CS3210 Assignment 3 Report

Joshua Tan Yin Feng (A0199502Y), Kishen Ashok Kumar (A0204777X)

Introduction

For this assignment, we designed 2 slightly varied MapReduce implementations which will be explained with the aid of diagrams. We then show our results with a varying number of Map and Reduce workers, different worker distributions across machines as well as varying input test cases.

Design

The general program sequence is as follows:

1. The master process reads the *ith* file into a buffer and sends it to a map worker *j*
2. Map worker *j* calls the mapping function on the data received before it partitions them appropriately into *k* partitions (where *k* represents the number of reducer workers).
3. Map worker *j* then sends each *k*th partition to the *k*th reducer worker respectively.
4. Reducer worker *k* has a hashmap *resultsMap* that maps a key string to an array.
    a. For each *key value* pair in the partitions that reducer worker *k* receives, it appends the *value* into the array *resultsMap[key]*.
5. Once all the work is done, every reducer worker will then iterate through resultsMap and perform the reduction operation on each entry in the table. After all the entries have been reduced, it then sends the entries in *resultsMap* back to the master program.
6. Master program then aggregates all the data together and writes it to the output file.

To indicate to the workers that there is no more data to be received, we use a *TERMINATION_TAG*. Upon receipt of this tag, each worker knows that there is no need to wait on any receive MPI function.

Fig 1 is a sequence diagram that helps to illustrate our program flow. Do note that the sequence diagram is for the purpose of providing a high level overview of the flow of our program and glosses over the specific implementation details.

Most notably, the two implementations we have written for this assignment differ in how they send and receive information between the Master and the Map worker in the first loop block in our sequence diagram. However since we are providing a high level overview with the sequence diagram, these differences are not captured.

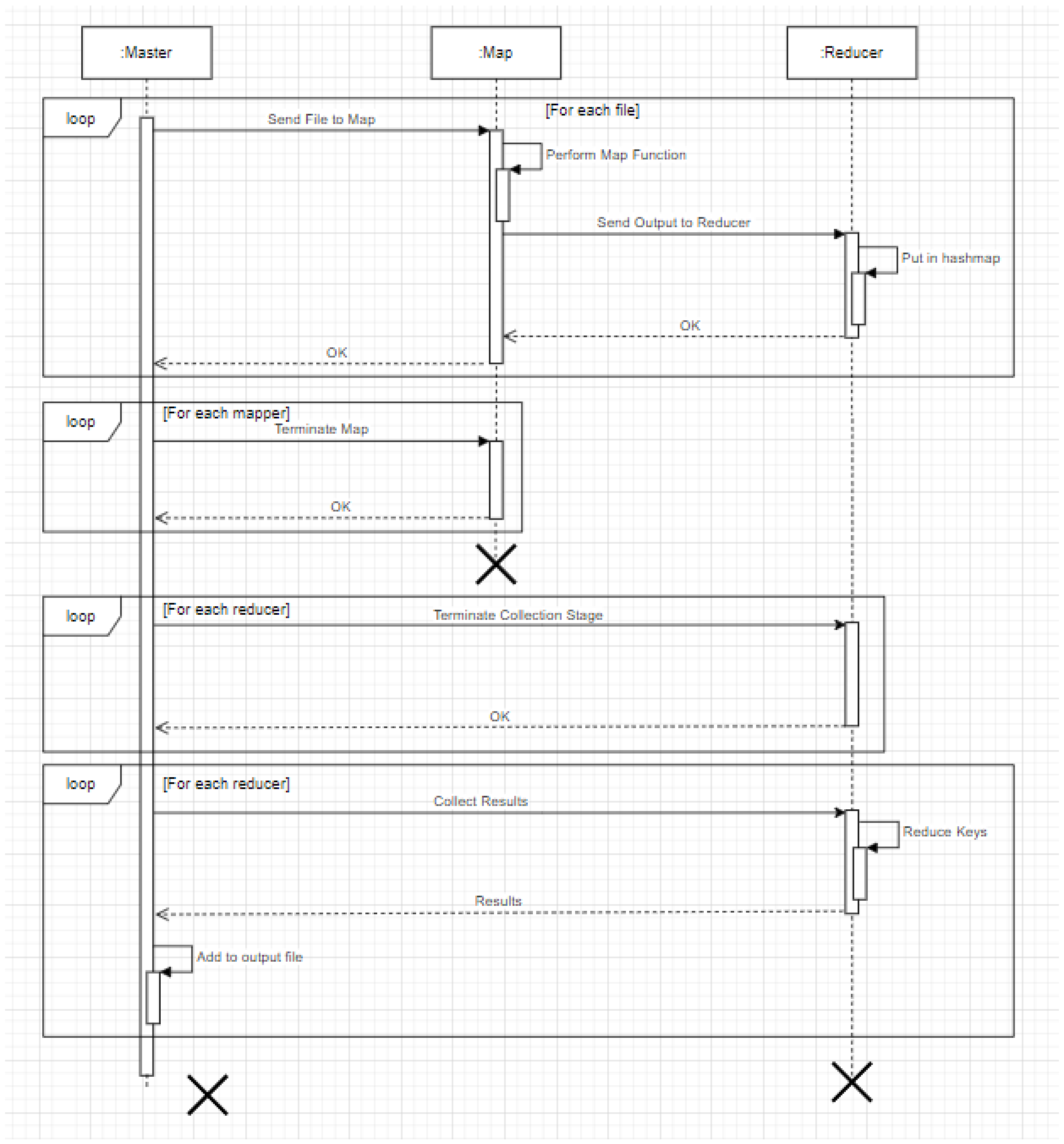

Figure 1: A high level overview of our implementations

Non-dynamic version

Description:

Our first version distributes the workload amongst the map workers in a block wise manner. Informally, the i^{th} file is given to the j^{th} worker where $j = i \% m$, m representing the total number of map workers. To achieve this, we perform an asynchronous send operation to send *m* files at a time to all the map workers. Once all map workers have received the *m* files, the master program will continue to send the next batch of *m* files to the map workers again. For a visual depiction of how the non-dynamic version works, refer to Fig 3.

Deadlock avoidance:

Fig 2 depicts the different MPI commands that are used at different parts of our non-dynamic version to facilitate the passing of messages between processes. We shall refer to the different loop blocks from top to bottom to understand why we avoid deadlocks with this implementation.

1st Loop block:
The master process first distributes the first batch of files to the map workers by using **MPI_Isend** and monitors the success of the operation using **MPI_Request_get_status** to ensure that the sending operation has completed before sending the next batch. During this time, the master process blocks and waits for all the **MPI_Isend** calls to complete successfully before moving on to the next batch. We are guaranteed that this will happen as the map workers block on an **MPI_Recv** for the receiving of data from the master process. Thus there will not be any deadlock between the master process and the map workers. The map workers interact with the reduce workers in a sequence of blocking **MPI_Send** and **MPI_Recv** calls as can be seen in the diagram. Since these calls are blocking and each **MPI_Send** is met with a **MPI_Recv**, we can be sure that there will be no deadlock.

2nd Loop block:
Similar to the 1st Loop block, the master process sends a message with a *TERMINATION_TAG* to all the map workers by using the non-blocking call **MPI_Isend** and waits for all the sending to complete before proceeding by making use of **MPI_Waitall**.Since each map worker blocks on **MPI_Recv** in anticipation of the message from the master processes, no deadlock will occur.

3rd Loop block:
This block works in the exact same way as the 2nd Loop block. The only difference is that the communication is between the master process and the reducer workers.

4th Loop block:
In the final block, the reducer worker sends the aggregated output to the master via a **MPI_Send** call. Since each **MPI_Send** call by the reducer worker is met by a **MPI_Recv** by the master process, we will avoid deadlocks in this block.

