(i, F[i]);
45      mesh->getElem(e)->setM(i, M[i][i]);
46      for(int j=0; j<nen; j++)
47      {
48          mesh->getElem(e)->setK(i,j,K[i][j]);
49      }
50  }
```

After the previous computations the explicit solver is entered. The outer loop is an iteration loop in which the convergence criterion is checked at the end (Code excerpt 2). If convergence is reached, the code breaks out of the loop. After clearing all entries of the RHS storage MTnew, the second inner loop iterates over all elements, evaluating the RHS at element level (line 9 to 22 in Code excerpt 2). This is followed by the computation of the new temperature, using the inverted mass matrix, which inverse corresponds to it's reciprocal as the mass lumping algorithm was previously applied to diagonalize the mass matrix. Subsequently, the global RMS error can be computed and the convergence criterion is checked.

Listing 2: Explicit Solver

```
1       for (int iter=0; iter<nIter; iter++)
2       {
3           // clear RHS MTnew
4           (...)
5           // Evaluate right hand side at element level
6           for(int e=0; e<ne; e++)
7           {
8               (...)
9               for(int i=0; i<nen; i++)
10              {
11                  TL[i] = T[elem->getConn(i)];
12              }
13
14              MTnewL[0] = M[0]*TL[0] + dT*(F[0]-(K[0]*TL[0]+K[1]*TL[1]+K[2]*TL[2]));
15              MTnewL[1] = M[1]*TL[1] + dT*(F[1]-(K[3]*TL[0]+K[4]*TL[1]+K[5]*TL[2]));
16              MTnewL[2] = M[2]*TL[2] + dT*(F[2]-(K[6]*TL[0]+K[7]*TL[1]+K[8]*TL[2]));
17
18              // RHS is accumulated at local nodes
19              MTnew[elem->getConn(0)] += MTnewL[0];
20              MTnew[elem->getConn(1)] += MTnewL[1];
21              MTnew[elem->getConn(2)] += MTnewL[2];
22          }
23
24          // Evaluate the new temperature on each node on partition level
25          partialL2error = 0.0;
26          globalL2error = 0.0;
27          for(int i=0; i<nn; i++)
```

```
28              {
29                  pNode = mesh->getNode(i);
30                  if(pNode->getBCtype() != 1)
31                  {
32                      massTmp = massG[i];
33                      MT = MTnew[i];
34                      Tnew = MT/massTmp;
35                      partialL2error += pow(T[i]-Tnew,2);
36                      T[i] = Tnew;
37                      MTnew[i] = 0;
38                  }
39              }
40              globalL2error = sqrt(partialL2error/this->nnSolved);
41              globalL2error = globalL2error / initialL2error;
```

3.3 Parallelization with OpenMP

The two most time consuming loops, which are the computation of the RHS and the computation of the global RMS error are to be parallelized on loop level with a shared memory approach. This means multiple threads share the work of the comutation inside a loop, and have access to the same memory address space.

Therefore OpenMP directives are used. Two different approaches are implemented, denoted with "A)" and "B)" which are assessed in Sec. 4.2.1. The corresponding directives are presented in Code excerpt 3.

For approach "A)" MTnew is declared as firstprivate upon entry in the parallel region. This ensures, that the entries MTnew are initialized at the entry of the parallel section, which means MTnew is copied from the parent thread to the child threads. For approach "B)" no initalialization of MTnew is performed on the child threads, as it is a shared array among all threads. Both approaches declare the element object elem, the mass matrix M, the stiffness matrix K, the source vector F, the local temperature, the node loop index i and the local RHS MTnewL as private. This means all these objects are private (local) to the thread and undefined upon entry to the region. For approach A), critical region are used for the accumulation of the RHS at the local nodes and the computation of the partial L2 errors.

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 behavior, which can result in incorrect computations.

For approach B), reduction operations are 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.

In this context different scheduling options, with different chunk sizes can be compared (Sec. 4.2.2). Explaining the different scheduling options briefly, static scheduling accounts for a static chunk size, meaning a chunk is consisting of a constant number of iterations. These chunks are then distributed in a round-robin fashion among the threads. "Dynamic" scheduling however allows the chunk size to be allocated dynamically, based on the load of a thread. Guided scheduling works similar to "dynamic" scheduling, but with a chunk size that is constantly decreasing, based on the unassigned iterations left. The scheduling option "auto" leaves this decision to the compiler.

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

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

3.4 Parallelization with MPI

Listing 4: Determination of number elements on current rank and number nodes in current

```
1   // Determine nec, mec
2   nec = (ne-1)/npes + 1;
3   mec = nec;
4   if ((mype+1)*mec > ne)
5   nec = ne - mype*mec;
6   if (nec < 0)
7   nec = 0;
8
9   // Determine nnc, mnc
10  nnc = (nn-1)/npes + 1;
11  mnc = nnc;
12  if ((mype+1)*mnc > nn)
13  nnc = nn - mype*mnc;
14  if (nnc < 0)
15  nnc = 0;
```

Listing 5: Read file for every rank with specific offset

```
1   (...)
