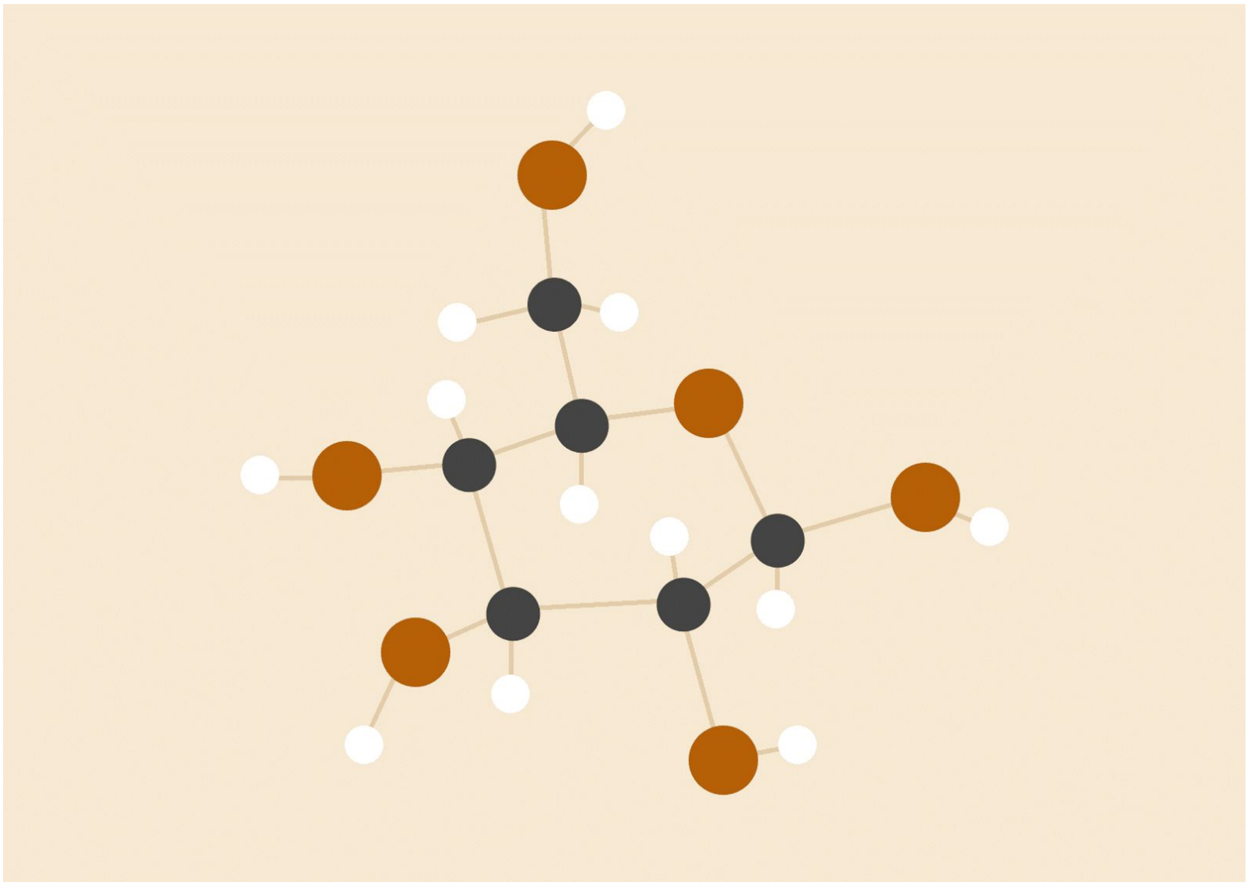
CMPUT 481 Assignment 2 Report

Distributed Memory Parallel Sorting by Regular Sampling



CCID	**jxie2**
Student Name	**Jiahui Xie**
Student Number	**1372777**

Table of Contents

Introduction

Instructions on building and running the main parallel sorting by regular sampling algorithm can be viewed in the “README.md” file under root source directory. The “PSRS” program is implemented entirely in the “C” programming language with the assistance of “MPICH” variant of the message passing interface library.

