PROGRAMMING OF SUPERCOMPUTERS

FINAL REPORT

Srdjan Krivokapic - 03646772

Kunal Mody - 03653987

ASSIGNMENT 1

The first assignment aimed at understanding the different parameters that influence the optimization of a single-core code. To optimize a code in a single core environment, it is necessary to understand the Code structure and the Architecture of the Machine where the simulation will be performed. We make use of different tools to understand these things.

First, we understand the Architecture of our Machine, SuperMUC in this case. To exploit the system to its peak performance, we study the node and core layout, processors and cache specifications.

When the code was first executed on the thin node of SuperMUC, the performance was very low. The first step toward optimization of the system was the Compiler-based optimizations. Different flags were used and vectorization reports were generated to check the present code vulnerabilities.

To understand the performance characteristics, we made use of the PAPI tool. This allowed us to see the Cache Miss rates, Floating Point Operations (FLOPs) and Execution time. We also analyzed the effects of various flags on the Cache misses, FLOPs and execution time.

Problem Encountered:

1. **Incorrect use - PAPI_FP_OPS**

   Reason - We had originally used the PAPI_FP_OPS floating point event for calculating the FLOPs, this event calculated vector as a single operation instead of the actual number of operations.

   Solution – Use PAPI_DP_OPS.

2. **Limit to number of PAPI_Events**

   Reason – The hardware cannot count more than a certain number of events. Hence, we had limitations of the parameters we could calculate. This led us to initially using 2 different versions of *gccg.c* to accommodate both, the cache rates and FLOPs.

   Another Solution – PAPI_multiplex functionality.

Observations:

1. Use of Flags -- Vectorization

The *–vec-report* flag gives a better idea about the vectorized and non-vectorized loops along with the dependencies. This gives us a better perspective of the loops in our code.

-xhost is a very useful flag when the code has been written for a particular architecture and is being run on the same system as it was designed for.

-O3 flag offers a more aggressive vectorization and hence, achieves faster results than –O1. However, the –O3 flag must be used carefully as the forceful vectorization can lead to undesired results.

2. Cache Size Influence

The input file size has a major influence on the run-time due to the memory hierarchy and cache sizes. If the size is very small, it can fit in the cache, it enables fast calculations. If the file fits into the L3 cache, the speed and efficiency is high. In this memory, the bandwidth and transfer rate is not a bottleneck.

When we use a file bigger than the L2 + L3 cache size, the data needs to be replaced in the caches by transfer from external memory. This process causes a loss in efficiency and considerable decrease in performance.

The efficient use of the caches is very important as it affects performance. Therefore, one of the aims when optimizing is to reduce the cache miss rate by optimal use of data already available in the cache.

3. Performance Achieved

Owing to all the bottlenecks and inefficiencies, the achieved performance was only 10% of the Peak performance. This is a major concern and indication for the need of code refinement and the need to better utilize the resources.

4. Binary Files

The encoding in a regular file increases the time for reading and interpreting the data. This can be changed by providing binary files that can be easily read by the machine. The performance increase is significant and this can be very well agreed by the results in Assignment1.

All of these inferences are based on the results obtained in the first assignment.

ASSIGNMENT 2

The second assignment dealt with parallelization and optimization of the present serial code. The assignment was divided into various milestones to provide a step-by-step approach to build on the progress toward a final parallelized efficient code.

MILESTONE 1

Milestone 1 involved the distribution of a decomposed domain and the approaches to do the same. The details of the task and problems is listed in the following sections.

- **DOMAIN DECOMPOSITION**

The first step toward a parallel code is the division of the computational domain. The domain decomposition was achieved by two algorithms.`

1. **Classic Distribution**

   The straightforward decomposition where the number of cells are divided equally to all processes. First k cells to the first process, next k cells to the second, and so on. In case of any residual, the additional cells are given to as many processes.

2. **METIS Distribution**

   METIS is a sophisticated tool used to partition graphs and meshes. It uses complex algorithms to partition the cells depending on various parameters to achieve optimized results. We made use of the following two partitioning algorithms.

   A. ***DUAL*** - This algorithm emphasizes on reducing the *total number of edgecuts* in the graph.
   B. ***NODAL*** - This algorithm emphasizes on reducing the *total communication* volume.

- **DISTRIBUTION APPROACH**

The partitioned elements need to be distributed to their respective processes. This is done using two approaches. One approach is allowing one process to decompose and distribute cells to all other processes correctly. This is the Oneread approach. In the other approach, each process gets the entire graph and partitions it and selects the cells corresponding to its own process only. This is the Allread approach.

There is a certain advantage and disadvantage to each approach as shown in the table below.

	ONEREAD	ALLREAD
Input Data available in	1 Process	N Processes (Memory: N * FileSize)
Partitioning performed by	1 Process	N Processes (Work: N times)
Distribution of Data	1 -> N (1 Process needs to gather and send elements of the respective other process)	No Distribution (Selecting elements as partitioned for respective process)

As we see in the table above, Oneread needs less memory and only performs the partition once. In contrast, Allread needs N times the memory and performs the partition N times.

However, Oneread only needs to filter the partition results and scatter respective elements to the processes. But Allread does not need any scattering, only selection of its respective elements.

Main Challenges:

1. **Use METIS correctly**

   METIS partitioning requires many parameters to be passed to the function call. These variables need to be defined and passed correctly. Few important parameters are highlighted below:

   ncommon - Specifies the number of common nodes that two elements must have in order to put an edge between them in the dual graph. We have used the value 4 because there are 4 nodes between a common face in a cubic framework.

   eptr – Vector with the pointer to the next element in the graph.

   eind – Contains the list of 8 nodes for each element as pointed by eptr.

   epart – Vector which stores the process number for the respective element.

   As we see, the METIS set-up can be tricky and it is important to understand its parameters and apply them correctly for a good partition.

2. **Distribute Cells Correctly**

   The *epart* output from the METIS partitioning indicates the respective process number for the particular cell. This is illustrated clearly in the next diagram.

CELL INDEX No.	…	**10**	**11**	**12**	**13**	**14**	**15**	**16**	**17**	**18**	**19**	**20**	**21**	**22**	**23**	…
RESPECTIVE PROCESS	…	0	1	1	0	2	2	3	0	2	2	0	0	3	3	…
GLOBAL LOCAL INDEX	…	3	2	3	4	1	2	1	5	3	4	6	7	2	3	…

Now, to distribute these elements correctly, we first need to know how many elements are there for each process and then collect and scatter these elements to the respective process.

This appropriate distribution will lead us to the *local_global_index* and global_local_index which is important to map all the element properties from the global to the local scale.

LOCAL ELEMENT INDEX	…	**3**	**4**	**5**	**6**	**7**	…
GLOBAL ELEMENT INDEX	…	**10**	**13**	**17**	**20**	**21**	…
PROCESS	…	0	0	0	0	0	…

MILESTONE 2

Milestone 2 involved the setting-up of the send and receive list for the communication between the distributed domains. The distribution of the domain leads to many shared/common boundaries. The calculation from one domain needs to be transferred to the neighbour to achieve continuity. For this reason, many cells need to be updated in the neighbouring domain. To assist this process, we set up a send and receive list. This list represents all the communication necessary to achieve the same effect as a continuous domain.

Main Challenges:

1. **Define Ghost Cells**

As we have now set up subdomains, boundaries are redefined for each subdomain. These (local) boundaries now include both real boundaries and ghost cells. Their definition is done with the help of LCC array, an array that lists all the six neighbours of an element. So, going through the neighbours of all the elements, we are looking for the ones, that are outside the considered domain. If they are not real boundaries, they represent the ghost cells and are added to the Receive list.

For the neighbours inside the domain, LCC needs to be adjusted to the local indexing. The change is performed during previously described process.

2. **Define Send List**

In order to be able to exchange data in the computation phase, we introduce a Send list. For a pair of neighbouring subdomains, Send list of the one should contain the same elements as Recv list of the other.

In this first solution, the natural approach was to define Send list at the same time as Receive list for one particular subdomain. This would mean that the elements, we add to the lists at a time are neighbours and would lead us to a different order in the Send - Receive list pair.

Hence, additional sorting of list will become necessary after the set-up of the lists

MILESTONE 3

Milestone 3 involved the final step to complete the parallelization process. The Send and Receive list created in the previous milestone is now used in the *compute_solution.c* to calculate the variables while maintaining the continuous domain. As discussed earlier, we will need to synchronize the data that is common to two subdomains.

