Programming of Supercomputers

Final Project

Parallelization and optimization of GCCG solver

Team 12:

Ajay Karthik - 03643141

Mahyar Valizadeh - 03636571

Winter semester 13/14

Introduction

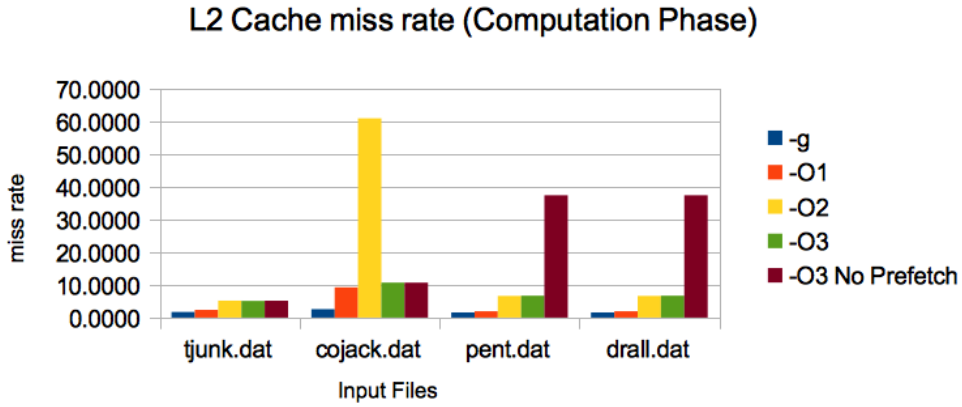
Brief summary of first assignment and introduction to final assignment

Outlook of first assignment

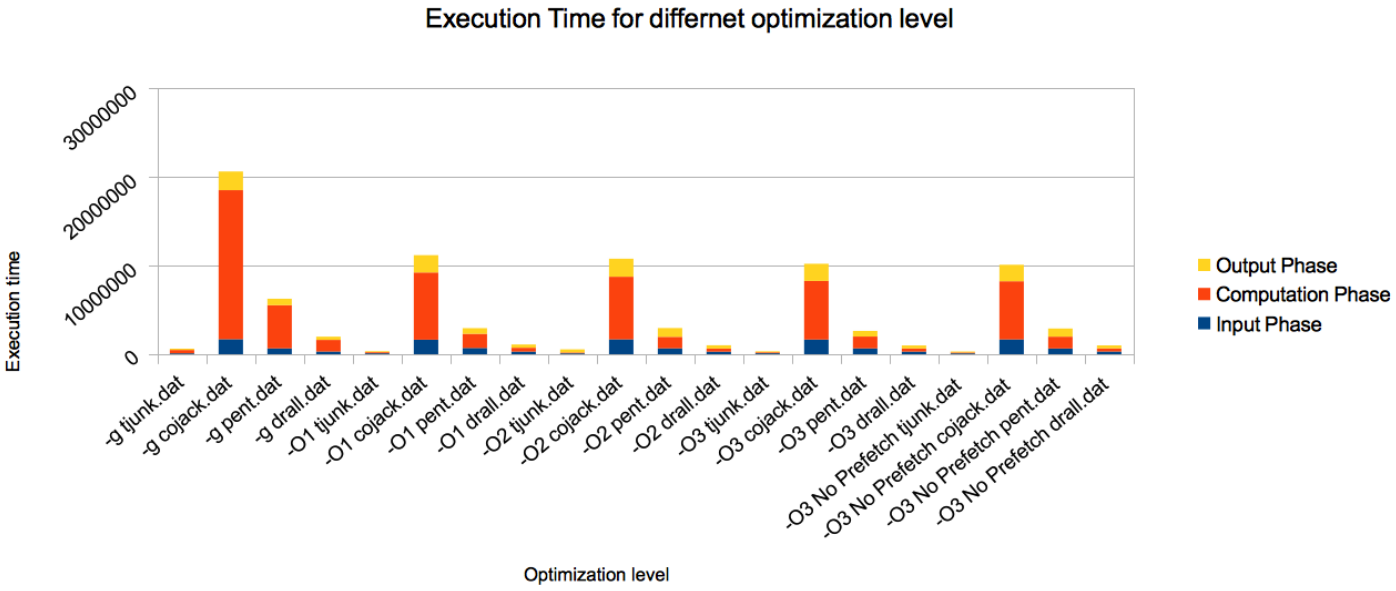
The main objective of the first assignment was analyzing sequential generalized conjugate gradient fire benchmark solver called "gccg" and optimizing it by applying different compiler flags and measuring performance using PAPI library and in the meantime getting familiar with running environment. This has major importance in performance analysis of parallel code since the gained speedup should be compared to the optimized sequential version.
Main features of PAPI library, which has been used here, were not only measuring the execution time with PAPI counters but also checking cache misses of different cache levels. The amount of cache misses can play an important role in performance of our simulation code in that we have faster access to data needed to proceed the computation. Achieving less cache miss rate can be exploited in parallelize version as well since by data decomposition one can have local memory access to data required because with less amount of data there is higher chance that it fits into cache and in result less cache miss rate and higher performance and higher speedup.

