Programming of Supercomputers WS12/13

Final report

Shulin Gao

January 27, 2013

1 Introduction

In this lab course, I am dealing with a Computational Fluid Dynamics solver framework for arbitrary geometries which is written in C language and developed by AVL LIST Gmbh, Graz, Austria. The working phases can be divided into two major parts.

The first assignment is the sequential code optimization, I/O and visualization. In this part, I tried different intel compiler optimizations and converted the text input files to binary input files to analyze the performance behavior of the framework.

The second one is parallelization of the code and scalability analysis. In this part, I used MPI to do the parallelization and some performance measurement infrastructures to analyze and optimize the parallel performance, which I will go into details in the following sections.

2 Sequential optimization

2.1 Compiler optimization

The running environment is MPP cluster in LRZ, and by using Performance Application Programming Interface (PAPI), which is a library for measurement of hardware counters, I collected the metrics including ”Execution time”, ”Mflops”, ”L1 cache miss rate” and ”L2 cache miss rate” with different compiler optimization flags for 4 different input files.

We can see the execution time is apparently lowered down when using different levels of optimizations [-O0, no optimization; -O1, optimization for larger size of code; -O2 optimization for higher code speed; -O3 optimization for code with intensive loops and floating point operations]. Figure 1 is the graph showing this property, and we can also observe from the figure that computational phase takes most percentage of the total execution time. As a consequence, looking for ways to lower the cost in computational phase can be profitable for our total performance. I will discuss this in detail in the parallelization section.

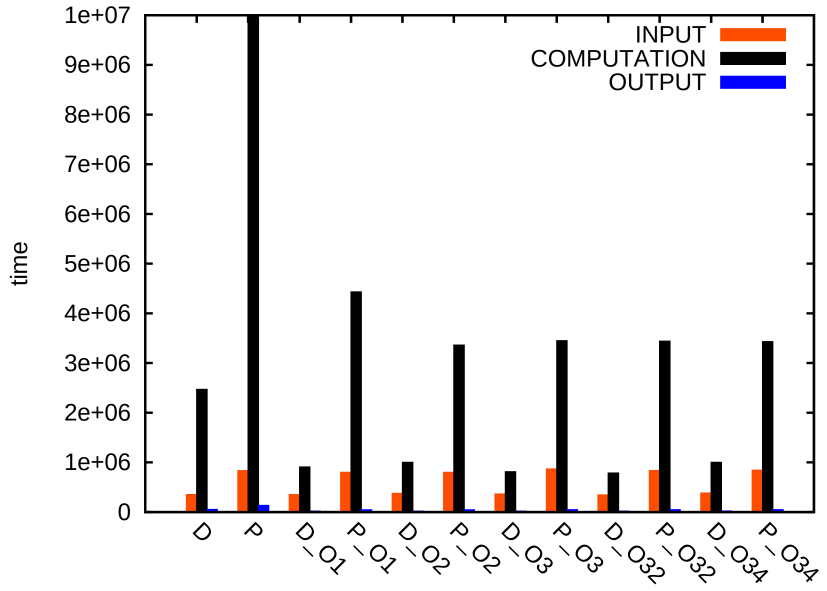


Figure 1: execution time of three phases for Drall and Pent with different optimizations

2.2 I/O

The other way to speed up the sequential code is to think about I/O. That is to lower the time reading data from the file. Here, I converted the text file or ASCII file into the binary file, which can obtain a much better performance.

We can see from figure 2, file reading time is dramatically decreased using the converted binary file. This is because in the normal text editor, the numbers are treated as characters(ASCII) which is represented using 8 bits which is 1 byte. But for the binary representation of the numbers, the capacity is smaller. For example, the number ”25”, in ASCII representation is should be ”x32 x35”, however in binary case it will be ”11001”.