2   offset = mype*nsd*mnc*sizeof(double);
3   MPI_Type_contiguous(nnc*nsd, MPI_DOUBLE, &mxyzftype);
4   MPI_Type_commit(&mxyzftype);
5   MPI_File_open(MPI_COMM_WORLD, writable, MPI_MODE_RDONLY,
6   MPI_INFO_NULL, &fileptr);
7   MPI_File_set_view(fileptr, offset, MPI_DOUBLE, mxyzftype,
8   "native", MPI_INFO_NULL);
9   readStream = new char [nsd*nnc*sizeof(double)];
10  MPI_File_read(fileptr,readStream, nsd*nnc, MPI_DOUBLE, &status);
11  swapBytes(readStream, nsd*nnc, sizeof(double));
12  for(int i=0; i<nnc; i++)
13  {
14      node[i].setX(*((double*)readStream + nsd*i));
15      node[i].setY(*((double*)readStream + nsd*i+1));
16      xyz[i*nsd+xsd] = *((double*)readStream + nsd*i);
17      xyz[i*nsd+ysd] = *((double*)readStream + nsd*i+1);
18  }
19  if (mype==0) cout << "> File read complete: " << dummy << endl;
20
21  MPI_File_close(&fileptr);
22  MPI_Barrier(MPI_COMM_WORLD);
```

- explain mapping from local to global node list - calculate element matrices is performed on every core

Listing 6: Explicit solver for MPI parallelization

```
1   MPI_Win winMTnew;
2   MPI_Win winTG;
3
```

```
4      MPI_Win_create(MTnewG, nnc*sizeof(double), sizeof(double), MPI_INFO_NULL,
5      MPI_COMM_WORLD, &winMTnew);
6      MPI_Win_create(TG, nnc*sizeof(double), sizeof(double), MPI_INFO_NULL,
7      MPI_COMM_WORLD, &winTG);
8      MPI_Win_fence(0, winMTnew);
9      MPI_Win_fence(0, winTG);
10
11     localizeTemperature(winTG);
12
13     for (int iter=0; iter<nIter; iter++)
14     {
15
16         // Similar to serial Code, RHS is evaluated at element level
17         // and accumulated at local nodes
18         (...)
19
20         // local node level MTnew is transferred to partition node level (MTnewL to MT
21         accumulateMTnew(winMTnew);
22
23         // Evaluate the new temperature on each node on partition level
24         for(int i=0; i<nnc; i++)
25         {
26             pNode = mesh->getNode(i);
27             (...)
28         }
29
30         // Transfer new temperatures from partition to local node level (from TG to TL
31         localizeTemperature(winTG);
32
33         MPI_Allreduce(&partialL2error, &globalL2error, 1, MPI_DOUBLE, MPI_SUM,
34         MPI_COMM_WORLD);
35
36         globalL2error = sqrt(globalL2error/(double)this->nnSolved);
37
38         // Output of iterations and globalL2error on parent processor
39
40     MPI_Win_free(&winMTnew);
41     MPI_Win_free(&winTG);
42
43     partialsumT = 0.0;
44     for (int i = 0; i < nnc; i++)
45     {
46         partialsumT += TG[i];
47     }
48     MPI_Allreduce(&partialsumT, &sumT, 1, MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD);
```

-accumulate mass -localize temperature -accumulate MTNew

3.5 R1 a) Validation

For the validation of the FEM solver, the solution of the coarse mesh is compared to the analytical solution in Fig. 2. 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.
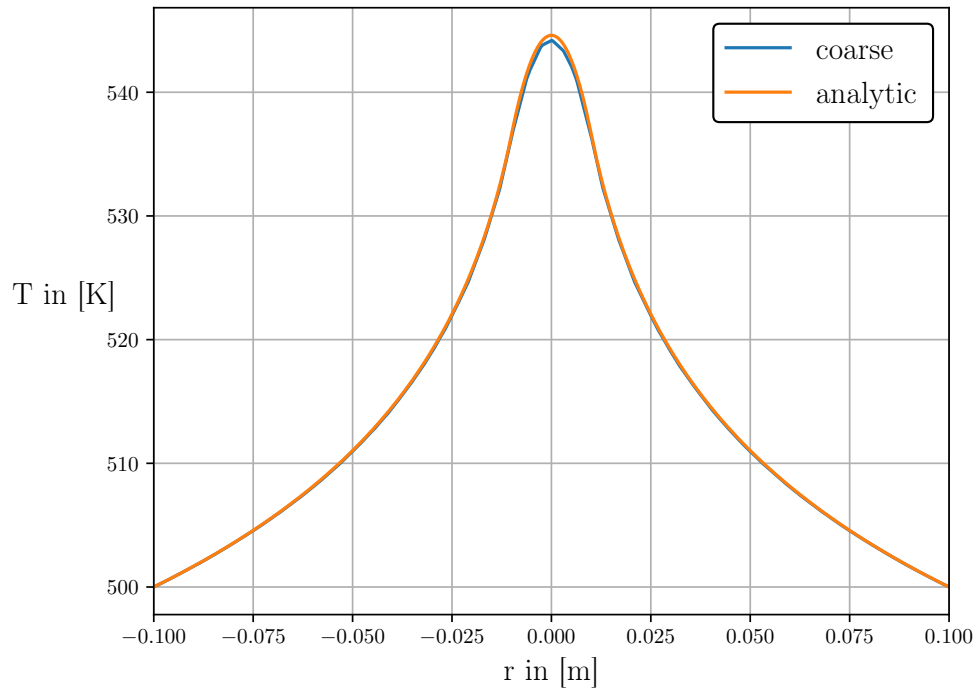
Figure 2: Comparison of the steady-state temperature distribution of the analytical solution (refEQ) and the numerical solution on the coarse mesh

4 Results and Discussion

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.

4.1.1 R1 a)

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. 3. 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 architecture 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 inter-procedural 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.
[2, 3, 4, 5]

The runtimes that were measured for the different compiler flags are shown in Tab. 3. 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 optimized code (-O0). For further investigations, assessing the parallel optimization with OpenMP and MPI, the "-O3 -axSKYLAKE-AVX512" flag combination was used.

4.1.2 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.

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 3: Timings of the various compiler flags for serial optimization, the shortest achieved runtime is printed in bold font

4.2 Parallel OpenMP Solver

In this section the performance gain by parallelization with OpenMP of the two most time consuming loops is investigated (Code excerpt 3).

