Programming of Supercomputers WS12/13

Final report

Dmitry Pinaev

1 Introduction

The proposed project was aimed at parallelization and benchmarking conjugate gradient-like solver with MPI library. The source code itself is subdivided into several functional units for particular problems:

- Binary data-file reader
- Initializer
- Solver
- Finalizer
- Plotter (vtk files)

Mainly, work was done in initialization and solver parts of the program. The special library Metis was used in order to exploit grid's geometry and configuration and obtain optimal communication faces among different partitions of a comutational domain.

During the execution, each running process solver obtains its own part of the domain and performs computing minimizing the residual over the number of iterations. The communication

After the parallelization, obtained code showed performance impovements (time) with same number of iterations and residual order.

The following report goes through different steps of the implementation and describes obtained results.

2 Sequential optimization

Before the parallelization some imporvements and benchmarks were done for a sequential code.

We can split program's execution into three main parts:

- Intialization
- Computing
- Finalization

Each part was analized with a library PAPI. The main counters were number of cycles and cache miss rate.

Serious impovements were obtained for the initialization part. Initially, data was stored in text files. That implies slow data access and its distribution into arrays. After implementing special converter from text to binary format the speed-up gained one order of magnitude (Figure 1).

Besides, the measurements for different compiler opitimization flags were done. They show that largest speed-up we get after switching from -O0 to -O1. For example, for cojack geometry -O1 reduced the exucution time from 87 to 37 seconds, -O2 imporoved time to 31 seconds.

For large input text files (cojack – 42 MB) initialization phase took up to 15% of whole time, for smalll input files (drall – 6 MB) the initialization phase is around 30% of the whole time. Of course, converting to binary format made time for file reading neglectable.

Also we can notice than number of floating point operations is significant just for a computational phase. For data reading and writing it is almost zero.

3 Benchmark parallelization

3.1 Data distribution

3.1.1 Distribution algorithms

There three data distribution approaches were proposed:

- Classical
- Dual
- Nodal

The first one is the most simple approach when an array of cells is split into several sequential parts. Since we do not take into account spartial location of cells relative to each other, resulting partitioning looks inside the computational domain like randomly scattered cells (Figure 3).

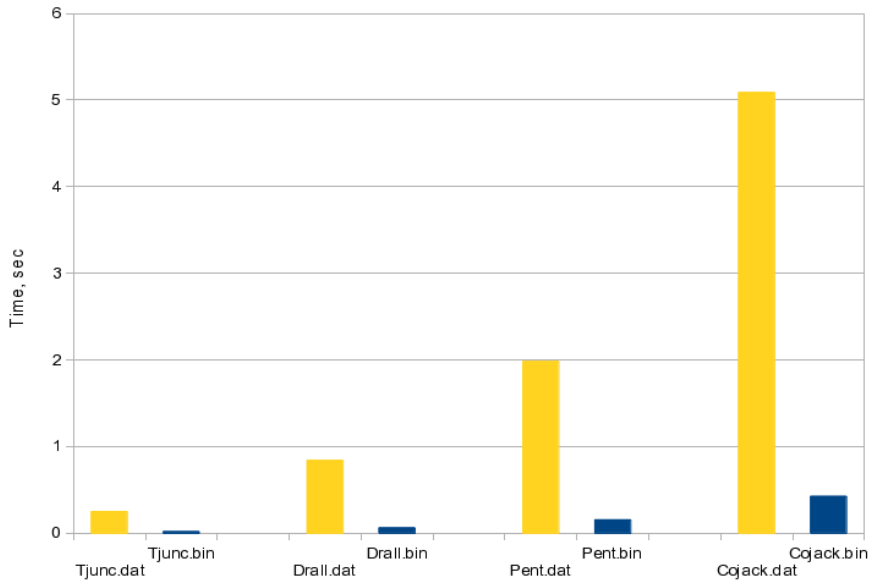

Figure 1: Comparison of reading time for text and binary input data

Next two algorithms are implemented inside the Metis library.

Typically, the graph or mesh is being partitioned so that a computational task can be correspondingly divided up among a number of processors for parallel computing. Thus, it is important that each part of the partition have roughly the same number of elements, and that "connections" between distinct partitions be limited, as much as possible, to reduce the amount of interprocess communication necessary [**?**].