Result and conclusion of first assignment

- Following figure is an example of comparison of the level 2 cache miss rate. It might be strange that the amount of cache misses seems it goes higher with more optimized compiler flag but actually a missing point here is that the amount of cache accesses are also higher. This can be verified also by the fact that the amount of floating point operations (flops) are higher for these cases, therefore cache misses are not relatively shown and if one check the relative cache misses then higher level of optimization (-O3) will have less cache misses as it is more reasonable.



- The most appropriate result providing us with a conclusion on how we should



choose the compiler flag is drawn from the following figure which represents the execution time.

It can be observed from above graph that for each dataset the execution time declines with higher optimization level. Therefore, for obvious reason the compiler flag -O3 -Wall is used for later assignments.

- Last note on first assignment is devoted to the reason of usage of binary files instead of ASCII in later milestones. As expected, reading a binary files which

has direct access to the value in terms of bytes so much faster than the ASCII text files. Besides, It can be observed that size of the binary file in this example is almost one third of the size of the text files which can save storage.

Introduction on final assignment

The final or second assignment was divided in four step by step milestones which is a good representation of the parallelization process of sequential code and furthermore optimization to gain the accurate, bug free, fast, optimized final simulation code. These steps were each dealing with a part of common source code, i.e. first milestone focused on how the data should be distributed on different processes and second milestone focused on communication model and localizing the data on processes. Each step can also be roughly regarded as a unit or integrated test model for the whole process. Third milestone was based on the fact that if the other two milestone are working correctly then one should be able to parallelize the actual algorithm of “gccg" solver and get the same exact result as serial version of code. The last yet most important part of the process is performance analysis of the implemented parallel code using some other tools and libraries provided. Detail discussion about each of these steps are going to be presented in following sections.

Milestone 2.1

Data distribution and domain decomposition

The approach for the domain partitioning

1. Classical Approach:

   The most basic partitioning algorithm which divides the elements equally and sequentially ordered in the input file into different processes.

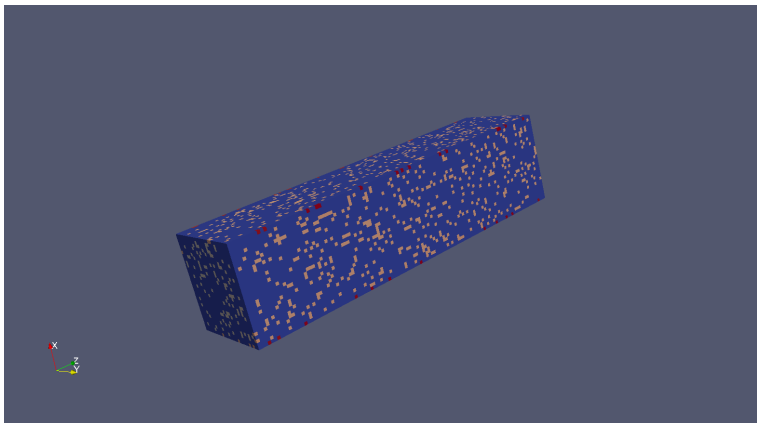
2. Metis Graph Partitioning:

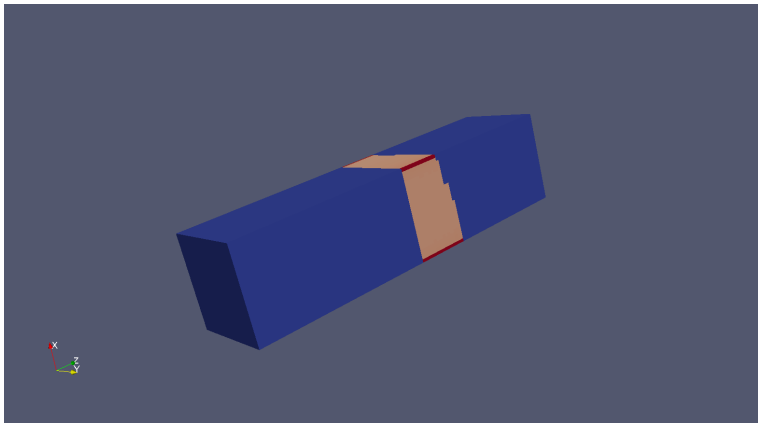
   In this phase, the domain elements is to be distributed to all the processes. In our implementation each processor reads its part of the elements and the way it is achieved is shown in the following algorithmic pseudo code.

The algorithm for data distribution model

```
read_binary_geo{
        All the processes read the global domain sizes.
        If ( rank == root){
                if (partition == classical){
                        fill epart with the rank of the processes sequentially by maintaining the
                        equal domain sizes
                }else if (partition == myclassical ){
                        fills the epart with the neighboring cells of the current element and
                        maintains the total size of elements
                }else {
                        use the METIS graph partition for filling in the epart vector.
                }
        }
        Now filling the global_local_index[domain_size][2] and also
        local_global_index[local_size] from the epart.
        Read the neighboring cells for the current local indexes into LCC.
        Read the coefficients for the local cells.
        Read the global list of elems
        Read the global list of points.
}
```

Now the counters local_int_cells (local internal cells) and elemcount (total elements in a partition) are explicitly calculated and the var, cgup and cnorm are allocated based upon these sizes. Finally the distribution pattern has been observed in the test_distribution function. As you can see the locality of classical distribution (left) is really bad but using METIS we get to much better portioning (right).





Milestone 2.2

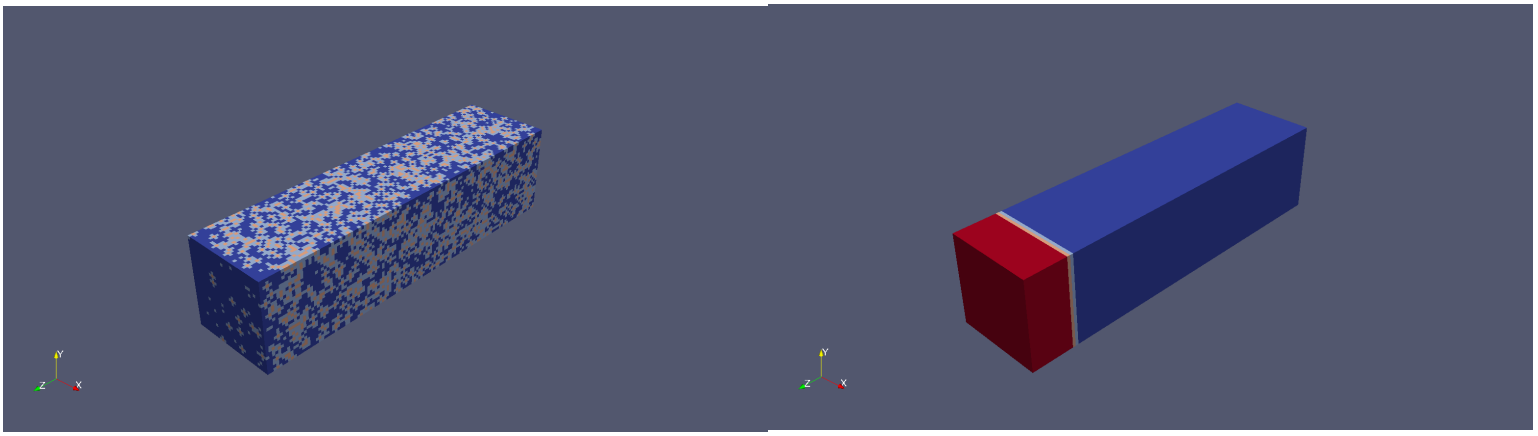
Communication model and data localization

