Distributed Memory

(Parallel Sorting by Regular Sampling)

Saeed Sarabchi
Dept. of *Computing Science, University of Alberta* *sarabchi@ualberta.ca*

Abstract— In this technical report, the details relating to implementing Parallel Sorting from Regular Sampling algorithm in a distributed memory framework would be elaborated. Plus, some experimentation and analysis on its performance will be provided. The goal is to assess the performance improvements when Distributed Programming style is used.

I. Introduction

This report is comprised of two sections. Throughout the first section, the implementation details are going to be discussed. In the second part, the performance aspects of Distributed Memory PSRS algorithm will be investigated, in detail and phase by phase. Moreover, the analysis on speedup graphs with variable input size will be elaborated.

II. Implementation Detail

In this project, the algorithm is implemented in C programming language on Linux, using the MPI library and is compiled and run using MPICH. the algorithm used in this project is based on [1]. It is comprised of 4 major phases which will be discussed in detail.

Initialization:

In the initialization part, random numbers are generated and scattered among the processes by the first process (MASTER process), since it is assumed that the data to be sorted is already distributed before the timing is begun:

```
   MASTER
   {

  GenerateData = (int*)malloc((NUM_GLOBALINPUTDATA)*sizeof(int));

for( counter =0; counter < NUM_GLOBALINPUTDATA; counter++ )
          GenerateData[counter] = random() %MAX_NUMBER;


     int LocalSize = NUM_GLOBALINPUTDATA/numprocs;
     int scounts[numprocs];
     int displs[numprocs];
```

```
     for(counter=0;counter<numprocs;counter++)
     {
           displs[counter]=counter*LocalSize;

           if(counter!= numprocs-1)
                  scounts[counter]=LocalSize;
           else
                  scounts[counter]=NUM_GLOBALINPUTDATA-
displs[counter];
     }


     InputData= (int *)malloc((scounts[myid])*sizeof(int));


     MPI_Scatterv(
                  GeneratedData,
                  scounts,
                  displs,
                  MPI_INT,
                  InputData,
                  scounts[myid],
                  MPI_INT,
                  0,
                  MPI_COMM_WORLD);
```

Phase1:

In this phase, the chunks of distributed data are sorted by each process using quicksort library function and regular samples are extracted:

```
    //for capturing start and end time
   //startTiming
      MPI_Barrier(MPI_COMM_WORLD);
      MASTER
      {
            start = MPI_Wtime();
      }

  // Phase 1 :
  // Sorting Chunks of Data which is Related to the Process
      MASTER
      {
            phase_start = MPI_Wtime();
      }


qsort(InputData,scounts[myid], sizeof(int), compare);

// Performing the Regular Sampling
FillRegularSample(RegularSample, InputData, myid, scounts[myid]);




              MPI_Barrier(MPI_COMM_WORLD);
              MASTER
              {
                     phase_stop = MPI_Wtime();
                     t_phase1=phase_stop-phase_start;
              }
              // End of Phase 1
```

The function `FillRegularSample` is as follows:

```
void FillRegularSample(int *SampleArray, int* inputDataArray, int
ThreadID, int size)

{

          int LocalSize = NUM_GLOBALINPUTDATA/(numprocs);
          // Omega in (n/p^2) in the original paper
          int Omega =  LocalSize/numprocs;

          int counter;

for(counter=0;counter<= size; counter = counter + Omega)
{

SampleArray[((counter)/Omega)] = inputDataArray[counter];

}

}
```

Phase 2:

In the Second phase, Regular Samples are gathered from the processes and it is sorted by the MASTER process, and broadcasted to all of the processes:

```
Phase 2
Gathering and Sorting Regular
Sample, Extracting Pivots and Broadcasting them
                  */
MASTER
{
phase_start = MPI_Wtime();
}
MASTER
{
          GatheredRegularSample =
(int*)malloc(( numprocs*numprocs)*sizeof(int));
}
          MPI_Gather(
                    RegularSample,
                   numprocs,
                   MPI_INT,
                   GatheredRegularSample,
                   numprocs,
                   MPI_INT,
                   0,
                   MPI_COMM_WORLD);

MASTER
{

qsort(GatheredRegularSample,numprocs*numprocs, sizeof(int), compare);
ExtractPivots(Pivots, GatheredRegularSample);
}
                    MPI_Bcast(
                              Pivots,
                              numprocs-1,
                              MPI_INT,
                              0,
                              MPI_COMM_WORLD);



               MPI_Barrier(MPI_COMM_WORLD);
               MASTER
               {
                     phase_stop = MPI_Wtime();
                     t_phase2=phase_stop-phase_start;
               }
                    // End of Phase 2
```

The function `ExtractPivots` is as follows:

```
void ExtractPivots(int *inputPivotArray, int* SamplingArray)
{
          int counter;
          int Phi = numprocs/2;

for(counter= (Phi+numprocs-1); counter<=numprocs*numprocs; counter =
counter + numprocs)
inputPivotArray[(counter-(Phi+numprocs-1))/numprocs] =
SamplingArray[counter];

}
```

Phase 3:

In this phase, each chunk of data is partitioned using the pivots, and the partitions are exchanged to their related process:

```
/*
Phase 3 :
Partitioning the data and Exchanging the Data
                    */
                MASTER
                {
                       phase_start = MPI_Wtime();
                }
                PartitionDisps =(int *)malloc((numprocs)*sizeof(int));
                PartitionCnt =(int *)malloc((numprocs)*sizeof(int));

          FillPartition(Pivots, InputData, myid, 0, scounts[myid]);
              int Rcvdispls[numprocs];

int RcvCnt[numprocs];
          MPI_Alltoall(
                    PartitionCnt,
                    1,
                    MPI_INT,
                    RcvCnt,
                    1,
                    MPI_INT,
                    MPI_COMM_WORLD);

int sum=0;
for(counter=0;counter<numprocs;counter++)
          sum+=RcvCnt[counter];
OutputData =(int *)malloc((sum)*sizeof(int));

Rcvdispls[0]=0;
for( counter=0;counter<numprocs-1;counter++)
{
          Rcvdispls[counter+1]=Rcvdispls[counter]+ RcvCnt[counter];
}

          MPI_Alltoallv(
                    InputData,
                    PartitionCnt,
                    PartitionDisps,
                    MPI_INT,
                    OutputData,
                    RcvCnt,
                    Rcvdispls,
                    MPI_INT,
                    MPI_COMM_WORLD);



MPI_Barrier(MPI_COMM_WORLD);
                MASTER
                {
                       phase_stop = MPI_Wtime();
                       t_phase3=phase_stop-phase_start;
                }

                //END of Phase 3
```

The Function `FillPartition` uses BinarySearch in order to locate the pivots, and accordingly, partitions the local data. This function is as follows:

```
void FillPartition(int* PivotArray, int* inputDataArray, int
ThreadID, int StartIndex, int Endindex)
{
// In this Function we Partition each Thread's Data to (#Threads-1)
Chunks of Data.

int PivotCounter;
PartitionDisps[0]=0;
for(PivotCounter=0; PivotCounter<numprocs-1; PivotCounter++)
{


PartitionDisps[PivotCounter+1]=PivotPostionByBinarySearch(inputDataAr
ray,StartIndex,Endindex,PivotArray[PivotCounter]);
                    PartitionCnt[PivotCounter]=
PartitionDisps[PivotCounter+1]-PartitionDisps[PivotCounter];
```

```
}
PartitionCnt[numprocs-1]=Endindex-PartitionDisps[numprocs-1];
}

int PivotPostionByBinarySearch(int* inputDataArray,int StartIndex,int
Endindex,int SearchKey )
{
int first = StartIndex;
int last = Endindex;
int middle = (first+last)/2;

while (first <= last) {
  if (inputDataArray[middle] < SearchKey)
          first = middle + 1;
  else if (inputDataArray[middle] == SearchKey) {
           while(middle+1<=(Endindex)&&
inputDataArray[middle+1]==inputDataArray[middle])
                    middle++;

          return middle;
  }
  else
          last = middle - 1;

  middle = (first + last)/2;
}
if (first > last)
{
   if(first>(Endindex))
            first=(Endindex);
  return first;
}
return -1;

}
```