The reason for using special tools for domain partitioning is the reducing of communication between processes.

Using initial information about vertices and cells Metis library can split computational domain into several almost equal partitions (Figure 4).

3.1.2 Structure of the initialization code

The goal of initialization is to read binary file with initial data and fill following arrays: BS, BN, BE, BW, BL, BH, BP, OC, CGUP, SU, VAR, CNORM. Moreover we need to split LCC array and create two mapping arrays for local number to global number and vice versa transformations.

There are two approaches for initialization:

- Read all the data in one process and the distribute among the processes using MPI
- Each process reads binary file and takes part of the data needed just for this process

For this work second approach was chosen because it simplifies source code drastically without serious any drawbacks in performance.

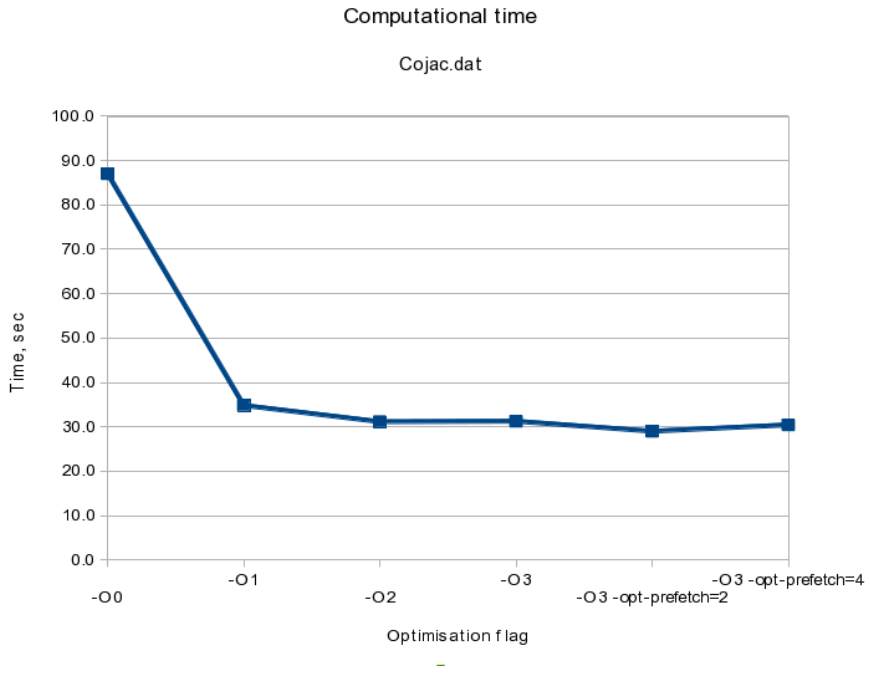

Figure 2: Execution time against different levels of optimization

In sake of simplification, initialization code was divided into several functions:

- `read_binary_geo`. Function reads whole data arrays from binary file
- `partitioning`. Here we use requested partitioning algorithm and distribute cells among processes
- `allocate_memory_for_distributed_arrays`. After we knew number of cells belonging to the current process we can allocate necessary amount of memory for all arrays
- `allocate_memory_for_mapping_arrays`. This function fills `global_to_local` and `global_to_local arrays`
- `fill_local_arrays`. Now we can fill all local Bs and CGUP arrays. The rest arrays are initialized with zeros
- `fill_send_recv_arrays`. The function creates and fills arrays which store information for partitions communications
- `fill_local_lcc`. To know about all the neighbours of all cells we need to store part of lcc array in each partition

