Garching, 26.01.2015

Group 5

Yunqing Dong

Mariusz Bujny

Programming of Supercomputers
Final Report

A1. Single-core optimization

1. **Observations**

   In the first assignment we were required to perform sequential optimization and compare the performance for manipulating data in different formats.

2. **Sequential optimization**

   In sequential optimization we tried to compile the code with different compiler flags and with/without automatic vectorization. With compiler flag -O3 we have achieved the best performance (shortest processing time). It reduced the L2/L3 cache miss rate, as well.

   By using vectorization, we tried to utilize the full length of register in order to process more data for the same instruction in one cycle. For scientific computing, which involves a lot of floating point operations, it really improves performance a lot. However, we should be aware which registers and which SSE instruction set we are using, in order to use an appropriate compiler flag.

   In order to understand the benefits of vectorization, we compiled the code with objdump -d to check which lines were vectorized.

   According to our tests with different combination of flags and/or vectorization, it turns out that we could reach 69% peak performance while executing tjunck.dat. However, this is dependent on the input file.

3. **I/O Performance**

   Manipulating data and reading/writing data can be very time-consuming. The most popular data types people are using are binary files and ASCII files. In ASCII files each character is coded as a single byte, therefore in many cases much more storage is needed to code a text file than a binary file. In our test case the size of file could be reduced as high as 65% of the original size by converting from ASCII to binary format.
   When reading the file, the advantage of binary files is even more obvious. The reading time for binary file is normally only 2%-3% of the time for the same file in ASCII format. That is because for ASCII format we need to internally convert the file into binary format before writing to memory. On the other hand, for binary format, such time-consuming process is not required.

   In general it is always recommended to write/read data in binary format for scientific computation. This is mainly due to the fact that the data size could be very big and very often we need to write data frequently to files.

A2.1. Data distribution

In this assignment we were required to implement three domain distribution methods with reading data by one processor and by all processors. The data from input file is distributed among these processors and later on each processor calculates one part of the domain. In this sense we speed up the calculation by involving more processors to share the workload.

During the implementation we have confronted two main issues. For domain distribution methods, nodal and dual can be done relatively easily by using metis library. But for classic method, we did not really have clear instructions how to distribute the data and how to deal with the ghost cells. In the end we distribute the internal cells as evenly as possible and external cells according to the distribution of the inner cells.

In order to make the code well-structured, we followed the same paradigms both for classic and metis methods. Namely, we created the same data structures in both cases, what turned out to be very helpful in the later milestones.

Comparing oneread and allread methods, we put much more effort into oneread case since we had to take care of the data communication. In this case, special care had to be put into the order of sending and receiving data.

The biggest problem that we were facing in this milestone was the memory allocation. We believe there was a problem of compatibility of compilers on Supermuc and on our local computers. Since at the beginning we compiled the code and run on our local machines, it caused several problems later on. Namely, although we had correct results and no error messages on our local machines, when we tried to run the code on Supermuc for the first time, we were getting the segmentation fault error. This was due to adding many new pointers and handling memory allocation, what was not an easy task. What has to be mentioned is that the places we are freeing and allocating memory are different. So this requires us to treat them really very carefully. It took us a very long time to check and fix the memory allocation bugs. From now on, we were testing our codes directly on Supermuc.

A2.2. Communication model

In this assignment we were required to prepare the send list and the receive list for each of the processes for handling the communication with neighboring processes. Since we distribute the whole domain among different processors, the exchange of information between neighboring cells is needed. Therefore we prepare the send list and receive list that hold the indexes of the cells to be exchanged with the corresponding neighbors.

We were required to avoid an exchange of the same cells several times between neighboring processors. Therefore, we thought that it would a good solution to use the “epart” array, which contains the information about to which processor a particular cell belongs to. In the initialization phase we loop over the cells and generate the sendlist and receive list accordingly.

Generation of the send and receive list is coupled with the domain distribution. Especially in oneread case we need to pay attention to the data communication, as well. It is also important to make sure that all of the required data is communicated.

A2.3. Main loop parallelization

The main goal of the assignment A2.2 was to finalize the parallelization of the Fire benchmark. Since in the previous milestones we already prepared the whole framework for efficient communication of the necessary data, the task was just to parallelize the code in compute_solution.c and finalization.c.

In order to parallelize the compute_solution function properly, we had to find all parts of the code, where some data might need to be communicated. In order to apply the stop condition for the solver, the global residue sum has to be compared with the required value. Therefore, at several stages of the computation, we need to communicate the residual values. For that purpose we use MPI_Allreduce as an efficient MPI collective

operation that makes the final results available for all of the processes. This included communication of the following variables (arrays): resref, occ, cnorm, omega, res_updated.

Before the actual computation in the while loop starts, we create an array of variables of the MPI_Datatype type. MPI derived datatypes in this case provide an efficient method of communicating non-contiguous or mixed types in a message. In our case, this idea can be easily employed due to the prior preparation of global_local_index and send_lst (for calculation of displacements) as well as recv_cnt and send_cnt. For each of the processes we use MPI_Type_indexed and MPI_Type_commit to create an array indextype that we use within the while loop to send those values of direc1 that need to be communicated to the other processes.

For communication of direc1, we decided to use coupled non-blocking MPI_Isend and blocking MPI_Recv functions. During the measurements in the A2.4 milestone, this solution turned out to be optimal in terms of performance, as well.