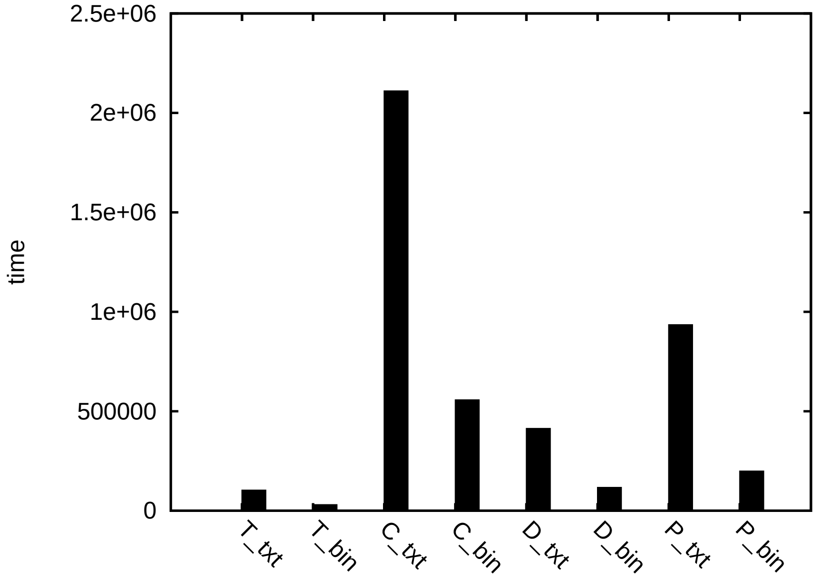


Figure 2: reading time of ASCII and binary files for 4 different input files

2.3 Visualization with paraview

After the solver calculates the input data, we get the output data and we want to visualize it. So we have to write the output data and the corresponding geometry into VTK files and we can visualize and analyze the data in a physical geometry perspective.

3 Benchmark parallelization

After finishing the sequential code optimization procedure above, I have to parallelize the code in order to have a considerable performance. In the parallelization of the code process, there are 4 basic milestones. Namely, data distribution, communication model, computational loop and performance analysis.

3.1 Data distribution

There are two possible directions to distribute the data.

- one core reads the file and distributes the data to all other cores.
- all cores read the file and only keep what they need.

Which one is better?

Firstly, I thought that if I read the file using all of the cores, probably they will not read the file concurrently. Then that can become a bottleneck afterward. Then I chose one core reads the file and distributes to others, in which I use the MPI communication.

How to distribute other than replicate
Basically there are two ways to distribute the data.

- using METIS.
- classical distribution.

METIS is a software package for Partitioning Unstructured Graphs, Partitioning Meshes, and Computing Fill-Reducing Orderings of Sparse Matrices. For geometry partition, it is better that the whole entity is partitioned into several blocks so that the communications between blocks are low. However, the data order I have read from the file is not consistent with the geometry. That is why I need the METIS to know which value in the original data should be in which block(core). In the initialization of the source code, I use function $METIS\_PartMeshDual()$.

On the other hand, classical distribution is to divide the data array directly into blocks without considering the geometry. The following graphs have shown the feature of the two data distribution methods.

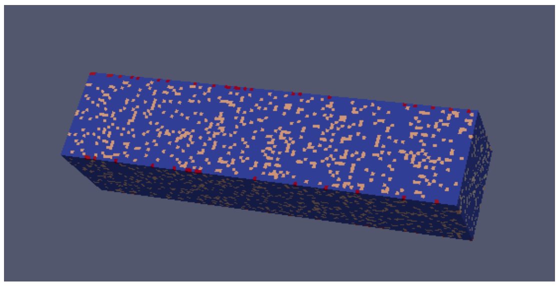
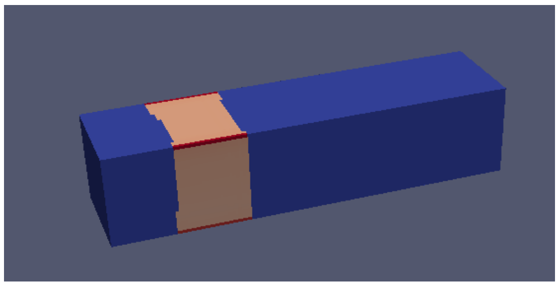



Figure 3: classical distribution(left) and METIS distribution(right)

In METIS distribution, I created another array variable called "$local\_global\_index\_full$" store the sorted global indices with the order of core ranks. And according to this array to sort the other arrays that I need to distribute. In the end, I used $MPI\_Scatterv()$ to distribute the data to the other cores.