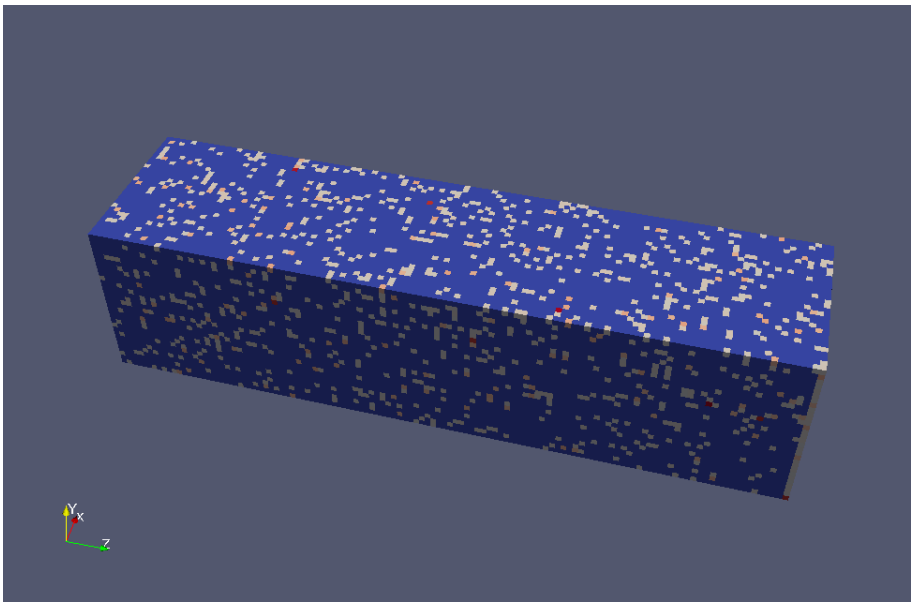

Figure 3: Classical distribution for one partition

3.1.3 Using Metis

All we need to use the Metis library is to create two arrays. First `eptr` is used to store the numbers of all the nodes for each cell in current domain. The second `eind` contains "pointers" to beginning of each cell subarray in first array.

```
if ( strcmp(part_type, "dual") == 0 ) {
    METIS_PartMeshDual(&ne, &nn,
                       eptr, eind,
                       NULL, NULL, &ncommon,
                       &procs, NULL, options,
                       &obvl, *epart, *npart))
} else if ( strcmp(part_type, "nodal") == 0 ) {
    METIS_PartMeshNodal(&ne, &nn,
                        eptr, eind,
                        NULL, NULL,
                        &procs,
                        NULL, options,
                        &obvl, *epart,
                        *npart);
} else if (strcmp(part_type, "classical") == 0) {
    int elems_per_node = ne / procs + 1;

    for ( i = 0; i < ne; ++i ) {
        int p = i / elems_per_node;
```

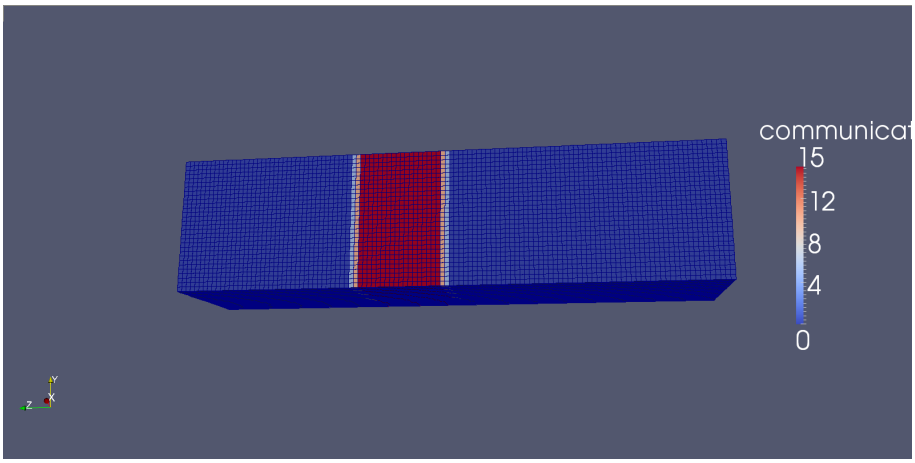

Figure 4: Dual distribution with send and receive arrays for one partition

```
20          (*epart)[i] = p;
21      }
22  } else {
23      printf("unknown␣partition␣type\n");
24      MPI_Abort(MPI_COMM_WORLD, -1);
25  }
```

It is worth to mention that even in case of classical partitioning we still use `epart` array. That simplifies the following code and allows us to work uniformly despite the partitioning algorithm.