The second stage of the work involved parallelization of the finalization function. Since the requirement of the milestone was to return exactly the same summary file irrespective of the number of processors used for communication (particularly comparison between serial and parallel version), the framework had to be adapted accordingly.

We separated the finalization function into 2 parts: executed if a serial version is running (where we placed the original code of finalization) and executed when a parallel version is used (here we needed to adapt the code to handle writing just a single file).

The adaption of the code of finalization to handle the parallel execution involved use of the MPI_Reduce for the num_internal_cells (to collect the total number of cells in the simulation), and also MPI_Isend executions for nintcf, var and local_global_index. All of the messages were gathered by the process number 0 and written to a single file. In this process we needed to make use of the mapping form local to the global system in order to collect the var values in the same order as in the serial version. Next, the arrays were passed to the stor_simulation_stats function the same way as for the serial version, resulting in the identical output files. This has proven the correctness of our program after the parallelization.

A2.4. Performance tuning

1. Performance analysis

1.1. Introduction

In order to perform the performance analysis and code optimization, two main methods were used. Firstly, to the original version of the code, we added PAPI function calls to measure the process and real time of the program execution in the computation phase. The measurements with PAPI can be easily turned on/off by uncommenting/commenting “#define PAPI” preprocessor directive.

Secondly, “#define SCOREP” preprocessor directive was added, as well. In the corresponding “#ifdef” sections, SCOREP regions for manual instrumentation were introduced. In the main function we defined the following regions: INITIALIZATION, COMPUTATION, FINALIZATION. This way we can measure the MPI overhead in each of those 3 phases of the program run as specified in the worksheet. The main reason to introduce the additional regions was to monitor not only the initialization, compute_solution and finalization functions, but also the other parts of the code in the main function. Moreover, in the compute_solution function, several regions were defined to detect the most computationally costly parts of the code.

The framework that we used for the performance analysis is the following:

- Measurements of the input scalability – instrumentation of the initialization function with PAPI.
  For both drall and cojack input geometries we tested the influence of the input algorithms on the initialization execution time.

- Measurements of the compute_solution function execution time with PAPI.
  Since the instrumentation just with PAPI causes much less overhead than the automatic/manual instrumentation with SCOREP, we treated it as the main tool for execution time (speedup) measurements (calculations).
- Profiling with SCOREP and analysis with CUBE.
  For that purpose we used the automatic SCOREP instrumentation and just compiled the program with the “scorep” flag.
- Manual tracing with SCOREP and analysis with VAMPIR.
  This time we included several (nested) SCOREP regions to detect precisely the regions consuming the most of the computational resources. This time we had to compile the program with the “scorep –user” flag. In order to monitor the number of the floating point operations, we had also to set the environmental variable: “export SCOREP_METRIC_PAPI=PAPI_DP_OPS”.

1.2. Performance analysis of the original version

In order to start the optimization process, we needed to carry out a detailed analysis of the code. For that purpose, we followed the methodology described in the previous section.

First, the scalability of the input algorithms was investigated. Below the results for drall and cojack geometries are presented.

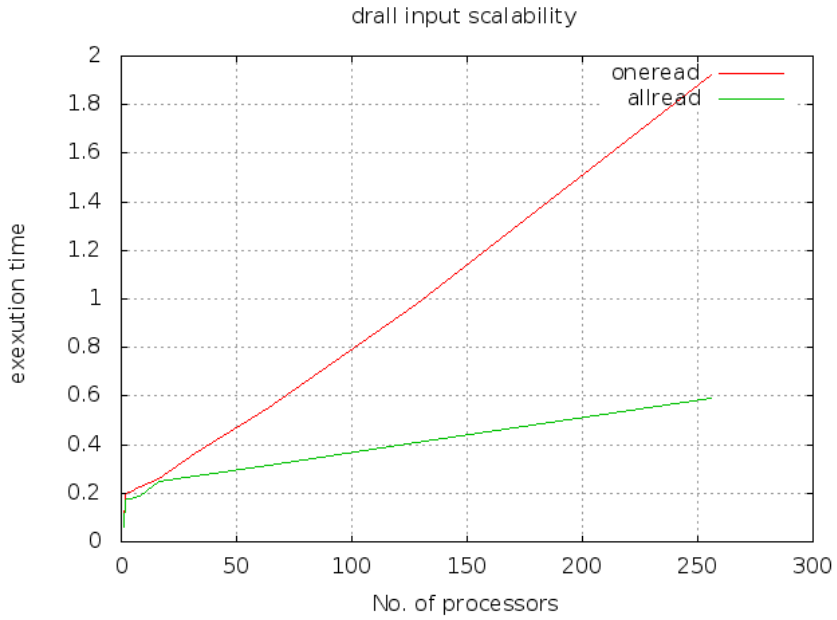

Fig. 1. Execution time in the initialization phase for drall.geo.bin geometry and dual partitioning.