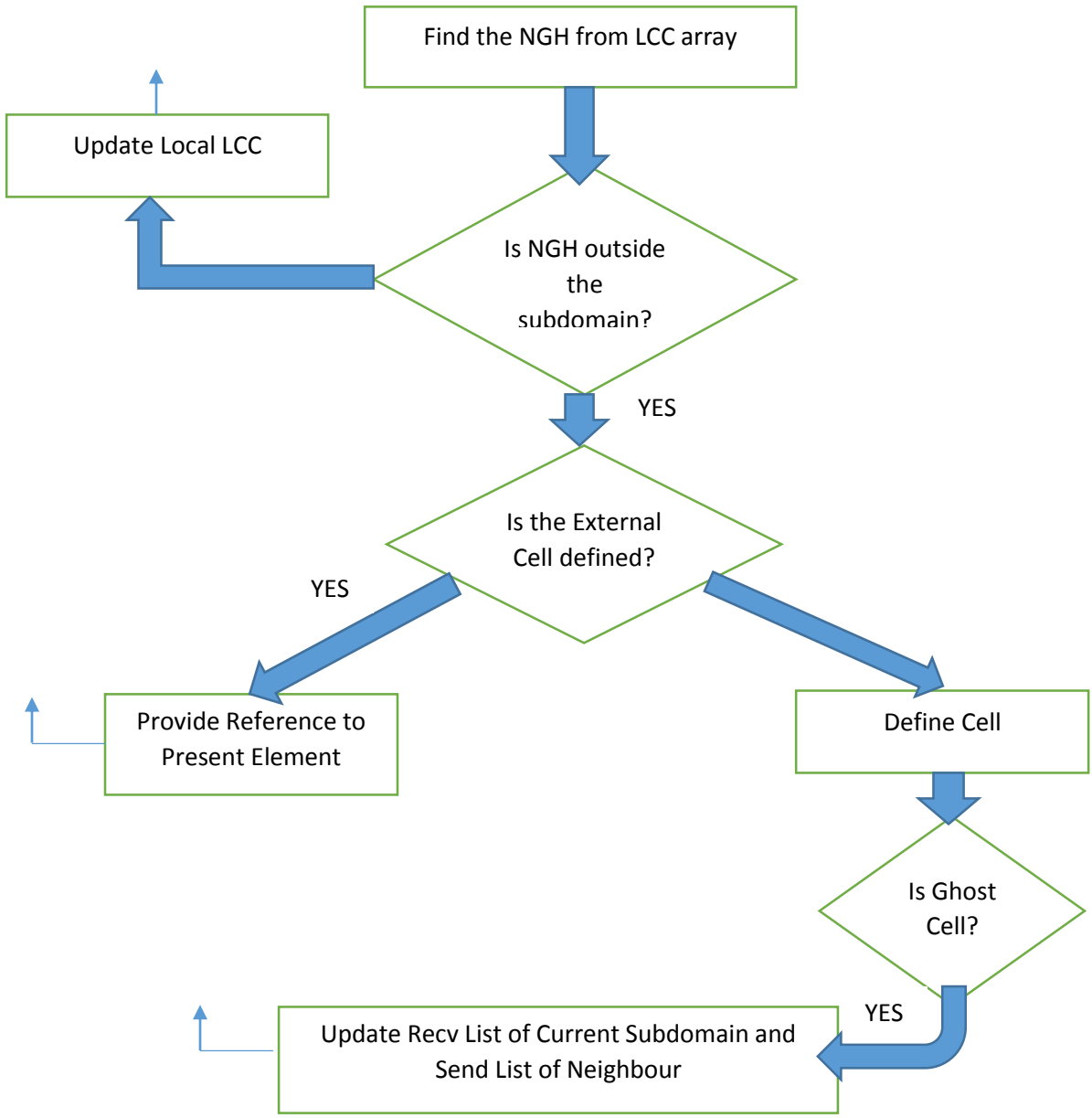
Change Of Approach:

1. **Re-Defining Send & Receive list**

   The definition of send list in one subdomain and the corresponding receive list in the neighbour was not arranged in the same order. Hence, we needed to reorder the list to have an exact Send and Receive list.

   Another approach to this problem would be to implement a code that uniquely arranged the ghost cells into the Receive list of the present subdomain and Send list of the neighbouring subdomain. This ensures that the Send and Receive list of one subdomain matches exactly with that of its neighbour.

   The algorithm adopted can be schematically described in the following flowchart.



The change of approach gave us a list that was ordered and exactly corresponded with the neighbouring domain without any further change of code. This set up the next tasks in this milestone.

Main Challenges:

1. **Communicating Ghost Cells**

   The Send and Receive list of a domain now need to be transferred to the neighbour to assist in the calculation of the *direc2* array. MPI provides different blocking and non-blocking functions to communicate between the processes.

   A. Send & Recv Buffers

   The first approach was to use buffers to communicate all the values between processes. This was achieved straightforward as we had already obtained a list that exactly corresponded.

   B. Blocking or Non-Blocking Communication

   In the communication model, we chose the simplest approach by using the blocking communication. MPI_Sendrecv() is the most commonly used communication model.

2. **Synchronizing the Computation Phase**

   In the computation phase, the calculation in the while loop is carried on until we have a certain residual. The residual is calculated by certain parameters that depend on all elements in the domain. Thus, we need to sum up certain parameters and make them available to all the processes.

   To ensure that all the processes performing the calculation don't have different values of the residual (parameters to calculate the residual), we need to communicate and synchronize these values in all of the processes.

   To achieve this, we use the MPI_Allreduce() functionality. This allows us to sum up all the values in different processes and provide the total sum in all of the processes.

3. **Reassemble VAR array**

   After the convergence is achieved, other than a few variables, rest all are only local. To present the solution as continuous, we need to reassemble the array. To achieve the same value as the VAR array from the sequential implementation, we need to gather the VAR values from each subdomain and arrange them properly using the *local_global_index*. It is important to re-order the elements to their own respective positions.

To sum up the procedure, we first set up the buffers for communication using Sendrecv() functions. Next, at the beginning of every iteration, we communicate all the neighbouring boundaries. Then we ensure correct synchronization of the variables by using the Reduce functions. This procedure goes on until we reach desired convergence. The total iterations indicate the number of iterations to reach convergence.

Finally, the synchronized and reassembled variables are presented as output.

MILESTONE 4

Milestone 4 was to analyze the previously completed code, improve the performance and achieve better results. The milestone aims at using the profiling and tracing tools to better visualize and understand the implementation of the code.

The tools used in this assignment include:

1. SCOREP
2. CUBE
3. VAMPIR/8.0
4. PAPI
5. VALGRIND

Note: The execution time values obtained from simulations with SCOREP were very different from those obtained using PAPI. We assume that this is due to the overhead in compiling and execution in SCOREP environment. To verify our results, we used MPI_Wtime() and PAPI instructions. We realized that the values from SCOREP were different and lead to different calculations of Speedup. We have used the Time measurements with PAPI but some SCOREP measurements are also highlighted (but not used as reference). In places where SCOREP was exclusive, we have tried to avoid absolute values and instead give percentage values to understand the significance better.

STEP 1: CHECK OLD CODE FOR BENCHMARK

The first task was to check the previous code for its performance. The benchmark was the PENT DUAL configuration with linear speedup until 8 processors. The original code with MPI_Sendrecv() and Buffers was almost linear until 8 processes. And hence, we needed to analyze the code better and reduce the bottlenecks.

STEP 2: COMPILER BASED OPTIMIZATION

Vectorization is the first step toward an optimized code. To start preliminary measurements, we used the –O1 and –O3 flags. This allowed us to have a better understanding of our code with compiler based optimizations. The vectorization gave us faster results without any effect on the code results.

With the help of –vec-report=5, we created reports to check for the non-parallelizable loops. We tried to remove the dependencies in the loops, however, we were not successful in changing the vectorization potential.

OPTIMIZATION 1 – Non-Blocking Communication

IDEA – Blocking communication always is a slower option. The function call waits for all the data to be transferred from the send buffer to the system buffer and does not return until a matching Receive call is made or the data is stored somewhere. In contrast, the Non-Blocking call don't wait for the communication to be finished. They return immediately and are available for re use. Though the Non-Blocking can be tricy, if used properly, it can lead to better results.

IMPLEMENTATION – Change of the MPI_Sendrecv() call to MPI_Isend() and MPI_Irecv(). To ensure the function has completed, we made use of the MPI_Wait() call. This ensured that even though the function is non-blocking, the call is completed.