4.2.1 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 Code excerpt 3. 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. 3, the runtimes of the approach A) (Fig. 3a) for different number of threads is in general significantly higher than the runtimes of approach B) (Fig. 3b), 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 behavior 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
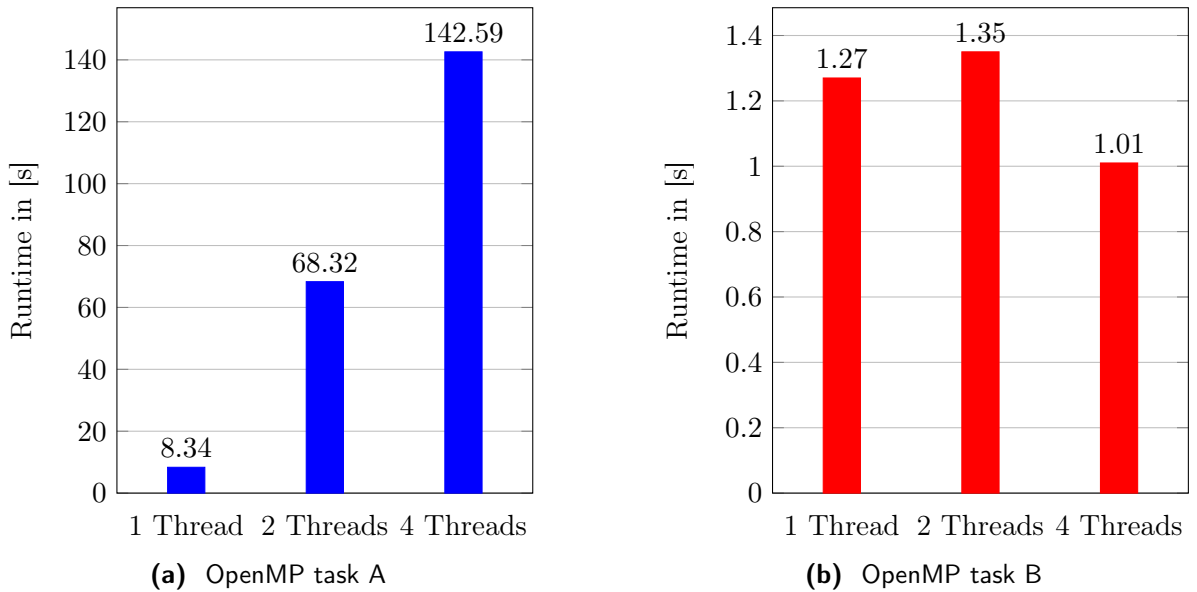
Figure 3: Timings for solver with different OpenMP optimizations using the coarse mesh

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. 3a. 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 in approach B), which is explained in Sec. 3.3.

4.2.2 R2 b)

To further optimize the parallelization with OpenMP, different scheduling options and chunk sizes are investigated for approach B) (Code excerpt 3). Therefore simulations on the fine mesh with a constant number of 4 threads were conducted. Some unexplained behavior 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 initialized 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 Code excerpt 3) for the computation of the RHS on element level. For the second loop (line 45 Code excerpt 3) a static scheduling with default chunk size was used. The results are presented in Fig. 4. 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. 4.
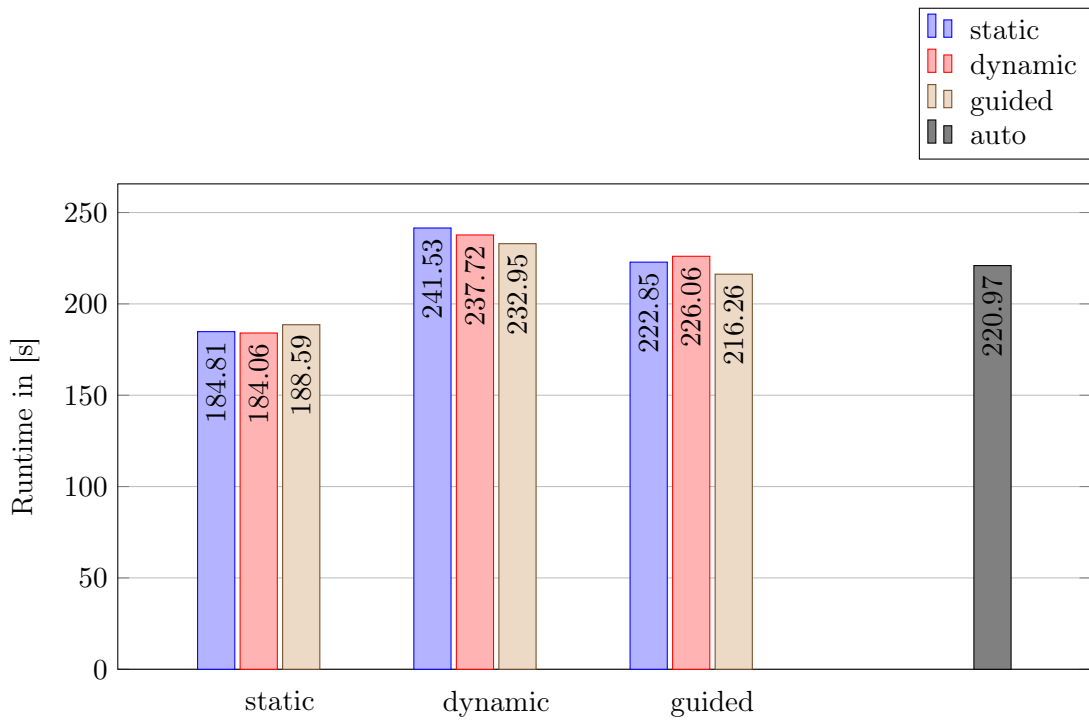
Figure 4: Comparison of different scheduling options (static, dynamic, guided, auto) for different chunk sizes(128, 256, 512) in first parallelized loop for 4 threads using the finest mesh

In addition to the introductory investigations regarding the different scheduling options, static scheduling was inspected further. Therefore, the reduction operations for the parallelization of the first and second loop are both equipped with static scheduling and the effects of different chunk sizes are compared in Fig. 5. 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.