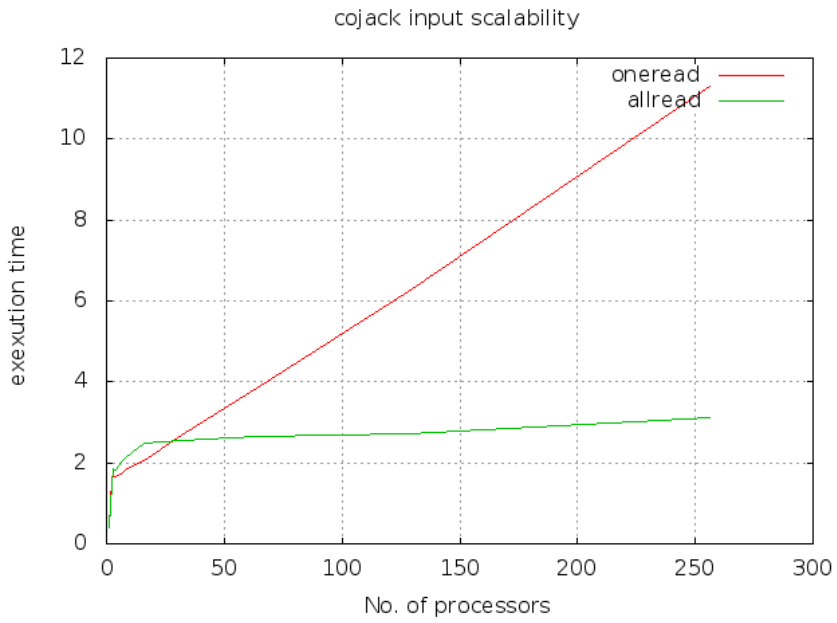

Fig. 2. Execution time in the initialization phase for cojack.geo.bin geometry and dual partitioning.

As one could expect, the execution time of the initialization function is higher for oneread strategy and grows significantly with the number of processors. This is mainly due to the overhead produced by MPI while exchanging the input data by the zeroth process (reading the files) and the rest of the processes (receiving the data). Nevertheless, also for the allread strategy we observe an increase in the reading time as the number of processors increases. This might be caused by the concurrent access to the input geometry file (one should also note that the performance drop is bigger in case of a smaller geometry, what is consistent with this reasoning) and could be possibly easily improved by providing multiple files with the same geometry.

Below we present the data exchanged during the initialization phase for the dual input algorithm (obtained from the profiling data and analysis with CUBE):

	allread [B]				**oneread [B]**			
geometry / cores	**1**	**2**	**4**	**8**	**1**	**2**	**4**	**8**
drall	0	0	0	0	36	2920000	4660000	6300000
cojack	0	0	0	0	36	16900000	28700000	38400000

As one could expected, there is no information exchange for the allread approach. On the other hand, we have a lot of information exchanged for the oneread approach and it grows significantly with the number of processors involved in the communication process.

In the next step, we gathered the execution times of the compute_solution part for pent and cojack geometries and calculated the corresponding speedups:

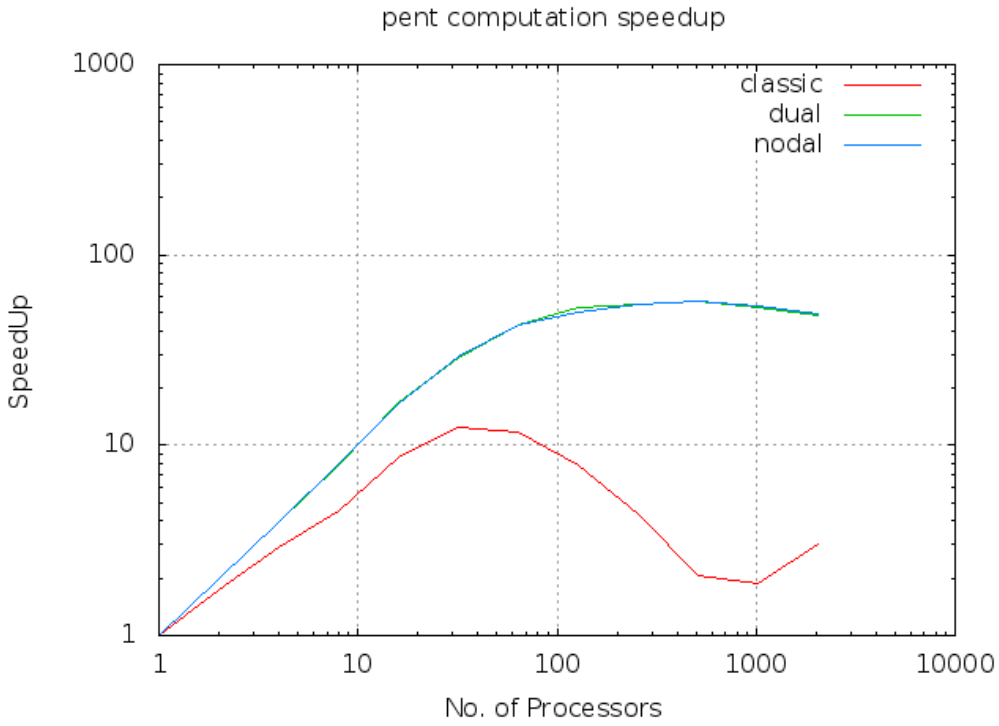

Fig. 3. Speedup in the computational phase for pent.geo.bin geometry, dual partitioning and allread strategy.