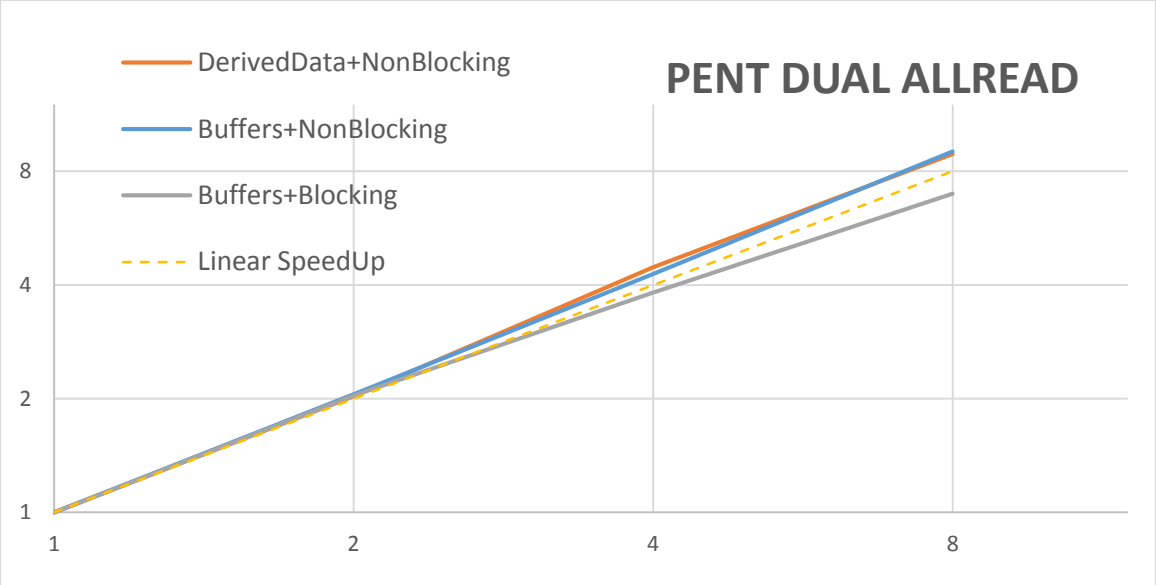
RESULT – The code was faster than the previous version. The performance achieved was linear until 16 processes.

OPTIMIZATION 2 – MPI Derived Datatypes

IDEA - We were made aware of the different techniques to achieve better communication in MPI. This was done by setting up a contiguous block of memory for the data exchange between two processes. MPI provides certain functions that facilitate this formation of contiguous block for different and non-contiguous data.

IMPLEMENTATION – To suit our requirements, we used MPI_Indexed_type() function. This allows the non-contiguous block to be suitably presented as a contiguous block. The derived data type required certain parameters to be correctly passed and the new datatype being committed.

RESULT – We hoped, the derived data type would lead to better results. Surprisingly, the results obtained with the derived data type was not as good as those with previous implementation of buffers. The speedup however, was as good as with Buffers but the Wall clocktime for the functions was slower.



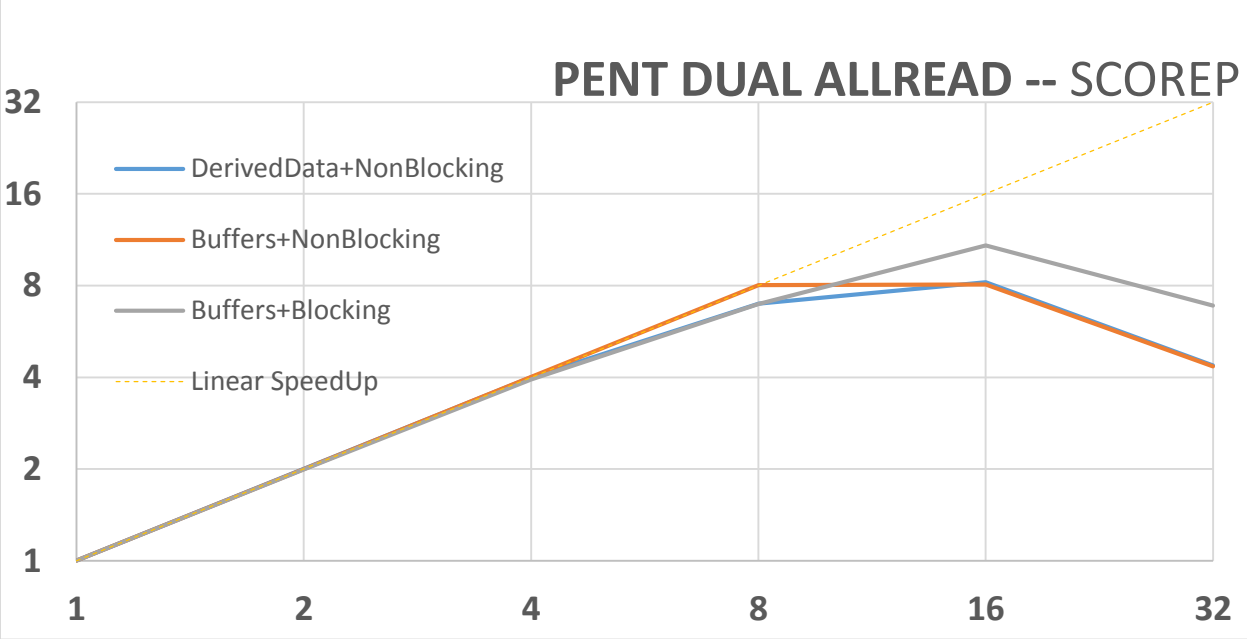
The above graph shows the different possible combinations for the code with Optimizations 1 and 2. As we see, the code with Blocking communication is quasi-linear and hence not good enough. The 2 configurations with Non-Blocking communication are providing linear speedup. However, the code with Derived datatype is slower than the Buffers. This leads us to select the code with Buffers for the Data and Non-Blocking function for Communication as our primary code for analyzing.

Configuration	**AVG TIME 3 Runs (sec) compute_solution.c PENT DUAL ALLREAD**			
No. of Processes	1*	2	4	8
Derived Datatype + Non-Blocking	1.87738	0.925876	0.421233	0.2112133
Buffers + Non-Blocking	**1.80035**	**0.876398**	**0.420843**	**0.1994143**
Buffers + Blocking	1.84493	0.905980	0.483240	0.2645773

*PENT CLASSIC ALLREAD

Results are available in the data/optimization2/2ndOptimization.xlsx.

Note: The tests were also conducted using SCOREP and similar results were achieved as shown below. However, the linear speedup was only achieved with Buffers and Non-Blocking configuration. Hence, we selected this code.



QUESTION 2: MINIMUM PERFORMANCE REQUIREMENT

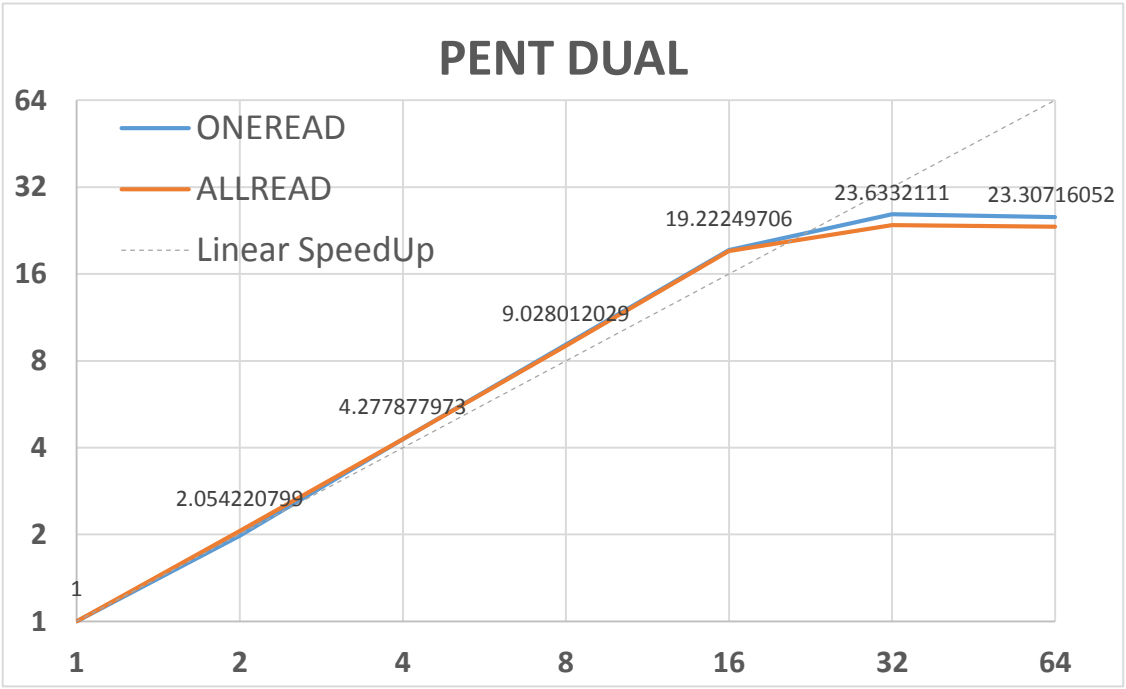
In the first part of the milestone, we need to obtain a certain minimum requirement. This we had also used as our Benchmark. The results below are presented for the PENT DUAL – ONEREAD and ALLREAD configuration. The results were obtained until 64 processes. The speedup achieved was linear until the 16 processes.