After the main program is successfully built, an additional “driver” program is required to perform the experiment described in the “Experiment” section below; this program is written in the “Python” programming language with the assistance of “Matplotlib” plotting library. To replicate the result obtained by the author, one can simply follow the instructions under “Speedup Comparison” section of the “README.md” file. Please pay particular attention to that file if experiment needs to be conducted in a cluster environment due to the fact that the stock version of the “driver” program would not work without modification.

To simplify the steps taken to install dependencies required by the “PSRS” experiment suite, a shell script is supplied to automate this process; however, this script is only tested under a “Ubuntu 16.04 LTS” linux environment.

Design Overview

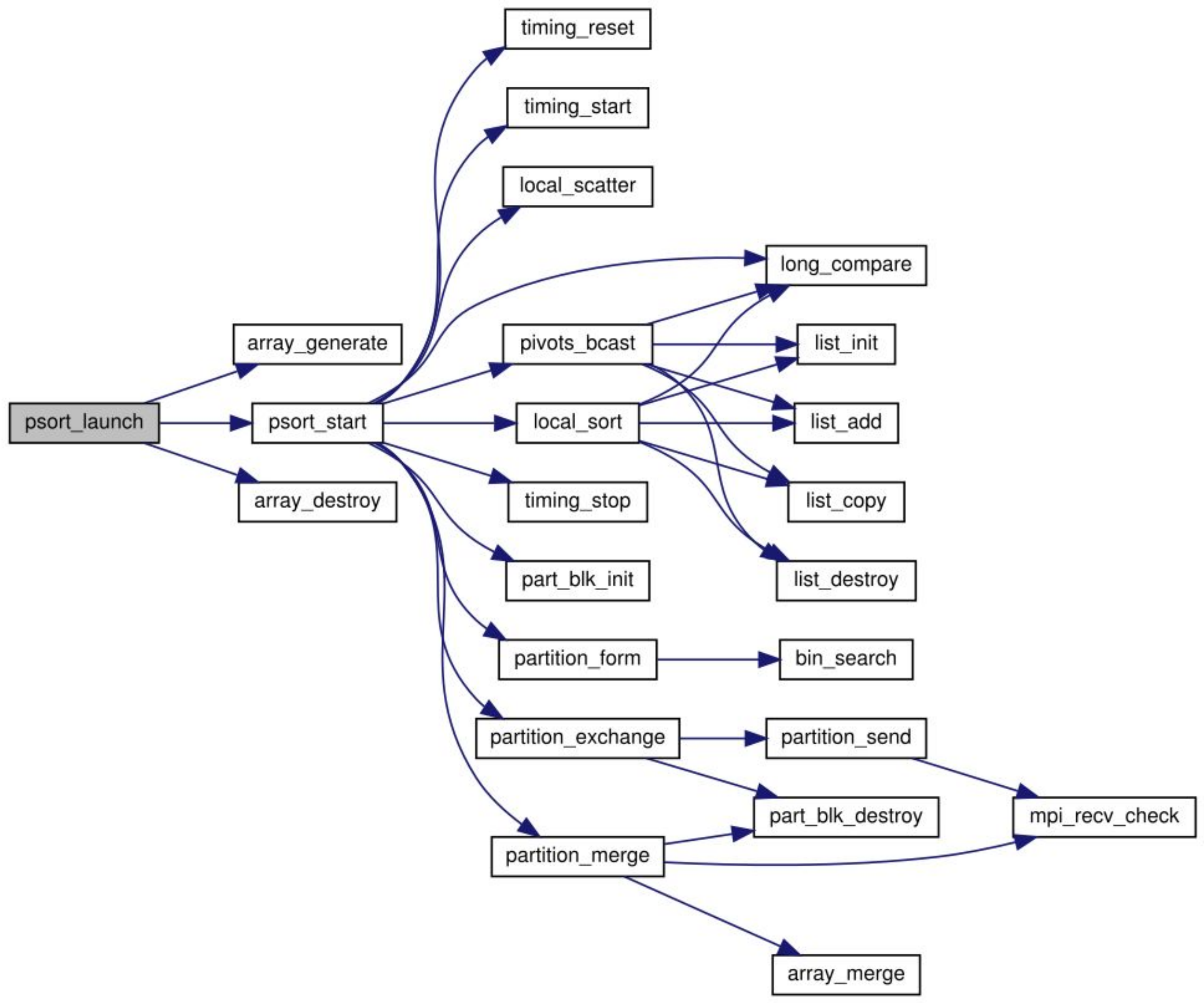


Figure 1: Call Graph with “psort_launch” as the Root Node

The “PSRS” program embraces a modularized “separation of concerns” design principle: all of its source code is divided into separate “modules” that each implement a set of functions that typically operate on a “data structure” where applicable (in the “software engineering” discipline, making functions or methods “revolve” around a common “object” is named “Object-Oriented Design”; even though the “PSRS” program is written in “C”, a programming language that does not have built-in support for that). For example, the functions that directly interact with the users through a command line interface reside in “src/psrs.c”, this source file has the following overall structure (details are abbreviated as “...” because their irrelevance to the discussion) :

