Programming of Supercomputers

Final Report

Chris Scheingraber, Mat.-Nr. 3016120

CSE Master Program

Technical University Munich

January 27, 2013

Contents

1 Introduction

1.1 The Project: Optimization and Parallization of a CFD solver

In the Programming of Supercomputers WS 12/13 Lab course, part of the curriculum of the CSE Master program at Technical University of Munich, students were required to individually apply their knowledge about parallelization using the Message Passing Interface (MPI) learned in previous courses, e.g. Parallel Programming.
This report should convey a comprehensive overview of this project and introduce the most important tasks and problems, as well as the ideas and algorithms that were developed to solve them.
In the course of the semester, several consecutive assignments building upon each other had to be handed in to the tutor. This way, students were guided through the Parallelization process.
The project centered around the optimization and parallelization of a given CFD program called "Fire Benchmark", a C code developed by AVL LIST GmbH (Graz, Austria) for two- or three-dimensional steady or unsteady heat and flow simulation. It can work on arbitrary geometries, and it can also deal with moving boundaries.
The Fire Benchmark (GCCG, generalized orthomin solver with diagonal scaling) implements the linearized continuity equation:

$$A_p \varphi_p = \Sigma_{c=E,S,N,...} A_c \varphi_c + S_\varphi$$

In this equation, the given terms are the source value S_{varphi}, the boundary cell and pole coefficients (A_c and A_p, respectively), while the equation has to be solved for the variation vector (flow var to be transported) φ_p.
The domain is discretized using volume cells with an unstructured grid with neighbouring information (using

a link-cell-cell array *LCC*, see Figure 1) and indirect addressing. The computation iterates until an acceptable result is achieved (residual small) or - in case the solver does not converge for some reason - until a previously set maximum number of iterations is reached. The loop consists of two computation phases: phase 1 computes the new directional values from the old ones, while phase 2 normalizes and updates the values and computes the new residual.
The data structures consist of 1D arrays for the cell and pole coefficients, for the source term and for the link-cell-cell array. The computed arrays are the directional vectors direc1, direc2, the variation vector var which keeps the result, and the residual vector resvec. The indirect addressing uses the link-cell-cell array to obtain values of neighbouring cells.

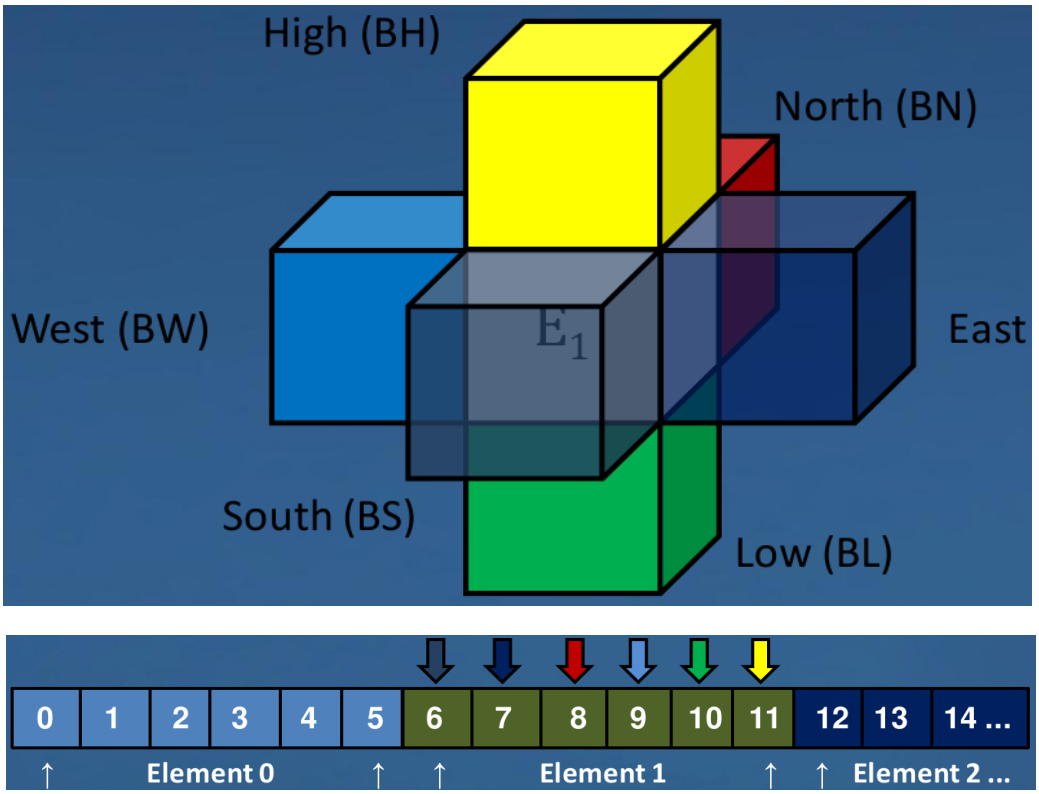

Figure 1: The Link-Cell-Cell (LCC) structure. For each element in the domain, the LCC array holds 6 entries, which are the indices of the neighbouring cells. In the solution provided by the author, a linearized version of this array is scattered among the processes and converted to local indices (including ghost cells).