	Time (sec) – compute_solution.c PENT DUAL ONEREAD						
No. of Processes	**1***	**2**	**4**	**8**	**16**	**32**	**64**
Run1	1.78349	0.909252	0.420839	0.197146	0.09292	0.069748	0.071484
Run2	1.8123	0.909117	0.420701	0.197303	0.09292	0.0697	0.071517
Run3	1.80164	0.909011	0.420648	0.197274	0.09295	0.069742	0.071474
Avg	1.7991433	0.90912667	0.420729	0.197241	0.09293	0.06973	0.0714917
SpeedUp	**1**	**1.99897983**	**4.276249**	**9.121548**	**19.3599**	**25.80157**	**25.165777**

*PENT CLASSIC ONEREAD

	Time (sec) – compute_solution.c PENT DUAL ALLREAD						
No. of Processes	**1***	**2**	**4**	**8**	**16**	**32**	**64**
Run1	1.835356	0.876165	0.421081	0.20064	0.093238	0.076437	0.077012
Run2	1.755474	0.876771	0.420666	0.198969	0.093911	0.07658	0.077424
Run3	1.810115	0.876258	0.420782	0.198634	0.093821	0.075515	0.077293
Avg	1.800315	0.876398	0.420843	0.199414	0.093657	0.076177	0.077243
SpeedUp	**1**	**2.0542208**	**4.277878**	**9.02801**	**19.22249**	**23.63321**	**23.3071**

*PENT CLASSIC ONEREAD



Observations:

The graph achieved linear speedup until 4 processes and after that it was superlinear until 16 processes. This would be because of the cache related discussion in Assignment 1. The file size when divided between 8 and 16 processes would be leading to sizes that easily fit in the L2 + L3 cache, thus, it does not need to transfer and process data from external memory. This leads to easier and faster communication. Until 4 processes, the file part in each process will not fit in the cache and hence data needs to be transferred when required from the external memory.

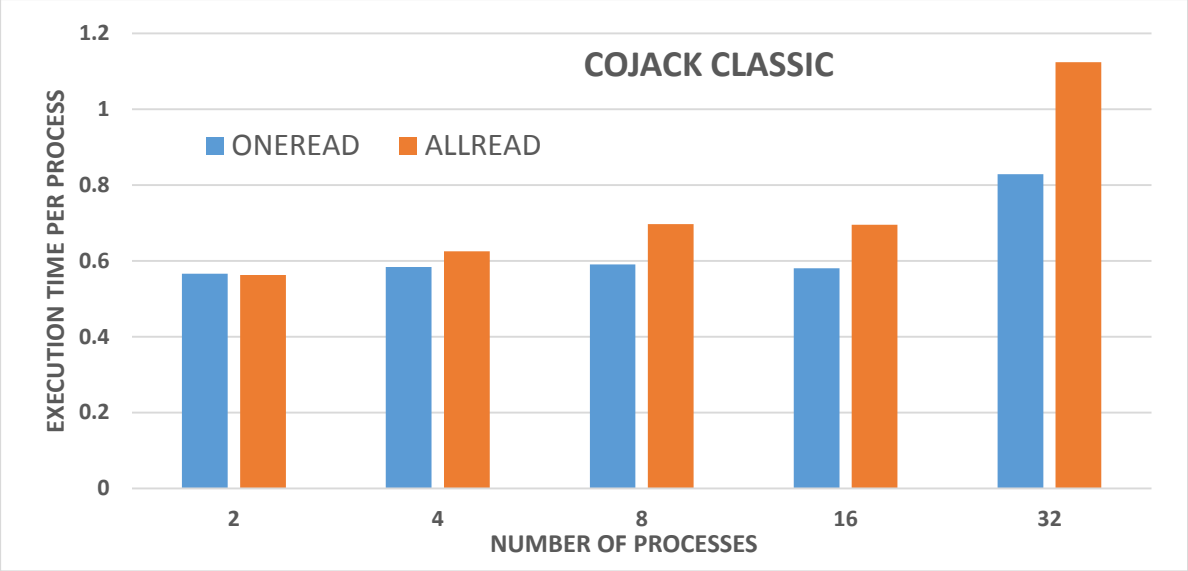
In these stages, the communication does not strongly affect the performance. But after 16 processes, the data in each process is too small and the communication it requires is a lot. This communication overhead causes a decrease in performance. It is strongly influential as the processes increase. The graph decreases sharply for 64 processes, as it not only needs to communicate between many processes but also the processes may be in different nodes.

QUESTION 3.1: SCALABILITY OF ONEREAD & ALLREAD – INITIALIZATION PHASE

The scalability of an algorithm is an important measure for achieving good results for a parallel code. The two approaches used are ONEREAD – where only one process reads all the input data, decomposes it and then distributes it to the others and second, other, ALLREAD – where all the processes read and perform the distribution algorithm but only use values for their own process.

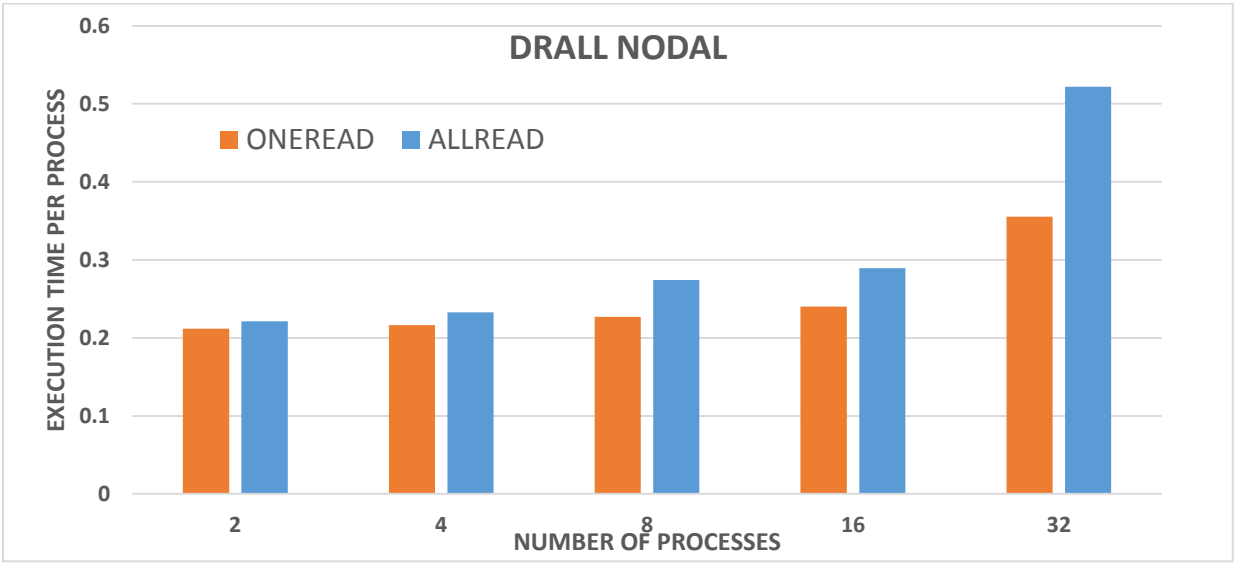
The range and scalability of the two algorithms is summarized in the data tables and their graphs. We have chosen the graphs only for COJACK CLASSIC distribution and DRALL NODAL distribution.

