

PARALLELIZATION OF A RADIATIVE TRANSFER SOLVER

Semester Thesis

written by
Simon Härdi

Supervisor
Prof. Dr. Christoph Schwab

Advisor
Konstantin Grella

Seminar for Applied Mathematics
ETH Zürich

Spring Semester 2012

Contents

1	Introduction	2
2	Discrete Ordinates Method	3
2.1	Discretization	3
2.2	Solution Method	4
2.2.1	Full Tensor DOM	4
2.2.2	Combination Technique	5
3	Parallelization	6
3.1	Motivation	6
3.2	Distribute Problems	6
3.3	Divide Problems	7
3.4	Scheduling	7
3.5	Implementation	8
4	Results	11
4.1	Convergence	11
4.2	Performance	12
4.2.1	Strong Scaling	13
4.2.2	Weak Scaling	16
5	Conclusion	17

1 Introduction

The goal of this semester project is the parallelization of a radiative transport solver and to leverage the parallel computing potential of the ETHZ cluster "Brutus". The tackled problem is the stationary monochromatic radiative transfer problem without scattering

$$\begin{aligned} \mathbf{s} \cdot \nabla_{\mathbf{x}} u(\mathbf{x}, \mathbf{s}) + \kappa(\mathbf{x}) u(\mathbf{x}, \mathbf{s}) &= \kappa(\mathbf{x}) I_b(\mathbf{x}), & (\mathbf{x}, \mathbf{s}) \in D \times S \\ u(\mathbf{x}, \mathbf{s}) &= g(\mathbf{x}, \mathbf{s}) & x \in \partial D, \mathbf{s} \cdot \mathbf{n}(\mathbf{x}) < 0 \end{aligned} \quad (1)$$

where S is the d_S -dimensional sphere, $\kappa \geq 0$ the absorption coefficient, $I_b \geq 0$ the blackbody intensity, $g \geq 0$ the radiation entering the domain and $u(\mathbf{x}, \mathbf{s})$ the unknown radiative intensity.

The theoretical work is done by Konstantin Grella and Prof. Dr. Christoph Schwab [4]. To solve the problem numerically they propose a sparse discrete ordinates method, in which the problem is divided into several discrete directions. This leads to ordinary PDEs, where existing solvers can be applied.

This method was implemented by Konstantin Grella in C++, using the FEM library Deal II [2]. In the existing solver it was possible to set up a radiative transfer problem and let the code run on a single core.

To enhance this implementation and make the solution of bigger problems possible, the code was extended to run the calculations on multiple CPUs using MPI. The partition of the original problem into several subproblems leads to a natural parallelization where every process solves one subproblem. To divide these subproblems on the available cores, a scheduler was implemented and tested.

As some of the subproblems are noticeably bigger than others, it was necessary to divide these problems further and solve them on more than one core. The FEM library Deal II has built-in wrappers for the linear algebra library PETSC [7], it suggested itself to use this functionality to divide the problems.

The code written during this thesis is available as a git repository [3]

2 Discrete Ordinates Method

2.1 Discretization

One possibility to discretize (1) is the discrete ordinates method. A discrete set of directions $S_N = \{\mathbf{s}_j\}_{j=1}^{M_s}$ is chosen for which the parametrized RTE is to be solved:

$$\begin{aligned} \mathbf{s}_j \cdot \nabla_x u(\mathbf{x}, \mathbf{s}_j) + \kappa(\mathbf{x})u(\mathbf{x}, \mathbf{s}_j) &= \kappa(\mathbf{x})I_b(\mathbf{x}), & (\mathbf{x}, \mathbf{s}_j) \in D \times S_N \\ u(\mathbf{x}, \mathbf{s}_j) &= g(\mathbf{x}, \mathbf{s}_j) & x \in \partial D, \mathbf{s}_j \cdot \mathbf{n}(\mathbf{x}) < 0 \end{aligned} \quad (2)$$

To solve the resulting PDEs, a Galerkin FEM discretization in the physical domain is applied. The solutions $u = u(\cdot, \mathbf{s}_j)$ of (2) are in the Hilbert space

$$\mathcal{V}_0^{(j)} := \{u \in L^2(D) : \mathbf{s}_j \cdot \nabla_x u \in L^2(D)\} \quad (3)$$

with zero inflow boundary conditions:

$$u|_{\Gamma_-} = 0, \quad \Gamma_- = \{x \in D, n(x) \cdot \mathbf{s}_j < 0\} \quad (4)$$

The resulting weak form is

$$\int_D (\mathbf{s}_j \cdot \nabla_x u + \kappa u) v \, dx = \int_D f v \, dx \quad \forall v \in \mathcal{V}_0^{(j)} \quad (5)$$

As the physical problem is a hyperbolic type equation, the standard Galerkin approach results in an unstable scheme. The stabilization is done with a streamline upwind Petrov-Galerkin (SUPG) method proposed by Kanschä *et al.* in [6] where an additional diffusion term in transport direction is added:

$$\begin{aligned} \int_D (\mathbf{s}_j \cdot \nabla_x u + \kappa u) v \, dx + \int_D (\mathbf{s}_j \cdot \nabla_x u + \kappa u) \delta \mathbf{s}_j \cdot \nabla_x v \, dx \\ = \int_D f v \, dx + \int_D f \delta \mathbf{s}_j \cdot \nabla_x v \, dx \quad \forall v \in \mathcal{V}_0^{(j)} \end{aligned} \quad (6)$$

As proposed by Kanschä when using $\kappa = 1$, the SUPG parameter is chosen as $\delta \approx 0.3h$, where h is the grid spacing.