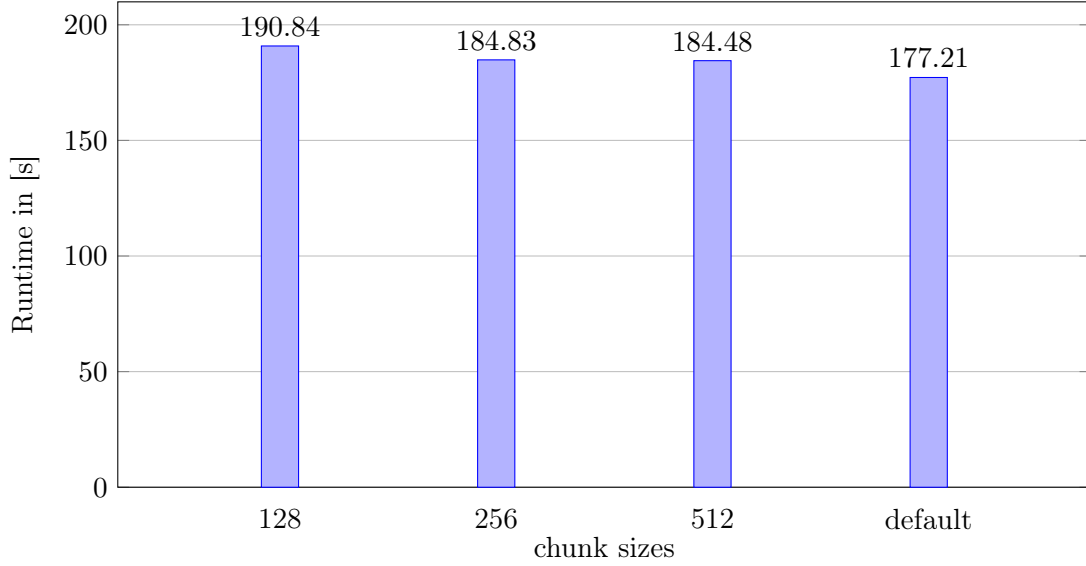
Figure 5: Runtimes for parallelization of loop in line 15 and loop in line 45 in Code excerpt 3 with static scheduling and the same option for chunk size

4.2.3 R2 c)

To evaluate the performance gain by using multiple threads for the parallelization approach B) with static scheduling and default chunk size, the runtime, the speed-up (Eq. (20)) and the Efficiency (Eq. (21)) are computed for 1, 2,4, 6, 8 and 12 threads and plotted in Fig. 6 and Fig. 7. The Speed-up is defined as

$$S_p = \frac{T_1}{T_p} \ , \tag{20}$$

where T_1 denotes the serial runtime of the respective code and T_p the time of parallelized code with p number of threads (cores when considering MPI). The Efficiency is defined as the quotient of speed-up and number of processors

$$E_p = \frac{S_p}{p} \ . \tag{21}$$

Ideal efficiency would correspond to $E_p = 1$, which means a direct proportional dependency of Speed-up and number of processors. This means x number of processors would result in an x times faster code in comparison to the serial code. Ideal Efficiency and Speed-up can not be achieved in a real situation. First, there are always parts of the code that just can be executed serially or have to be executed by each thread. Second, there will always be some overhead or idle time, if threads are waiting for other threads to complete their tasks at the end of parallel regions. If time is the most important feature, speed-up has to be considered. If the core-hours are to be minimized and used most efficiently, the efficiency should be the measure of choice.

In Fig. 6 the different runtimes for the different simulations on three different meshes (coarse, medium, fine) are visualized. For the coarse mesh the shortest runtime was found for 8 threads with 1.13 seconds (Fig. 6a). In comparison to the medium (Fig. 6b) and fine grids (Fig. 6c), which show good scaling behavior,with a monotonous decreasing runtime for an increasing number of threads, the maximum runtime for the coarse mesh was found for 2 threads with 1.5 seconds. This behavior can contributed to the short runtime of the simulations on the coarse mesh itself, as they are in the order of possible fluctuations. Also the number of loop iterations is for the coarse grid not high enough, that parallelization would pay off. For the medium mesh the shortest runtime is 22.76 seconds and 107.75 seconds for the fine mesh, both obtained with 12 threads.