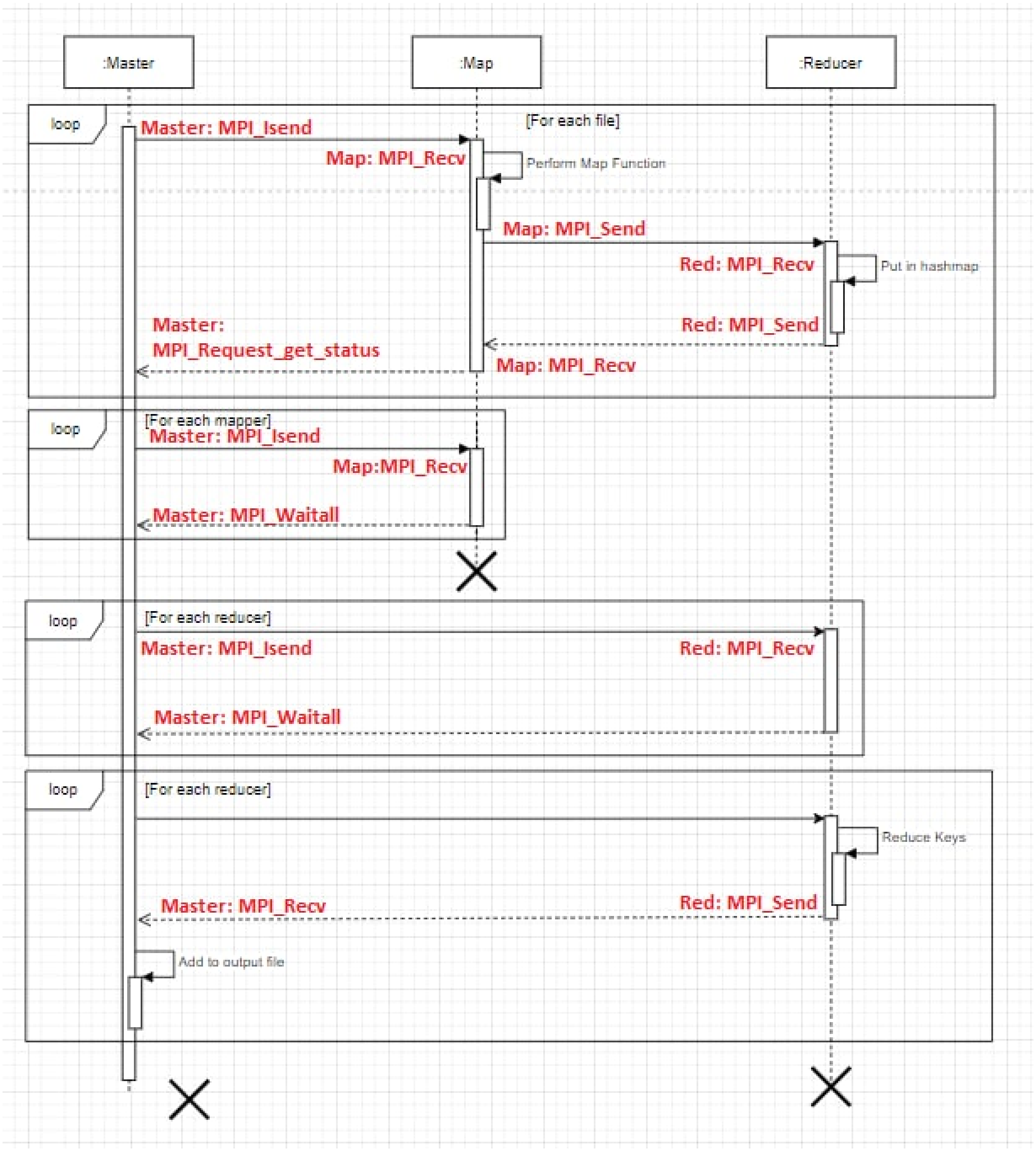

Figure 2: MPI Commands used in non-dynamic version

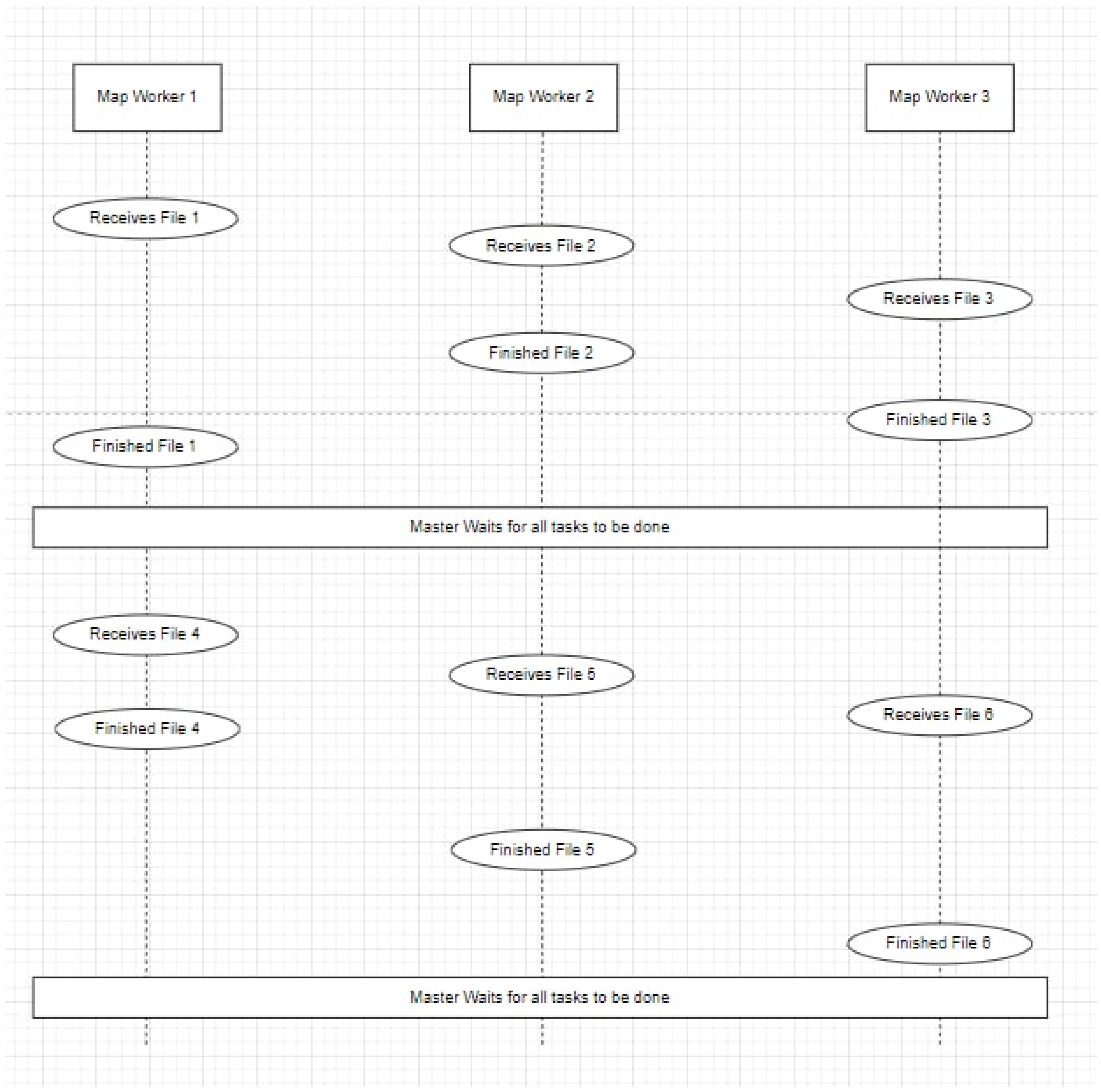

Figure 3: File distribution of non-dynamic version on 3 map workers and 6 files

Dynamic version

Description:

In this second version, the master program will just send the files to whichever map worker is free and available. Signalling is done by sending an empty message from the map worker to the master once the map worker has finished its computation. Upon receipt of this signal, the master knows which map worker to send the next task to. For a visual depiction of how the dynamic version works, refer to Fig 5.