The number of directions is chosen as

$$\begin{aligned} M_S &= 2N + 3 & \text{for } d_S = 1 \\ M_S &= (N + 1)^2 & \text{for } d_S = 2 \end{aligned} \quad (7)$$

depending on the parameter N . For a two-dimensional problem the M_S directions are distributed evenly over the angular space. In three dimensions the minimum determinant points as proposed by Sloan *et al.* in [8] are used.

As in [4] it is assumed that the solution u_j is in $H_1(D)$. The space $H_1(D)$ is discretized by choosing a hierarchical sequence of spaces V_D^l on dyadically refined meshes T_D^l , $l = 0 \dots L$ over the physical domain:

$$V_D^l := S^{p,1}(D, \mathcal{T}_D^l) \subset H^1(D) \quad (8)$$

They consist of continuous piecewise polynomial functions of degree $p \geq 1$. To satisfy the boundary conditions in a strong sense and make the function space directional dependent again, the trial and test space is chosen as $V_{D,0}^{L,j} = V_D^L \cap \mathcal{V}_0^{(j)}$.

This leads to the linear variational problem: Find $u_{j,L}(\mathbf{x}) \in V_{D,0}^{L,j}$ such that

$$a_\delta(u_{j,L}, v_{j,L}) = l_\delta(v_{j,L}) \quad \forall v_{j,L} \in V_{D,0}^{L,j} \quad (9)$$

As basis functions we use bilinear nodal functions on a quadrilateral mesh or a hexahedral mesh for 2D and 3D respectively. The resulting complexity of a physical subproblem for a single direction is $M_D = (2^L + 1)^d$.

By selecting a basis and a maximal level L one gets a linear system of equations

$$Au = f \quad (10)$$

for each subproblem. The evaluation of the integrals is done with a Gaussian quadrature, where the degree was chosen such that the integration is exact for piecewise linear coefficient functions.

2.2 Solution Method

If the set of directions S_N is chosen according to (7), the RTE can be solved numerically by solving the resulting M_S purely physical transport subproblems. The linear system of equations is solved with a GMRES-method. The used preconditioner is an incomplete LU-decomposition with the fill-in parameter $k = 1$.

2.2.1 Full Tensor DOM

If all of the subproblems are discretized with the same grid size, meaning the same amount of basis functions M_D , the resulting computational complexity is

$$M_{(L,N)} = M_D \cdot M_S \quad (11)$$

However, with such a distribution over the angular space the method suffers from the curse of dimensions. For real world problems with three physical and two angular

dimensions, an unreasonable amount of degrees of freedom is needed to achieve a given level of error. In [4] it is proven that the convergence rate of the full tensor DOM method is

$$\|u - I_S^N P_D^L u\|_{H^{1,d_S}(D \times S)} \lesssim \max \left\{ 2^{-sL}, N^{-t+(d_S-\alpha)} \right\} \|u\|_{H^{1+s,d_S+t}(D \times S)} \quad (12)$$

where $u_{L,N} = I_S^N P_D^L u$ is the approximation of a function $u \in H^{s+1,d_S+t}(D \times S)$, $s \in [0, p]$, $t \in \mathbb{N}_0$. The parameter $\alpha \in \mathcal{R}$ is depending on the angular dimension d_S and the set of points on the unit sphere which are needed to interpolate the solution to the angular space. For $d_S = 1$ it can be assumed that $\alpha \approx 1/2$ or smaller, for $d_S = 2$ $\alpha \approx 1$ or smaller.

2.2.2 Combination Technique

To overcome the curse of dimensionality, one can apply the combination technique to the radiative transfer problem as proposed by Grella and Schwab [4].

To derive a sparse solution it is often assumed that the domain of the problem is a Cartesian product domain $D_1 \times D_2$ on top of which the function space is approximated by a tensor product of discrete spaces $V_1^{L_1} \otimes V_2^{L_2}$. This discrete spaces are assumed to consist of hierarchic families of spaces $V_1^{l_1}, V_2^{l_2}$ with $V_i^{l_0} \subset V_i^{l_1} \subset \dots \subset V_i^{L_i}$, $i = 1, 2$. The increment between two subspaces is denoted by $W_i^{l_i}$:

$$V_i^{l_i} = V_i^{l_i-1} \oplus W_i^{l_i} \quad (13)$$

A sparse approximation of the approximate full tensor solution

$$u_{L_1, L_2} = \sum_{l_1=0}^{L_1} \sum_{l_2=0}^{L_2} Q_{l_1, l_2} u \quad (14)$$

can then be given by restricting the summation to a subset of the indices l_1, l_2 :

$$\hat{u}_{L_1, L_2} = \sum_{0 < f(l_1, l_2) < L_1, L_2} Q_{l_1, l_2} u \quad (15)$$

where Q_{l_1, l_2} is a projection operator on the tensor product of detail spaces.