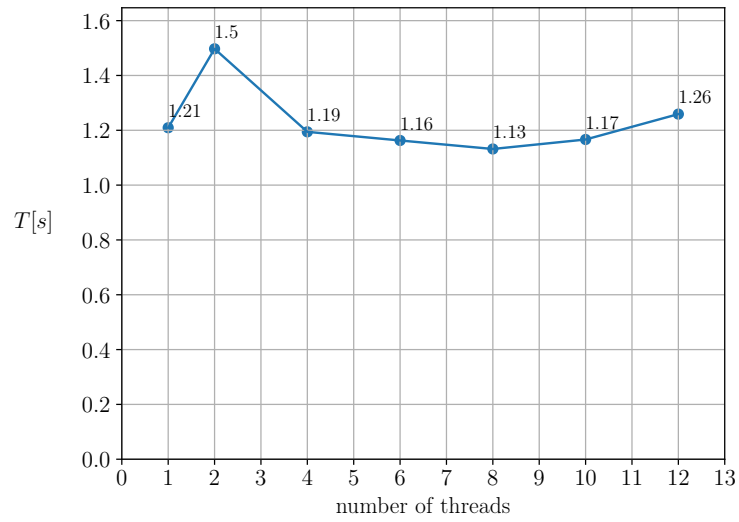
(a) coarse mesh
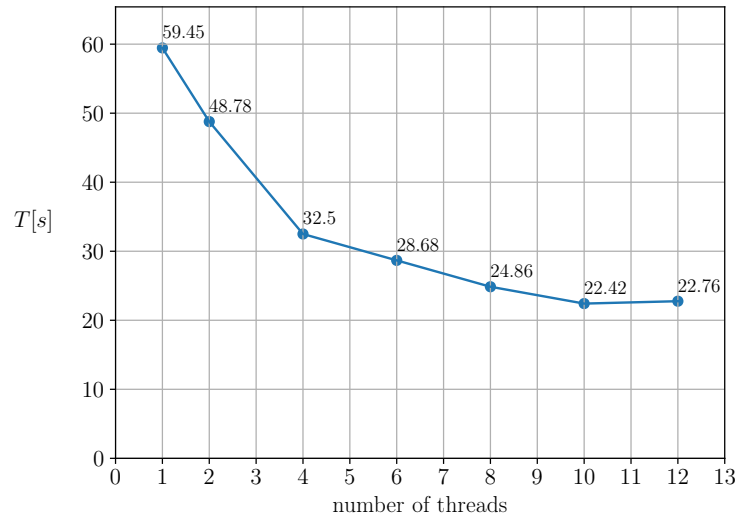
(b) medium mesh
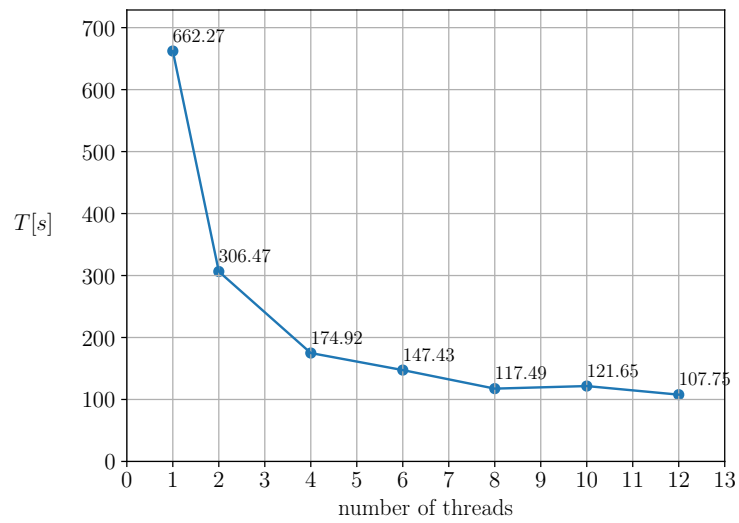
(c) fine mesh

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

Overall the highest speed-up is observed for 12 threads the fine mesh with a 6.15 times faster execution than the serial runtime. This corresponds to an efficiency of 0.51 . The fine mesh shows the highest speed-up and efficiencies for the different number of threads. This is due to the amount of computations. The coarse mesh has not enough elements and nodes to achieve great performance gain through parallelization as the overhead gets fast to large for a higher number of threads. This results in 8 threads being fastest for the coarse mesh, where as 10 threads for the medium mesh and 12 threads for the fine mesh result in the fastest execution time (Fig. 7).

(a) Speed-up
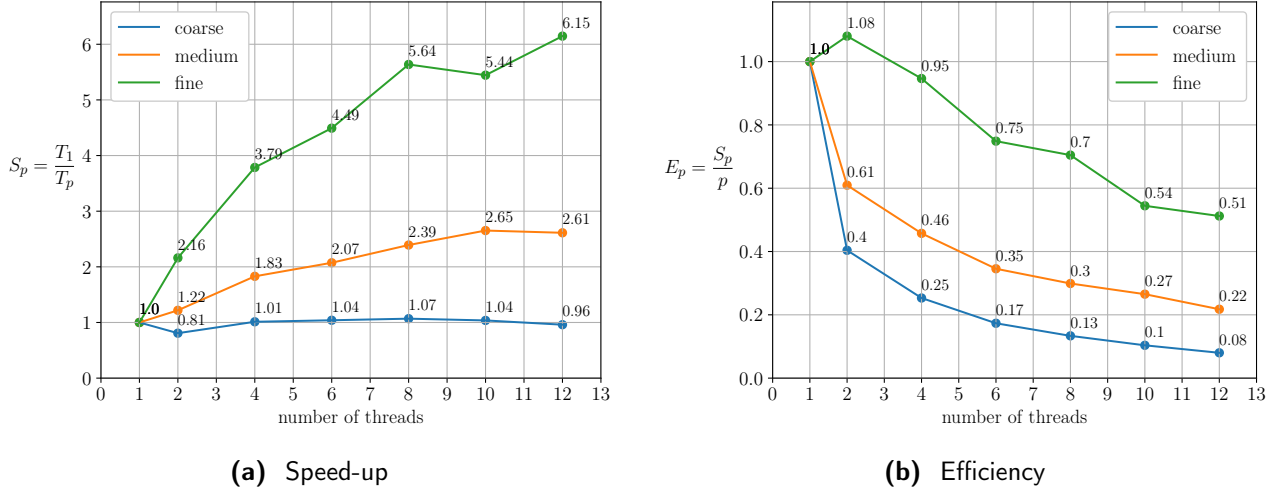
(b) Efficiency

Figure 7: Speed-up and efficiencies for coarse, medium and fine mesh for different numbers of threads

4.2.4 R2 d)

This short subsection is dedicated to the default values of the chunk sizes for different scheduling methods. The default scheduling for OpenMP usage on an intel compiler is static. The default chunk size of static is determined by the quotient of the number of loop iterations and the number of threads. For dynamic scheduling the default chunk size is 1. Lastly, the dehault chunk size for guided scheduling is proportional to the number of unassigned iterations divided by the number of the threads. This means the chunk size dynamically decreases for guided scheduling. The scheduling option auto delegated the decision to the scheduling type to the compiler or runtime of the system [6].