Deadlock avoidance:

Fig 4 depicts the various MPI commands used in our dynamic implementation. The reasons for deadlock avoidance are similar to that of the non-dynamic version. The only difference with the dynamic version is that instead of using **MPI_Request_get_status** to block and wait for all the **MPI_Isend** operations to complete, the dynamic version replaces this with a pair of **MPI_Send** and **MPI_Recv** calls between the map worker and the master process. This is done so that the master process has more fine grain control over the sending of data to the map processes so that it can perform dynamic allocation of tasks.

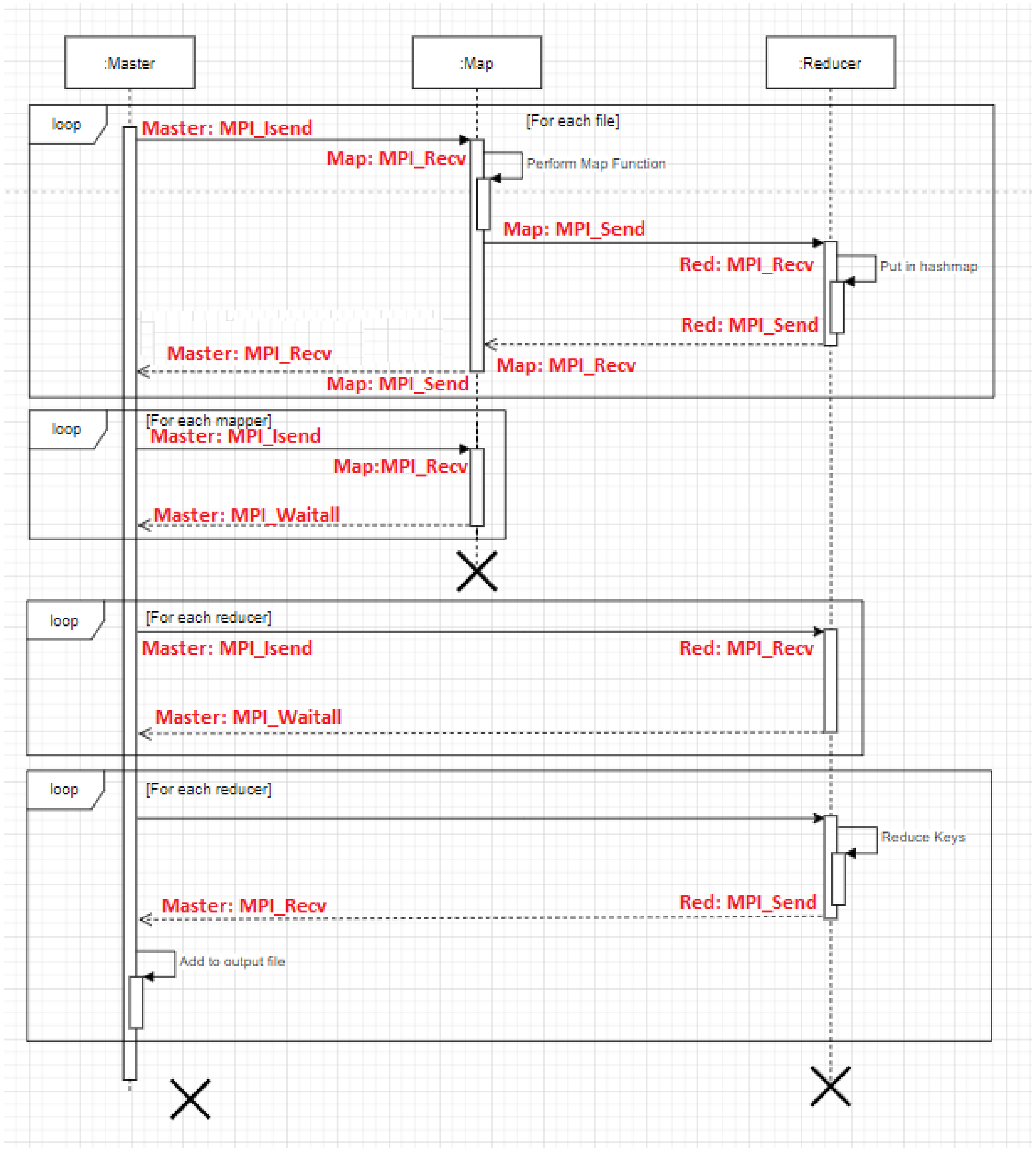

Figure 4: MPI Commands used in dynamic version

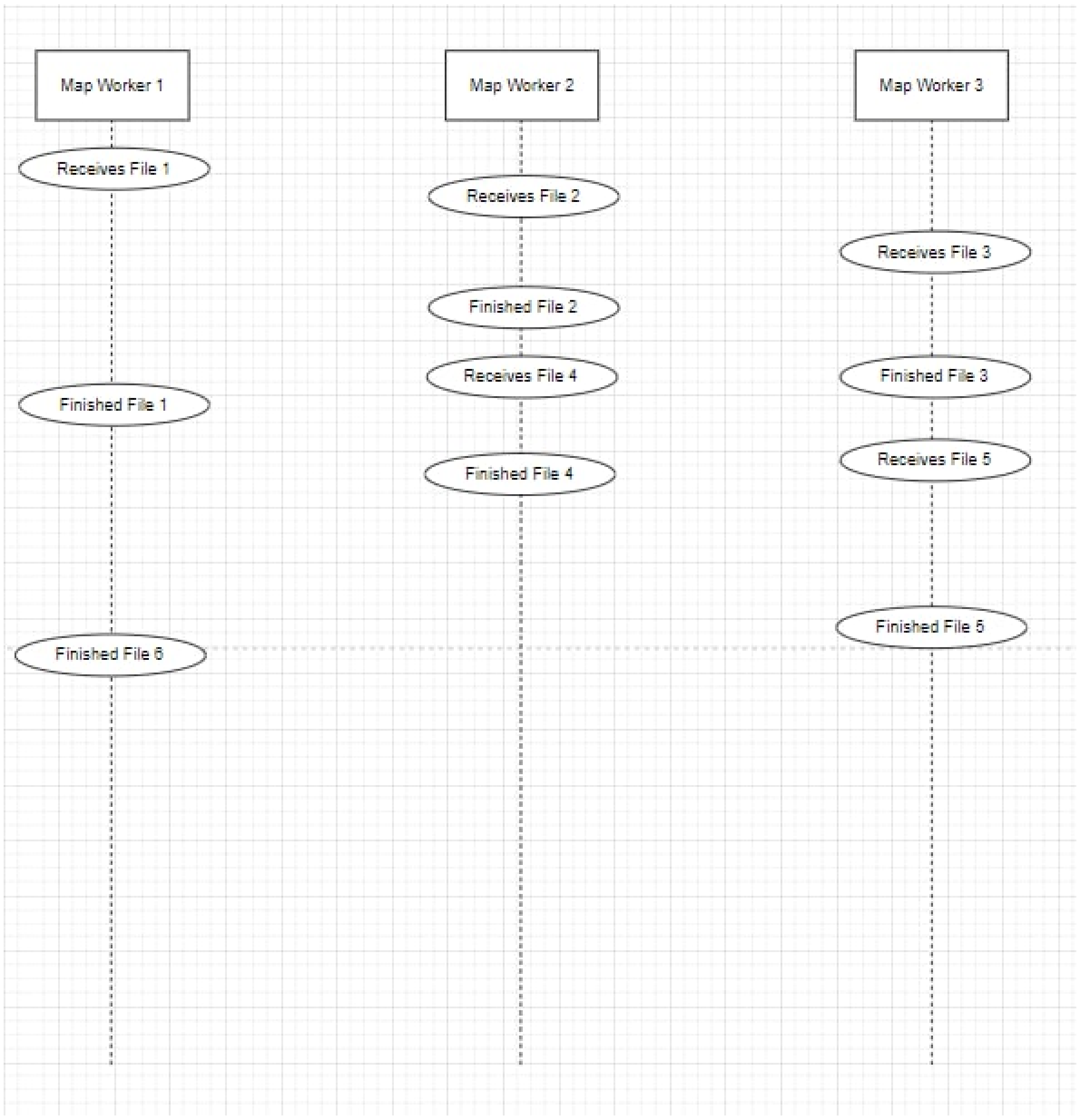

Figure 5: File distribution of dynamic version on 3 map workers and 6 files

Methodology

As mentioned in the introduction, we performed our tests with varying factors to examine their impact on correctness, and more importantly, the performance of our program. In this section we outline some of the details of our data collection methods as well as factors of interest.