In this phase, the communication pattern between the different processes has to be defined through send list and the receive list.

The algorithm for communication model

```
for i in 0 to local_int_cells do
        if( lcc[i][j] not in current process){
                recv_count++;
                if ( current cell not already in send_list){
                        send_count++;
                }
        }
}
```

fill the send_list with the global_local_index[local_cell|_index].
Now, filling the recv_list by sending the send_list to the respective processes.
For the communication, the non-blocking communication is used with respective calls to MPI_wait.
In the following figures, the result of communication model and formation of send and receive list are shown. Again from this figures you can see that the classical distribution (left) has a lot of nonuniform pattern which makes more communication overhead but METIS (right) tries to give an optimized pattern.



Milestone 2.3

MPI Implementation of computation phase and finalization

At this point, the send_list and recv_list of each of the processes have been setup. So, we can start with the computation phase.

- **First approach**
  A. Use of MPI_Type_indexed – sends the non-contiguous block of data for sending the send list values
  B. recv_buffer is allocated based upon the recv_count.
  C. The usage of neighboring elements was done in direc1 vector. (Direc1 is vector calculation through solve macro)
  D. define solve (lcc_index)
  E. binary search the recv_list and access the corresponding element in the recv_buffer.

Overall Computation phase algorithm for first approach

```
compute_solution(...){
        MPI_TYPE_INDEXED(....send_list)
        Allocate recv_buffer (from the recv_counts)
        Main iteration loop {
                Updating direc1
                MPI_Irecv – for receiving the direc1 values from other processes
                MPI_Isend – to send the direc1 values to other processes
                Call to MPI_wait
                direc2 vector computation using solve macro
                Proceed with computations and using MPI_Allreduce  in the required places
                ....
        }
        MPI_TYPE_free – for the user defined indexed type
}
```

• **New approach (after performance analysis)**

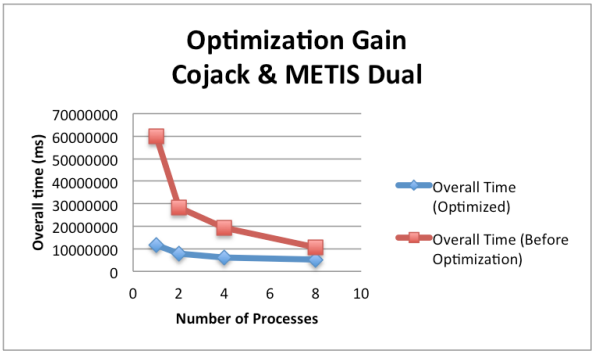
After analyzing the results through Scalasca, the hotspots found in our approach was the binary search function. So in order to prevent this flaw, the size of the direc1 has been extended to the total domain size. This way we can have direct access to values of direc1 to compute direc2.