```
int main(int argc, char *argv[])
{
   MPI_Init(&argc, &argv);
   struct cli_arg arg = { .binary = false };
   …
   if (0 == rank) {
      argument_parse(&arg, argc, argv);
   }
   argument_bcast(&arg);
   sort_launch(&arg);
   MPI_Finalize();
   …
}
```

Figure 2: Simplified Source Code Listing for “src/psrs.c”

Judging solely from the structure of the “main” function it is clear to see that “PSRS” program still follows the general design guidelines in which conditional execution of parallel instructions all happens between the functions “MPI_Init” and “MPI_Finalize”. In this case, only the “root” process with a rank of “0” would parse the command line arguments, then it broadcasts the validated arguments to all processes through “argument_bcast”. Furthermore, another design pattern that can be seen employed is “chain of responsibility”, where a function is dedicated to take or delegate certain “responsibilities”; an obvious use of this pattern is seen in the “kernel” “psort_start” sorting function which implements the “PSRS” core algorithm: it invokes a series of functions in sequence and assigns certain “ownership” of block scope variables to and from them in order to progressively proceed from phase one to phase four (the second column of stack frames that start with “pivots_bcast” up to “partition_merge”). Another benefit that is gained through this explicit declaration of “ownership” is the chance of “memory leak” is minimized due to unawareness of the programmer when there is an abundant amount of pointer variables.