Machines Used

We did our measurements on Intel Core i7-7700 (soctf-pdc-012, soctf-pdc-010) and Intel Core Xeon Silver 4114 (soctf-pdc-002, soctf-pdc-003) machines. Their capabilities are as follows:

Intel Core i7-7700:
- 3.60GHz
- 4 cores (8 threads)
- 32GB DDR4
- 500GB 7200RPM SATA HDD

Intel Xeon Silver 4114
- 2.20GHz
- 10 cores (20 threads)
- 32GB DDR4
- 500GB Seagate 7200RPM HDD

Factors of interest

Number of workers

With multiple workers, the final result must still be correct. Varying the number of workers would also allow us to assess the performance speedup.

Distribution of workers

If we distribute the workers amongst different nodes, we should be able to assess the impact on communication times between processes, as compared to having all the workers on the same node.

Number of and size of files

With a varying number of files, we can examine the difference in performance between our dynamic and non-dynamic allocations. A non-dynamic allocation, which sends the files in batches at a time, would probably take more time especially if one node repeatedly receives the biggest files; all other nodes will be left idle while waiting for this one node to finish its receiving operation.
To simulate this situation, we created a set of 3 custom test files and repeated them so that we have a total of 99 test files. In this set, test files 2 and 3 are very small compared to test file 1. When testing with 3 map workers, the first worker will always take test file 1 under a non-dynamic allocation. Hence, workers 2 and 3 will always be left idle because their computations can be done relatively faster. This will not be the case for dynamic allocation as idle workers will request for jobs.

Reproducing results

There are two commands in our Makefile that can be used to build our Non-dynamic and Dynamic version of the code.
To build the Non-dynamic version run: **make build**
To build the Dynamic version run: **make build_dynamic**

Each section uses a combination of different parameters in the execution of our program. As a general rule, use the following format to reproduce any results from the appendices A - D.

Non-dynamic:
mpirun -np <NUM_MPI_PROCESSES> -hostfile <PATH_TO_HOSTFILE> ./a03 <INPUT_FILES_DIR> <NUM_INPUT_FILES> <NUM_MAP_WORKERS> <NUM_REDUCE_WORKERS> <OUTPUT_FILE_NAME> <MAP_REDUCE_TASK_ID>

Dynamic:
mpirun -np <NUM_MPI_PROCESSES> -hostfile <PATH_TO_HOSTFILE> ./a03_dynamic <INPUT_FILES_DIR> <NUM_INPUT_FILES> <NUM_MAP_WORKERS> <NUM_REDUCE_WORKERS> <OUTPUT_FILE_NAME> <MAP_REDUCE_TASK_ID>

Appendix E uses the following commands:
Non-dynamic:
mpirun -np <NUM_MPI_PROCESSES> -rankfile <PATH_TO_RANKFILE> ./a03 <INPUT_FILES_DIR> <NUM_INPUT_FILES> <NUM_MAP_WORKERS> <NUM_REDUCE_WORKERS> <OUTPUT_FILE_NAME> <MAP_REDUCE_TASK_ID>

Dynamic:
mpirun -np <NUM_MPI_PROCESSES> -rankfile <PATH_TO_RANKFILE> ./a03_dynamic <INPUT_FILES_DIR> <NUM_INPUT_FILES> <NUM_MAP_WORKERS> <NUM_REDUCE_WORKERS> <OUTPUT_FILE_NAME> <MAP_REDUCE_TASK_ID>

Custom Test Cases

In our submission folder we have testcases folder that contain 99 input files.

The output files in the testcases directory were generated for the following MapReduce CLI parameters:

0.out

MAP_REDUCE_TASK_ID: 3
NUM_INPUT_FILES: 99
INPUT_FILES_DIR: testcases

1.out

```
MAP_REDUCE_TASK_ID: 1
NUM_INPUT_FILES: 50
INPUT_FILES_DIR: testcases
```

2.out

```
MAP_REDUCE_TASK_ID: 2
NUM_INPUT_FILES: 10
INPUT_FILES_DIR:testcases
```

Results and Discussion

Our full test results can be found in the appendices at the end of this report. In this section, we will cover 3 main observations that we found out from our experiments.

Observation 1

Our first observation was that both our dynamic and non-dynamic implementations do not have any significant differences in performance times. For the non-dynamic version, we expected the execution time to be much longer since the files are sent in batches. As mentioned in the previous section, we expected this difference to be prominent if in one batch of files, one of the files is much larger than the others. The node processing it would take much longer than the rest and thus the other nodes are left idle. However, our results proved otherwise, shown specifically in Fig 6.