The first assignment dealt with sequential optimization including extensive benchmark tests using the Performance Application Programming Interface (PAPI). The second assignment, consisting of 4 milestones, was about parallelization using MPI. The individual milestones corresponded to the most important steps of parallelizing a program: data distribution, communication model including ghost cells, parallelization, and performance analysis and tuning.

For both assignments, students were asked to comply to the PoS course code guidelines to improve readability and comparability of the code. The guidelines were based on current industry standards, i.e. the Google and ID software code guidelines.

1.2 Execution Environment

The specifications of the execution environment have been obtained from web resources (LRZ website, 2012) and by using an interactive session on the cluster.

The execution environment used for this lab course is the LRZ Linux Cluster. The segment which was used is the MPP Cluster with 16-way AMD-based nodes and Infiniband interconnect. Like all segments of the LRZ Linux cluster, it has 64 bit address space. Its 2848 processor cores are divided among 178 nodes: each node has two sockets with 8 cores each. The cores have a peak frequency of 2 GHz and can dynamically scale the

frequency to 2.00 GHz, 1.70 GHz, 1.40 GHz, 1.10 GHz and 800 MHz. The cores have a L1 cache size of 128K (64K data cache and 64K instruction cache). The L2 cache is 512K. On each socket, the 8 cores share 5118K L3 cache. The system does not provide hyperthreading, thus the number of threads per core is limited to one.
The peak performance is 0.0079 TFlop/s per processor, and 0.1275 TFlop/s per node. The aggregate peak performance is 22.7 TFlop/s.

The MPP Cluster uses SLURM as batch job system, which has been used for all benchmarks for this report. No interactive sessions were used, since there are only 8 simultaneous interactive sessions available in total. All batch jobs have been scheduled from the login nodes *lxa191* and *lxa192*.

2 Sequential Optimization

In assignment 1 of the course, compiler-based sequential optimization and the effect of input/output formats has been investigated. For the measurements, the Performance Application Programming Interface (Browne *et al.*, 2000) library has been used.

2.1 Derived metrics

These formulas for derived metrics have been obtained from a web resource (National Center for Supercomputing Applications, University of Illinois at Urbana-Champaign, 2012). The level 1 cache miss rate is the ratio of the total cache misses to the total cache accesses:

$$R_{L1CacheMisses} = \frac{PAPI\_L1\_TCM}{PAPI\_L1\_TCA}$$

and analogous for level 2 cache:

$$R_{L2CacheMisses} = \frac{PAPI\_L2\_TCM}{PAPI\_L2\_TCA}$$

The MFlop/s have been obtained from the total floating-point instructions (hardware counter *PAPI_FP_INS*) total CPU cycles (hardware counter *PAPI_TOTAL_CYC*), and the clock frequency:

$$MFlop/s = \frac{PAPI\_FP\_INS}{PAPI\_TOT\_CYC} \cdot freq_{clock}$$

The CPU utilization has been computed via

$$util = \frac{PAPI\_TOT\_CYC}{time_{wall} \cdot freq_{clock}}$$

As clock frequency, the peak frequency per core of 2 GHz has been used.

2.2 Sequential Benchmark Procedure

The sequential code was tested using different compiler optimization levels: *-O0 -g*, *-O1*, *-O2*, *-O3*, *-O3 -opt-prefetch=2*, *-O3 -opt-prefetch=4*. Since not all hardware counters could be accessed at the same time, an optional command line parameter was implemented and documented. For each optimization, each measurement was performed 3 times. To avoid tedious repetition, an automated procedure using a Makefile with several targets for the different optimization flags and a bash script was utilized (cf. Report of Assignment 1).
The achieved floating point performance for the computation phase was, depending on the scenario and input file, in the range of about (450 ± 100) MFlop/s. This is small compared against the peak system performance, which is about 22.7 TFlop/s. However, since at this stage, the code was not parallelized yet, one has to consider the peak performance per core (the peak system performance divided by the number of cores). For the MPP cluster, it is about 7970 MFlop/s.

2.3 Results of the Sequential Optimization

Flop/s, CPU Utilization and Cache Miss Rate A reason that the achieved MFlop/s were about one order of magnitude smaller than the peak performance per core may be non-floating-point operations, since the CPU utilization for the computation phase was very close to 100%. For the input and output phase, the CPU utilization was around 90%.

In general, the miss-rate for both the level 1 and level 2 cache was becoming larger with increasing optimization level and the execution time decreased, most significantly from optimization level 0 to 1 (see Figure 2). However, no significant improvement could be made with prefetching over plain -O3 optimization. In fact, the execution binaries for *-O3*, *-O3 -opt-prefetch=2* and *-O3 -opt-prefetch=4* did not differ as tested with the *diff* program.