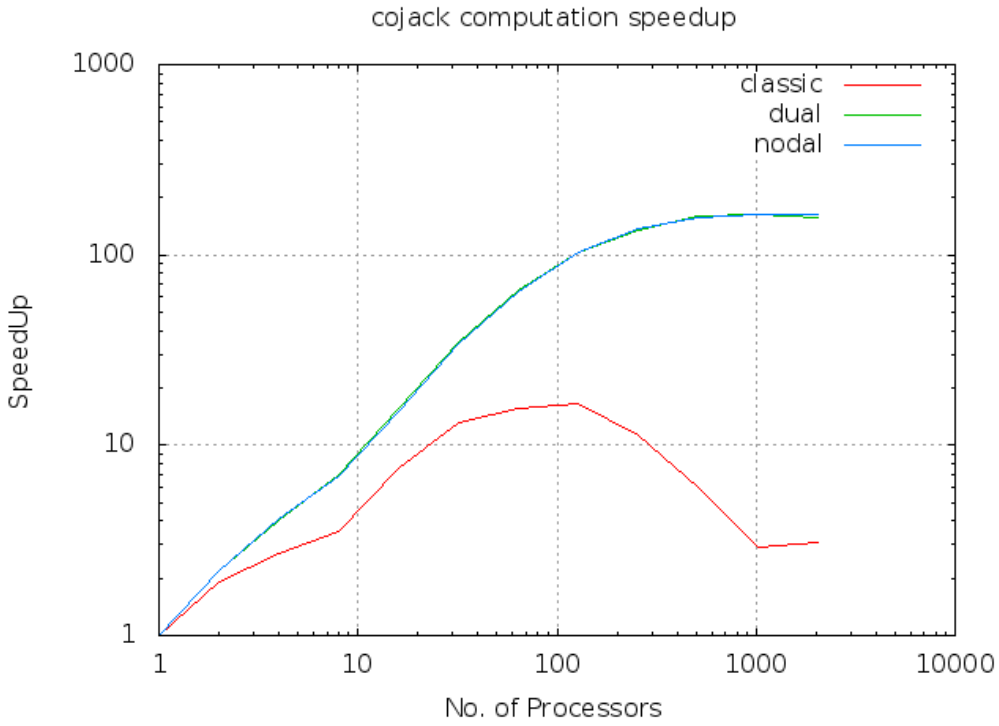

Fig. 4. Speedup in the computational phase for cojack.geo.bin geometry, dual partitioning and allread strategy.

It turns out that the minimal requirement for the milestone A2.4. is already satisfied by the initial version of our code. Actually, we get a linear speedup up to 16 cores for the pent geometry and the dual strategy. As mentioned before, already at the beginning of the parallelization of the compute_solution function we tried to write an optimal code. We used efficient collective operations,

non-blocking communication and derived datatypes. Therefore, such result might have been expected. Nevertheless, we still tried to improve the performance of our code.

The next step of the performance analysis was to compare the MPI overhead in the three execution phases and for the pent geometry:

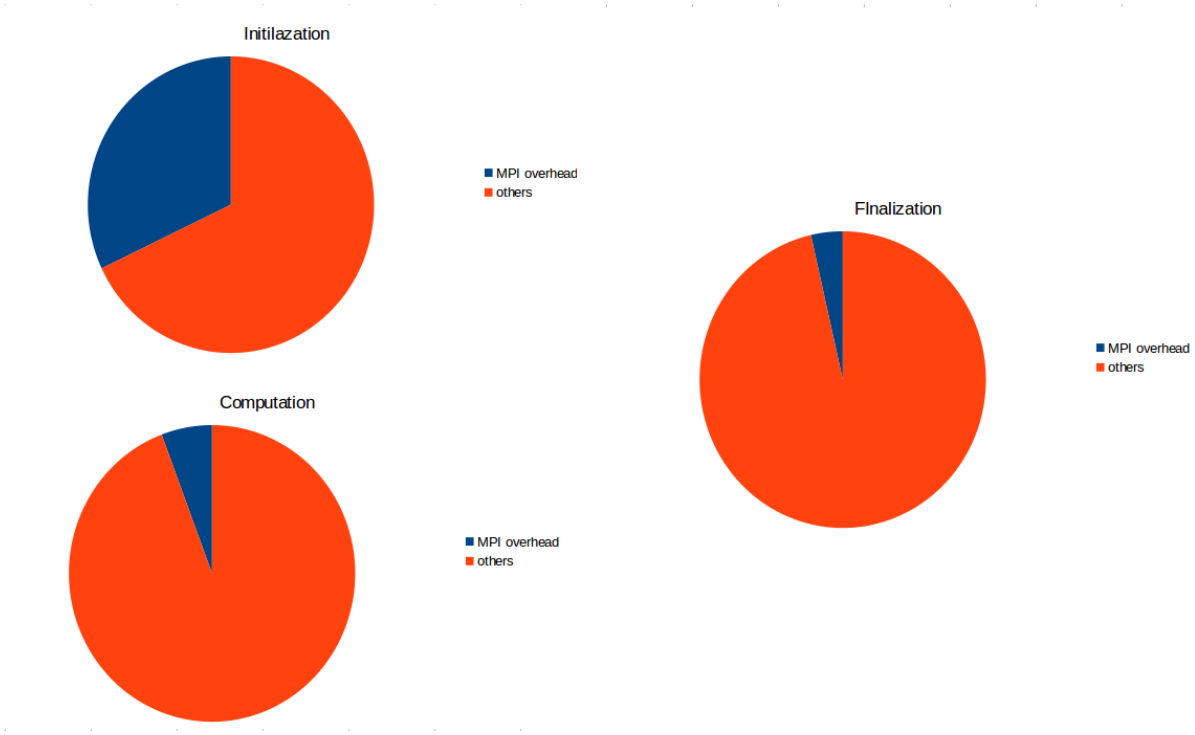

Fig. 5. MPI overhead of the three execution phases: initialization, computation and finalization, for the pent file on 8 processes, dual distribution and oneread strategy. Others – the other computations of the program.