Data

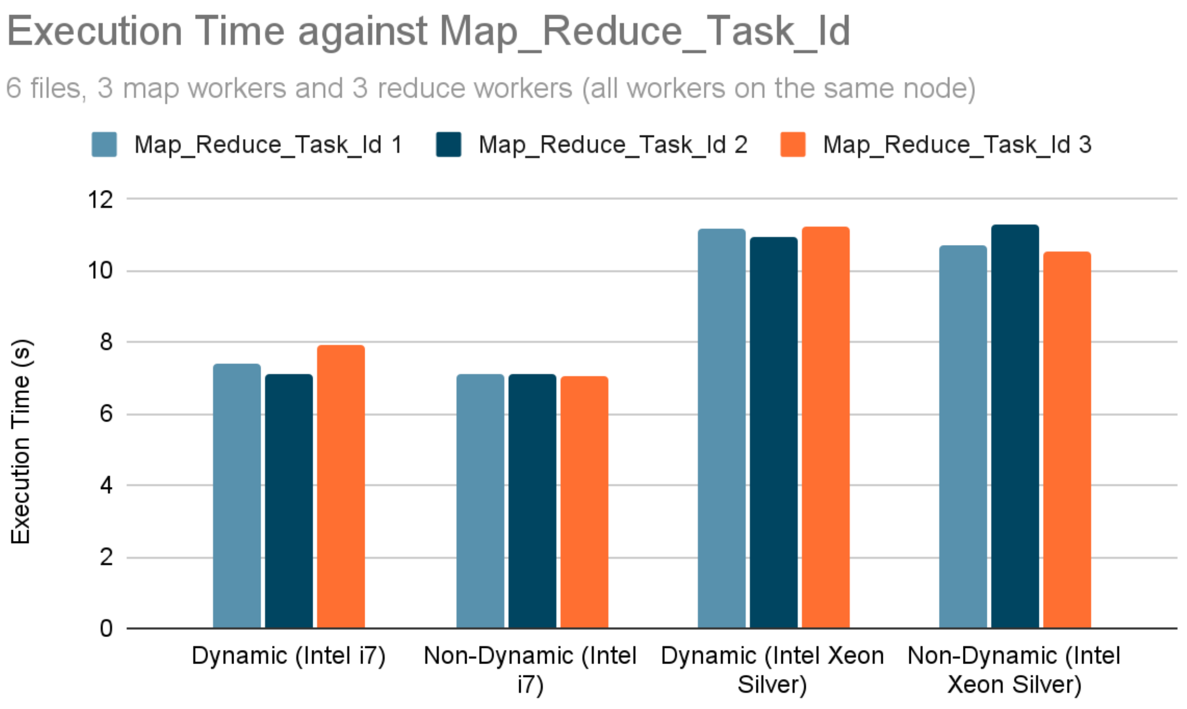

Figure 6

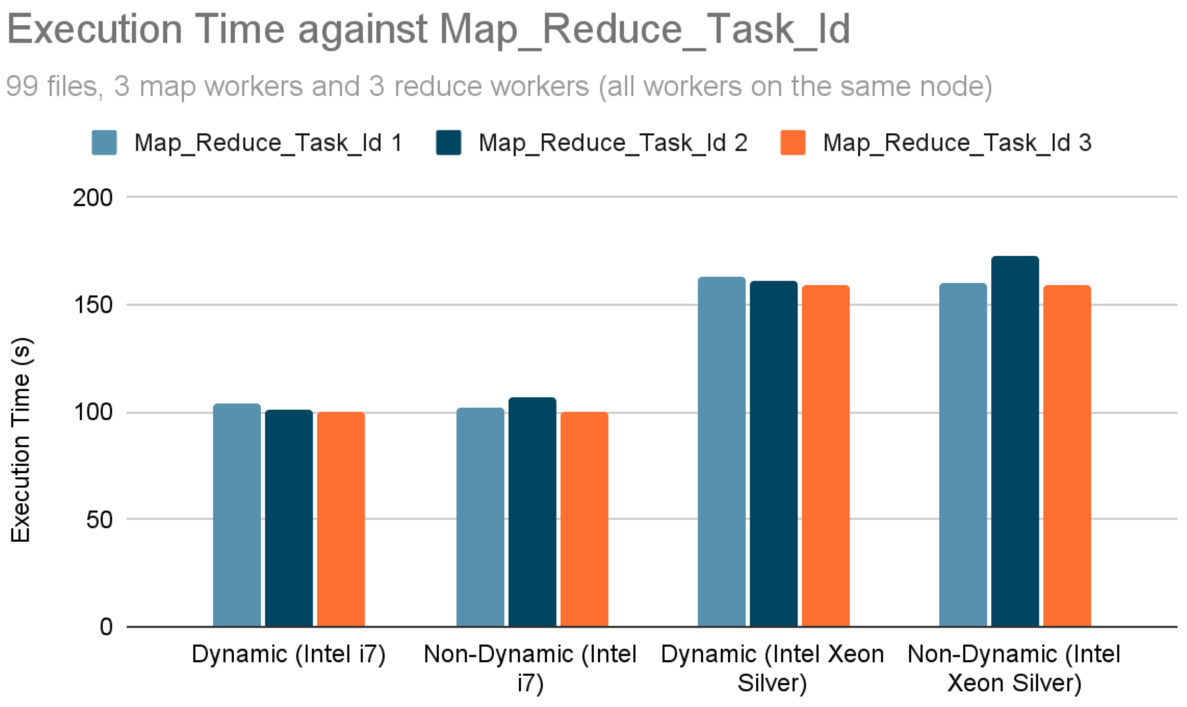

Figure 7

Analysis of results

We postulate that the dynamic version performed an additional blocking receive which the non-dynamic version does not implement. This blocking receive is used to receive a signalling message from whichever node is available to do work. This overhead from blocking might have reduced whatever benefits we expected to gain from a dynamic allocation of work.

Observation 2

Our second observation was that the number of map workers seems to be the bottleneck to the performance of the program and that an increasing number of reducer workers slowed down performance considerably. In our tests, we varied the number of map workers and reducer workers in 3 sets, (3 Map Workers, 3 Reducers), (6 Map Workers, 3 Reducers) and (3 Map Workers, 6 Reducers). We conducted this experiment on 2 sets of input files - the 6 sample input files provided to us and our own custom 99 file test case. Furthermore, we repeated the experiment for each different map function. Since the differences between the different map functions were negligible, we shall just discuss the results for running the experiment with map function number 3. The experiment was repeated for both the non-dynamic and dynamic versions on both the Intel Core i7-7700 and Intel Core Xeon Silver 4114 machines. For the full results, please refer to Annex A-D respectively.