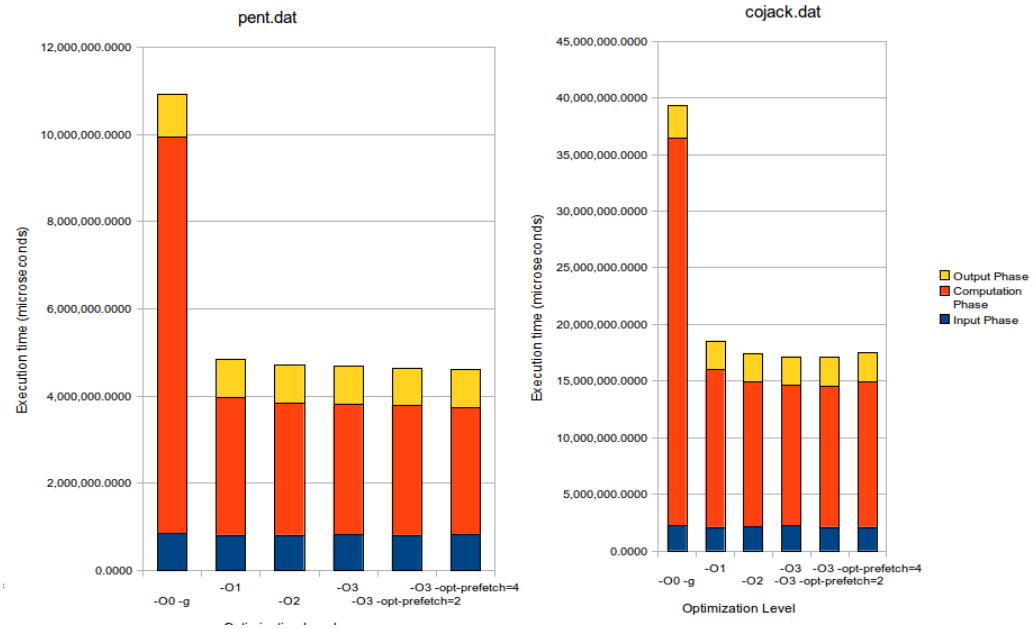

Figure 2: Sequential execution time for different compiler optimization levels.

Text vs Binary Input To investigate the effect of input on performance, a small tool to convert the text input files to a binary format was written. A significant speedup over text input could be achieved when using the binary input. The file size decreased to about 1/3 of original file size for all scenarios. This manifested in a speedup of about 10% for *-O3*.

3 Parallelization of the Fire Benchmark

To achieve further speedup and benefit from the large scale parallel architecture, in the second assignment the Fire Benchmark was parallelized using the Message Passing Interface (MPI). Each sub-assignment (milestone) corresponded to a fundamental step of parallelization.
As briefly indicated in Section 1.1, the Fire Benchmark supports arbitrary unstructured grids with linked neighbour cells. This is implemented via the three arrays *points*, *elems*, and *LCC*.
The *elems* array stores which points belong to a specific element. Since an element consists of 8 points (in the 3-dimensional case), each element consists of 8 consecutive entries in this array. These entries are indices of the *points* array, which stores coordinates of all points used in the discretization. It is later used to write the unstructured grid header for output.
The *LCC* (Link-Cell-Cell) array stores in the neighbouring information. For each element it holds 6 entries, which are the indices of the adjacent cells.

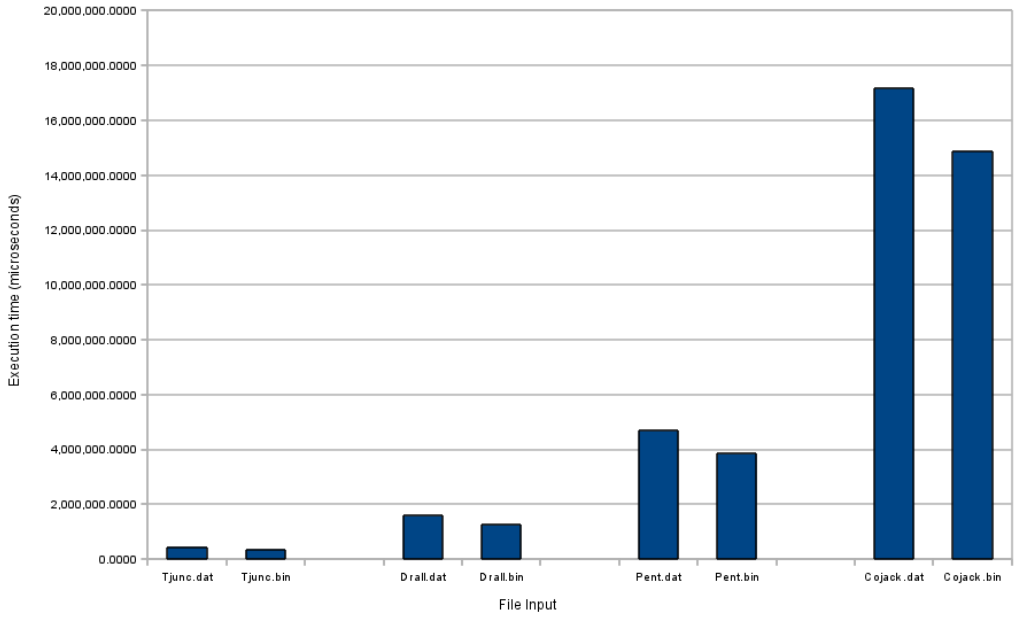

Figure 3: Sequential execution time for binary and text input files for all scenarios for -O3.