If the splitting in incrementing subspaces is not available, a sparse solution can still be given by the combination technique. As this sparse solution is a linear combination of the contributions from the detail spaces, it can be assembled from addition and subtraction of different full solutions. For the example of $L_1 = L_2 = L$ and the sparsity profile $f(l_1, l_2) = Ll_1 + Ll_2$ the combination formula is given by

$$\hat{u}_{L, L} = \sum_{i=0}^L \sum_{l_1=0}^{L-i} \sum_{l_2=0}^i u_{l_1, l_2} - \sum_{i=0}^{L-1} \sum_{l_1=0}^{L-i} \sum_{l_2=0}^i u_{l_1, l_2} \quad (16)$$

In general, the solution of the combination technique is

$$\hat{u}_{L_1, L_2} = \sum_{l_1=0}^L u_{l_1, l_2^{\max}(l_1)} - \sum_{l_1=0}^{L-1} u_{l_1, l_2^{\max}(l_1+1)} \quad (17)$$

where $u_{(l_1, l_2)}$ is the solution of the full tensor subproblem. Applied to the radiation transport problem L_1 and L_2 are set to L and N respectively, $l_2^{\max}(l_1)$ is set to $2^{\lfloor \log_2(N+1) \rfloor / L(L-l_1)}$ and l_1 and l_2 are the physical and the angular resolution index of the subproblem.

This solution is in general not identical to the direct sparse solution, but according to Griebel *et. al.* [5] equivalent in its complexity and convergence properties.

Using this technique, the convergence rate stays up to a logarithmic factor the same:

$$\|u - I_S^N P_D^L u\|_{H^{1, d_S}(D \times S)} \lesssim L \max \left\{ 2^{-sL}, N^{-t+(d_S-\alpha)} \right\} \|u\|_{H^{1+s, d_S+t}(D \times S)} \quad (18)$$

Whereas the complexity reduces to

$$M_{(L, N)} \lesssim L^\theta \max \left\{ 2^{dL}, N^{d_S} \right\} \simeq (\log M_D)^\theta \max \{M_D, M_S\} \quad (19)$$

where $\theta = 1$ if $N^{d_S} \simeq 2^{d_S}$ and zero otherwise.

This has the advantage of reducing the scaling of the complexity to one of a problem on a single domain while maintaining the convergence rate, both multiplied with a logarithmic factor.

3 Parallelization

3.1 Motivation

With the combination technique the necessary number of degrees of freedom for a two dimensional problem leads to a computational complexity that can be handled with a single core. However, if applied to real world 3D problems the computation time grows to an unreasonable span of time. This is the reason why the code was adapted to run on several cores. The code was compiled and tested on a Linux machine and the ETH-cluster Brutus.

3.2 Distribute Problems

Both the full tensor DOM and the sparse DOM via combination technique lead to several subproblems. It was a natural choice to do the parallelization by distributing

the subproblems on the cores, each process gets its own PDE to solve.

This distribution has the advantage of a minimal required communication between the nodes which is limited to the set-up and the evaluation of the solution.

3.3 Divide Problems

If the combination technique is used, some angular directions are resolved finer than others. This leads to a highly unbalanced amount of work between the subproblems and only distributing them is no longer an option.

To get an even distribution of work between the available cores, it is necessary to assign several cores to the biggest problems. This requires the possibility of splitting a problem, which was implemented with the library PETSC, as Deal II provides wrapper to the PETSC functions to parallelize the assembly and solution of a linear system of equations.

3.4 Scheduling

To distribute and divide the subproblems on the available cores a scheduler was implemented. Its tasks are calculating an optimal ratio of splitting and dividing the problems to gain an optimal work distribution.

As finding the optimal workload distribution is an NP hard optimization task, it requires too much time to be solved exactly.

Instead, we chose the following heuristics called longest processing time (LPT) to produce an approximation to the optimal schedule. This algorithm orders problems according to their weight in a descending way and then assigns each problem to the node with the currently smallest work load.

A pseudo code would look like this:

```
1  % Calculate the idealLoad per core
2  % The list problemSizes contains the weight of all problems,
   nCores is the number of available cores
3  idealLoad=sum(problemSizes)/nCores;
4
5  for i=1:nProblems
6      % How many cores are assigned to the current problem
7      corePerProblem = ceil(problemSizes[i] / idealLoad)
8
9      for j=1:corePerProblem
10         % Find the core that has currently the smallest work
           load
```

```

11         core = findCoreWithSmallestLoad();
12
13         % Assign the problem and its work load to this core
14         core.addLoad(problemSizes[i]/corePerProblem)
15         core.assignProblem(i);
16     end
17 end

```

The function `findCoreWithSmallestLoad` checks all available cores for their work load and returns the one with the smallest. Since the subproblem sizes of the combination technique are factors of 2^d , the LPT algorithm and static workload splitting should be combinable quite well.

To be able to test different scheduling algorithms without much effort, the scheduler was implemented so that the actual algorithm could easily be exchanged. For this end, the scheduler class delegates the actual scheduling to a scheduling algorithm class for which a specific implementation can be provided by the user in a template parameter of the scheduler class.