3.2 Indexing and mappings

To communicate with neighbours each partition must store information about cells located near its boundary. This implies communication via global indexing valid for all cells. But to operate with cells locally it is much more easier to use special local indices.

To fulfil our purposes we create two mapping arrays. The length of first array is equal to the total number of cells in computational domain. The array `global_local_mapping` is stored and used in each partiton (Figure 5). The second array `local_global_mapping` contains the same amount of cells as its partition (Figure 6).

We fill these two arrays during the initialization and then pass to computations routine.

3.3 Communication lists and ghost layers

Solver works individually in each partition. Generally, each cell communicate with its six neighbours. While the communication is done within the partition no problem arises.

If a cell lies near the boundary of whole geometry we assume that external boundary cell contains zeros. In case the cell is situated on the boundary between two different

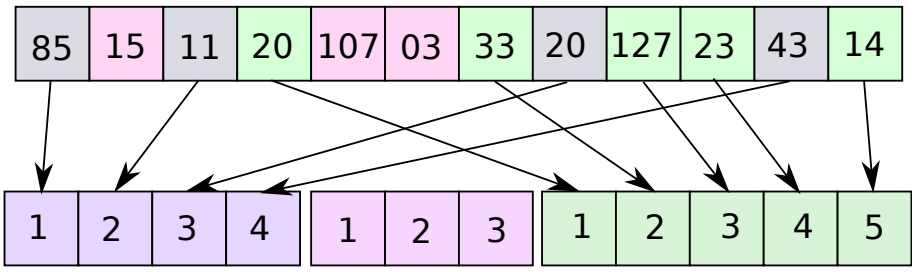

Figure 5: Local to global mapping

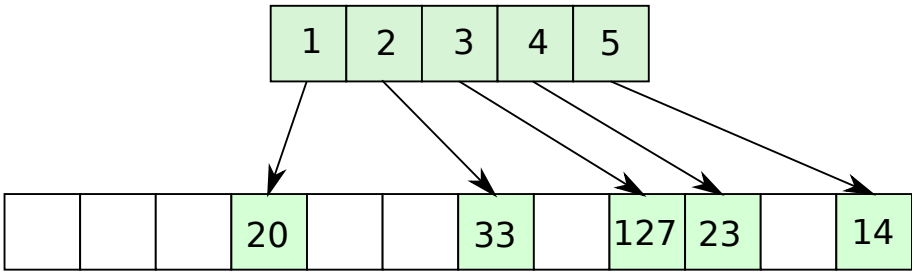

Figure 6: Global to local mapping

partitions we must somehow transfer values of this cell to the neighbours.

The communication model uses so-named ghost layers. Each boundary layer of cells from any neighbouring partition is a ghost for current partition (Figure 7).

The communication happens each iteration of the solver. To perform this action we need to store two special arrays: `recv_array` and `send_array`. Each of arrays contains local numbers of cells supposed to be send neighbour or received as a ghost layer from neighbour.

3.4 MPI implementation

Since each process reads its own data from data file we do not need any MPI communication at all at initialization stag e.

MPI is used just in solver to enable data sending and receiving.

The communication is done for a vector named `direc1` which stores current state of the solution. It contains all inner cells of a partition and then additional cells to be received from all neighbours.

We know that `send_list` and `recv_list` may point to arbitrarily distributed cells within the vector `direc1`. This configuration of the data does not allow to use simple `MPI_Send` or `MPI_Recv` functions with an array of `double` as we do it with sequential data.

To overcome aforementioned problem we use `MPI_Type_indexed` datatype.

```
1 MPI_Datatype send_datatypes[neighbours_count];
2
3 for ( i = 0; i <  neighbours_count; ++i ) {
```