Overall Computation phase algorithm for new approach

```
compute_solution(...){
        MPI_TYPE_INDEXED(....send_list)
        MPI_TYPE_INDEXED(....recv_list)
        Main iteration loop {
                Updating direc1 through local_global_index access.
                MPI_Irecv – receiving the direc1 values directly from other processes
                MPI_Isend – to send the direc1 values to other processes
                Call to MPI_wait
                direc2 vector computation by direct access to the direc1 vector
                Proceed with computations and using MPI_Allreduce  in the required places
                ....
        }
         2 MPI_TYPE_free – for the user defined indexed type
}
```

Here is the solution to our problem using "pent" dataset and classical distribution using 2 processes. Obviously, the result should be and is the same for sequential or any other number of processes. Comparison of this methods are shown in figure below for one case.

```
=======================================
= AVL -  Linear Equation Solver - GCCG =
=======================================


Input File:  ../data/cojack.geo.bin
=======================================

Output File:  pent_classic_summary.out
=======================================

No. of Active Cells:  318044
=======================================

Iterations Count: 643
=======================================

Residual Ratio: 9.889198e-11
=======================================

Addresses Solution (Minima)      Addresses Solution (Maxima)
===========================      ===========================
  271306         -833.551929            22214        3540.166928
   57463         -833.509690           145399        3537.832057
  271409         -833.167008           150127        3530.774072
  183281         -833.063311           283421        3520.405396
  310505         -831.267625           106716        3510.272569
   71371         -831.019758           103238        3509.574978
  104744         -830.705206           280214        3507.400546
    1603         -830.221264           227257        3499.193983
  267855         -829.428607           208041        3495.213751
  126569         -829.138362            64551        3491.750594
=======================================
~
```



• The Finalization phase

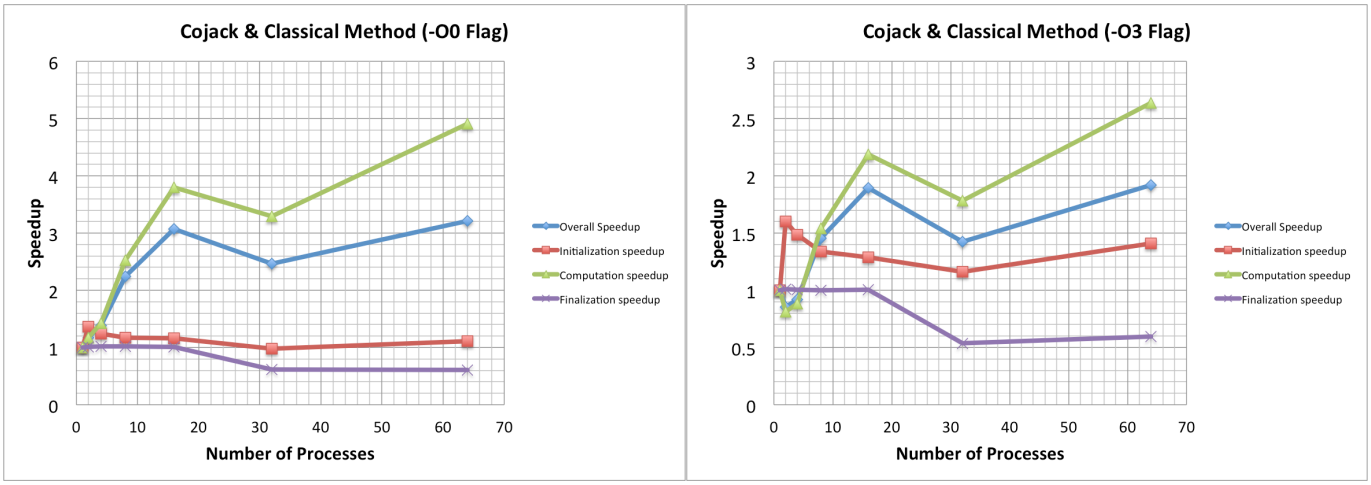
After the computation phase, in the finalization function all the local "cgup" arrays are collected in to a global "cgup" array using the local_global_index into the writing processor.