Data

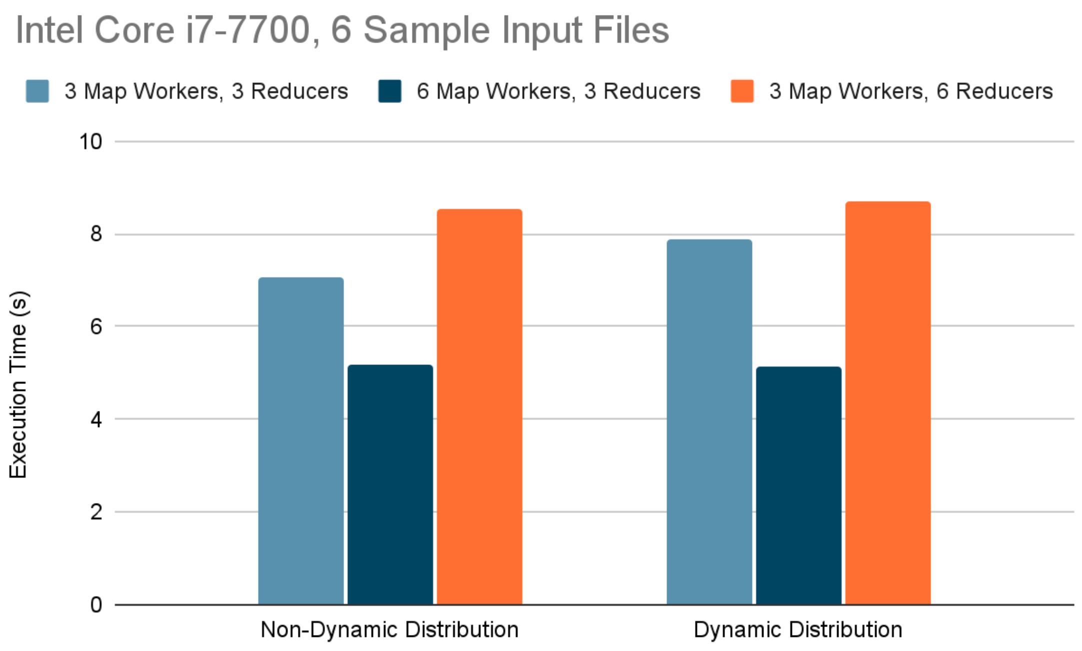

Figure 8

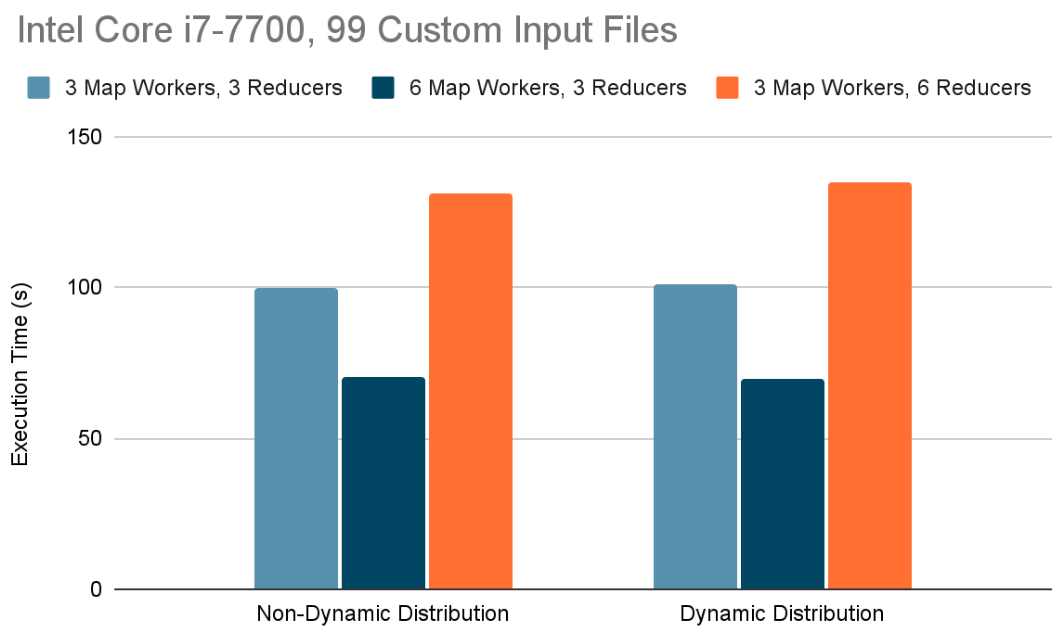

Figure 9

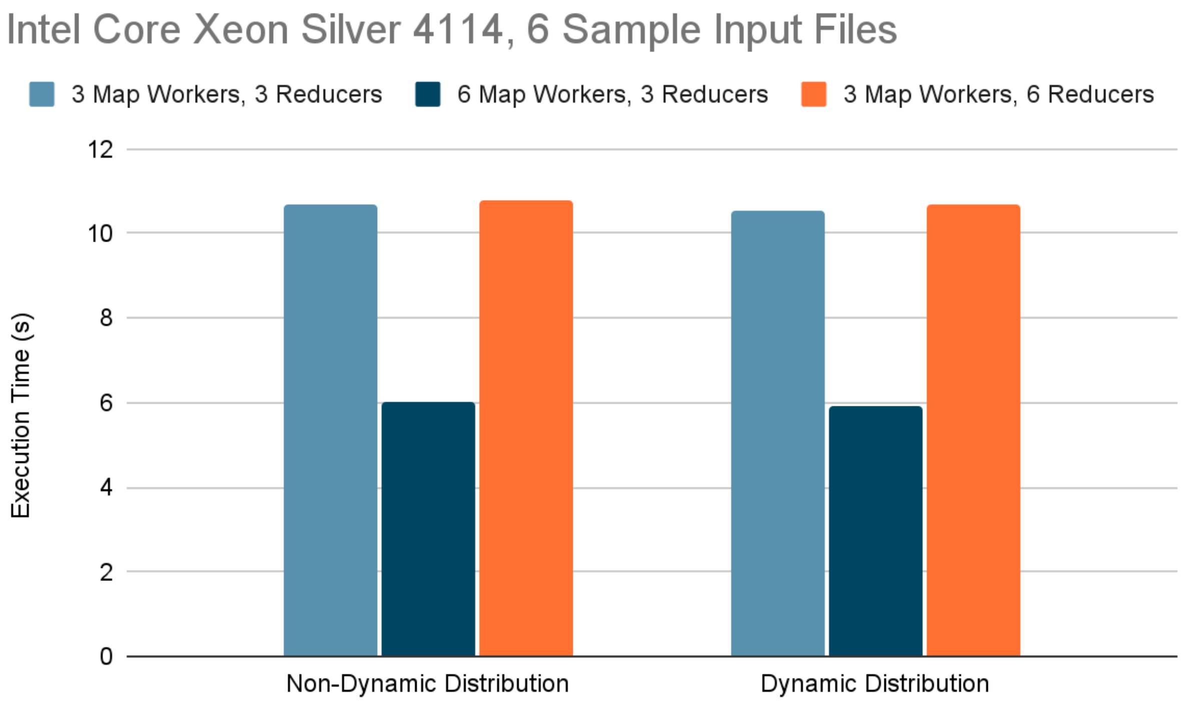

Figure 10

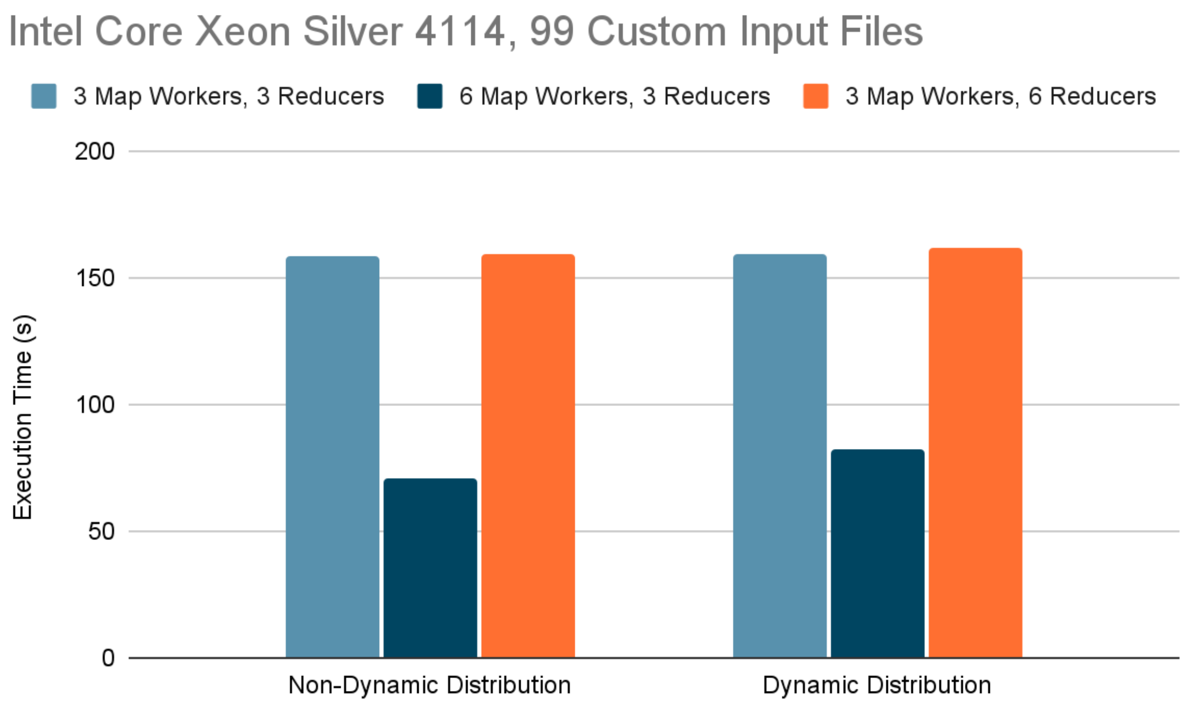

Figure 11

Analysis of Results

There are 2 trends that we can see from the results of our experiment. The first of which is that increasing the number of map workers leads to a decrease of the execution time of the program. This result applies to all the experiments run and hence we can conclude that for our implementation of MapReduce, the number of map workers seems to be a bottleneck. The second trend from our results has to do with the effect that increasing the number of reducer workers have on the performance of our program. We can see that the impact on the performance is different for the Intel Core Xeon Silver 4114 and Intel i7-7700 machines. On the i7-7700, we see that increasing the number of reducer workers actually leads to an increase in execution time. However on the Xeon Silver, there was little impact to the execution time. When we increase the number of reduce workers, we increase the number of partitions available for the map worker to send the map task output to and hence there is greater variance in the destination of the sending of map task output from map worker to reducer worker. Our hypothesis is that the Intel i7-7700, which has fewer cores, suffers in performance as it has to perform context switches more often in order to facilitate this sending to a larger group of reducer workers. However, since the Xeon Silver has a larger number of cores, this effect is not significant and hence we see that there is no impact to its performance.