Experiment

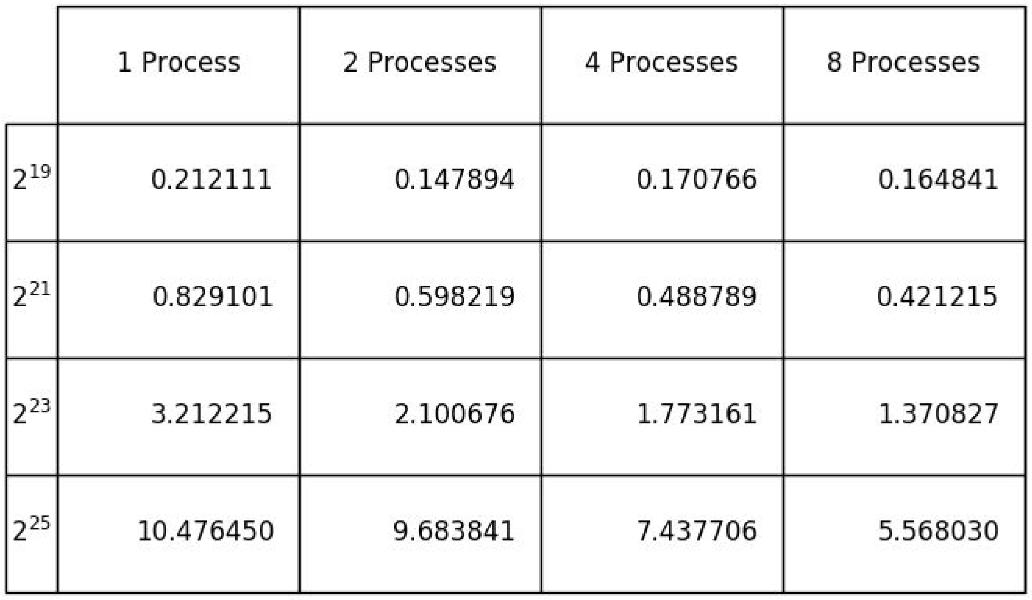
The experiment is conducted in a virtualized cloud environment named “Cybera Rapid Access Cloud”; in total four virtual machines are used, and each machine comprises of two physical CPU cores (“Intel(R) Xeon(R) CPU E5-2650 v3 @ 2.30GHz”, as reported by the “lscpu” program) and two gigabytes of RAM. The product between the previously specified amount yields eight CPU cores in total. To ensure the fairness of distribution of the aggregate workload, eight MPI processes are launched to execute the job from the “head” node with designated “floating” IP address. As for the software environment, all virtual machines use the same “Ubuntu 16.04 LTS” linux distribution with a kernel version of “4.4.0-45-generic” (reported by the “uname -r” command). Before the experiment is performed, “uptime” is issued to query the current load average, which yields a result of “load average: 0.00, 0.00, 0.00” for the last field.

As mentioned in the “Introduction” section, the experiment is initiated through the “tools/plot.py” “driver” program on the head node after tweaking it according to the “README.md” file. Both the number of processes and the length of array are incremented progressively from 1 to 8 and 2^{19} to 2^{25} (exponent increases with a step of 2), respectively. The actual sorting times displayed in the table below (Figure 3) are calculated by applying a simple moving average algorithm (refer to “src/stats.c” for implementation details) with a window size of 5 on the original sorting time obtained from a total of 7 runs; to get a “concrete” feeling of the range of sorting times, both the number of processes and the length of array are used as categorical variables for a inter-group sorting time comparison chart (Figure 5). Additionally, the “PSRS” program also outputs standard error along with the averaged sorting time, those errors for each test run is recorded in a separate table (Figure 4). By alternating the length of array to be sorted, different groups can be categorized with the array length as the key and sorting times for distinct number of processes as values. The “driver” program calculates speedup values as T_1 / T_p after obtaining the data according to the previously stated scheme. The speedup calculation procedure can be expressed in the following “Python” pseudo instructions (details can be inspected in the “speedup_plot” function of “tools/plot.py” program):

```
vector = list()
for length in [2 ** e for e in range(19, 26, 2)]:
    vector.clear()
    for process in [2 ** e for e in range(4)]:
        mean_time, std_err =program.run(length, process)
        if 1 != process:
            speedup = vector[0] / mean_time
        else:
            speedup = mean_time
        vector.append(speedup)
```

Figure 6: Speedup Calculation Pseudo-Code

Sorting Time in Moving Average (second)



	1 Process	2 Processes	4 Processes	8 Processes
2^{19}	0.212111	0.147894	0.170766	0.164841
2^{21}	0.829101	0.598219	0.488789	0.421215
2^{23}	3.212215	2.100676	1.773161	1.370827
2^{25}	10.476450	9.683841	7.437706	5.568030

Figure 3: Mean Sorting Time for the Last 5 Runs

Standard Deviation for Sorting Time

	1 Process	2 Processes	4 Processes	8 Processes
2^{19}	0.010498	0.014965	0.019788	0.018838
2^{21}	0.116273	0.055317	0.036792	0.017899
2^{23}	0.604618	0.432632	0.157764	0.067076
2^{25}	0.524618	0.728085	0.233319	0.670895