As expected, the overhead due to the MPI communication is highest for the initialization phase – for oneread, we have to decompose the whole domain, what involves a lot of communication.

In the next step we analyze in detail the computation phase of our program:

compute_solution	0.256 s
handle9 - Computation phase2. residual_ratio computation - before break	54.190 ms
handle9 - Computation phase2. residual_ratio computation - after break	20.516 ms
handle8 - Computation phase2. occ computation	27.973 ms
handle7 - Computation phase1. direc2 computation	93.445 ms
handle6 - Computation phase1. direc1 communication	34.649 ms
handle5 - Computation phase1. direc1 update.	37.644 µs
handle4 - Memory allocation.	157.353 µs
handle3 - Calculation of the residue sum.	3.352 µs
handle2 - 1st Allreduce.	33.384 µs
handle1 - Initialization of variables and reference residuals.	47.213 µs

Fig. 6. Analysis of a single computational iteration. MPI overhead, execution steps mapping to the corresponding source code operations. Floating point operations distribution.

From the analysis performed above we can conclude that the most time is consumed by direc2 computation, residual_ratio computation and direc1 communication. Also by looking at the statistics of the floating point operations we can think of overlapping computation and communication as about an efficient method of optimizing the code.

1.3. Performance analysis of the final (optimized) version

Below we present the speedup charts for pent and cojack geometries for the optimized version of the code:

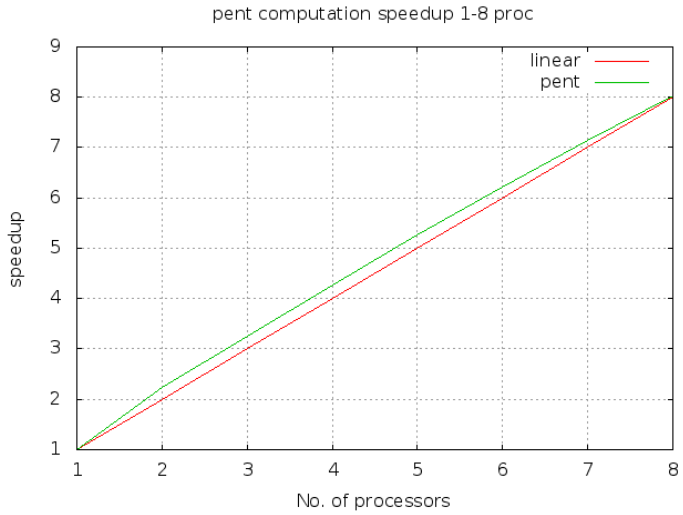

Fig. 7. Speedup in the computational phase for pent.geo.bin geometry, dual partitioning and allread strategy.

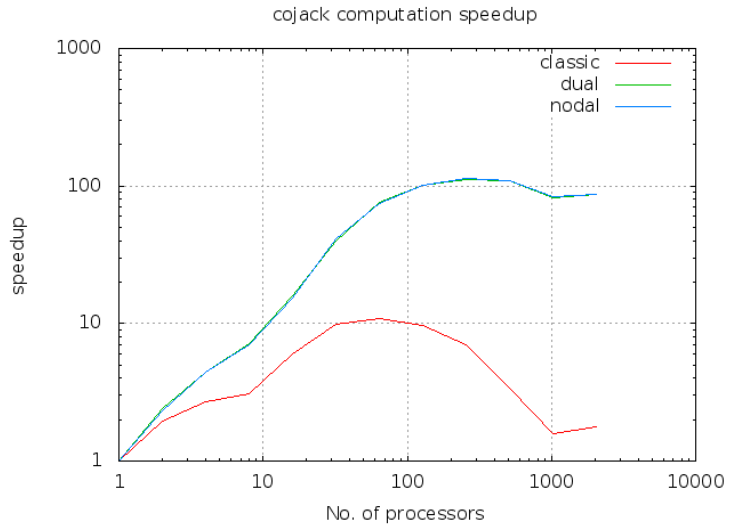

Fig. 8. Speedup in the computational phase for cojack.geo.bin geometry, dual partitioning and allread strategy.