Observation 3

We attempted to vary the configuration of our program execution to investigate how this would impact the performance of our program. To do so, we came up with four different configurations that we wanted to test. The four configurations are:

Name	Configuration	What we wanted to investigate
A	We attempted to pair map and reducer workers on the same core and run all the processes within the same machine. An example hostfile would be: rank 0=soctf-pdc-012 slot=0:0 rank 1=soctf-pdc-012 slot=0:1 rank 2=soctf-pdc-012 slot-0:2 rank 3=soctf-pdc-012 slot=0:3 rank 4=soctf-pdc-012 slot=0:1 rank 5=soctf-pdc-012 slot=0:2 rank 6=soctf-pdc-012 slot=0:3	Since the Map workers had to send messages to the reducer workers, we wanted to see if running them on the same core would speed up this process
B	We attempted to pair the master and map workers on the same core and rull all the processes within the same machine. An example hostfile would be: rank 0=soctf-pdc-012 slot=0:0 rank 1=soctf-pdc-012 slot=0:1 rank 2=soctf-pdc-012 slot-0:2 rank 3=soctf-pdc-012 slot=0:3 rank 4=soctf-pdc-012 slot=0:0 rank 5=soctf-pdc-012 slot=0:0 rank 6=soctf-pdc-012 slot=0:0	Since the master process has to receive the aggregated outputs from all the reducer workers, we wanted to see if running them on the same core would speed up this process
C	We attempted to pair the map and reducer workers on the same core. The map and reducer workers were run on the same machine while the master process was run on another machine. An example hostfile would be: rank 0=soctf-pdc-012 slot=0:0 rank 1=soctf-pdc-010 slot=0:1 rank 2=soctf-pdc-010 slot-0:2 rank 3=soctf-pdc-010 slot=0:3 rank 4=soctf-pdc-012 slot=0:1 rank 5=soctf-pdc-012 slot=0:2 rank 6=soctf-pdc-012 slot=0:3	We wanted to test how having the map workers on a separate machine would impact the performance of the program
D	The reducer workers and master process were run on the same machine while the map workers were run on another	We wanted to test how having both the map workers and reducer workers on a separate machine would impact the

	machine. An example hostfile would be: rank 0=soctf-pdc-012 slot=0:0 rank 1=soctf-pdc-010 slot=0:1 rank 2=soctf-pdc-010 slot-0:2 rank 3=soctf-pdc-010 slot=0:3 rank 4=soctf-pdc-010 slot=0:1 rank 5=soctf-pdc-010 slot=0:2 rank 6=soctf-pdc-010 slot=0:3	performance of the program