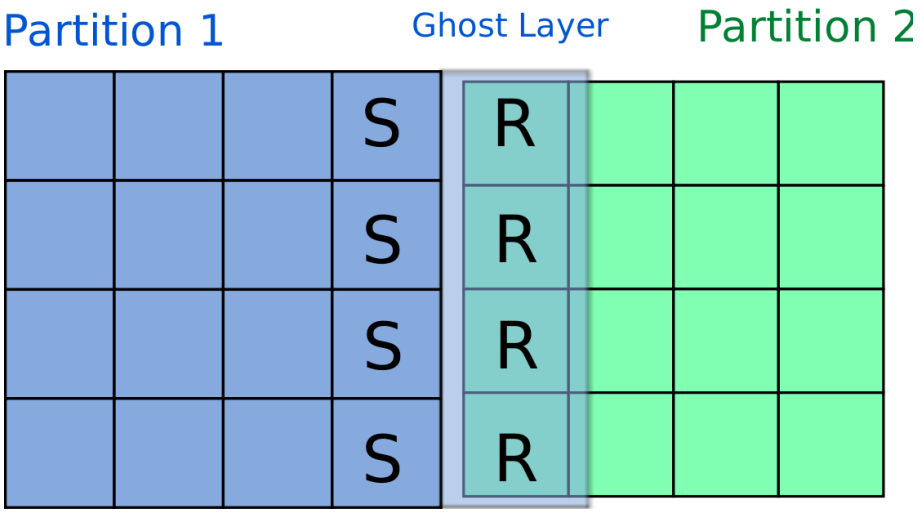

Figure 7: Cells marked with "S" are sent to neighbouring partition. Cells marked with "R" are received from neighbour and used as a ghost layer.

```
    int j;
    int* b = (int*) calloc(send_count[i], sizeof(int));
    for ( j = 0; j < send_count[i]; ++j )
        b[j] = 1;

    // type for sending to neighbour #i
    // send_list contains cells to be send
    MPI_Type_indexed(send_count[i], b, send_list[i],
                     MPI_DOUBLE, &send_datatypes[i]);
    MPI_Type_commit(&send_datatypes[i]);
    free(b);
}
```

On the each time-step we update vector `direc1` and then do an update for ghost cells. First, we need to send updated data to all the neighbours of the current partition. Second, we receive updated values from neighbours.

To avoid a dead lock `MPI_Isend` is used. It allows program to immediately proceed without blocking sending and receiving routines.

To control the convergence of a solution we need evaluate global residual for all processes. It is done with a reduce operation.

```
MPI_Allreduce(MPI_IN_PLACE, &res_updated, 1,
              MPI_DOUBLE, MPI_SUM,
              MPI_COMM_WORLD);
```

As a result, computational function looks like:

- Creating indexed MPI datatypes

- Start computational loop
  - Update vector `direc1`
  - Send and receive ghost cells
  - Update vector `direc2`. We use remapped to local cells `lcc` vector
  - Computer residual and other clobal values
- As soon as necessary magnitude of residual is reached, stop iterations
- Result is stored in vector `var`

4 Performance analysis and tuning

4.1 Scalability

Tho geometry files were used for an analysis: pent.geo.bin and cojack.geo.bin. They have sizes 13.7 MB and 40.3 MB accordingly.

During the measurements of computational phase we can observe several phenomena:

- The problem remains scalable as we increase a number of processes for large files (Figure 8, top)
- For a relatively small file communication overhead grows faster and for 32 processors speedup decreases (Figure 8, down)
- Classical distribution is the worst case from view point of communication overhead, therefore, scalability drops down very fast (Figure 9)
- In general, scalability of a code with dual distribution is obviously better that with classical one
- For the pent.geo.bin file processed with dual distribution scalability reaches 19 for 16 processes. Assuming that code has no errors, it could be reasoned by perfect fitting the data into the processors' cache. Also, it could be caused by incorrect time measurements for a sequential code (see section 5.1)

4.2 Initialization phase

Since the approach was chosen when each process reads its own data it causes the problems of simultaneous access to one file by different nodes. It significantly slows down the start-up (Figure 10).