One can notice a drop of parallel performance of the program (at least in certain regions). Nevertheless, the program still satisfies the minimal speedup performance of the milestone. On the other hand, the graph presented below explains why such effect occurs:

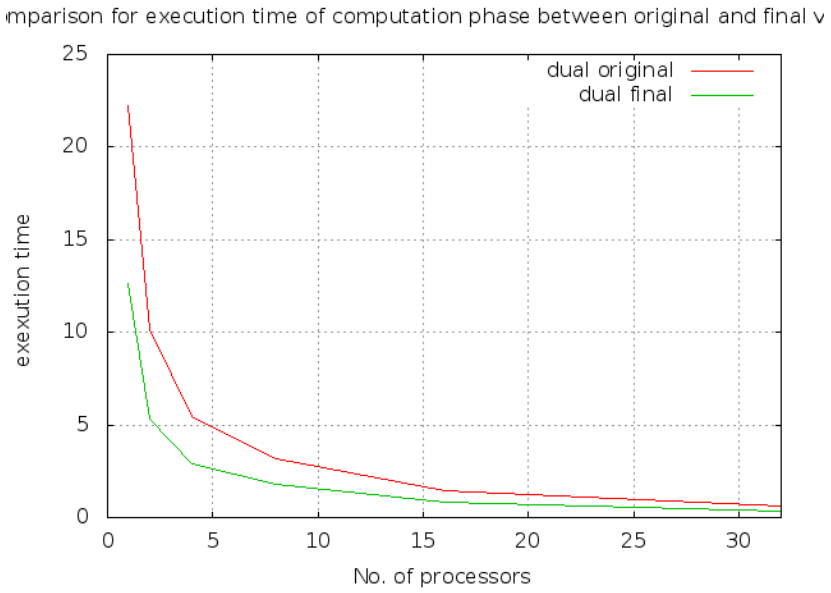

Fig. 9. Comparison of the execution time of the computational phase for the original and final (optimized) versions of the code. All tests carried out with cojack.geo.bin geometry, dual partitioning and allread strategy.

It is clear that the improvement in the time of serial as well as parallel execution for the final version is significant. If the time of serial execution is much shorter, only a small (even random) change in the runtime results in a big speedup difference. Therefore, the results might be affected much more both by some random variations of the execution time as well as the limits of the MPI communication.

Below we present the MPI overhead for three different phases for pent geometry:

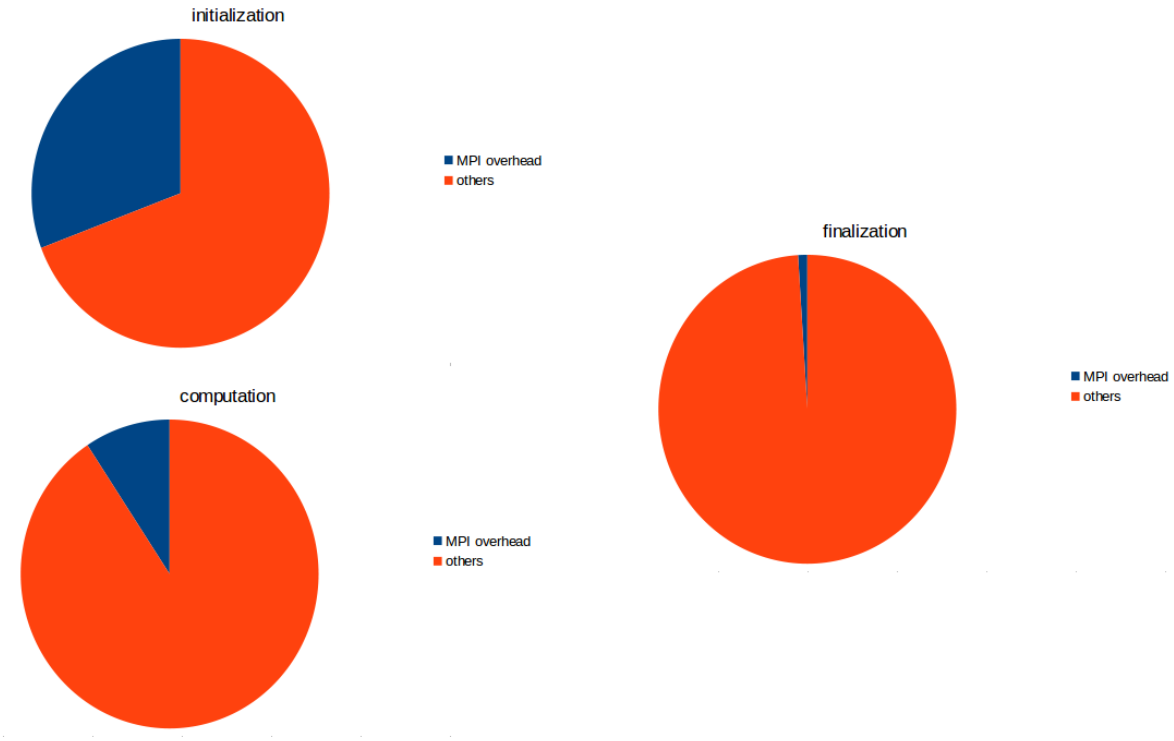

Fig. 10. MPI overhead of the three execution phases: initialization, computation and finalization, for the pent file on 8 processes, dual distribution and oneread strategy. Others – the other computations of the program.

Comparison of the MPI overhead for original and optimized version of the code for the pent geometry, dual distribution and oneread strategy:

original	**total time [s]**	**MPI overhead [s]**	**others [s]**
initialization	19,82	5,4	14,42
computation	7,06	0,87	6,19
finalization	0,08	0,001	0,079
final	**total time [s]**	**MPI overhead [s]**	**others [s]**
initialization	14,57	4,39	10,18
computation	3,62	0,81	2,81
finalization	0,03	0,0005	0,0295

From the pie charts we can see that MPI overhead in the computation phase increased compared to the original code. Nevertheless, the table above shows that it is due to both optimization of communication and computation. We can clearly see that the main factor limiting the further code optimization is the communication overhead.