3.5 Implementation

The main coding work besides the implementation of the scheduling class was done in the class `DOMTransportSolver` and its classes `PhysicalTransportSolver` and `-Solution`. Their parallel versions are provided in the namespace `parallel`, making the change to the parallel version as easy as writing `parallel::` in front of the declaration of the `DOMTransportSolver`.

To set up the RTE problem the code generates the list of subproblems inside the `DOMTransportSolver`. This proceeding is the same in the serial and the parallel version of the program and is done in the function `fillCompProbList` which implements the subspace partition of the combination technique.

The used function `addCompProb` creates an instance of a `ComputeProblem`, in which all necessary information is stored, with the given parameter and adds it to the list.

In the parallel version, the scheduler is applied after the call of `fillCompProbList`. The task of the scheduler is not only to calculate a computation plan of which CPU calculates which problem but also to make sure this plan is followed. This is done by giving the scheduler access to the `probList`, allowing it to remove all problems that are not calculated on the current node.

The work flow in the scheduler is therefore to first calculate a scheduling plan, which

is done in its function `schedule()` and then to modify the `probList`, which is done in the function `adaptProbList`.

```
1 void Scheduler<SchedulingAlgorithm, spacedimV>::schedule() {
2 // Get number of available cores
3 int nCores = MPI::COMM_WORLD.Get_size();
4
5 // Create instance of scheduling algorithm
6 SchedulingAlgorithm schedulAlgo;
7 // And call its scheduling method
8 schedulAlgo.schedule(
9     // number of available cores
10    nCores,
11    // List with problem weights, problem i has a
12    // computational complexity of problemSizes[i]
13    problemSizes,
14    // List of lists to be filled with the scheduling,
15    // problem i is calculated on all cores contained in the
16    // list problemToCore[i]
17    problemToCore
18 );
19 }
```

The scheduling itself is outsourced to an instance of the class `SchedulingAlgorithm`, which gets only the number of available cores and a list with the problem weights.

```
1 void Scheduler<SchedulingAlgorithm, spacedimV>::adaptProbList() {
2 /*
3  * Allocation of needed variables
4  */
5
6 // Find rank of this process
7 thisCore=MPI::COMM_WORLD.Get_rank();
8
9 // Iterate over all Problems
10 for( int problem = 0; problem < problemToCore.size(); ++problem,
11     ++probListIt){
12
13     nCoresPerProblem = problemToCore[problem].size();
14
15     // Create new MPI group
16     temp_group = MPI::COMM_WORLD.Get_group().Incl(
17         problemToCore[problem].size(), problemToCore[problem]
18         ].data());
19     // Create communicator of this group
20     temp_comm = MPI::COMM_WORLD.Create(temp_group);
21 }
```

```

19 // Add communicator to ComputeProblem
20 probListIt->communicator = temp_comm;
21
22 // Remove ComputeProblem from probList if current
   process is not involved
23 if( std::find(problemToCore[problem].begin(),
   problemToCore[problem].end(), thisCore ) ==
   problemToCore[problem].end() ){
24     probListIt = probList.erase(probListIt);
25     --probListIt;
26 }
27 }
28 }

```

The MPI variables used in this function are needed to give PETSC the number and ranks of the nodes involved in the computation of each problem. The MPI communicator where these parameters are stored in is given to the `ComputeProblem` where it can be retrieved by the time it is needed.

After this step, the scheduling is done and the `DOMTransportSolver` generates an instance of the `PhysicalSolver` and the `PhysicalSolution` for each subproblem contained in `probList` and runs them.

The (simplified) code of the modified method run in the `DOMTransportSolver`:

```

1 void DOMTransportSolver<VeloGen, Method, TransApp>::run() {
2 // Allocate needed variables
3
4 // Create the problem list and fill it
5 std::list<transo::ComputeProblem> problist;
6 fillCompProbList(problist);
7
8 // Create an instance of the scheduler and give it a handle to
   the problist
9 Scheduler<SchedulingAlgorithmSimple> schedul(problist);
10 // Run the scheduler
11 schedul.run();
12
13 for(probit = problist.begin(); probit != problist.end(); ++
   probit){
14     // Create a PhysicalTransportSolution for each
       subproblem in schedule
15     sollist.push_front(transo::TransportSolution(
16         transo::parallel::
           PhysicalTransportSolution<TransApp::
           spacedimV> );
17

```

Results

```
18 |         // Create a PhysicalTransportSolver for each subproblem
19 |         in schedule
20 |         transo :: parallel :: PhysicalTransportSolver<Method,
21 |         TransApp> ts(*probit, ((*solit)));
22 |
23 |         // Solve the problem
24 |         ts.run();
25 |     }
26 | }
```

As the distribution of the problems is already done, the modifications in the two classes `PhysicalTransportProblem` and `-Solution` are specifically to make the problem splitting possible. To achieve this all the contained matrices and vectors had to be changed into parallel version. This was done by using PETSC, exchanging their classes to the corresponding Deal II PETSC wrappers.

Also, the assembly function was adapted to calculate only the contributions to the part of the system matrix that is stored on the current process, and the solver routines were changed to parallel ones.