Phase 4:

In this phase, the partial ordered partitions are merged by each process and it is assumed that the sorted data is the concatenation of the processes' memory:

```
Phase 4
Merging the Partitions by Merging Partial OrderedLists
*/
MASTER
{
          phase_start = MPI_Wtime();
}

MergePartialOrderedLists(OutputData, sum, myid, RcvCnt);

MPI_Barrier(MPI_COMM_WORLD);
MASTER
{
          phase_stop = MPI_Wtime();
          t_phase4=phase_stop-phase_start;
}
//End of Phase 4
```

The function `MergePartialOrderedLists` uses a pointer for each sublist and creates the sorted output list in one pass. This function is as follows:

```
void MergePartialOrderedLists(int* InputDataArray, int size, int
ThreadID, int* RcvCnt )
{
int* Outputptrs[numprocs];
int counter;
for(counter=0;counter<numprocs;counter++)
{
Outputptrs[counter]=(int *)malloc((RcvCnt[counter])*sizeof(int));
}
int innercnt;
int tmpcnt=0;
for( counter = 0; counter < numprocs; counter++ )
{
                    {
          for(innercnt=0;innercnt<RcvCnt[counter];innercnt++)
                                {
                                          Outputptrs[counter]
[innercnt]=InputDataArray[tmpcnt];
                                          tmpcnt++;
                                }

                    }
}
int buffer[numprocs];
int* PointersToThreadData[numprocs];
for( counter = 0; counter < numprocs; counter++ )
{
          PointersToThreadData[counter]=Outputptrs[counter];
          if(RcvCnt[counter]!=0)
                    buffer[counter]=*(PointersToThreadData[counter]);
          else
                    buffer[counter]=MAX_NUMBER;

}


int MinIndex;

for (counter=0; counter<size; counter++)
{
          MinIndex=FindMinIndex(buffer);
          InputDataArray[counter]=*(PointersToThreadData[MinIndex]);
          PointersToThreadData[MinIndex]++;
          if(Outputptrs[MinIndex]+RcvCnt[MinIndex]-
1>=PointersToThreadData[MinIndex])


          buffer[MinIndex]=*(PointersToThreadData[MinIndex]);
          else
                    buffer[MinIndex]= MAX_NUMBER;
}
char tempStr[1000];

for( counter = 0; counter < numprocs; counter++ )
{
          free(Outputptrs[counter]);
}
}
```

Finalization:

In this part, the time analysis takes place, the local data are Gathered in MASTER process and the malloc data is freed. Note that the Gathering part is just for correctness testing purposes which aill be discussed in the next section:

```
//Finalization
MASTER
{
stop = MPI_Wtime();
t_total=stop-start;
printf("\n Time elapsed: = %f\n",t_total );
//Time Analysis :
printf("\nPortion of Phase 1: %f",t_phase1/t_total);
printf("\nPortion of Phase 2: %f",t_phase2/t_total);
printf("\nPortion of Phase 3: %f",t_phase3/t_total);
printf("\nPortion of Phase 4: %f\n",t_phase4/t_total);

}
int sizes[numprocs];
MPI_Allgather(
&sum,
1,
MPI_INT,
sizes,
1,
MPI_INT,
MPI_COMM_WORLD);

int sizeDspl[numprocs];
sizeDspl[0]=0;
for(counter=0;counter<numprocs-1;counter++)
{
 sizeDspl[counter+1]=sizeDspl[counter]+sizes[counter];
}

          MPI_Gatherv(
                    OutputData,
```

```
                    sum,
                    MPI_INT,
                    GeneratedData,
                    sizes,
                    sizeDspl,
                    MPI_INT,
                    0,
                    MPI_COMM_WORLD
                    );

//Free the mallocs
MASTER
{
free(GeneratedData);
free(GatheredRegularSample);

}
free(InputData);
free(OutputData);
free(PartitionDisps);
free(PartitionCnt;
```