Again, a detailed analysis of the single computational iteration was performed:

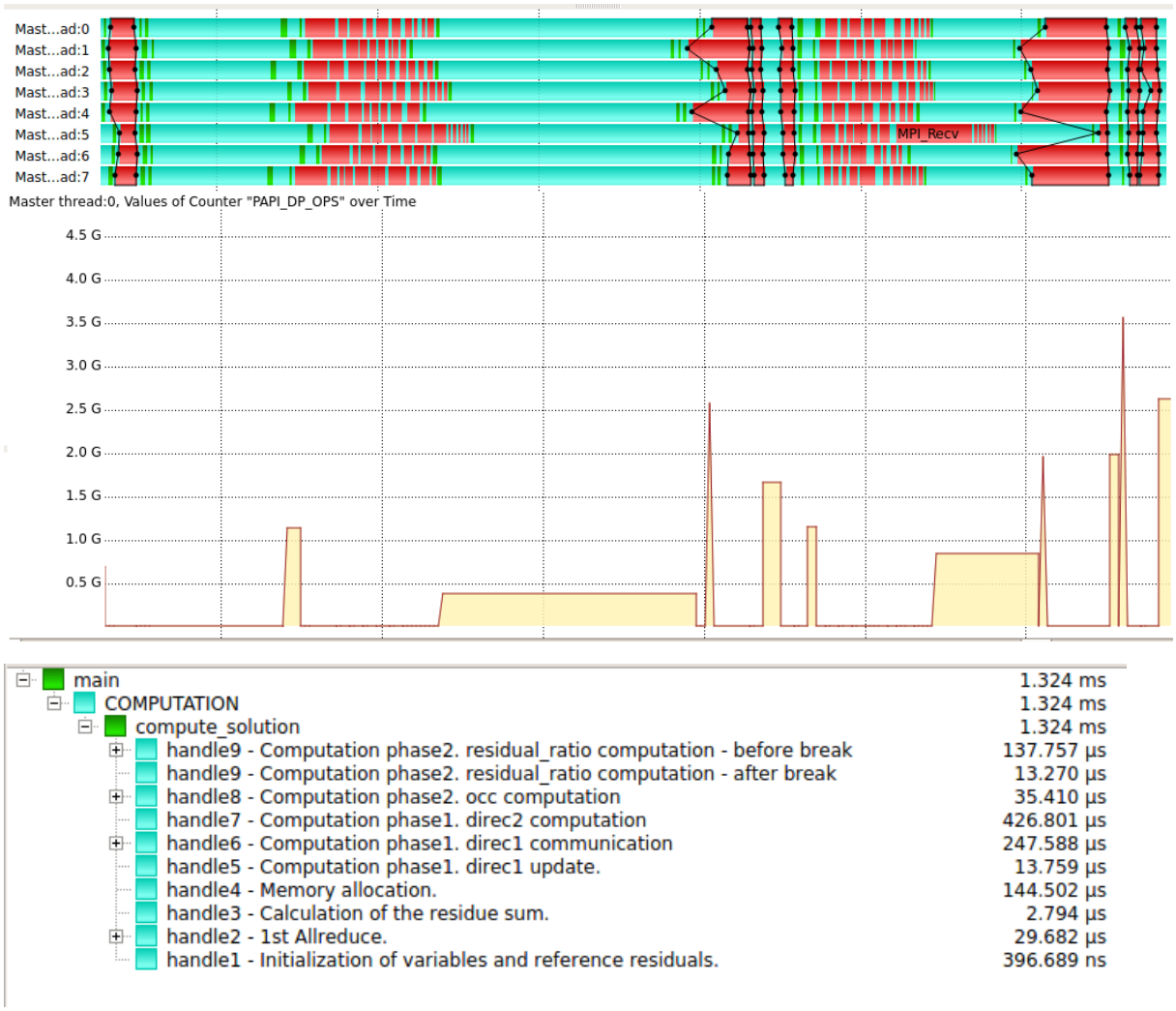

Fig. 11. Analysis of a single computational iteration. MPI overhead, execution steps mapping to the corresponding source code operations. Floating point operations distribution.

The execution times of all the regions were significantly decreased. We can observe also a big increase of the floating point operations intensity, what was achieved thanks to the vectorization, cache optimization and the overlapping of computation and communication.

2. Performance optimization

During the previous milestones, we paid a special attention into an appropriate structuring and optimization of the code in the initialization and finalization parts. Since the most time in the professional numerical simulations (usually much bigger cases than the one used in our benchmark) is consumed by the computational phase, we focused especially on optimizing that part in this milestone. Below we present the optimizations we did and describe precisely different variants we have tried. The measurements for different cases were done both with SCOREP and PAPI, the latter one being always the final benchmark of the obtained performance increase.

What is worth mentioning is that the original code for our optimizations (coming from the assignment A2.3) was already highly optimized – we used non-blocking communication, efficient All_reduce collective operations and derived MPI datatypes. Therefore, the initial performance measurements with PAPI showed that with this version of the code we already satisfy the minimal requirements for this milestone. Nevertheless, we carried on looking for the further code optimizations.