There might be three possible solutions for this problem:

- Rewrite code in a way when one process reads data and sends to neighbours

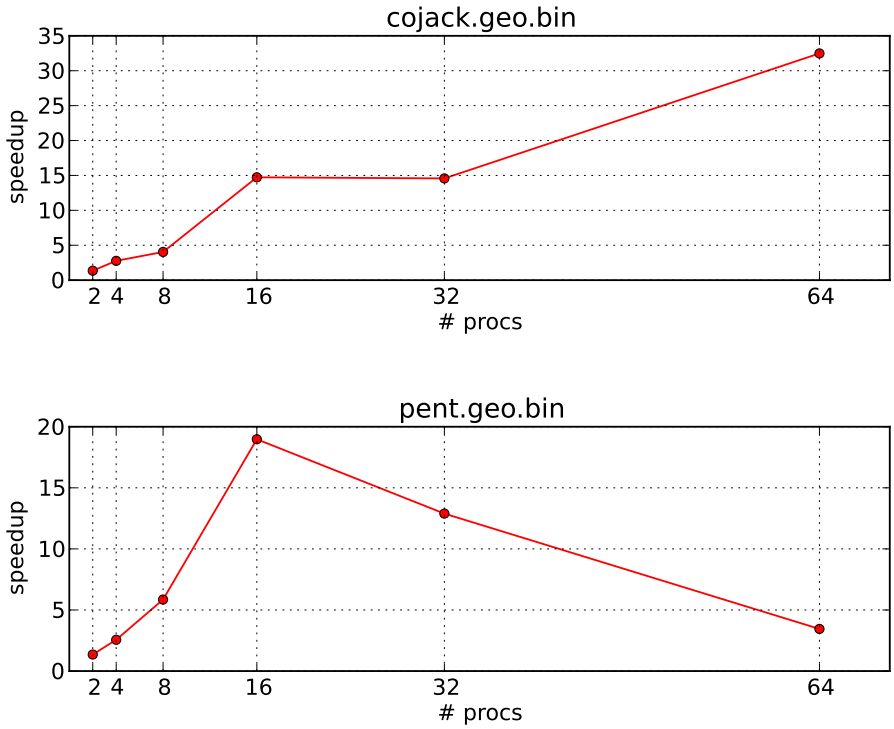

Figure 8: Scalability for dual distribution (computational phase)

- Use MPI functions for files read/write operations
- Organize ordered access to the data file for all nodes

4.3 Bottlenecks

Analysis of Periscope files showed that function `MPI_Allreduce` takes a lot of time during the computational phase. We can not eliminate it because global residual is required for the correct computing.

As all processes read their own part of data they do not idle waiting while one master process makes a distribution. After optimizing simultaneous file reading it will no longer be a problem.

5 Overview

5.1 Problems

To measure speed-up we have to compare obtained results with serial version of the program. There was strange behaviour of cluster which caused wrong execution time for a

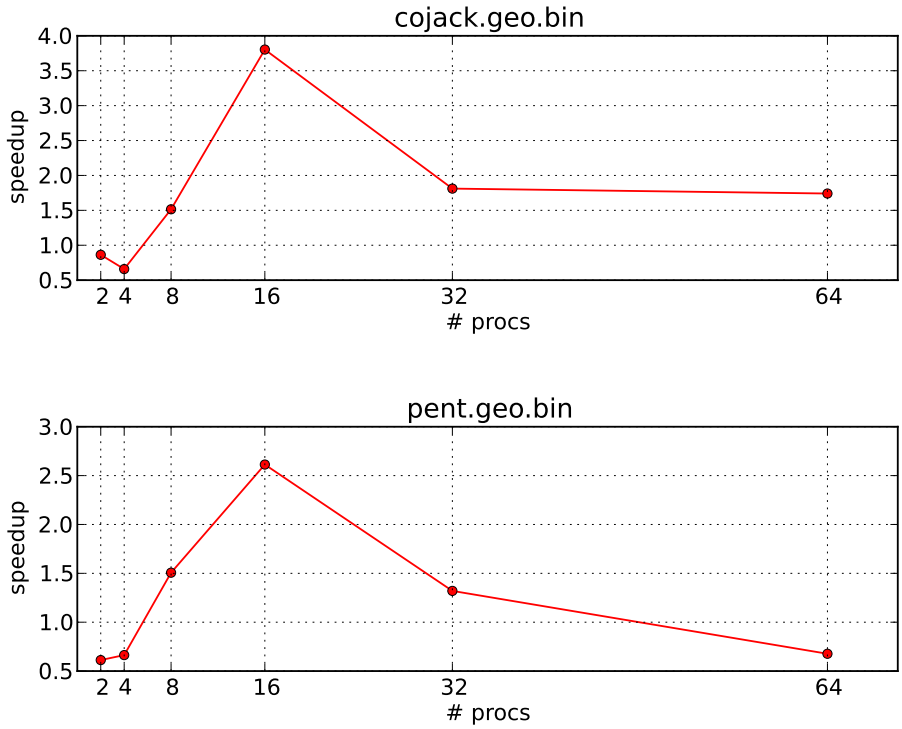

Figure 9: Scalability for classical distribution (computational phase)

serial program.