To efficiently solve the linear systems of equations, an ILU preconditioner was used. Unfortunately, PETSC does not support a parallelized ILU preconditioner out of the box. To circumvent this problem, PETSC was compiled with the additional package HYPRE [1]. As this is not supported by Deal II, it is necessary to give some command line options to PETSC to configure the preconditioner.

4 Results

The parallelization was tested with a model problem with a degenerate Gaussian on the right hand side:

$$I_b(x) = \exp\left(-8\left(x - \begin{pmatrix} 0.5 \\ 0.5 \end{pmatrix}\right)^T \begin{pmatrix} 4 & -2 \\ -2 & 1 \end{pmatrix} \left(x - \begin{pmatrix} 0.5 \\ 0.5 \end{pmatrix}\right)\right) \quad (20)$$

This rhs forms a beam which is decaying like a Gaussian in all directions except the beam direction. The problem was solved on the hyper-rectangle $D = [0, 1]^d$, $d = 2, 3$ with zero inflow boundary conditions, a constant absorption coefficient $\kappa = 1$ and without scattering.

4.1 Convergence

While starting some test runs with this problem, some serious convergence problems appeared. Even for small problems the code took a long time to complete and the

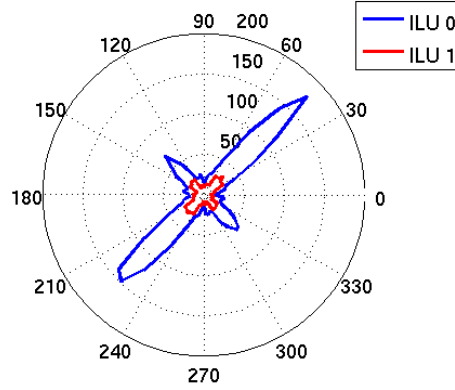


Figure 1: This plot shows how many iterations were needed until convergence depending on the direction s_j . The blue line is with a fill in parameter $k = 0$, the red line shows $k = 1$. It is easy to spot the problematic directions, and the influence of the fill in parameter.

iteration counts to reach the convergence criterion of a residuum $r < 10^{-12}$ were unreasonably high. A closer investigation showed that for certain directions the needed iterations were considerably higher than for others.

This problem could be solved by choosing a higher fill-in parameter for the ILU preconditioner. Figure 1 shows the iteration numbers for each direction before and after this adaption. The used parameters were $L = 8$ and $N = 32$.

4.2 Performance

The performance measurements were done on Brutus with the CPU model Opteron8380 to get consistent timings. This quad-core CPUs run on a frequency of 2.5 GHz and have 32 GB of RAM.

The varying parameters were the spatial resolution parameter L with a complexity of $M_D = (2^L + 1)^d$ for a single physical subproblem, the angular resolution parameter N with a direction resolution of $M_S = 2N + 3$ and the number of used CPUs C .

All timing runs were performed three times to reduce random timing effects.

4.2.1 Strong Scaling

In a strong scaling test a fixed-problem size W is ran with a varying number of processors. The speed up S is the time one processor needs to solve this problem divided by the time C processors need:

$$S(W, C) = \frac{t(W, C = 1)}{t(W, C)} \quad (21)$$

The efficiency of the parallelization can then be measured by dividing the speed up by the number of processors. The result is a percentage of how good the program scales compared with a linear scaling:

$$E = \frac{S(W, C)}{C} \quad (22)$$

Figure 2 shows the timing (a) and the efficiency plot (b) for the RTE solver with the full tensor solution method. As expected the efficiency number decreases for increasing number of processors, since the non-parallelizable part of the code comes into play. Also a non-monotonic behavior can be observed. This results from the fact that the parallelization is done in two ways, the distribution and the splitting of the problems.

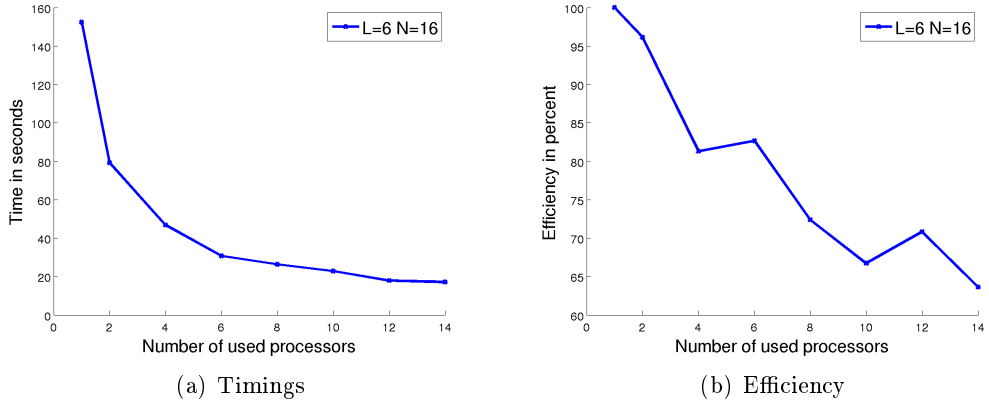


Figure 2: Strong scaling of the RTE solver with the full tensor method. On the left the timing results are shown, on the right the according efficiency numbers are plotted.