The difference in classical distribution is that I do not need to sort the arrays that need to be distributed. What I should do is to divide the arrays into n(num_cores) average parts and also use $MPI\_Scatterv()$ to send data to other cores.

3.2 Communication model

The second step after distributing the data is to consider the communication between cores. In our case, the double array "lcc" contains the neighbor indices of all the internal cells, of course, also external indices. But I just need to distribute the internal cells. In order to communicate the data correctly, I need to record the number of sent data, received data and also the indices of sent data and received data. In my code they are called respectively "*send_count*","*receive_count*","*send_list*" and "*receive_list*".

To build "*send_list*", we need to go over all the cells for each core in a for loop and check their neighbors to see whether they are in the same core or not, if not, we need to count the root cell into the *send_list* and *send_count*.

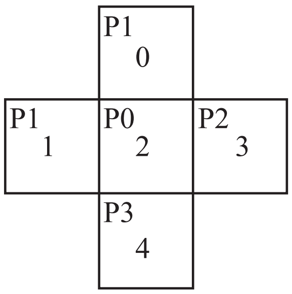


Figure 4: checking neighbors of the root cell

However, it can not be counted twice if there are more than one cell in the same other core. Figure 4 gives a clear explain of that. cell 2 has neighbor 1, 0, 3, 4 and cell 0 and cell 1 belongs to core 1, but cell 2 just needs to be sent to core rank 1 once. I excluded this occasion in my code "initialization.c"(*line 238-320*). And also for building the "*receive_count*" and "*receive_list*", we have to exclude the duplicated ones.

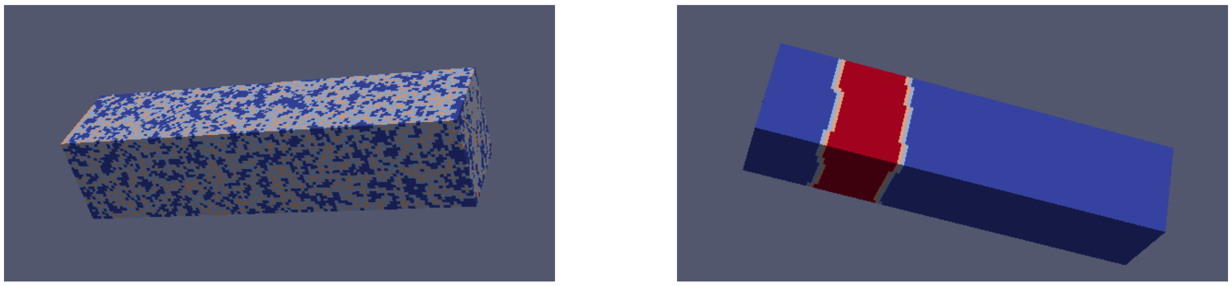


Figure 5: classical communication(left) and METIS communication(right)

In the end, assuming data communication from a to b and we should make sure that the number of sent data to b is equal to the number of received data from a. In my case, the indices stored in the *send_list* are local indices and the ones in "*receive_list*" are global indices. In the following sections, I can manipulate the array using the corresponding local global index mapping and global local index mapping.

3.3 Computational loop

Since I already have the "*send_count*", "*receive_count*", "*send_list*" and "*receive_list*" which are the prerequisite for exchanging data in the computational part. I have to think of ways how to exchange the values and where to put the sent and received data.

In order to send the data, I used *MPI_Type_indexed*() to package all the data in one core to a new type and send them to the other cores(*see my code "compute_solution.c" around line 74*). I decided to put the received data from other cores right after the "*direc*1" in the computational loop, like the graph illustrates below.