Figure 4: Standard Deviation for the Last 5 Runs

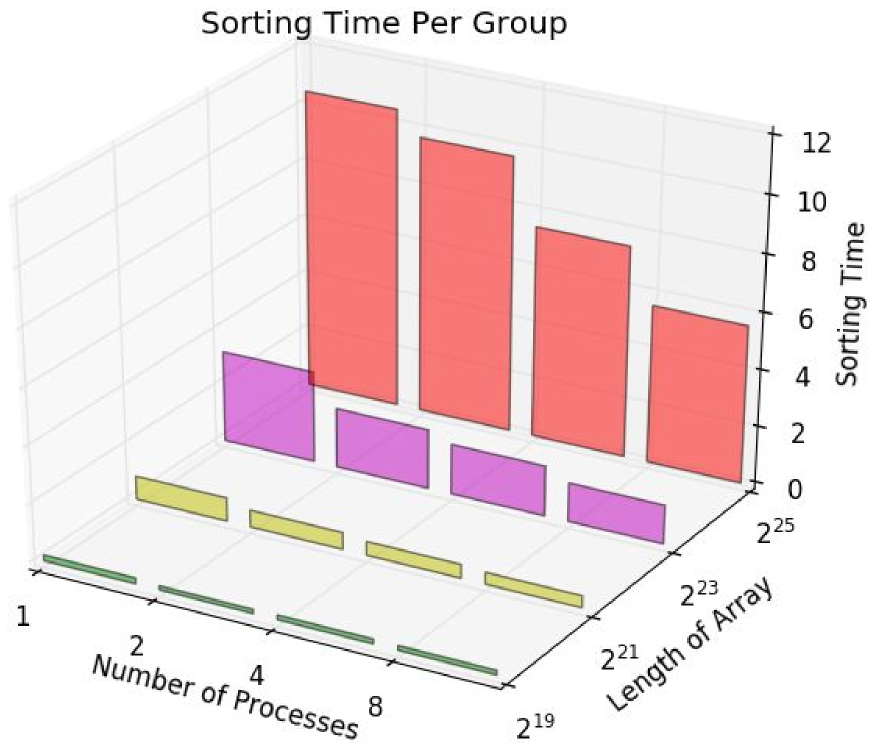


Figure 5: Inter-Group Comparison of Sorting Time

By following the pseudo-code listed in Figure 6, a speedup graph can be plotted as in the following (Figure 7):