3.1 Data Distribution

Considering the implementation of the data distribution, two strategies were to be taken into consideration: letting every process read in all data and keep only the needed parts, or let one process read in all data and distribute it to the other processes.
The first strategy completely avoids delay due to communication during the initialization phase, but has higher memory requirements is prone to slow disk input when all processes try to access the input file simultaneously. The second strategy, chosen by the author, results in less duplicated work being performed, including less computation and less disk access. This manifests in lower energy consumption and lower component wear, therefore less overall cost to execute the simulation.

Implementation Process 0 (henceforth called *root*) reads in the data using the *read_binary_geo* function. It also obtains the distribution of the elements among itself and the other processes. Three strategies were implemented:
Classical Partitioning, which assigns the cells in the way they happen to be located in the elements array (i.e., if p is the total number of processes, the first $1/p$ elements to process 0, the second fraction to process 1, and so on).
Dual partitioning and *nodal partitioning* are implemented using the API of the graph partitioning library METIS (Karypis Lab, 2013) via the functions *METIS_PartMeshDual* and *METIS_PartMeshNodal*. The output of these functions is two vectors which store the partitioning of the mesh. The array *epart* stores the partitioning of the elements. For each index, it stores the rank of the process the respective element has been assigned to. Likewise, the array *npart* stores the partitioning of the nodes.
In order to avoid code replication using both *METIS* and classical partitioning, the classical partitioning produces equivalent *epart* and *npart* vectors:

```
} else if (strcmp(part_type, "classical") == 0) {
    // just set epart and npart to classical values
    for (int i = 0; i < nelems; i++) {
        (*epart)[i] = (int) ((double) num_procs / nelems * i);
    }
    for (int i = 0; i < *points_count; i++) {
        (*npart)[i] = (int) ((double) num_procs / (*points_count) * i);
    }
```

Once the partitioning is obtained, *epart* is broadcasted to all processes. Each process counts how often his rank occurs in this array. This information is then communicated via *MPI_Gather* to the root process, which then creates for all arrays creates buffers sorted with respect to all process ranks (see line 250 of *initialization.c*):

```
int i = 0;
int idx_loc;
for (int p = 0; p < num_procs; p++) {
    idx_loc = 0;
    for (int j = 0; j < nelems; j++) {
        if ((*epart)[j] == p) {
            bp_buffer[i] = bp_global[j];
            su_buffer[i] = su_global[j];
            (...)
            i++;
        }
    }
}
```

These buffers are then scattered among all processes using *MPI_Scatterv*. Each process allocates only the amount of memory necessary for his local domain. The memory holding the global arrays and the buffers can then be deallocated in the root process.
The check whether the data distribution is correct, a test function *test_distribution* is implemented in *test_functions.c*. Called by a single process for which to test the distribution, it writes a VTK file to disk. The VTK shows the whole domain where only local values of the *cgup* array do not vanish. It can be examined using Paraview (Ahrens *et al.*, 2005).

3.2 Communication Model

3.2.1 Index Mappings

The next step in the parallelization is the derivation and implementation of an efficient communication model. Since the domain is divided into smaller local domains, a mapping from local to global indices and vice versa is created. This is necessary to correctly communicate with other processes.

Implementation The index mappings, called *local_global_index* and *global_local_index*, are created in the same loop as the buffers (for context: cf. the loop above in Section 3.1, or line 250 of *initialization.c*):

```
local_global_index_buffer[i] = j;
(*global_local_index)[j] = idx_loc++;
for (int n = 0; n < 6; n++) {
    lcc_buffer[i * 6 + n] = lcc_global[j][n];
}
```

While *global_local_index* is fully broadcasted to all processes, *local_global_index_buffer* and *lcc_buffer* are scattered in parts matching the local domain size.

3.2.2 Communication Lists, Ghost Cells, and Local LCC

Consider a cell at the boundary between two subdomains. For the computation of a value for this cell, data from adjacent cells is often necessary. Since at least one of the linked neighbour cells belongs to another process, data has to be exchanged between these two processes. In order to do so, communication lists are created on both processes. The list of cells to send from process A to process B (*send_list*) has to match the list of cells to receive at process B from process A (*recv_list*).
The cells that need to be retrieved from another process are commonly referred to as *ghost cells*, while cells outside the global computational domain are called *external cells*.

Implementation To create the communication lists, root first counts the number of cells to communicate between all pairs of processes and scatters this information. After all local *send_list* and *recv_list* arrays have been allocated, each process loops over his *LCC* and sets up the *recv_list* of cells to receive from neighbouring processes.