However, calculating the efficiency number by comparing the runtime of the parallel code on multiple and on one processor is some way of cheating as the parallelization

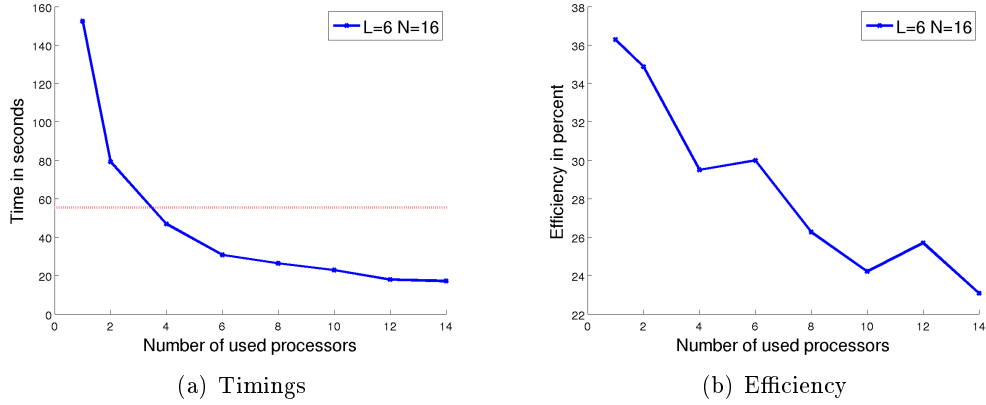


Figure 3: Strong scaling in comparison with the serial version. The red line in the left plot shows the time needed with the serial version.

itself introduces a lot of overhead. Therefore a more honest efficiency number is calculated by comparing the parallel runtime to the one of a serial code. This is done in Figure 3. In the timing plot the horizontal line indicates the runtime of the serial code. It needs four processors to be faster than one processor with the serial version. The efficiency number is now down to approximately thirty percent.

The same observations can be made for the combination technique as shown in Figure 4. Here, the efficiency is strongly dependent on the scheduling result. This can be seen at the efficiency jump, where the scheduling algorithm did a bad job in splitting up the problem on four processors, resulting in a distribution where one processor did only half of the work of the others.

In three dimensions the parameters $L = 3$ is used. The results are better, as the problem sizes are getting bigger and the parallelization is more effective. It only needs two processors to outrun the serial version, and the efficiency is near 70 percent as shown in Figure 5.

This shows that the parallelized code works, it is possible to speed up the calculation of a fixed problem size. But the overhead introduced by the parallelization has to be considered, since it is possible to actually slow down the calculation by using the parallel version with not enough processors.

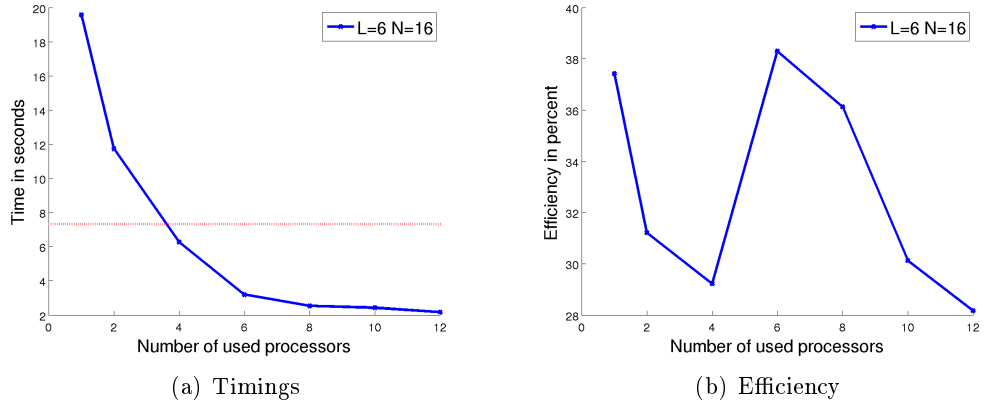


Figure 4: Strong scaling of the RTE solver with the combination technique. On the left the timing results are shown, on the right the according efficiency numbers are plotted. The red line indicates the time one processor needs to run the serial version.

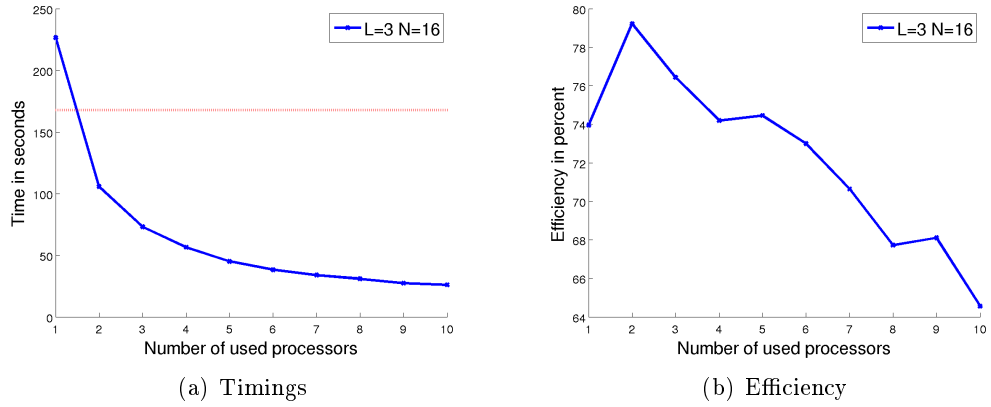


Figure 5: Strong scaling of the RTE solver in 3D. The red line indicates the time one processor needs to run the serial version. The results are remarkably better than in the 2D timings.