Algorithm of finalization process

```
Finalization(...){
        if(myrank == writingProc){
                4 MPI_Recv for the local_int_cells, local_global_index, local_cgup, local_var
                Write vtk for the "global_cgup" vector.
        }else {
                4 MPI_Send calls corresponding to the MPI_recv's
        }
}
```

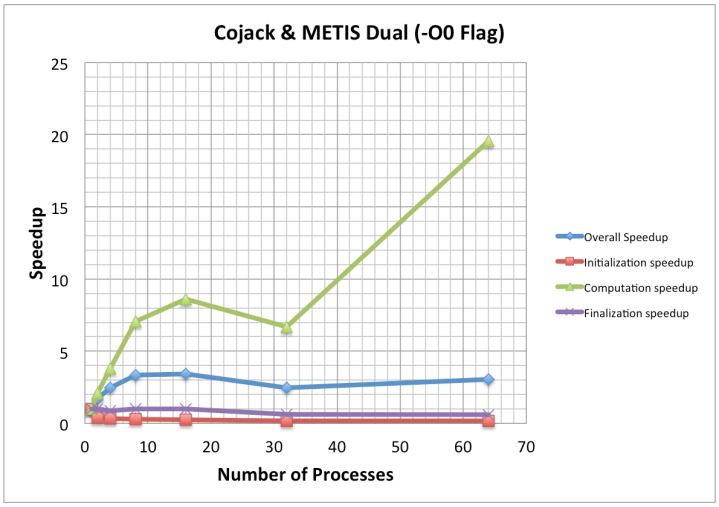
• Results

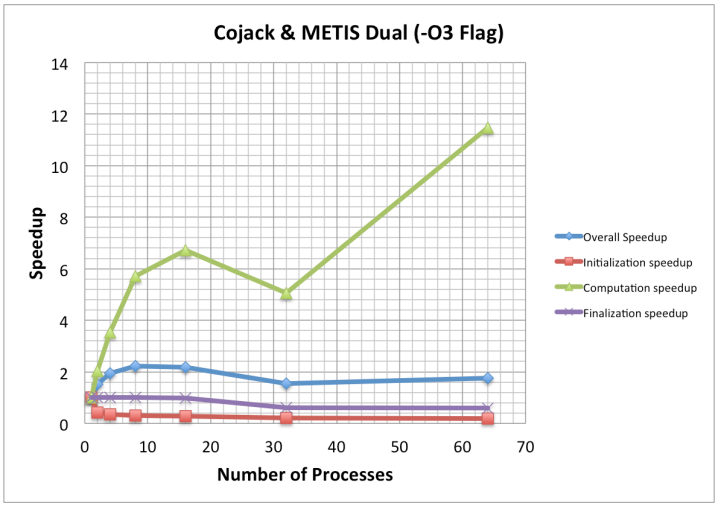
As identified during the first assignment -O3 is the optimized flag which has decent cache access rate and high Mflops. For the current solver we have the performance analysis with the -O3 compiler flag and the -O0 without any compiler optimization. The following are the general aspects of the results.



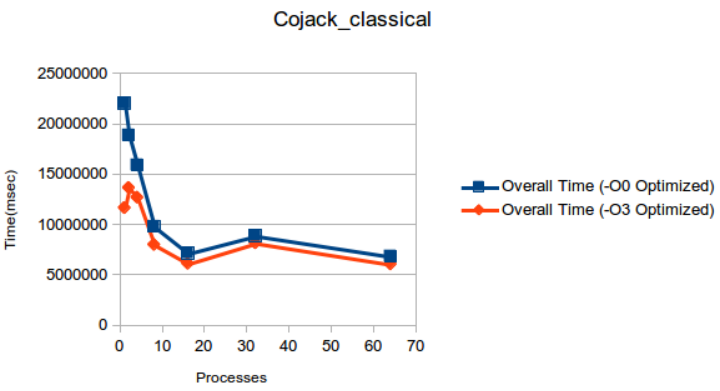
The linear speedup is observed for up to 8 processes in both the compiler flag considerations in METIS distribution. The execution time is much lesser in the