At the same time, he updates his *LCC* to local indices. For inner cells, *global_local_index* is used. The organisation of cells in the local array (*direc1*, see *compute_solution.c*) is: inner cells, external cell, ghost cells. All necessary ghost cells are therefore just appended to the local array. Therefore, in the local *LCC*, the respective indices of ghost cells are translated to the local indices at the end of the array (for details, cf. line 405 et seqq. of *initialization.c*). If an external cell is found, it is updated to the local index of the external cell, which is located between the internal cells and the ghost cells.
The check whether the communication lists are correct, another test function, *test_communication*, is implemented in *test_functions.c*. Its output is again a VTK file to be examined in Paraview, which has value 5 for ghost cells to be received, value 10 for cells to be sent, and value 15 for other internal cells.

3.3 Parallelization of the Computation

The main computational loop of the Fire Benchmark consists of two phases: phase 1 involves computing the new directional vectors *direc1* and *direc2* for all cells, while phase 2 executes the normalization steps. The loop is repeated until convergence ($residual\_ratio \leq 10^{-10}$) or until the maximum number of iterations is reached.
During phase 1, indirect addressing (LCC) is used to access the neighbour cells of *direc1*. Each cell has coefficients for south, north, west, east, low and high boundaries as well as a pole coefficient. These boundary coefficients are used to weight the contribution of each neighbour during the computation, the pole coefficient determines the contribution of the cell itself. During each iteration of the computation phase, it is necessary to frequently exchange values between processes; specifically before ghost cells of *direc1* need to accessed, or when global residuals need to be computed, or during normalization steps.

Implementation The parallel computation is implemented in *compute_solution.c* using a send buffer *direc1_buf* and *MPI_Scatterv* to directly place the ghost cells at the end of the respective process' *direc1*:

```
// update the old values of direc
for (i = nintci; i <= nintcf; i++) {
    direc1[i] = direc1[i] + resvec[i] * cgup[i];
}
// set up send buffer
for (int i = 0; i < send_total; i++) {
    direc1_buf[i] = direc1[global_local_index[send_list[i]]];
}
// send/recv direc1
for (int p = 0; p < num_procs; p++) {
    MPI_Scatterv(direc1_buf, send_count, send_idx, MPI_DOUBLE,
                 &(direc1[nintcf + 2 + recv_idx[p]]), recv_count[p], MPI_DOUBLE, p,
                 MPI_COMM_WORLD);
}
for (i = nintci; i <= nintcf; i++) {
    direc2[i] = bp[i] * direc1[i] - bs[i] * direc1[lcc[i * 6 + 0]]
              - bw[i] * direc1[lcc[i * 6 + 3]] - bl[i] * direc1[lcc[i * 6 + 4]]
              - bn[i] * direc1[lcc[i * 6 + 2]] - be[i] * direc1[lcc[i * 6 + 1]]
              - bh[i] * direc1[lcc[i * 6 + 5]];
}
```

In addition, *MPI_Allreduce* is used in several places, to sum up a variable among all processes, e.g. *resref*, *occ*, *cnorm*, *omega*, or *res_updated*.

4 Performance Analysis and Tuning

After the code was fully parallelized, it was sensible to analyse the effect of the efforts, i.e. compute the speedup that was obtained over the sequential runtime as well as identification and ideally exploitation of performance bottlenecks and tuning possibilities.

4.1 Code Instrumentation

For performance analysis, the code was first manually instrumented with the help of the *Score-P* User API (Scalable Performance Measurement Infrastructure for Parallel Codes, `http://www.score-p.org/`). To

graphically interpret the performance profile obtained with *Score-P*, the program *CUBE* was used. It is part of Scalasca (`http://www.scalasca.org/`), a tool to support performance optimization. In addition, the scalable automatic performance analysis tool *Periscope* was used. *Periscope* allows for automatic search of bottlenecks without any tracing by the user. For evaluation of the results, the *Periscope GUI for Eclipse* was used.
To get more precise results in conjunction with Periscope, the Score-P instrumentation included an online access user region and multiple nested phases, e.g. in the computational loop:

```
// user regions definitions
SCOREP_USER_REGION_DEFINE(OA_Phase);
SCOREP_USER_REGION_DEFINE(phase_1);
SCOREP_USER_REGION_DEFINE(phase_2);
SCOREP_USER_REGION_DEFINE(phase_2_norm);

while ( iter < max_iters ) {
    // online access phase for periscope
    SCOREP_USER_OA_PHASE_BEGIN(OA, "OA_Phase", SCOREP_USER_REGION_TYPE_COMMON);
    SCOREP_USER_REGION_BEGIN(phase_1, "phase_1", SCOREP_USER_REGION_TYPE_COMMON);
    [computation of phase 1...]
    SCOREP_USER_REGION_END(phase_1);
    SCOREP_USER_REGION_BEGIN(phase_2, "phase_2", SCOREP_USER_REGION_TYPE_COMMON);
    SCOREP_USER_REGION_BEGIN(phase_2_norm, "phase_2_norm", SCOREP_USER_REGION_TYPE_COMMO
    [norm computation of phase 2...]
    SCOREP_USER_REGION_END(phase_2_norm);
    [more phases...]
    SCOREP_USER_REGION_END(phase_2);
    SCOREP_USER_OA_PHASE_END(OA);
}
```

This way, it was possible to trace the MPI communication hotspots to the RFL (region first line) of relatively small phases.