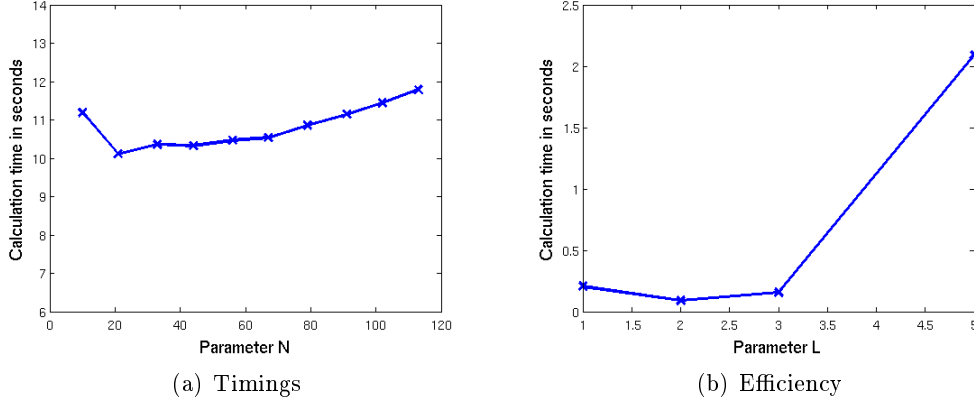


Figure 6: The timing results from the weak scaling test. In (a) the timing stays almost constant while an increasing number of CPUs solved an increasing number of subproblems, fixing the amount of work per processor and the size of the subproblems. In (b) the timing results from the weak scaling for increasing subproblem sizes are shown. The time to solve grows with an increasing problem size and more processors, as the communication overhead grows.

4.2.2 Weak Scaling

Weak scaling is the measurement of how a program performs if the amount of work per processor is fixed. For the RTE solver this can be done in two ways. Increasing the angular level N will increase the number of subproblems whereas increasing the discretization resolution L of the physical space will enlarge the subproblems without changing their number.

It is clear that these two possibilities have to be considered separately as their parallelization works very differently.

Testing the weak scalability by increasing the number of subproblems was done with the parameter $L = 5$ and $C = 1, \dots, 10$. The parameter N was chosen to keep the workload per processor as fixed as possible. The result is shown in Figure 6(a). The code performs very well, the execution time stays almost constant. Of course this will get worse if not only the solving of the subproblems but also an evaluation of the solutions is implemented.

To test the scaling with increasing subproblem sizes it turned out to be quite difficult to get timing data. Since the varying parameter has to be L and the complexity in 2D grows with $(2^L + 1)^2$, the number of needed processors to keep the workload fixed

grows really fast. The testing was done with $N = 0$ and $L = 1, \dots, 5$, resulting in a needed amount of processors of $C = 1, 3, 9, 32$ and 121 . The result is shown in Figure 6(b).

This setup scales worse than the one with an increasing amount of subproblems, but this was expected since the parallel solving of a linear system of equations generates overhead that is not present when just distributing independent jobs.

5 Conclusion

The implemented version of the code is able to perform in parallel and shows a decent speed up when using an increasing number of CPUs. It is also possible to perform faster than the serial version. The implementation is done in a very user friendly way, allowing a fast and easy switching between the serial and parallel version of the code.

However, the implemented scheduling algorithm is very simple and there were scheduling jams observed for specific combinations of parameters. In a future work it must be a goal to improve this algorithm. This applies especially when using the combination technique where the parallelization efficiency is highly dependent on the scheduler. Also, the `probList` that is created inside the `DOMTransportSolver` class is created on each node, and every node also has to do the whole scheduling. It would be an improvement to do this in parallel and reduce the overhead of the code. This would not only need an improvement of the scheduling algorithm but also of the scheduler itself.