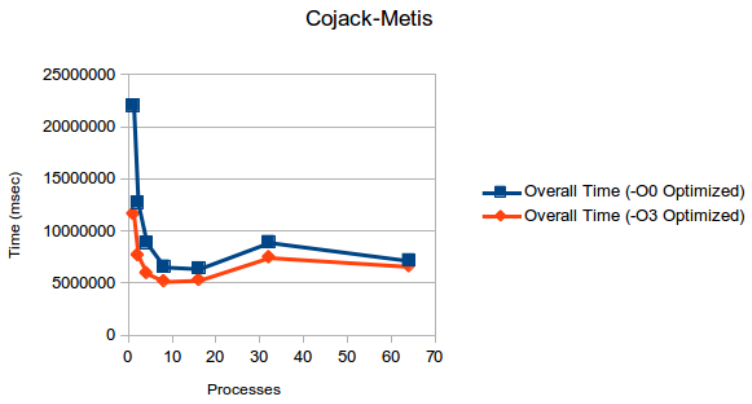
case of the -O3 optimized compilation. Hence, it explains the reason between reduced speedup during this case, since the percentage of overhead is increased for the application implementations.





Comparison between the execution time of -O0 and -O3 compiler flags can be observed in the following graph.





Milestone 2.4

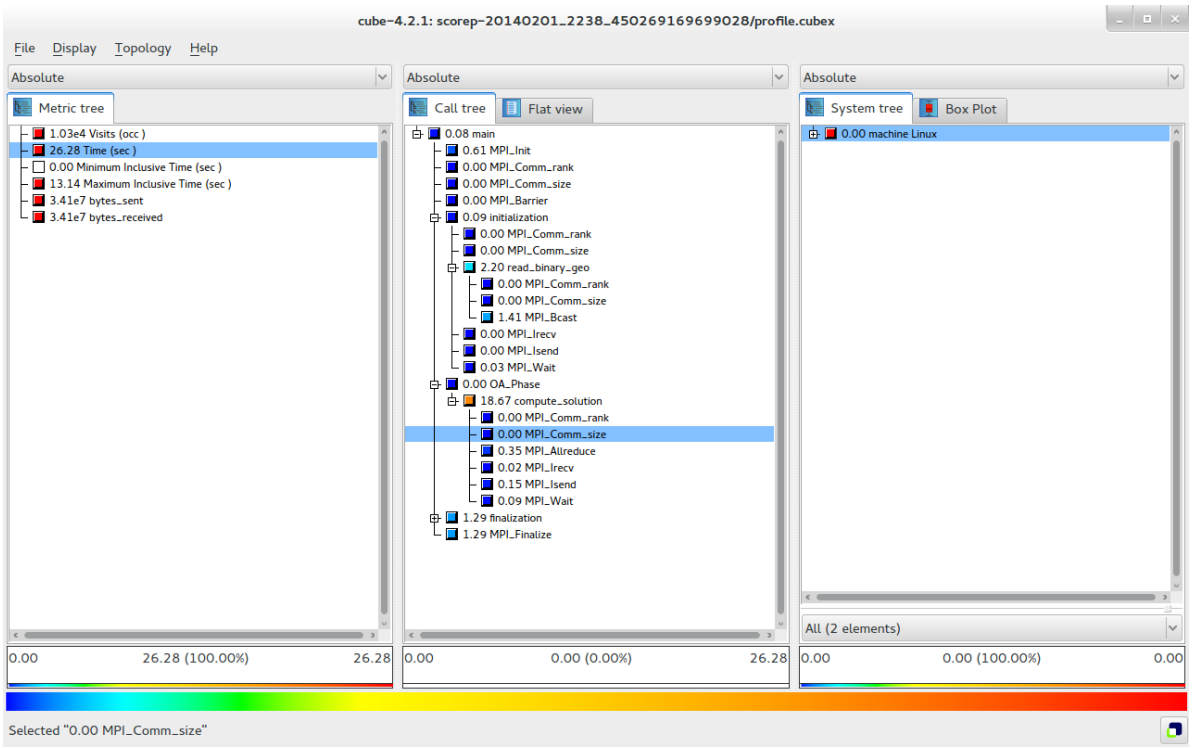
Performance analysis and tuning of MPI implementation using Score-P, Cube and Scalasca

Results related to the OverHead

The following analysis using Score-P measurement library and visualizing with cube shows that the overhead related to classical approach and the metis approach.