4.2 Scalability Analysis

For scalability analysis, the results of the profiling were analyzed using *CUBE*. The speedup of the code over the sequential version (optimized with *-O3*) was computed for all phases individually.
As expected, only the computation phase showed significant speedup. Naturally, for both scenarios (*Pent* and *Cojack*), the magnitude of the speedup was strongly dependent on the domain partition strategy. For *classical partitioning*, the immense communication overhead resulted in only moderate speedup for the computation phase, peaking at around 3.3 for *Pent* with 32 processors, and around 3.7 for *Cojack* with 48 processors (see Figure 4).
For *dual partitioning*, the parallelization yields much better speedup for the computation phase. For *Pent*, the speedup peaked at 27 for about 48 processes, while for *Cojack*, this peak was 40 for the same number of processes. The higher speedup of the code for *Cojack* is due to the higher number of cells in this scenario. This causes a higher ratio of internal cells to ghost cells, therefore less communication overhead and higher maximum speedup.
The relative slowdown of the initialization phase dulls the great results of the computation phase. As shown in Figure 4, the step from one processor to two processors is most significant. The plot shows the *Pent* scenario, but the graph looks very similar for *Cojack*. Going from 2 processors until 64 processors, the further slowdown is moderate.
The sudden overhead caused by the parallelization using only 2 processors suggests that a high percentage of it is caused by the mere necessity of communication. It is also apparent that the utilization of *METIS* causes a lot of overhead, which was confirmed by using *CUBE* (see Figure 6).
The time spent in the finalization phase was roughly constant. This is especially interesting since the root process has to gather the data from all other processes before writing the VTK file to disk. Since this is a similar procedure as in the initialization phase (only reversed), this suggests, backed by the data mentioned in the paragraph above, that a big bottleneck of the initialization phase is indeed the utilization of the sequential version of *METIS*.

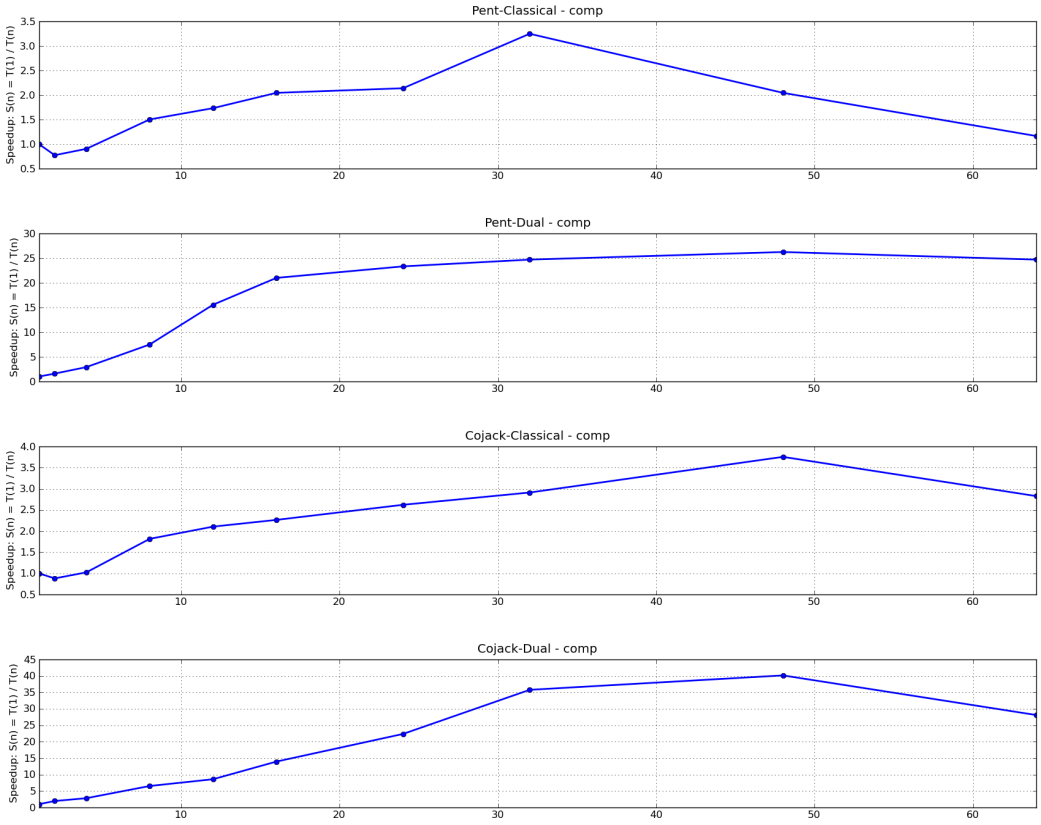

Figure 4: Plot of the speedup of the computation phase for the *pent* and *cojack* scenarios using *classical* and *dual* partitioning until 64 processors. These are the results using compiler-based optimization with *-O3*. For *-O0*, the speedup over the sequential version was significantly higher than shown here.