III. Experimentation

For the experimentation part, the program was executed on a **4 identical-node (small size) RAC cluster** running on **Ubuntu 16.04** provided by Cybera. The specification of the nodes is as follows:

Number of CPU: 2
CPU Speed : 1999.992 MHz
Memory: 2000 mb
Architecture: x86_64
CPU op-mode(s): 32-bit, 64-bit
Byte Order: Little Endian
On-line CPU(s) list: 0,1
Thread(s) per core: 1
Core(s) per socket: 1
Socket(s): 2
NUMA node(s): 1
Vendor ID: GenuineIntel
CPU family: 6
Model: 45
Model name: Intel(R) Xeon(R) CPU E5-2620 0 @ 2.00GHz
Stepping: 7
CPU MHz: 1999.992
BogoMIPS: 3999.98
Virtualization: VT-x
Hypervisor vendor: KVM
Virtualization type: full
L1d cache: 32K
L1i cache: 32K
L2 cache: 256K
L3 cache: 15360K
NUMA node0 CPU(s): 0,1

For the testing purposes, this program was compiled by MPICC command with optimization level of **-O3**. In order to investigate more about the performance of the executions, the speedup graphs and the phase-by-phase analysis will be provided. For the tests, **each data point is executed 7 times and the result is the average of its last 5 runs** in order to avoid start-up overheads. The tests were run automatically by running a **batch script** and by storing the results in an output text file.

Furthermore, the **correctness of the implementation** is tested in this manner that first, I copied the generated random data into an array and sorted it by quicksort library function, and then I compared each one of its elements with the elements of the result of the program, and it outputs the number of Different elements between those two arrays:

```
//CORRECTION TEST
MASTER
{
qsort(tempdata,NUM_GLOBALINPUTDATA,sizeof(int),compare);
          int DiffNum=0;
for(counter=0;counter<NUM_GLOBALINPUTDATA;counter++)
          if(tempdata[counter]!=GeneratedData[counter])
                    DiffNum++;
printf("\n DIFFNUM: %i \n", DiffNum);

free(tempdata);

}
```

This test was done for all of the cases that are used in this experimentation. In order for not interfering with the execution time and the memory consumption issues, the testing part was disabled for data point calculations.

Moreover, the `MAX_NUMBER` is set to `2147483647` which is the Maximum possible number for INT type to make the possibility of generating duplicate numbers as low as possible in order for the PSRS algorithm's performance to be optimal (since the performance of the algorithm diminishes with duplicate data)

For the speedup graphs and Phase-By-Phase analysis, 2 different number of keys are used: **16 Million keys** and **160 Million keys**. The reason for choosing 160M keys is for the execution time to be at least greater that 60 seconds for a single process in order for the compressions to be more measurable. At first, I decided to experiment with 80M and 800M keys, however, the program crashed on 800M keys, since size of INT type is 4 bytes and 800M*Sizeof(int)>2G (available memory on each node). Furthermore, in order to study the effect of using more processors on the execution time, each set of keys were

tested on 10 different numbers of processes which are: 1, 2, 3, 4, 5, 6, 7, 8, 16, 32. and 64.

For capturing the duration time for each phase and the total execution, the function `MPI_Wtime()` is used and before starting the timer and stopping it, the MPI_BARRIER is called to synchronize the starts and stops between the processes.

Speedup Analysis:

The tables below are the program's execution times and speedup graph on those two sets of data and the different processor numbers.