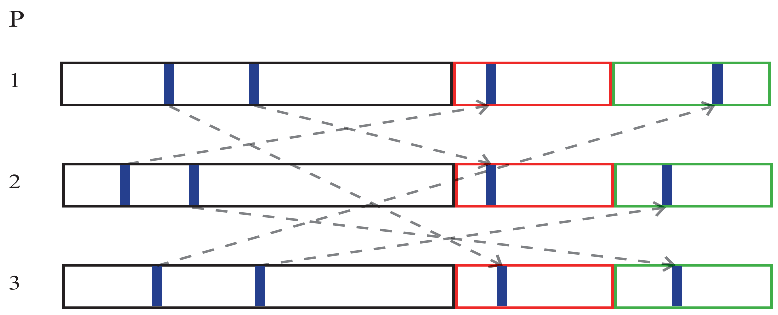


Figure 6: direc 1 sending and receiving data in each core

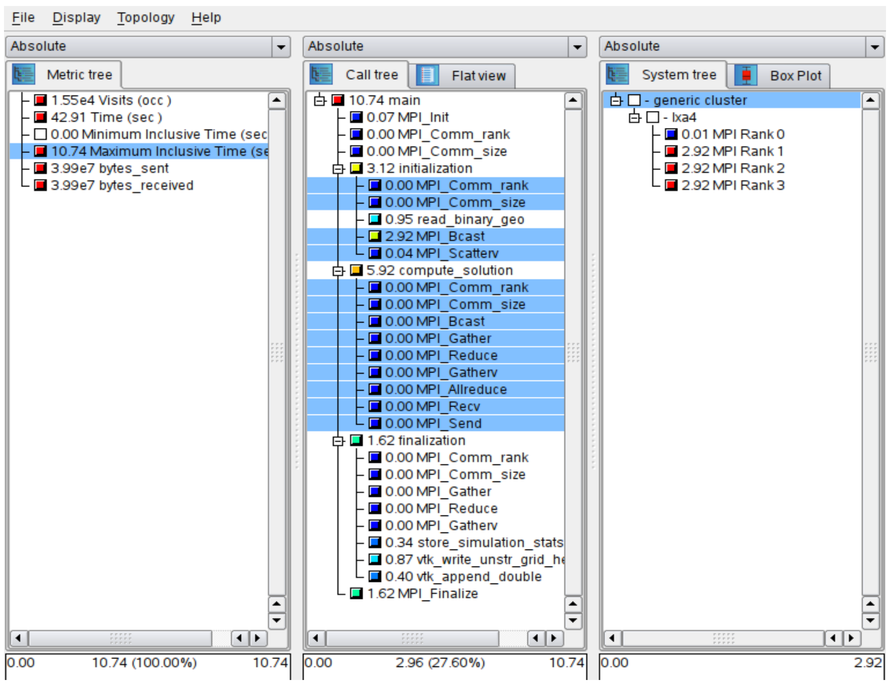
Firstly for easy thinking and implementing, I tried to use a global array in each core and send to itself the data it needs right after the local "*direc*1" array. To realize this, I need to gather the new "*direc*1" values every iteration. But this requires a lot of communication. To avoid this, I changed my strategy, directly send and receive data before computing the new "*direc*1" values. The values received from other cores have the same order as the ascending rank order of the cores.

But, in order to directly point to the values we need in the calculation of "*direc*1", I have to change the values in "*lcc_local*", which is realized in building "*receive_list*" process(*see source code "initialization.c" line 315-336*).

After calculating "*direc*1" in each iteration, I need to use *MPI_Allreduce*() to put intermediate data together and go on within the loop.

4 Performance analysis and tuning

After compiling and running the program successfully, we are accessing to a phase to analyze the parallel performance and tune the code according to the measured metrics. To collect the metrics, we use Scorep which is joint performance measurement infrastructure, Periscope and CUBE.