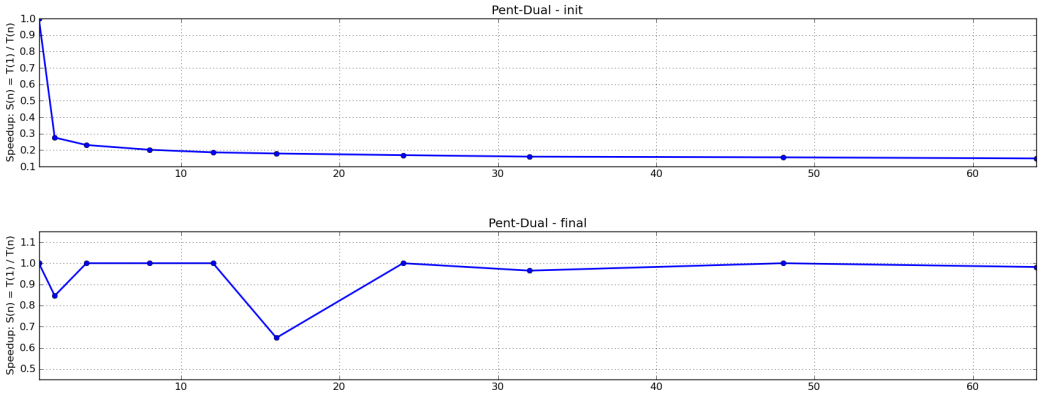

Figure 5: Exemplary plots of the speedup for the initialization and the finalization phases for the *pent* scenario using *dual* partitioning until 64 processors. This shows that while the time spent in the finalization phase is roughly independent of the number of processes, the initialization phase is not and slows down significantly. For the *cojack* scenario, the speedup diagrams look likewise.

Communication to computation ratio Depending on the number of processes and the size of the scenario, the time spend in communication can be a significant fraction of the total time. For *pent* using *dual partitioning* and 4 processors, the communication made up for 33.97% of the overall runtime.
This is slightly higher than the set goal of 25%. The most significant reason for this lies in the choice of letting process 0 take care of the entire initialization including calling the *METIS* partitioning function, while the other processes wait idle. This is particularly caused by invoking *METIS* only on root, since this takes some time and results in multiple *MPI_Wait* operations. However, this does necessarily mean that the chosen approach is slower in terms of overall execution time, but just that the amount of communication is comparatively high. Therefore, just analyzing the communication to computation ratio on its own may only be of limited value. More insight is gained by analyzing the speedup and bottlenecks of the individual phases.

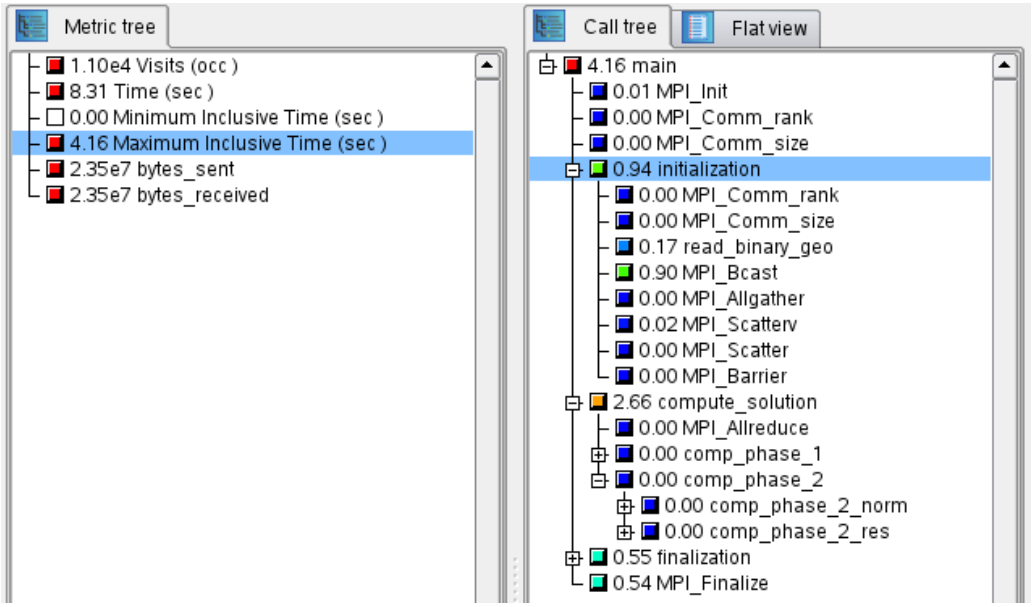

Figure 6: Using CUBE and multiple user regions to identify bottlenecks in the code.