4.3 Parallel MPI Solver

In this section the mpi parallelization of the FEM Solver (reference) is examined regarding it's parallel performance.

4.3.1 R3 a)

Fig. 8 presents the different runtimes for three different meshes that have different level of refinement. The FEM code was run for 1,2,4,6,8,12 and 16 cores, using the compiler flags for best serial performance (Sec. 4.1).

For the coarse mesh the shortest average runtime is at 1.52 seconds with 2 cores (Fig. 8a), for the medium mesh at 64.69 seconds also with 2 cores (Fig. 8b) and for the fine mesh at 342.57 seconds, which is achieved with 4 cores (Fig. 8c).

In Fig. 9, speed-up and efficiency are illustrated for the different meshes. As already observed in the runtime plots, the best speed-up for the coarse and medium mesh is obtained with 2 cores with a speed-up of 1.26 and 1.22, respectively. For the fine mesh the parallelized code ran fastest with 4 cores, resulting in a speed-up of 2.32. The efficiency graphs show an expected development as the efficiency monotonically decreases with an increasing number of threads. For the MPI parallelization with 2 cores on the fine mesh a noticeable high efficiency with 100% was measured, being at optimum efficiency, marked in orange in (Fig. 9f). This means that 2 cores do not introduce nearly any communication overhead and a perfect scaling behavior can be achieved. For the highest number of cores all meshes show a very poor efficiency between only 1-8%. This is also due to the used partitioning approach, which is discussed in (Sec. 4.3.2). Again the same principle applies for distributed memory parallelization. If time is the priority, then according to the speed-up plots, 2 cores should be choosen for the coarse and medium mesh and 4 cores for the fine mesh. If, on the other hand the core-hours are limited and the simulations are supposed to be as efficient as possible, a serial code execution would be most efficient for coarse and the medium mesh. For the fine mesh two cores would be optimal regarding efficiency as they showed excellent scaling behavior.