#Processes	Execution Time (in Seconds)	Speedup
1	5.52	1.00
2	2.37	2.33
3	2.31	2.38
4	1.76	3.14
5	1.37	4.02
6	1.25	4.40
7	1.15	4.79
8	1.10	4.99
16	1.47	3.76
32	2.60	2.12
64	4.71	1.17

Table 1: Execution Times for 16M keys

#Processes	Execution Time (in Seconds)	Speedup
1	64.00	1.00
2	27.33	2.34
3	19.91	3.21
4	14.61	4.38
5	12.11	5.28
6	11.19	5.72
7	8.81	7.26
8	8.29	7.72
16	9.67	6.62
32	14.26	4.49
64	20.29	0.27

Table 2: Execution Times for 160M keys

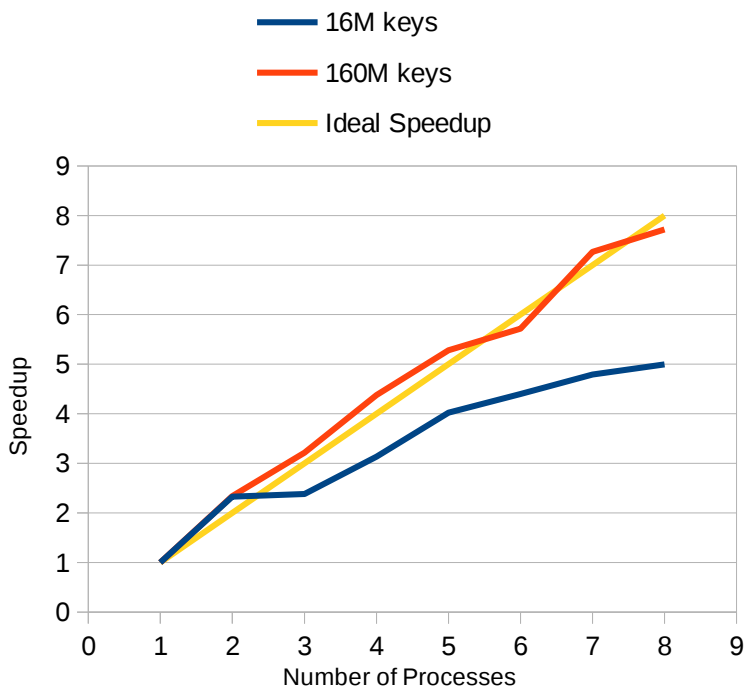

Illustration 1: Speedup Graph for running on upto 8 processes

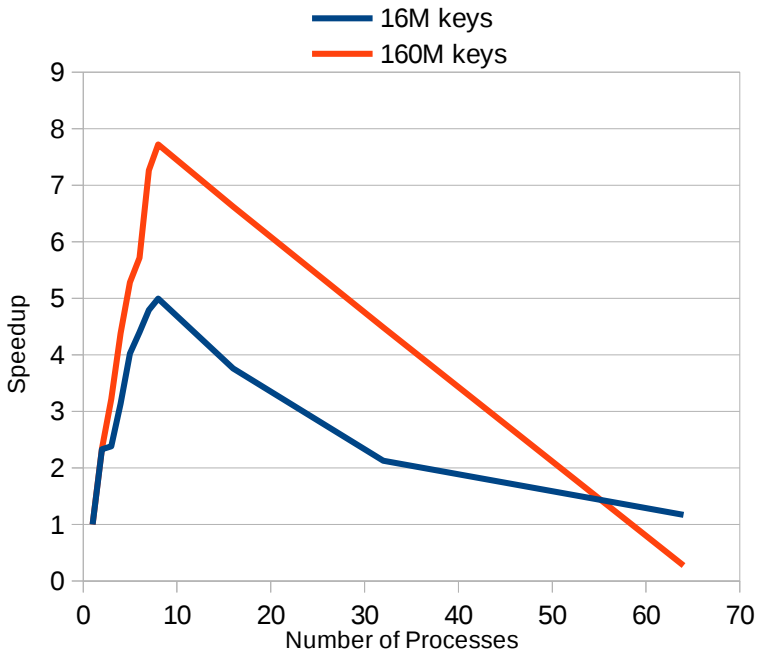

Illustration 2: Speedup Graph including 16, 32 and 64 processes

Since we have a 4-node RAC cluster, we expect to have an **increasing speedup curve** until running with 4 processes. However, we are seeing that the speedup curve is increasing until running with 8 processes. This is due to the **multi-core structure of the nodes**: Since each node has 2 cpu cores, then it is capable of running 2 threads in parallel without performance diminishing effects.

On the other hand, in Figure 2 (Illustration 2) it is obvious that after 8 processes, the speedup is decreasing. This is because of the fact that we have 8 cores in total in this RAC configuration and by creating processes above this number, the **overheads** of executing the distributed commands(namely scattering and gathering the data and thread management tasks) overcome the speedup effect.

Another characteristic of the speedup concept which is obvious in the above graphs is that **the more the granularity of data for each process, the better the speedup.** As we can see in Illustrations 1, for a fixed number of processes, the speedup for the greater input data is higher. Plus, the negative effect is true for speedup: the more corse-grained data for each process, the higher the negative effect would be due to overheads. As a proof, we can look at Illustration number 2 and see that on 64 processes, the speedup of the program with 160M key is even worse than that of 16M keys.

Another point in the speedup graphs is that for 160M keys, in many datapoints, we have a superlinear speedup and since the difference between the linear case is significant[1], then we could have an explanation for that.

Speculation: in this configuration, superlinear speedup can occur because of more efficient utilization of resources by the processors [2] and the fact that the single process program was overloaded with data, while there is a fixed 2Gig RAM and 15360K L3 cache on each processor. So in this case, for each increasing process, the effect of utilizing the reduced partitioned data by the full memory could be more significant.

Phase by Phase Analysis:

Figures below illustrate the percent of execution time that each phase has taken.