4.3 Identified Performance Bottlenecks

Using Periscope, it was possible to automatically identify slow regions with increased MPI communication. Since most time is spent in *compute_solution.c*, this is where optimization possibilities were especially examined. It paid out to use multiple user phases. This way, the bottlenecks with a high severity could be tied closer to the actual code statement, since Periscope gave the RFL (region first line) of the respective phases:

```
169         SCOREP_USER_REGION_END(comp_phase_2_norm);
170         SCOREP_USER_REGION_BEGIN(comp_phase_2_res, "comp_phase_2_res", SCOREP_USER_REGION_TYPE_COMMON);
171         // compute the new residual
172         cnorm[nor] = 0;
```

Periscope Properties View | Clustering Results View | Console | Periscope Table View

Name	Filename	RFL	Severity	Confide	Region	Proce	Threa
Excessive MPI communication time in MPI_SCATTER	compute_solution.c	83	9.83	1.00	Mpi call	4	0
Excessive MPI communication time in MPI_SCATTER	compute_solution.c	83	8.23	1.00	Mpi call	5	0
Excessive MPI communication time in MPI_SCATTER	compute_solution.c	83	6.56	1.00	Mpi call	1	0
Excessive MPI communication time in MPI_ALLREDUCE	compute_solution.c	170	6.74	1.00	Mpi call	1	0
Excessive MPI time due to late process in allreduce	compute_solution.c	170	5.50	1.00	Mpi call	0	0
Excessive MPI communication time in MPI_ALLREDUCE	compute_solution.c	170	5.93	1.00	Mpi call	4	0

Figure 7: Automatically identified performance bottlenecks using Periscope. For 4 processes, the *MPI_Allreduce* calls of phase 2 seemed to have higher severity, while for 8 and 64 processes, the *MPI_Scatterv* calls of phase 1 were more severe. This table shows the data for the *pent* scenario using 8 processes and *dual partitioning*.

Interestingly, the *MPI_Allreduce* calls of phase 2 had a high severity for 4 processes, while the *MPI_Scatterv* calls of phase 1 were most severe for a higher number of processes. Instead of using a buffer array and *MPI_Scatterv*, another approach would have been to use MPI_Type_indexed to package arrays together. This provides no fundamental advantage, though, since the communication call would have similar weight, and the slowdown due to cache-inefficient distribution of cells that need to be send would be the same.
A possible modification that brings some optimization potential would therefore be to sort all cells that need to be sent into consecutive locations in the local arrays, and adjust the *lcc* arrays accordingly. However, this does not increase the speed by orders of magnitude, since the communication is in comparison much slower. It turned out to be hard to achieve further speedup of the computation without major refactoring of the code or using a different method, since the communication of the ghost cells and the residual is necessary to maintain a correct computation.

5 Overview

This project provided very valuable insights into parallelization potential. The consecutive assignments and milestones were encouraging and helped to apply the principle of divide and conquer.
It can be said that the parallelization approach suggested in this assignment is not harvesting the full potential of a massively parallel architecture. Without parallel input/output, the initialization phase is bound to limit the possible speedup, since it runs, in one way or the other, quasi-sequential. As Amdahl's law states, if P is the parallelizable fraction of the algorithm, the possible speedup is ultimately limited by the inherently sequential fraction S:

$$Speedup \leq \frac{1}{(1-P) + \frac{P}{S}}$$

In the algorithm as described in Section 3, the initialization phase contains mostly sequential parts. In addition, the strategy to let the root process read all the data and distribute it accordingly to others has proven to be less than ideal. It might be more energy efficient and elegant, but it causes the other processes to wait idle. These factors cause the initialization overhead to impede high speedup.
For unsteady problems, this is probably be a less important issue, since the speedup of the computation phase can outweigh the overhead in the initialization phase for a long simulation. However, for a steady flow computation, to achieve close-to-linear speedup for the whole program, it seems to be crucial to use parallel input/output techniques and a mesh partitioning library designed for parallel usage (i.e. ParMETIS - Parallel Graph Partitioning and Fill-reducing Matrix Ordering).

Bibliography

Lecture material and notes of the PoS12/13 course.

J. Ahrens, B. Geveci and C. Law. 2005. Paraview: An end user tool for large data visualization. *the Visualization Handbook. Edited by CD Hansen and CR Johnson. Elsevier*.

S. Browne, J. Dongarra, N. Garner, G. Ho and P. Mucci. 2000. A portable programming interface for performance evaluation on modern processors. *International Journal of High Performance Computing Applications*, **14**, 189–204.

Karypis Lab. METIS - Serial Graph Partitioning and Fill-reducing Matrix Ordering. `http://glaros.dtc.umn.edu/gkhome/views/metis`, January 17th, 2013.

LRZ website. Overview of the Cluster Configuration. `http://www.lrz.de/services/compute/linux-cluster/overview/`, November 18th, 2012.

National Center for Supercomputing Applications, University of Illinois at Urbana-Champaign. PerfSuite Default PAPI-Derived Metrics. `http://perfsuite.ncsa.illinois.edu/psprocess/metrics.shtml`, November 18th, 2012.