	SCALABILITY – initialization.c EXECUTION TIME PER PROCESS - COJACK						
No. of Processes		**2**	**4**	**8**	**16**	**32**	
CLASSIC	**ONEREAD**	0.566666667	0.584166667	0.590833333	0.580416667	0.829166667	
	ALLREAD	0.562667667	0.625131167	0.696953333	0.695226396	1.123910625	
DUAL	**ONEREAD**	1.828333333	1.869166667	1.905416667	1.871041667	2.510833333	
	ALLREAD	2.257298167	2.64655	3.090487083	3.042744583	4.983386667	
NODAL	**ONEREAD**	1.521666667	1.5625	1.606666667	1.581041667	2.173020833	
	ALLREAD	1.661903667	1.796146667	2.02530125	2.006823854	3.388634375	



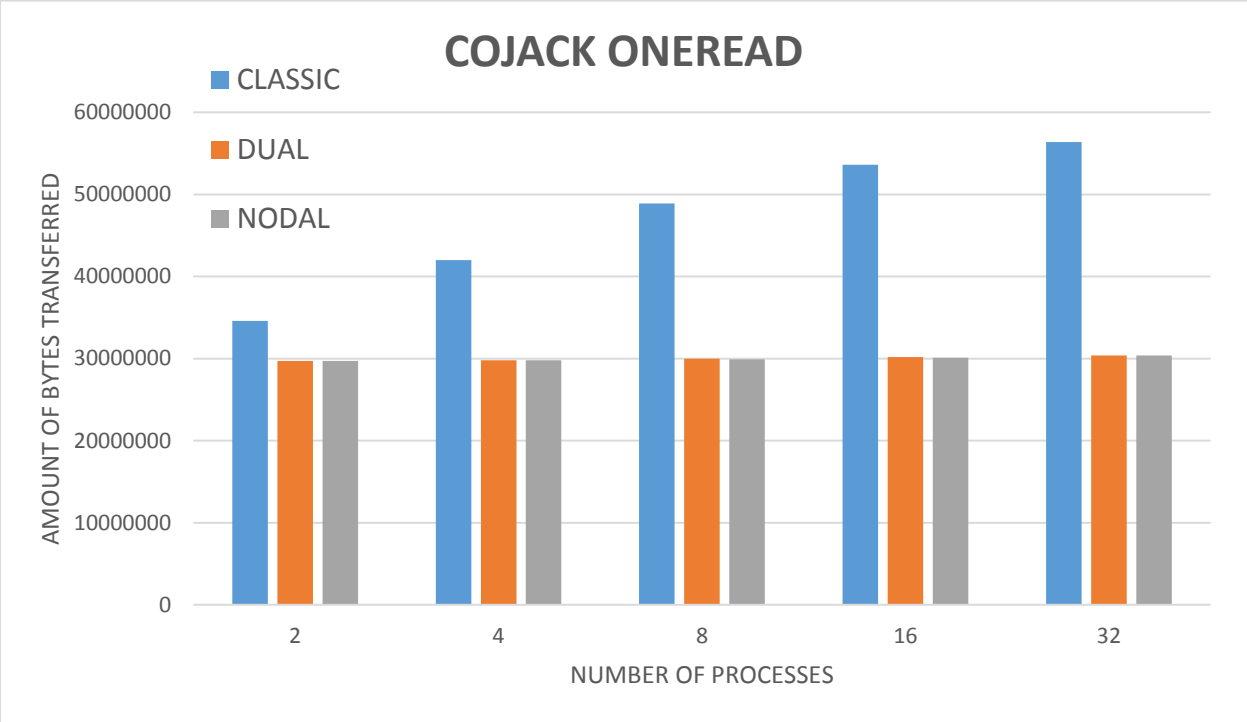
As the results show, the scalability with both the algorithms is good until 16 processes. After 16 processes, the execution time per process increases sharply. The same observation has been made for NODAL and DUAL configuration.

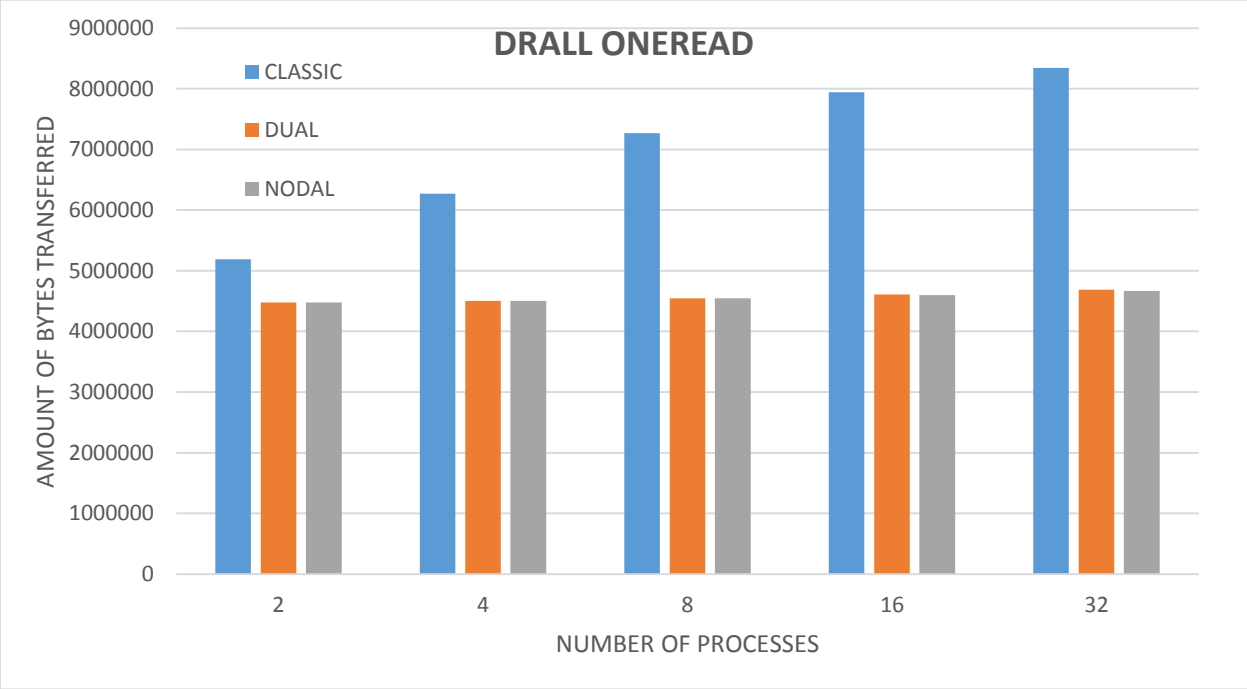
	SCALABILITY – initialization.c EXECUTION TIME PER PROCESS - DRALL						
No. of Processes		**2**	**4**	**8**	**16**	**32**	
CLASSIC	**ONEREAD**	0.083616667	0.085866667	0.085874583	0.085671083	0.121963646	
	ALLREAD	0.083341667	0.0850475	0.086004167	0.085592292	0.1336425	
DUAL	**ONEREAD**	0.212806667	0.216945833	0.224664583	0.229673125	0.346207083	
	ALLREAD	0.217663333	0.238029083	0.301345833	0.303703125	0.5716375	
NODAL	**ONEREAD**	0.21187	0.2161825	0.227030458	0.239940833	0.355240521	
	ALLREAD	0.221361667	0.232584083	0.274180417	0.289295833	0.521897083	



DATA EXCHANGE

During the Initialization phase, ONEREAD performs the transfer of data from the main process to the other processes. However, ALLREAD does not involve any exchange of data as all the work is performed by every process. The results below show the scalability of the ONEREAD algorithm.





For the CLASSIC distribution, we see a constant increase in the data transferred. This would be because of the random distribution of the CLASSIC algorithm. The distribution causes to have neighbours spread among different processes, this leads to the increase in the send and receive list. Thus, we see a constant rise in the data transferred.

For the METIS tools, NODAL and DUAL provide a near constant transfer of bytes from 2 processes until 32 processes. This is because of the sophistication of the algorithm. As detailed earlier, METIS tries to minimize communication or the edgecuts in a graph. This leads to minimum communication. This is reflected in the graph in the above examples.

QUESTION 3.2: SPEEDUP – COMPUTATION PHASE – COJACK & PENT

The computation phase is the most important part for achieving appropriate results. After the parallelization, it is necessary to be able to scale up the files to higher number of processes without affecting performance. As already mentioned, the effect of cache memory and communication greatly influence the performance.

The tables below show the speedup for cojack and pent. All the values are the average runtimes from 3 executions. The data is in sheet data/Data.xlsx/Q3.2.

	SPEED UP – compute_solution.c COJACK						
No. of Processes		**2**	**4**	**8**	**16**	**32**	**64**
CLASSIC	**ONEREAD**	1.7376291	2.3884322	2.89364253	5.368849878	6.743073545	7.34519896
	ALLREAD	1.7274832	2.4136131	2.8913018	5.3734261	6.6722854	6.57351966
DUAL	**ONEREAD**	2.064161	3.9352226	6.77434565	18.28251248	50.54945357	79.5812807
	ALLREAD	2.0734392	3.9572914	6.7895186	17.333438	47.376871	61.9605842
NODAL	**ONEREAD**	2.0087267	3.9301526	6.71906191	16.34086731	51.05541528	77.6027698
	ALLREAD	2.0309831	3.9288367	6.7055262	15.765755	46.809654	67.5726726

The Speed up in the Computation Phase for COJACK is shown above. As we see, for CLASSIC distribution, there is no speedup. There is minor improvement but there is not sufficient gain from the previous configurations.