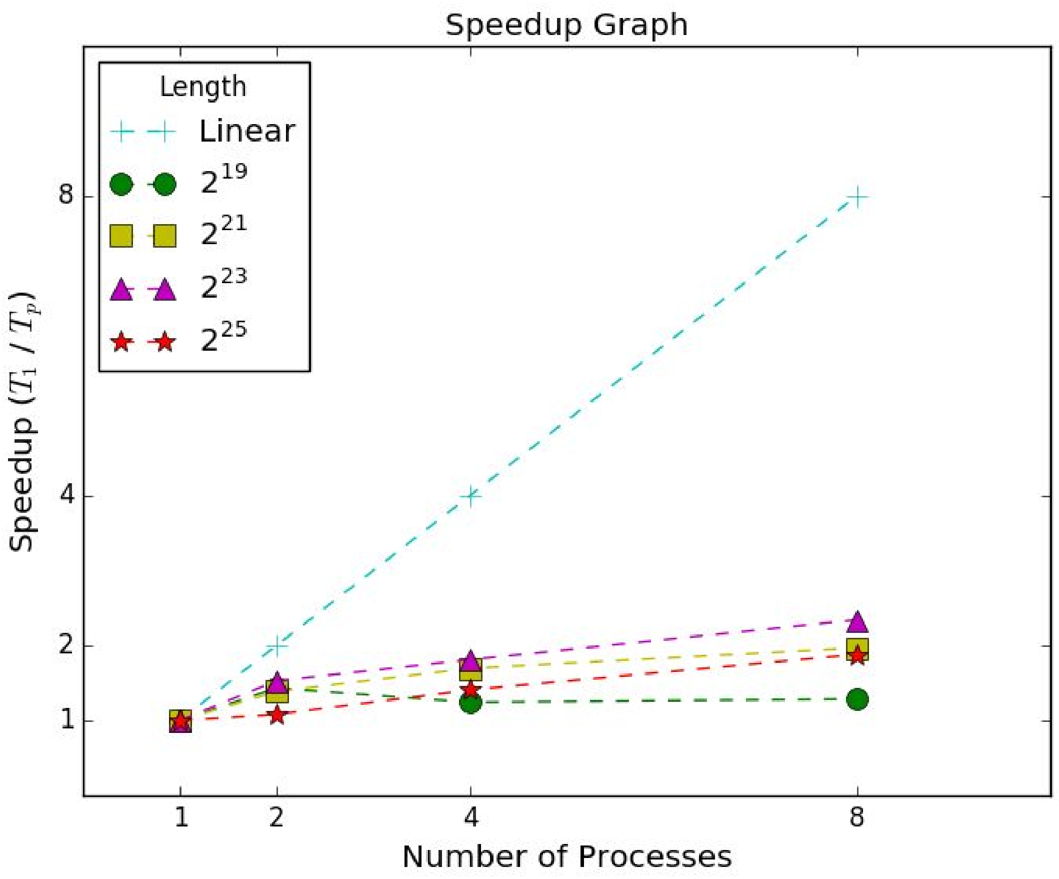


Figure 7: Speedup Graph

As documented in the “Getting Started” section of “README.md” file, the “PSRS” program supports a special per-phase mode in which a per-phase mean execution times are reported through a similar moving average algorithm; however, the “driver” program directly obtains the result in single run for two different setup of array length: 2^{21} and 2^{27} are used with 4 picked as the number of processes to launch since it is the median between 1 to 8. The reason for outputting two separate pie charts is to compare between the two and observe whether varying the length of array could play a significant role in changing the proportion of per-phase execution time (Figure 8, 9).