- **Classical Appraoch**

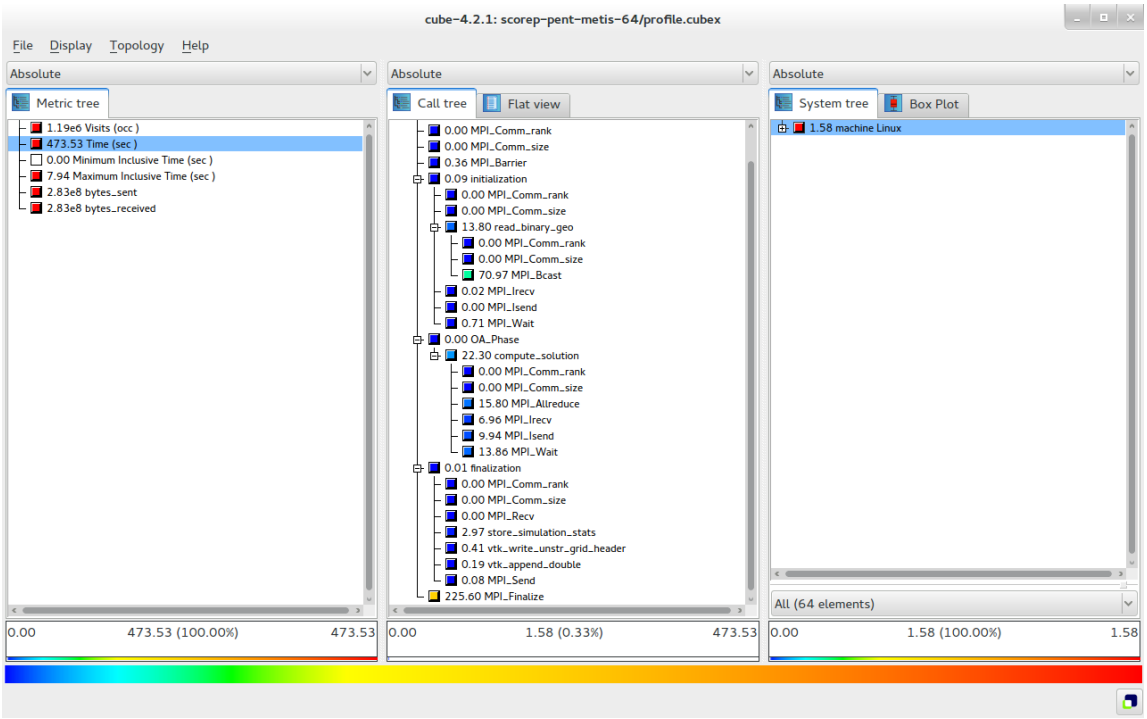
1. MPI_Bcast of epart in intialization phase is very low → probably owing it to very simple logic in the partitioning algorithm.
2. In computation, the non-blocking communication routines have large overhead owing to the large communications with many processors involved in this approach



• **Metis Approach**

1. In this approach, the MPI_Bcast takes time since the partitioning takes into consideration of optimizing the localization of the neighboring elements.
2. In the computation phase, the communication overhead is drastically reduced → since it has to communicate only with local processes. Overhead involved with increase in the number of processes.

In this case, it is overhead involved in the MPI_Init and MPI_Finalization that hampers the application.



Communication Overhead result

Finally, the table below is the result of communication overhead required as the goal of optimization for 4 processes to be less than 25%. It computed using performance measurement tools like Scalasca or Cube and Score-P together.

# of Processes	Cojack-Metis	Pent-Metis
2	7.80%	6.97%
4	17.87%	16.25%
8	28.86%	14.18%