For the NODAL and DUAL configuration, we have linear speedup until 64 processes. However, the benefit of the speedup isn't available after 32 processes because of the superlinear speedup for 32 processes.

	SPEED UP – compute_solution.c PENT ONEREAD						
No. of Processes		**2**	**4**	**8**	**16**	**32**	
CLASSIC	**ONEREAD**	1.477898	1.8805625	2.55864156	6.295077601	4.840830013	
	ALLREAD	1.4329737	2.0030632	2.5333742	5.7902673	4.4585776	
DUAL	**ONEREAD**	1.849568	4.1131734	9.41452409	19.90813863	26.21475734	
	ALLREAD	2.0723922	4.3919173	9.0734034	18.556625	21.554648	
NODAL	**ONEREAD**	2.0182693	3.944644	9.29755394	19.66424106	27.11470465	
	ALLREAD	2.0264892	4.4444443	8.7486163	19.232442	22.152074	

For the PENT file, the computation phase shows the same characteristics. There is no gain in the CLASSIC configuration. In the METIS distributions, the speed up is profitable until 16 processes where the gain is high. There is a sharp decline for 32 processes. The most probable reason being the communication overhead.

The serial execution time for 1 process is compared with the classic execution with 1 process in the table below.

	SERIAL EXECUTION TIME	CLASSIC 1 PROCESS
COJACK	9.81243 sec	9.766 sec
PENT	1.90341 sec	1.834 sec

The time for serial execution is the same and even a little higher than CLASSIC ONEREAD.

QUESTION 3.3: MPI OVERHEAD – PENT NODAL ONEREAD 8 Procs

The MPI Overhead is the amount of time MPI function calls take in a parallel execution. It is important to understand the overhead as it gives us an insight about the performance gain and result of increasing the number of processes. As already observed in Q2, the effect of MPI overhead greatly influences the performance after a certain increase in number of processes. We have chosen the ONEREAD approach as it includes MPI functions in the initialization function.

A. INITIALIZATION

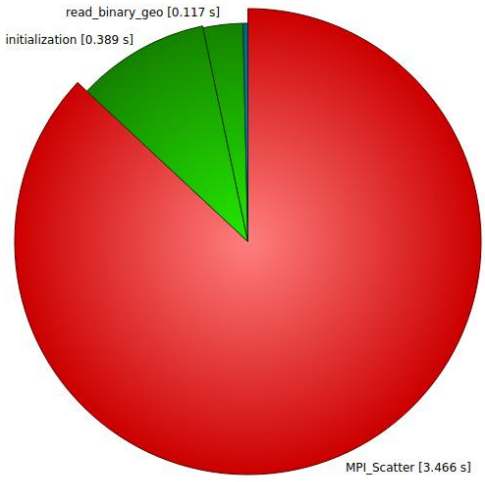
In the ONEREAD approach, one process reads the binary file, then it performs the distribution (in this case METIS Nodal) and finally, distributes all the respective values to other processes.

As we already understand, there is MPI_Scatter() that needs to be used to distribute the elements to the correct processes. In our code, we make use of MPI_Scatter() and MPI_Scatterv() to distribute the data.

INITIALIZATION	**Total Time for 8 Procs***	**4.01976 sec**	
MPI FUNCTIONS			
	MPI_Scatterv	0.01334 sec	
	MPI_Scatter	3.49676 sec	
	Total MPI Time	**3.51010 sec**	
MPI_OVERHEAD			**87.22%**

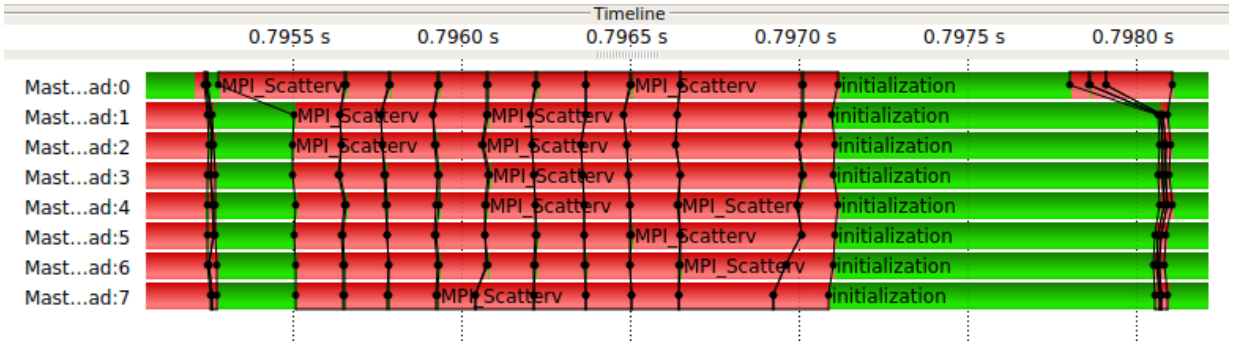
*Not Maximum Inclusive Time

The data has been procured using CUBE and VAMPIR/8.0. Below, is shown the corresponding graph from VAMPIR detailing the MPI Overhead and matching with values from CUBE.



ONE process does all the task until the end when the Scatter operations begin to take place. However, all the other processes await the data but cannot perform any other operations. This leads to a very high overhead and also a reason why ONEREAD becomes slower as compared to ALLREAD.

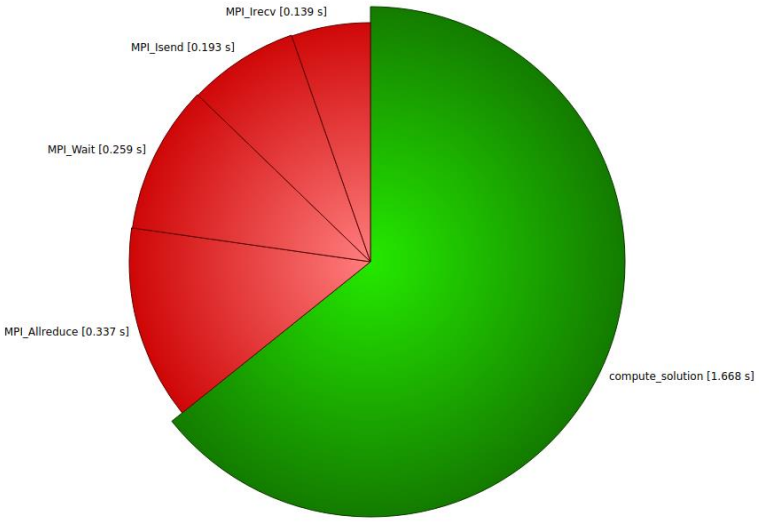
Enlarged view of the Scatterv operation.



B. COMPUTATION

The compute_solution.c is the computationally intensive part with a lot of calculations in a number of iterations. Due to the divided domain, at every iteration, there needs to be an exchange of data. This data ensures that the continuity of the domain is preserved. Also, in the calculation phase of the variables, some variables need to be reduced and communicated to all processes. All these communication overheads account to a certain percentage of the time. The following table shows the time values and later a few graphics to explain the data.

COMPUTATION	**Total Time for 8 Procs***	**2.596 sec**	
MPI FUNCTIONS			
	MPI_Allreduce	0.337 sec	
	MPI_Isend	0.193 sec	
	MPI_Irecv	0.139 sec	
	MPI_Wait	0.259 sec	
	Total MPI Time	**0.928 sec**	
MPI_OVERHEAD			**35.75%**



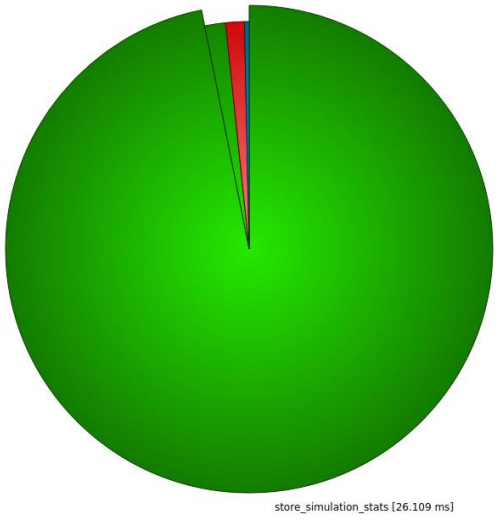
The piechart confirms the results obtained from CUBE and the percentage MPI Overhead.

In this figure, we clearly see the *red* lines indicating the MPI processes and the *green* with the function. We can notice that there is a lot of communication that takes place at regular intervals, however, unlike Initialization, the overhead isn't as large.