2.1. Optimization 1 – Compiler flags optimization

At the first optimization step, the compiler flags were chosen to give the best performance. According to the prior knowledge gained during the first assignment, the best effects were obtained for –O3 optimization flag. This time, the other optimization flags were tested, as well. Various configurations of the following flags were investigated:

- -Ofast
- -funroll-loops
- -march=native

Nevertheless, the best performance was obtained just for the –O3 optimization flag.

2.2. Optimization 2 – Overlapping computation and communication

In the second step of the program optimization, we focused on computation and communication optimization. Namely, we applied an overlapping computation of direc2 and communication of direc1.

The procedure is the following:

- Outside the compute_solution.c, we loop over the internal cells indices in order to find out if for a given cell, the computation of direc2 requires an access to the ghost cells. If yes, the cell index is stored in the ext_access array. If no, the cell index is stored in the int_access array. In the next step, we pass the arrays and the corresponding counters to the compute solution function.
- Within the body of the compute_solution function, we replace MPI_Recv function for the communication of direc1 with MPI_Irecv. Secondly, we split the direc2 computation loop into 2 parts – first we loop over the cells from int_access array and in the second step we loop over the cells from ext_access array. Between 2 computation loops, a MPI_Wait function is called in order to check if MPI_Irecv received the required data.

Although the idea of overlapping communication seemed to be good, the measurements indicated lower performance and scalability properties of such solution. We conclude that the overhead created due to determination of the int_access and ext_access arrays, as well as the memory accesses to those arrays, is bigger than the benefits from overlapping computation and communication in this case.

Therefore, the second idea was developed. Namely, instead of accessing int_access array in the first loop, we can simply calculate all of the entries of direc2 array as it was done in the original version. Of course, some of the newly calculated entries will not be correct, since they can be computed based on both new and old data. As a consequence, after MPI_Wait we recompute those entries by looping through the ext_access array for computation of the direc2. Although some computations in this case are doubled, we overlap computation and communication and avoid costly memory accesses in the direc2 computation loop. Nevertheless, this approach led even to a worse performance than in the previous case.

2.3. Optimization 3 – Overlapping computation and communication

According to the results from VAMPIR, the best effect of overlapping communication and computation might be done by overlapping direc2 computation and direc1 communication, as it was done in the Optimization 2. Nevertheless, since such solution was worse than the original one, we looked for the other parts of the code that could be overlapped with direc1 communication. The potential benefits of such solution are smaller than in the previous case (direc2 computation is the most costly part of the compute_solution function), but might be worth consideration. It turns out, that at the end of the while loop, there is a part of a code that can be overlapped with the communication of direc1 without introducing new overhead due to memory accesses as before.

The basic idea of overlapping in this case is to split the loop at the end (prepare additional arrays for the next iteration step) in 2 loops – one executed at the end of the while loop and the second executed at the beginning of the while loop (just after the MPI_Isend). Thanks to splitting the initial loop into a part that depends on direc1 (has to be executed before the update done at the beginning of the while loop) and a part depending on direc2 (can be overlapped with sending of direc1), we obtain a performance increase. Since we observe that behavior also for the sequential version, we suppose that this way we increase the cache efficiency, as well.

We tried also to overlap the whole loop mentioned before with the communication of direc1. This can be done, if the direc1 array is copied to a temporary array direc1_temp at the end of the while loop. Then, the overlapped loop can access just the direc1_temp array and direc2 array without any risk of data dependency between computation and communication. This way even more computation can be overlapped with the communication of direc1. Nevertheless, such solution showed worse performance than the previous optimization.

2.4. Optimization 4 – MPI_Allreduce optimization

In the next optimization step, we optimized the MPI_Allreduce operations by introducing MPI_IN_PLACE value as a send buffer. This should reduce unnecessary memory motion, resulting in higher performance of the program.

Nevertheless, we did not observe a significant improvement in performance of the optimized code. Perhaps the performance increase could be higher for bigger test cases.

2.5. Optimization 5 – Vectorization

Although the main bottleneck in the program performance is the communication, in some circumstances (e.g. limited resources) single-core optimization might play a vital role, as well. Therefore, we decided to go further with the single-core optimization and use the technique of manual AVX vectorization with Intel Intrinsics.

Nevertheless, like in the A1 assignment, better vectorization resulted in performance drop (perhaps due to the worse cache optimization associated with better vectorization). Therefore, we treated the optimization 4 as our final version.

5. Final remarks and further work

The Programming of Supercomputers Lab gave us an opportunity to familiarize ourselves with the whole process of using and optimizing codes for supercomputers. In particular, it was a privilege to use SuperMUC, one of the most powerful machines in the world.

During the work on the assignments we had a possibility to get to know the architecture of SuperMUC as well as its components (computational nodes and Sandy Bridge processors). We excelled in using supercomputer as an efficient tool for solving engineering problems.

What is more we managed to parallelize and optimize both for serial and parallel execution a commercial simulation software. This is especially important issue from the point of view of the simulation software industry.

Finally, we could profile and analyze the code with modern tools such as SCOREP, VAMPIR and CUBE that are being in use in many supercomputing centers.

Nevertheless, we are aware how much work can still be done to improve the code. We can think of further optimizations of the initialization and computation part, but it would involve a considerable redesign of the code and require much better knowledge about the program itself. Therefore, we believe it is out of scope of this subject. However, the awareness of further optimization directions could be very helpful in the future projects that we will be involved in.