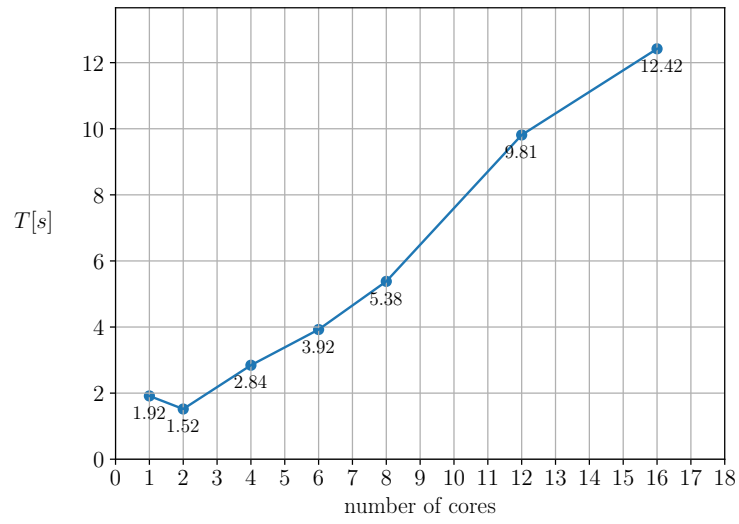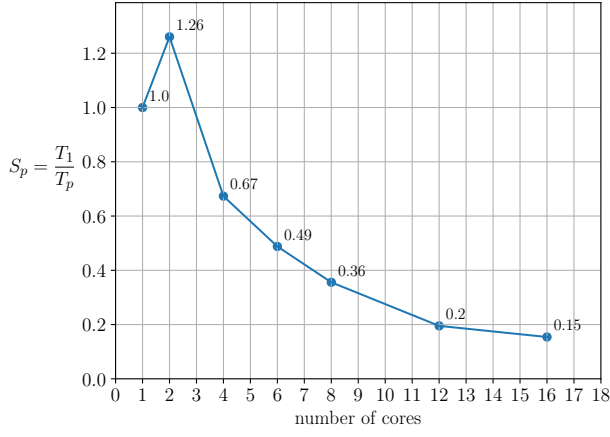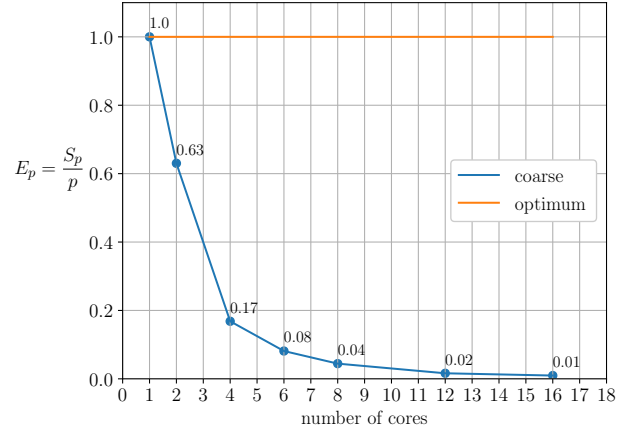
(a) coarse mesh
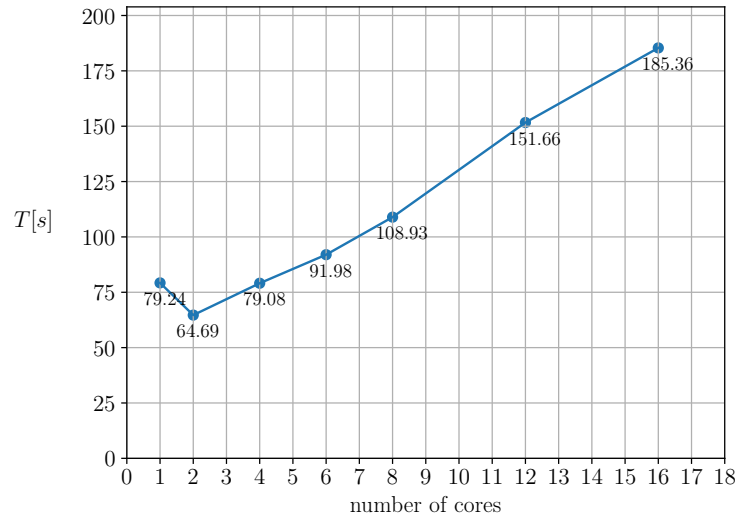
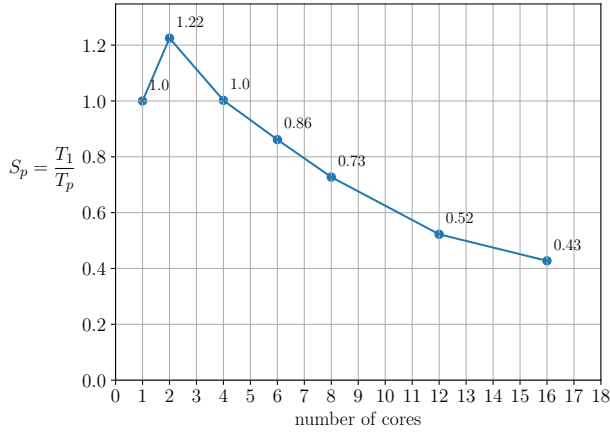**(b)** medium mesh
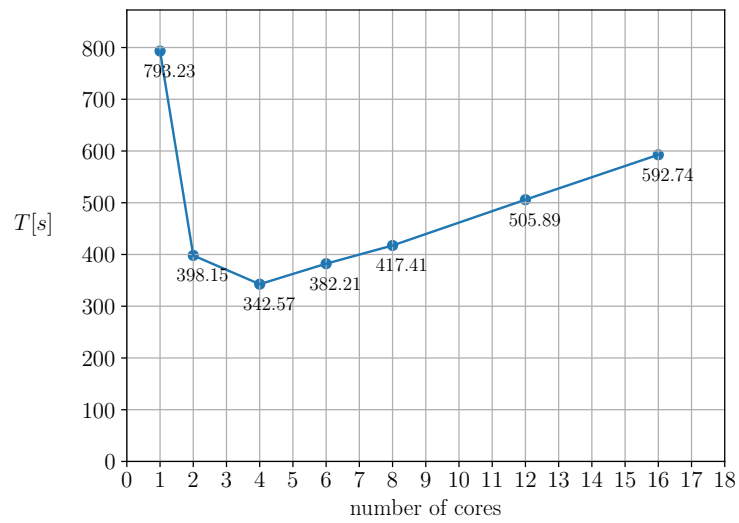
(c) fine mesh

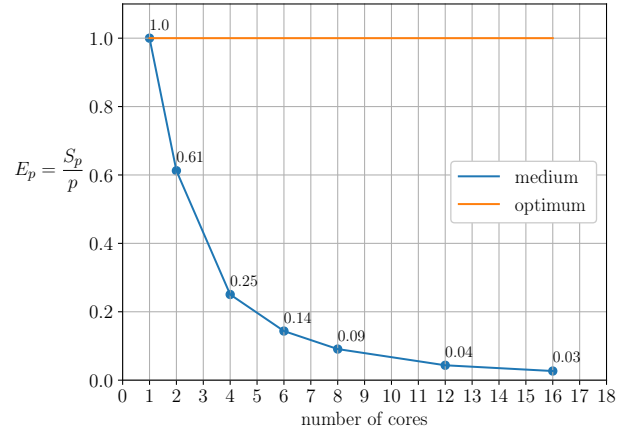**Figure 8:** Runtime for coarse, medium and fine mesh for different number of cores
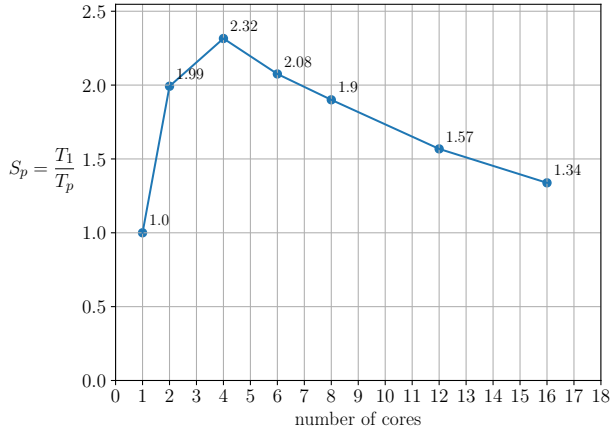
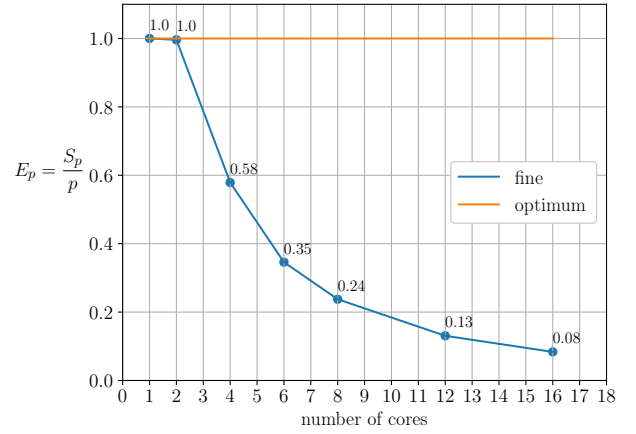**(a)** Speed-up coarse mesh

(b) Efficiency coarse mesh

(c) Speed-up medium mesh

(d) Efficiency medium mesh

(e) Speed-up fine mesh

(f) Efficiency fine mesh

Figure 9: Speed-up and efficiency for coarse, medium and fine mesh for different number of cores

4.3.2 R3 b)

The partitioning of the elements and nodes of the mesh among the individual cores is performed in a round-robin fashion, depending on how the mesh data structure is stored in the mesh input file. Each processors reads with a certain offset a specific amount of elements and nodes from this input file, that is distributed as evenly as possible between the different cores. This results in a more irregular distribution as it can be seen from Fig. 10. This partitioning is far from optimal as it is in general recommended to use a partitioning with minimum boundary surfaces between different partitions. This will reduce the necessary communications between neighboring partitions. For a circular domain like this heat diffusion problem on a flat circular plate, a distributions of p similar circle sectors is advisable, where p represents the number of processors used. This partitioning will result in a minimization of the boundary surfaces.