C. FINALIZATION

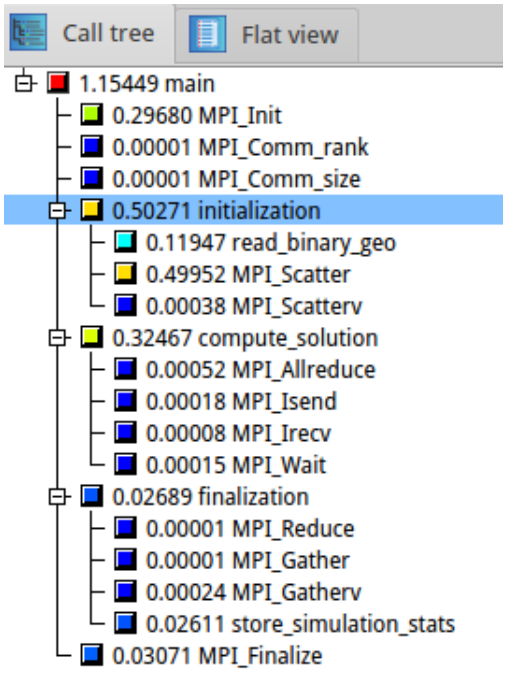
The finalization phase of our code involves two major tasks, one is the reassembling the VAR array and the other is the function call to store_simulation_stats.c which produces the output summary file. However, this is only performed by the main thread and all the others leave the function and go into Main and MPI_Finaliza(). A comparative showing the task for the other threads is also given. It gives us a better idea to understand the waiting time for all processes until process 0 finishes.

PROCESS 0			
FINALIZATION	**Total Time for 8 Procs***	**0.02899 sec**	
MPI FUNCTIONS			
	MPI_Reduce	0.00007 sec	
	MPI_Gather	0.00007 sec	
	MPI_Gatherv	0.00222 sec	
	Total MPI Time	**0.00236 sec**	
MPI_OVERHEAD			**8.139%**



PROCESS 1			
WHILE PROCESS ONE FINISHES FINALIZATION	**Total Time for 8 Procs***	**0.02899 sec**	
MPI FUNCTIONS			
	MPI_Reduce	0.000007 sec	
	MPI_Gather	0.000007 sec	
	MPI_Gatherv	0.000206 sec	
	MPI_Finalize	0.02618 sec	
	Total MPI Time	**0.02646 sec**	
MPI_OVERHEAD			**91.05%**

As a brief overview of just one process, the following figure shows the time division for 1 process (Maximum Inclusive Time) for the above configuration using CUBE.



QUESTION 3.4: ONE ITERATION OF COMPUTATION - PENT NODAL ONEREAD 8

As we have already looked at the computational phase of this configuration, it is easier to discuss the observations from the same configuration. The *compute_solution()* function is divided into 2 parts. First part is the set-up of the buffers for the communication. The second is the while loop which performs the calculation until certain residual ratio is achieved.