For example, program was executed in a following way:

```
1  salloc --nprocs=1 --partition=mpp1_inter
2  srun_ps ./gccg cojack.geo.bin
```

Afterwards program reported that computational phase took 50 seconds. But, if we run two (or more) instances of sequential program as below:

```
1  salloc --nprocs=2 --partition=mpp1_inter
2  srun_ps ./gccg cojack.geo.bin
```

each instance prints 25 seconds for computational phase. This result correlates with parallelized version of code.

5.2 Summary

This work included several main stages leading to the final working parallelized code.

First we were supposed to study performance analysis tools, particularly, the PAPI library. Having a wide spectrum of profiling and tracing tools it is very easy to identify problems in the performance and improve communication between nodes in a cluster.

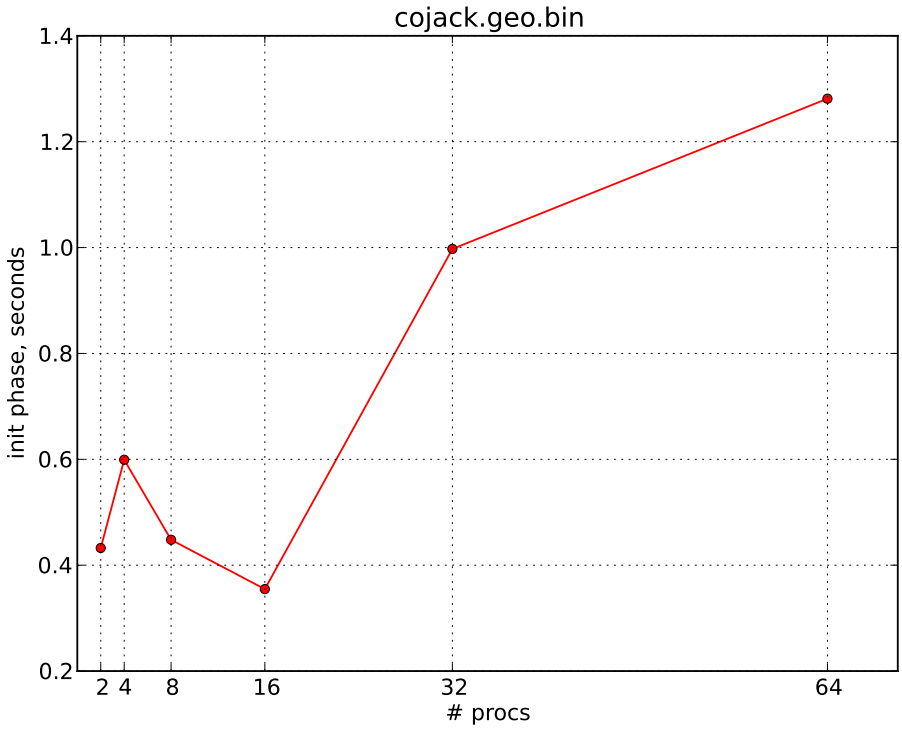

Figure 10: Initialization time depending on number of processes

Second part of the work was devoted to the Metis library which helps to efficiently split computational unstructured grid into almost equal parts reducing the communication cost. It is done by minimizing shared interfaces between partitions.

Finally, serial code was parallelized with MPI library. After evaluating cells supposed to be send and received we have managed to rework computational loop and insert a code for a data exchange between nodes. Several collective operations were used.

The parallelized code delivers the same result as the sequential one. The residuals' difference is around machine precision. The total number of iterations is conserved and remains the same for all number of processes.