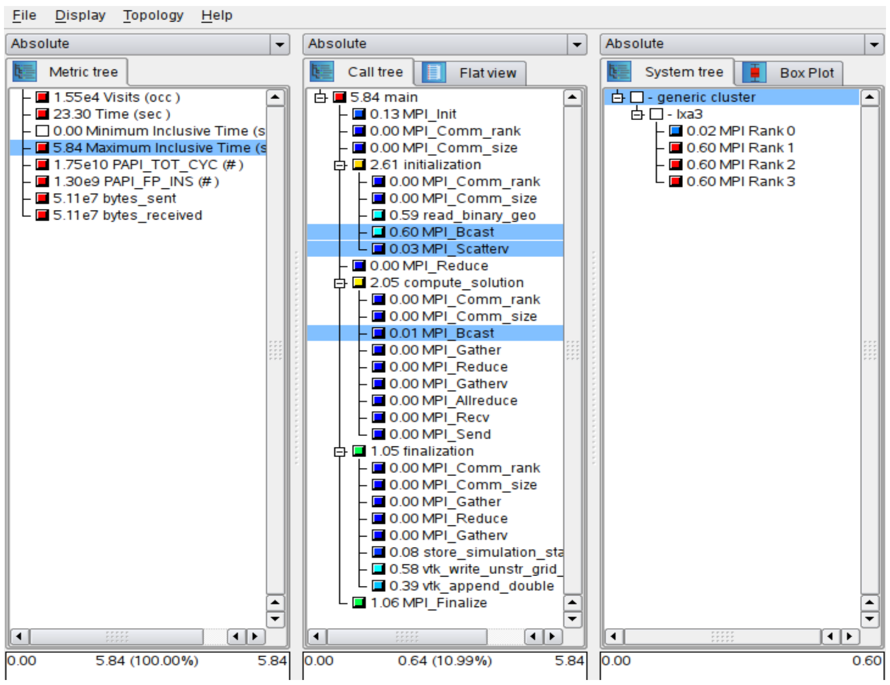
```
1.55e4 Visits (occ )
42.91 Time (sec )
0.00 Minimum Inclusive Time (sec
10.74 Maximum Inclusive Time (se
3.99e7 bytes_sent
3.99e7 bytes_received
```

```
10.74 main
  0.07 MPI_Init
  0.00 MPI_Comm_rank
  0.00 MPI_Comm_size
  3.12 initialization
    0.00 MPI_Comm_rank
    0.00 MPI_Comm_size
    0.95 read_binary_geo
    2.92 MPI_Bcast
    0.04 MPI_Scatterv
  5.92 compute_solution
    0.00 MPI_Comm_rank
    0.00 MPI_Comm_size
    0.00 MPI_Bcast
    0.00 MPI_Gather
    0.00 MPI_Reduce
    0.00 MPI_Gatherv
    0.00 MPI_Allreduce
    0.00 MPI_Recv
    0.00 MPI_Send
  1.62 finalization
    0.00 MPI_Comm_rank
    0.00 MPI_Comm_size
    0.00 MPI_Gather
    0.00 MPI_Reduce
    0.00 MPI_Gatherv
    0.34 store_simulation_stats
    0.87 vtk_write_unstr_grid_he
    0.40 vtk_append_double
  1.62 MPI_Finalize
```

```
- generic cluster
  - lxa4
    0.01 MPI Rank 0
    2.92 MPI Rank 1
    2.92 MPI Rank 2
    2.92 MPI Rank 3
```

```
0.00    10.74 (100.00%)    10.74
0.00    2.96 (27.60%)      10.74
0.00                        2.92
```