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


Input File:  pent
========================================

Output File:  pent_summary.out
========================================

No. of Active Cells:  108000
========================================

Iterations Count: 546
========================================

Residual Ratio: 9.695255e-11
========================================
```

This configuration involves 108000 active cells and 546 iterations that lead to a residual ratio of 9.695255e-11. First, we shall list the sequence of operations in the while loop that iterates until convergence is achieved.

For better understanding the ITERATION loop, we present a summary with labels that we show in the diagram.

WHILE loop {

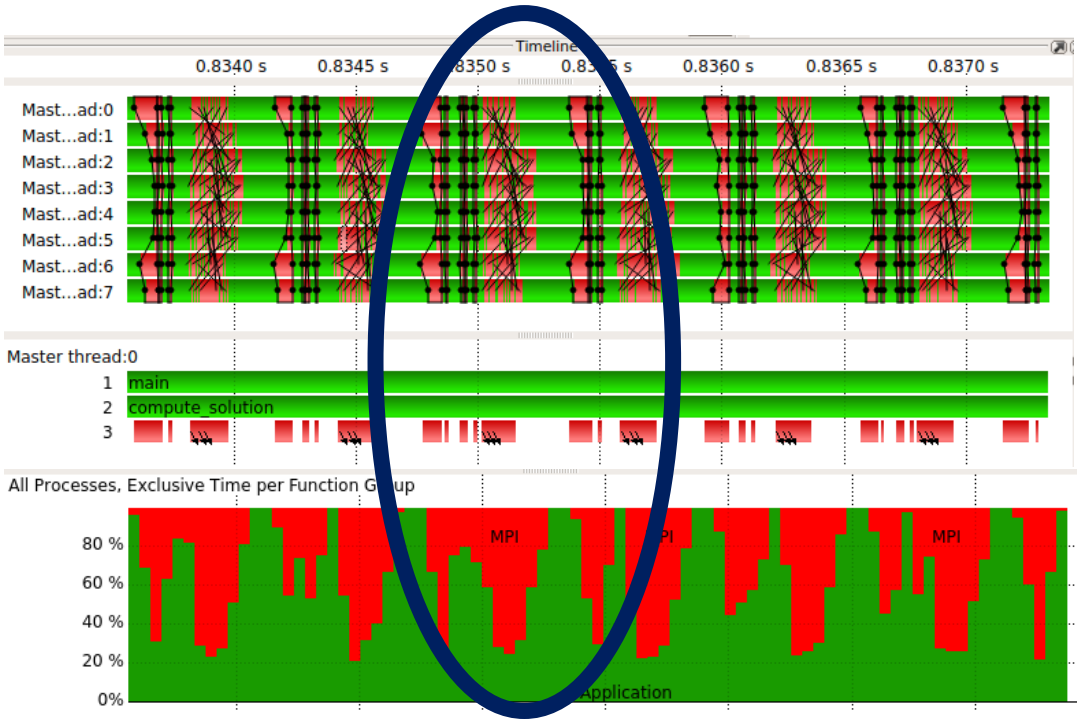
A. Computation Phase 1

1. Calculate ***direc1*** for external cells.
2. Load ***direc1***@*boundaries* into ***sendbuf***.
3. Non-Blocking communication **MPI_Isend** & **MPI_Irecv** with all neighours + **MPI_Wait()**
4. Unloading from ***recvbuf*** to ***direc1.***
5. Calculate ***direc2***.

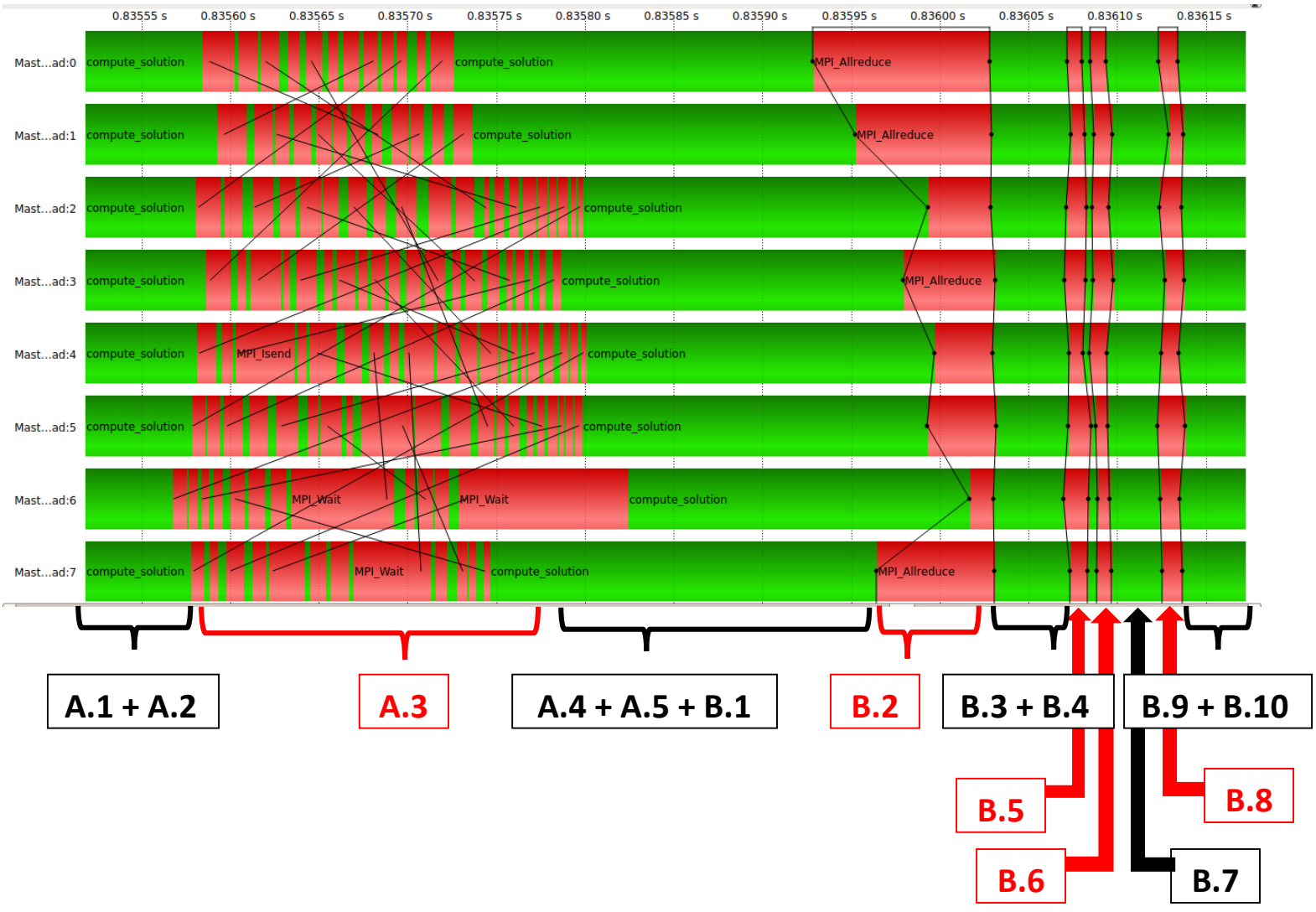
B. Computation Phase 2

1. ***occ*** normalization.
2. **MPI_Allreduce()** for occ. (Could have two instances)
3. Recalculate ***occ***, ***direc1*** and ***direc2.***
4. Calculate ***cnorm, omega, omega2.***
5. **MPI_Allreduce()** for **cnorm.**
6. **MPI_Allreduce()** for **omega**.
7. Calculate ***resvec***, ***res_update*** & ***var.***
8. **MPI_Allreduce()** for ***res_update***.
9. Calculate ***residual_ratio.***
10. Calculate ***dxor*** & ***adxor***. **}**

The diagram below shows the repeated occurrence of a pattern, we can easily conclude that this would be the area where the while loop goes on.



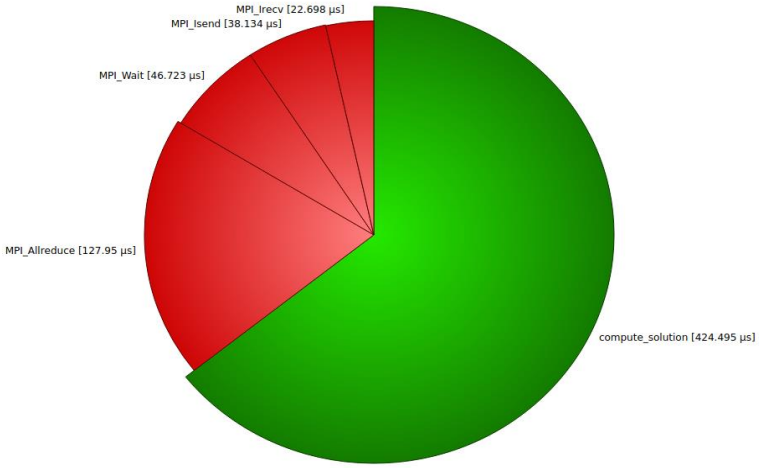
An enlarged view of the recurring pattern is shown below. As we have already listed the operations taking place above. The figure below maps these operations to the timeline.



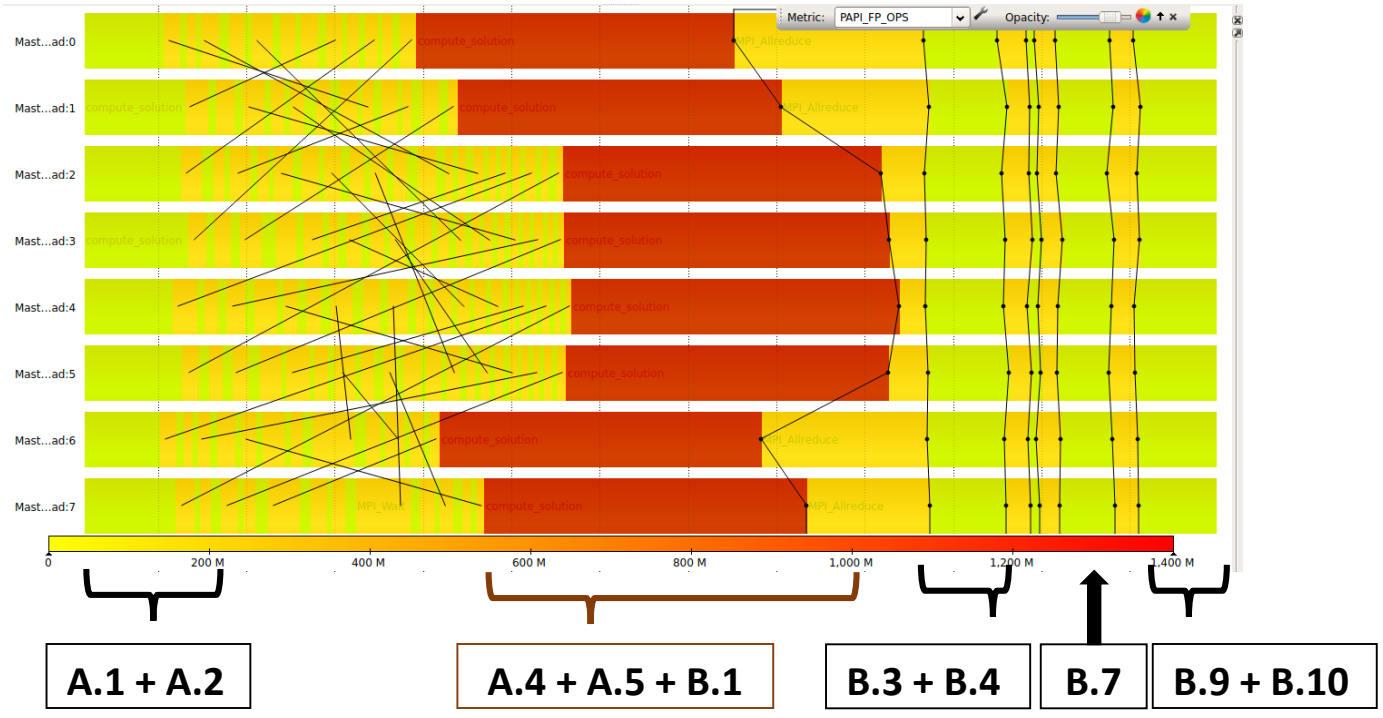
Every operation noted in the while loop has been mapped to the respective timeline. This clearly shows us the calculation phase, the time taken for each step along with the MPI calls. The diagram also shows the transfer of data to different processes.

Below, we have listed the timing for all the steps and MPI to indicate the Overhead.

1 ITERATION	**Total Time for 8 Procs***	**661.999 µsec**	
MPI FUNCTIONS			
	MPI_Allreduce	127.95 µsec	
	MPI_Isend	38.134 µsec	
	MPI_Irecv	22.698 µsec	
	MPI_Wait	48.723 µsec	
	Total MPI Time	**237.505** µsec	
MPI_OVERHEAD			**35.84%**



The MPI Overhead in one iteration is nearly the same as that for the entire computational loop.



As we observe, the most FLOPS are in **A.4 + A.5 + B.1** which is the unloading to receive buffer, calculation of **direc2** and **occ** normalization. If we take a closer look at the code, we observe that the most computationally intensive part would be **direc2** calculation as it accesses many elements and performs many calculations.

The areas with the least FLOPS are **A.1 + A.2 + B.3 + B.4 + B.7 + B.9 + B.10.** As we illustrated in the above while loop, these are the calculation of **direc1**, loading direc1 to **buffers**, **occ, direc1, cnorm, omega, omega2, resvec, res_update, residual, var, dxor & adxor.**

OPTIMIZATION 3 – RESTRUCTURING SEND & RECEIVE LIST

The exchange of data is greatly affected by how the ghost cells are ordered. To improve the performance of the code and exchange of data, we rearrange the ghost cells in such a way that the first *k* cells correspond to the exchange with process 1, next *k* cells with process 2 and so on. To achieve this, we also make changes in the LCC array to correctly arrange the new grouping. The receive list now contains ghost cells with the above mentioned arrangement.

We expected that the code with this new arrangement will be faster and much simpler to communicate hence reducing the overhead and access time.

However, the code would execute correctly on local machines in all configurations but did not give correct results when submitted on the SuperMUC. We eventually could not debug the problem and hence, no measurements from this Optimization are available.