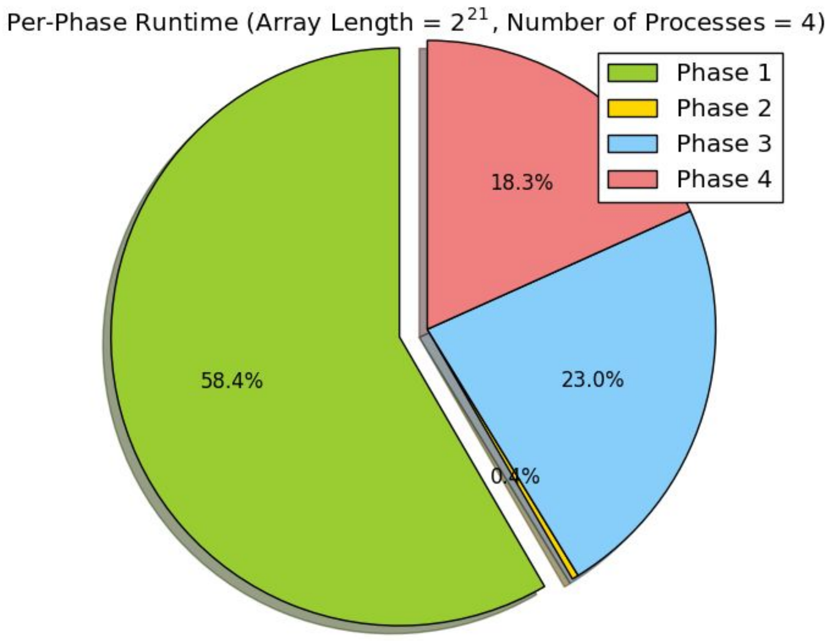


Figure 8: Per-Phase Runtime with Array Length 2^{21}

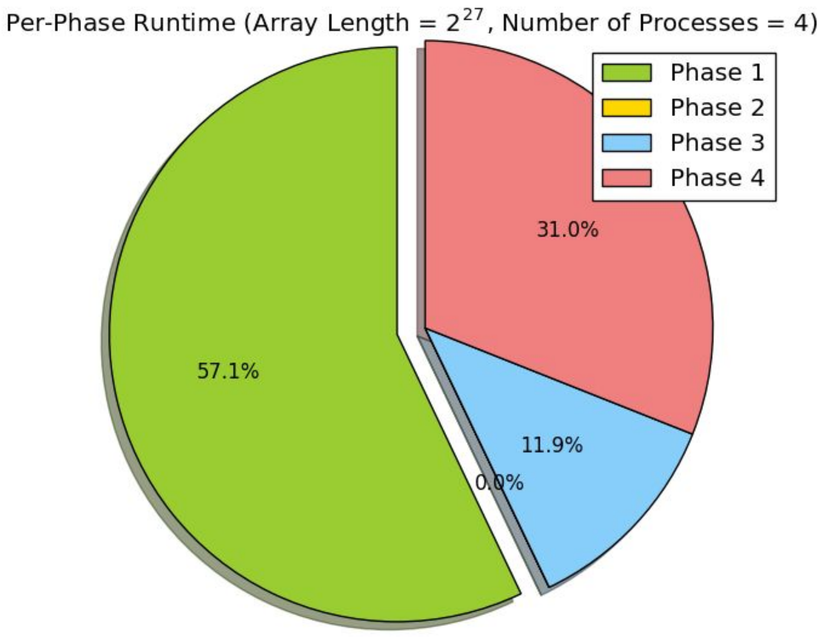


Figure 9: Per-Phase Runtime with Array Length 2^{27}

Analysis & Speculation

Based on the obtained sorting time (Figure 3), if the length of array to be sorted is fixed and number of processes is set to be an independent variable, then it is easy to observe that for all array lengths except the 2^{19} case, there may exist a negative association between the sorting time and number of processes: as the number of processes increases, the sorting time decreases. Even though there is a special case when array length is 2^{19} and number of processes is 2, it does not affect the overall decreasing trend when the table is viewed from left to right on a row by row basis.

For the standard error (deviation) gathered (Figure 4), it does not exhibit an obvious pattern; however, inference about the relationship between the length of array and the sorting time can be made: first, fix the number of processes to be eight and compare the sorting time between the 2^{19} and the 2^{25} case; we can see that the sorting time differs by a large amount (0.164841 and 5.568030), but whether it suggests that whether altering the array length could increase the sorting time is unknown. With the assistance of the standard error values from Figure 4, we can say with high degree of confidence that there may be a positive association between the length of array and the sorting time because the sorting time still differs by a large amount even subtracting the standard error for the 2^{25} case (5.568030 - 2 * 0.670895 = 4.22624); that is, as the length of array increases, the sorting time also increases, assuming the algorithmic complexity of the sorting algorithm is unknown.

In terms of speedup, all the cases achieve a speedup between 1 and 2. For all the cases other than 2^{19}, there is an upward trend for the speedup curve, which suggests that the speedup increases as the workload (length of array) increases.

Lastly, both pie chart (Figure 8, 9) exhibit similar proportions: phase one accounts for the majority of execution time for both cases (near 60 percent) while phase two takes the least amount of time to execute. As the workload (length of array) increases, phase four dominates phase three by a very large extent (phase four “merge” increased from 18.3 % up to 31.0 %, while phase three shrinked from 23.0 % to 11.9 %).

REFERENCES

Li, X., Lu, P., Schaeffer, J., Shillington, J., Wong, P.S., Shi, H. On the Versatility of Parallel Sorting by Regular Sampling. Parallel Computing 19, 1079-1103, 1993.