References

- [1] CASC at Lawrence Livermore National Laboratory. Hypre is a library for solving large, sparse linear systems of equations on massively parallel computers, July 2012. <http://acts.nersc.gov/hypre/>. (Cited on page 11.)
- [2] Wolfgang Bangerth, Timo Heister, and Guido Kanschat. A finite element differential equations analysis library, July 2012. <http://www.dealii.org>. (Cited on page 2.)
- [3] Konstantin Grella and Simon Härdi. Git repository, July 2012. <https://git.math.ethz.ch/sam/ggrella/transport/transo.git>. (Cited on page 2.)
- [4] Konstantin Grella and Christoph Schwab. Sparse discrete ordinates method in radiative transfer. *Computational Methods in Applied Mathematics*, pages 305–326, 2011. (Cited on pages 2, 4 and 5.)
- [5] M. Griebel, M. Schneider, and C. Zenger. A combination technique for the solution of sparse grid problems. In *Iterative Methods in Linear Algebra*, pages 263–281. IMACS, Elsevier, North Holland, 1992. (Cited on page 6.)
- [6] Guido Kanschat. Solution of radiative transfer problems with finite elements. In *Numerical Methods in Multidimensional Radiative Transfer*, pages 49–98. Springer Berlin Heidelberg, 2009. (Cited on page 3.)
- [7] Mathematics and Computer Science Division Argonne National Laboratory. Portable, extensible toolkit for scientific computation, July 2012. <http://www.mcs.anl.gov/petsc/>. (Cited on page 2.)
- [8] Robert S. Womersley and Ian H. Sloan. How good can polynomial interpolation on the sphere be? *Advances in Computational Mathematics*, 14:195–226, 2001. (Cited on page 3.)

Appendix

Installation on Linux

This is a step-by-step guide to install the library Deal II together with PETSC, Metis and HYPRE on a linux operating system.

It was tested on a 64bit Ubuntu 12.04 LTS distribution.

Download and Versions

The links in this list were valid by the date of writing this thesis, Jul 23 of 2012.

- Deal II
 - Version 7.1.0
 - <http://www.dealii.org/>
- PETSC
 - Version 3.2-p7
 - <http://www.mcs.anl.gov/petsc/>
- Metis
 - Version 4.0.3
 - <http://glaros.dtc.umn.edu/gkhome/metis/metis/overview>
- HYPRE
 - Version 2.7.0b
 - Downloaded with PETSC

Installation of the Libraries

- Compile Metis
 - Change to metis directory, open the 'Makefile.in' and add the compiler flag '-fPIC'

- Call 'make'
- Compile PETSC
 - Make sure the environment variable 'PETSC_DIR' is set to the top level directory of PETSC. Also set the environment variable 'PETSC_ARCH' to some meaningful name. These variables have to be set each time PETSC is used, so setting it in the '.bashrc' of your system may be a good idea.
 - Change to the PETSC directory and call 'configure' with the following options:

```
1 ./configure --with-x=0 --with-mpi=1 --with-hypre=1 --
  download-hypre=1 --with-shared-libraries=1
```

- Call 'make' with the options that are proposed by the configuring step.
- Compile Deal II
 - Change to the Deal II directory and call 'configure' with the following options:

```
1 ./configure --enable-mpi CC=mpicc CXX=mpicxx --enable-
  shared --disable-threads --with-metis=/home/simon/
  Documents/ETH/Semester-thesis/libraries/metis-4.0.3
  --with-blas
```

The path to metis has to be adjusted and if lapack\blas is not installed in the standard directory, the path must be given.

- Call 'make all'. This step takes quite a while, it can be speeded up by using more than one process with 'make -j x all' where x is the number of processes.

Configuring Cmake

After downloading the source code of the radiative transfer solver, create a directory where you want to build the code. Change to this directory and call cmake (CMake has to be installed) with the path to the source code top directory as argument. After trying to configure the first time by pressing 'c' (which will result in an error), press 't' to show all options and change the 'CMAKE_CXX_COMPILER' and 'CMAKE_C_COMPILER' to mpic++ and mpicc respectively. It is important to do this before you set any other options, as changing the compiler causes CMake to throw away all other cached options.

After setting the compiler, turn off the 'USE_TBB_LIB_FOR_DEAL_II_MT' and the 'brutus_FLAG' and set the paths to the libraries Deal II, PETSC and Metis. Sample paths are shown in listing TODO:MakeListingOfPaths. If everything is set correctly, pressing 'c' and 'g' should produce a makefile that can make all targets that are set in 'CMakeLists.txt'

Installation on Brutus

In principal the installation on Brutus is the same as the installation in the previous section. However, there are a few changes that are described here.

The libraries available on Brutus are organized in modules. It is important that all needed modules are loaded before starting the work. An elegant way to do this is via the '.bash_profile' in the home directory.

The modules needed for this thesis were

- open_mpi
- mkl
- cmake
- gcc/4.6.1
- boost

After loading the modules and setting the environment variables for PETSC, it is possible to compile the libraries. The used configuration flags were

```
1 ./configure --with-x=0 --with-mpi=1 --with-hypr=1 --download-hypr=1 --with-shared-libraries=1 --with-blas-lapack-dir=/cluster/apps/intel/mkl/10.1.1.019/lib/em64t --with-mpi-dir=/cluster/apps/openmpi/1.4.5/x86_64/gcc_4.1.2/lib
```

for PETSC and

```
1 ./configure --enable-mpi CC=mpicc CXX=mpicxx --enable-shared --disable-threads --with-metis=/cluster/home/math/shaerdi/rte/libraries/metis-4.0.3 --with-blas="mkl_intel_lp64 -Wl,--start-group -lmkl_intel_thread -lmkl_core -Wl,--end-group -lguide -lpthread"
```

for Deal II. Make sure the paths to the mkl blas and the metis library are set correctly. The rest of the procedure is the same, except of course the 'brutus_FLAG' which is now to be set.