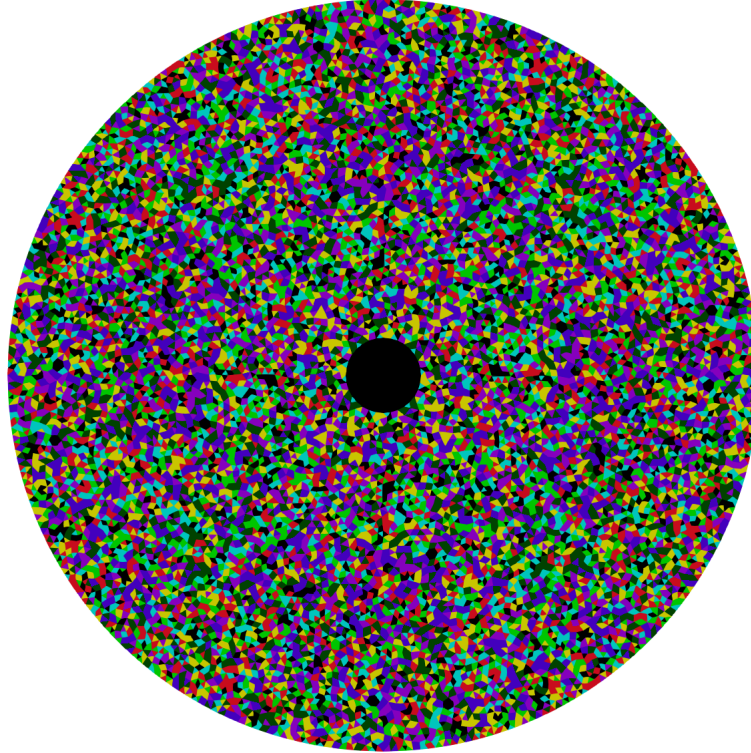

Figure 10: Partitioning of medium mesh using 8 cores

4.3.3 R3 c)

The overall performance of the code can be improved. Some speed-up is achieved, with the shortest runtime for two cores for the coarse and medium mesh and for four cores for the fine mesh. However the efficiency is poor, especially for a higher number of cores (Fig. 9). The current bottleneck is the communication overhead. A lot of loops (line numbers) run over all processors, which results in an increasing loop size for an increasing number of processors. Inside these loops a lot of MPI calls take place, which will result in high communication overhead. Therefore some kind of static data structure, which stores information for all nodes could help. Also a window-halo approach would be beneficial, which allows certain processors to store some data of directly neighboring elements and nodes, that are required for an efficient communication.

Another way to improve the performance of the code is to couple shared memory parallelization on a loop level with OpenMP with the distributed memory parallelization with MPI on global mesh

level. Meaning the loops, especially for computing the RHS on element level and for the computation of the L2 error can be parallelized with the proposed approach B) (Code excerpt 3) on each processor. So each processor executes or at least creates multiple threads. This approach is then called hybrid parallelization.

4.3.4 R3 d)

This section answers some general question regarding MPI commands and communication methods:

1) Why do we use *MPI_ Accumulate* in the code instead of *MPI_ Put*?

With the *MPI_ Accumulate* function a reduction among all the processors of a certain variable or array elements can be performed. Like a summation or finding the maximum value throughout all processors. Put simply overwrites a value.

2) What are the advantages of using one-sided MPU communication over two-sided MPI communication?

On one hand, one-sided communication has no risk of deadlocks as the individual processors do not have to wait for send and receives of other processors in a communicator. Also the communication patterns are in general easier to implement for one-sided communication. On the other hand, the processors simply can read and write to another processor's memory. This however, is error prone and requires a thought-through communication. Two-sided communication is more time-consuming or cumbersome to implement. But, through the send and receive approach, data encapsulation is ensured and the programmer is always aware of which data is exchanged between individual processors. But blocking methods that are required for two-sided communication can lead to an increase in waiting time.

3)What advantage does MPI offer over OpenMP, what is a main obstacle of MPI?

The difference in the MPI and the OpenMP parallelization approaches lies in the memory architecture. MPI is designed for a distributed memory approach, which does not make any limitations to the number of processors to be used. In contrast, OpenMP follows the shared memory approach, so this make the physical memory of a CPU the limitation for the number of threads. But the parallelisation with OpenMP is less elaborate as it can be easily added to existing code, when considering loop parallelization. MPI introduces a lot of additional overhead, when writing parallel code, due to specific MPI commands and the partitioning.

5   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] J. Donéa and Antonio Huerta. *Finite Element Methods for Flow Problems.* Wiley, Chichester ; Hoboken, NJ, 2003.

[2] 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.

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

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

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

[6] Jaka Špeh. OpenMP: For & Scheduling, June 2016.