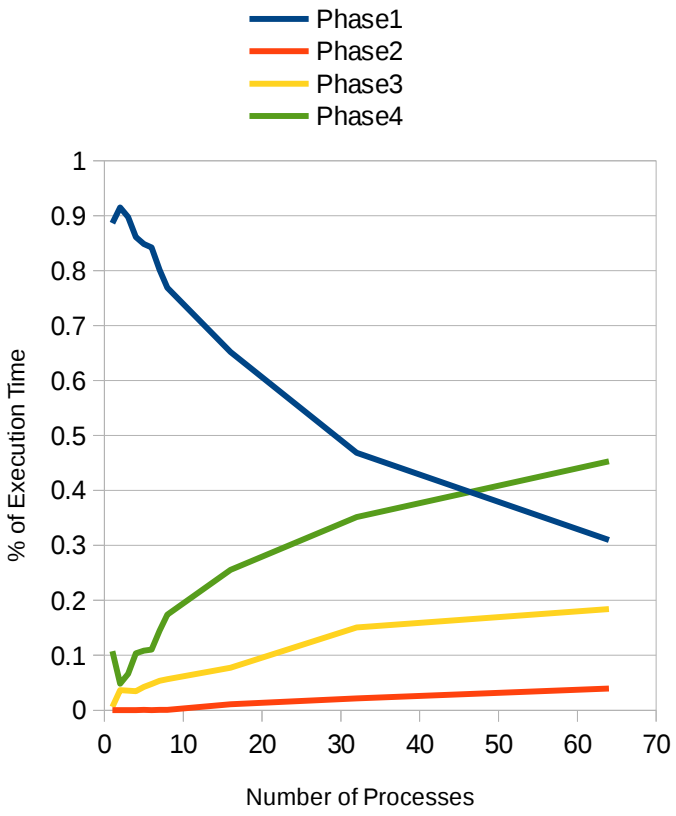

Illustration 3: Phase-By-Phase Analysis for 160M keys

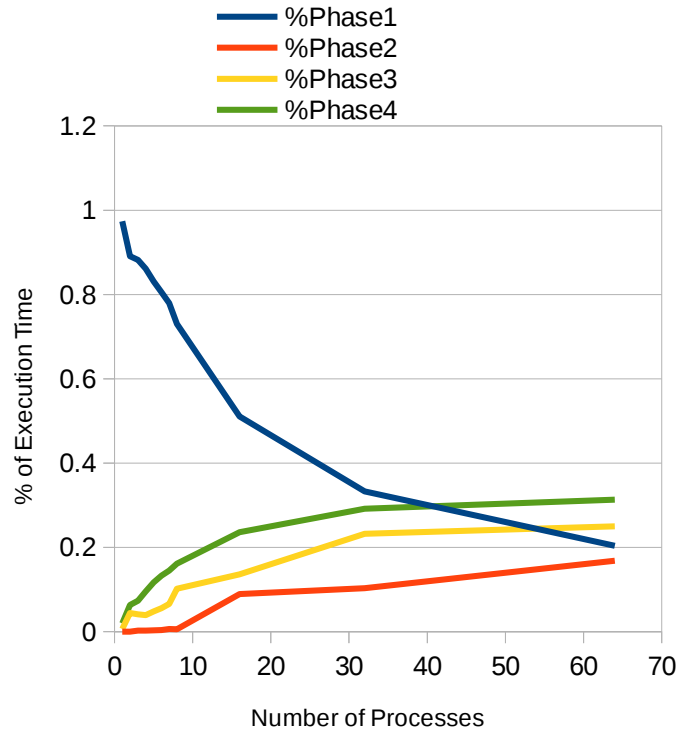

Illustration 4: Phase-By-Phase Analysis for 16M keys

1 the datapoint's difference with the ideal speedup is greater than the error-bar for that datapoint.

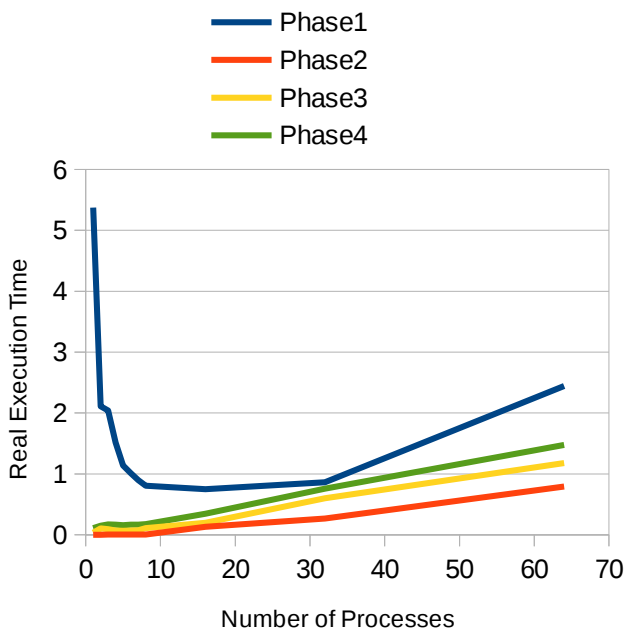

Illustration 5: Real Execution Time for 16M keys

Based on the figures above, we can see that phase1 takes the major part of the execution time since it has to sort its local data in this phase and as we see by increasing the processes, phase1's percentage decrease since there would be fewer data for each process but after 8 processes it increases due to the overheads, However other phases execution time increase since they have to handle the overheads of merging partitions.

IV. References

[1] Xiabo Li, Paul Lu, Jonathan Schaeffer, John Shilington, Pok Sze Wong, Hanmao Shi, “On the versatility of parallel sorting by regular sampling”, Parallel Computing, Volume 19, Issue 10, October 1993, Pages 1079-1103.

[2] http://www.ccs.neu.edu/course/com3620/projects/scalable/jshan/final1.pdf