Figure 7: profile.cubex file visualized in cube browser before optimization



```
1.55e4 Visits (occ )
23.30 Time (sec )
0.00 Minimum Inclusive Time (s
5.84 Maximum Inclusive Time (s
1.75e10 PAPI_TOT_CYC (#)
1.30e9 PAPI_FP_INS (#)
5.11e7 bytes_sent
5.11e7 bytes_received
```

```
5.84 main
  0.13 MPI_Init
  0.00 MPI_Comm_rank
  0.00 MPI_Comm_size
  2.61 initialization
    0.00 MPI_Comm_rank
    0.00 MPI_Comm_size
    0.59 read_binary_geo
    0.60 MPI_Bcast
    0.03 MPI_Scatterv
  0.00 MPI_Reduce
  2.05 compute_solution
    0.00 MPI_Comm_rank
    0.00 MPI_Comm_size
    0.01 MPI_Bcast
    0.00 MPI_Gather
    0.00 MPI_Reduce
    0.00 MPI_Gatherv
    0.00 MPI_Allreduce
    0.00 MPI_Recv
    0.00 MPI_Send
  1.05 finalization
    0.00 MPI_Comm_rank
    0.00 MPI_Comm_size
    0.00 MPI_Gather
    0.00 MPI_Reduce
    0.00 MPI_Gatherv
    0.08 store_simulation_sta
    0.58 vtk_write_unstr_grid_
    0.39 vtk_append_double
  1.06 MPI_Finalize
```

```
- generic cluster
  - lxa3
    0.02 MPI Rank 0
    0.60 MPI Rank 1
    0.60 MPI Rank 2
    0.60 MPI Rank 3
```

```
0.00    5.84 (100.00%)    5.84
0.00    0.64 (10.99%)     5.84
0.00                      0.60
```

Figure 8: profile.cubex file visualized in cube browser after optimization

First, I got a relatively high percentage of $MPI\_Bcast()$ communication in the initialization phase, also seeing from the corresponding psc file which shows that the overhead is excessive time due to waiting for the root. So I optimized the initialization phase, asking all cores calling METIS function in parallel instead of doing this always in root core. This makes the other cores not in a state idle and it reduced the communication effectively. The following figures show the speedup for three phases and the total phase.

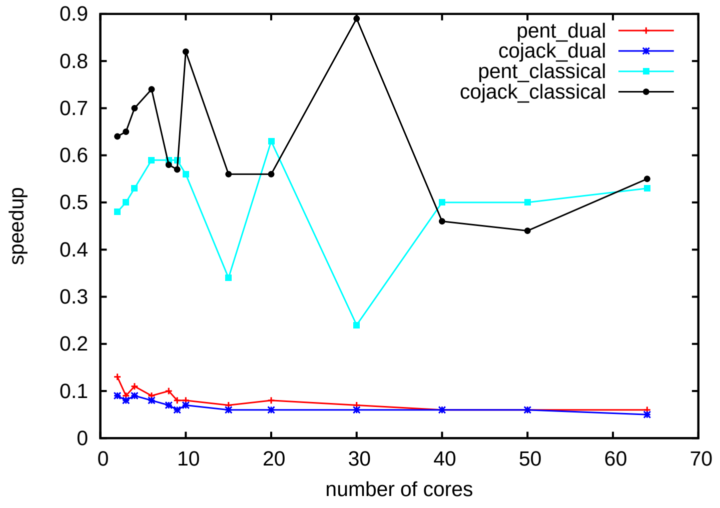


Figure 9: speedup in initialization phase

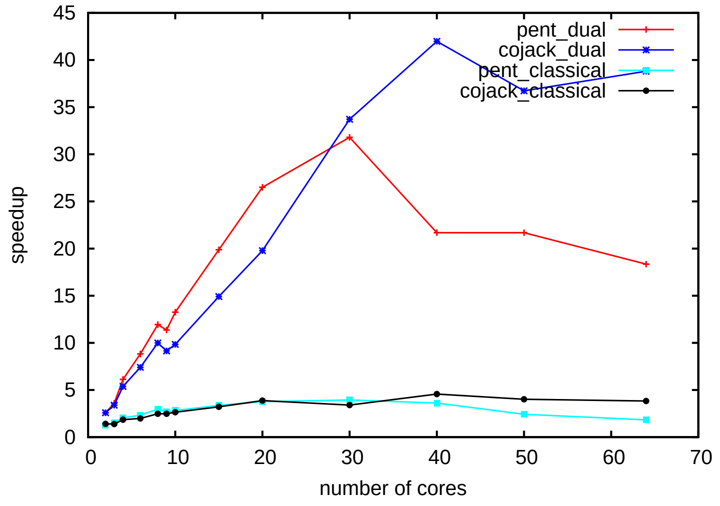


Figure 10: speedup in computation phase

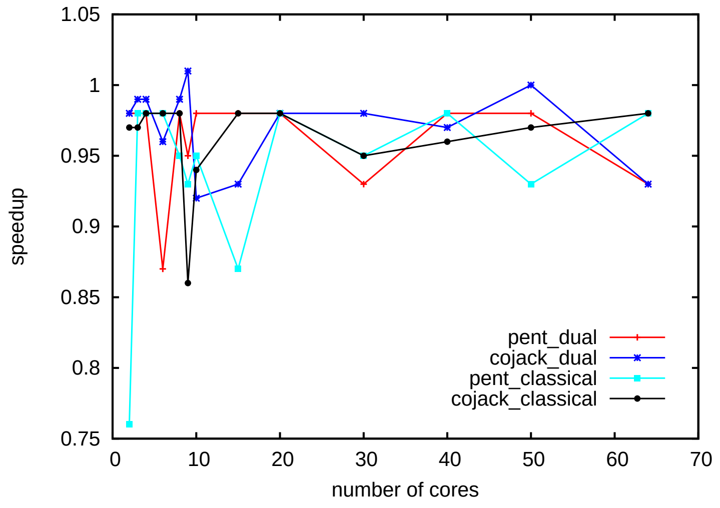


Figure 11: speedup in finalization phase

Looking at the speedup of the three phases respectively, we can conclude that there is no speedup in initialization and finalization phases, which is reasonable because what we actually parallelize is the computational part.

In the initialization phase, the execution time using METIS is almost 5 times the time using classical partition, because calling the METIS function takes a lot of time and some communication overhead.

As we can see, there shows nearly a linear speedup for both input files using METIS as a partition strategy in the computation phase, while the ones using classical partition have a much lower speedup mainly due to the excessive communication time. The speedup of pent input file using METIS begins to drop after 30 cores and the one of cojack input file using METIS begins to drop after 40 cores mainly because as the number of cores is increasing, the communication overall is increasing due to larger interface between partitions.

By checking the finalization phase, the speedup is almost 1, because in this phase, I still use only one core to generate the final results.

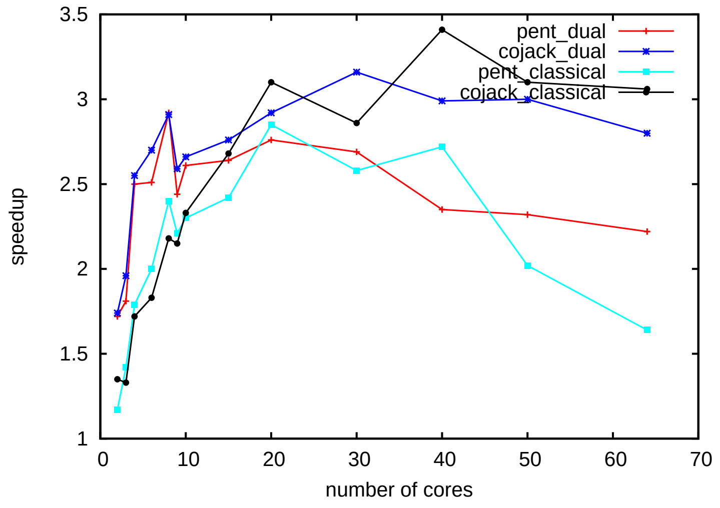


Figure 12: speedup in total phase

For the total phase speedup, locally, the speedup is higher using METIS partition is higher than classical partition if the number of running cores is less than 20. With the increasing number of cores, the advantage of using METIS is not that obvious. Both partition strategys have very low speedup because both of them have their own overhead. For METIS, doing initialization takes a lot of time. And for classical, computation is a time-consuming part. I think for much larger files which need a lot of computation, the METIS partition will show much more apparent advantages.

5 Overview

In this lab course, I got an overview of how to exploit the performance of framework. Including sequential code optimization, parallelization of the code and tunning the parallelization code to a higher stage. I think there are still potentials to enhance the performance in the initialization phase in my code. Maybe substitute the $MPI\_Bcast()$ with $MPI\_Send()$ and $MPI\_Recv()$ can lower the communication time to some extent.