Data

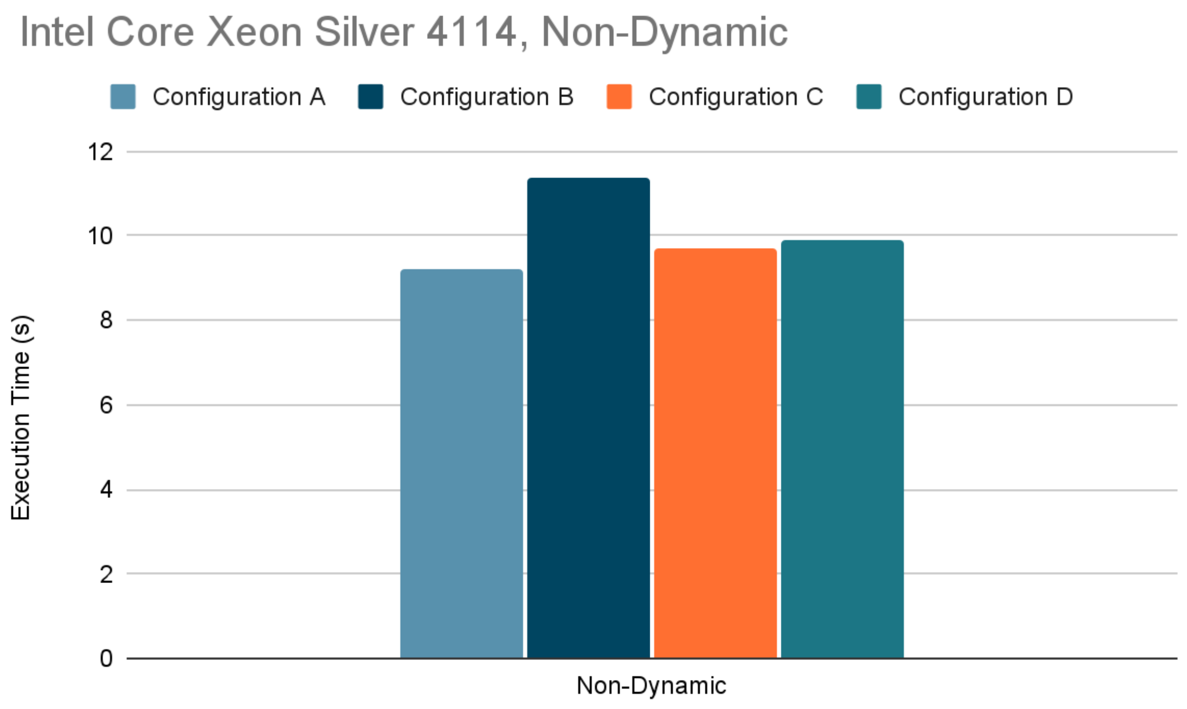

Figure 12

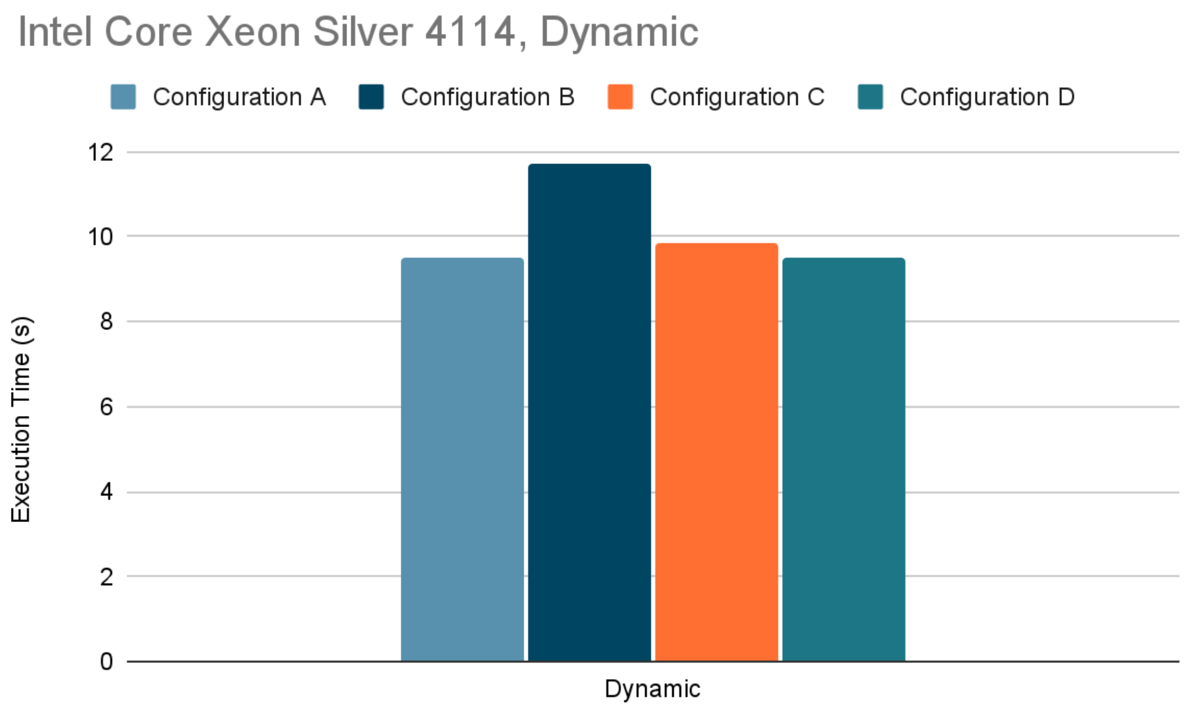

Figure 13

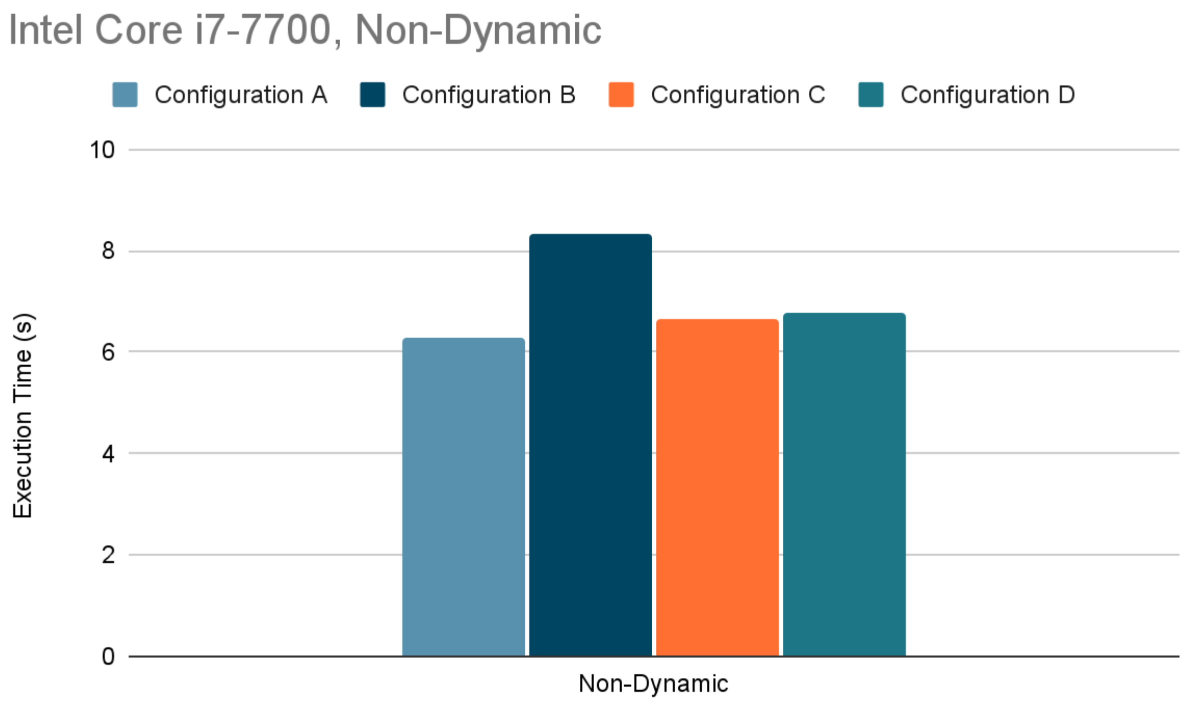

Figure 14

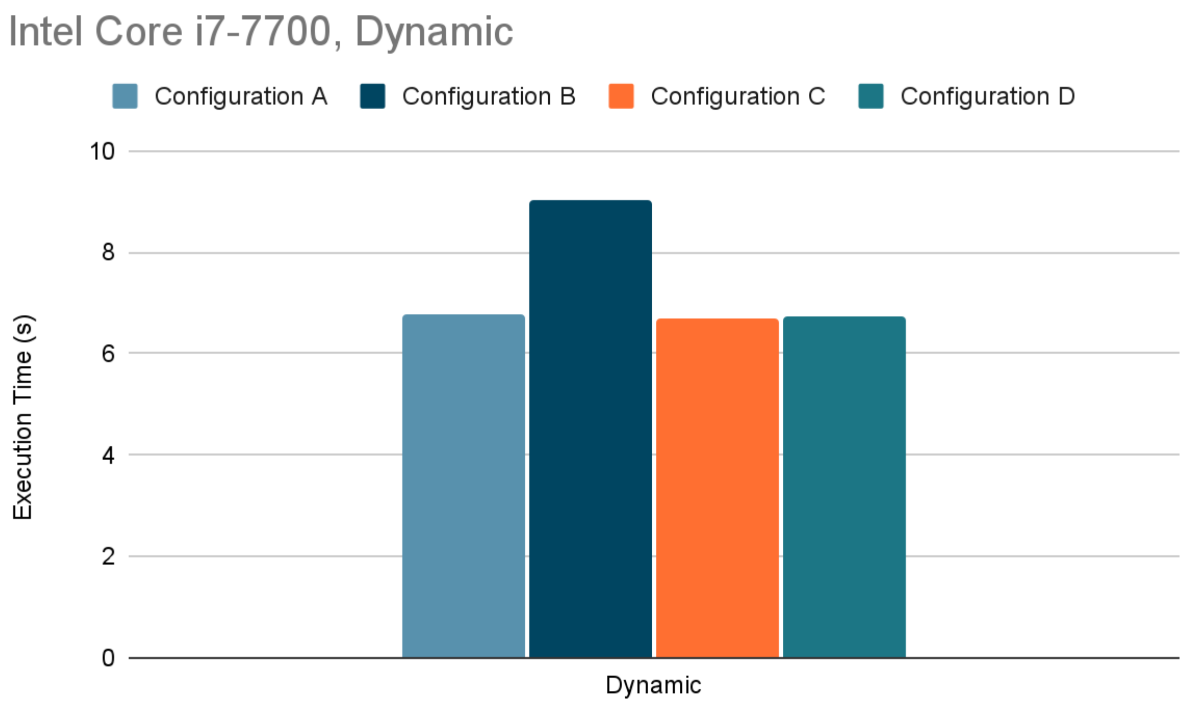

Figure 15

Analysis of Results

From our results, we can see that configurations C and D did little to impact the performance of our program. While we would expect additional latency due to MPI commands sending data across nodes, we do not see this effect in our data collected. This is probably due to the fact that the data we are sending is small in size and hence it is having negligible impact on the performance of our program. Most notably is the noticeable increase in runtime for configuration B. In configuration B we wanted to see if we could improve performance by running the master process and the map workers on the same core. However we ended up with the opposite effect instead. Our explanation for this is that by running these two types of processes on the same core, we increased the number of context switches needed for the Master to send data to the map workers and hence this is what resulted in the additional latency.

(Bonus) Comparison with original MapReduce framework

In this section, we will go through some of the differences identified between our program and the original MapReduce framework written by Dean and Gemawhat at Google[1].

Partitioning of Input Data

In the original implementation, the inputs are partitioned into files of 16-64MB per piece. This is to allow a fair distribution of work to each map worker. Our implementation does not do any initial partitioning and simply takes in each input file as it is. This means that in our implementation, our map workers will have an uneven share of work to do if the input files have a large variance in size.

Distinction between Map and Reduce workers

Our implementation defines the workers very clearly. If a worker is assigned to do mapping, then it will only do mapping, and similarly for reducing. The google implementation allows for the master program to arbitrarily change the workers roles at any point. For instance, if all the mapping tasks are done, the map workers can be converted into reducers to help finish up whatever uncompleted reducing tasks. This maximises the utility of resources and is considerably more efficient than ours.

Master control

In our implementation, our master program only controls the map workers. The master initiates the map reduce process by sending data to the map workers who in turn, will send the intermediate data to the reducers. The only interactions between the master and the reducer worker are 1) reducer sending back the final data to be aggregated by the master, and 2) master sending a terminating signal to the reducer.

In the google implementation, the master program is in constant communication with the map and reducer workers. The master program oversees the whole operation and keeps track of the important details at all times. These details include the state of workers as well as memory locations of the intermediate and final data outputs. In our implementation, the master program does not know anything about these intermediate processes and data.

Fault tolerance

Based on the previous fact that the master program has control and knowledge of its workers in the google implementation, the overall MapReduce operation can have extra fault tolerant mechanisms. For instance, if the overall operation is almost complete and yet one of the map or reduce workers is still taking a long time to finish its task, the master program can utilise idle

[1] Dean, J., & Ghemawat, S. (2004). MapReduce: Simplified data processing on large clusters.

workers as backups. This is helpful if the original worker is somehow performing poorly. The backup worker is used as a replacement and hopefully it can finish faster than the original slow worker. Even if not, only the results from the faster worker will be taken, hence preventing duplicates.

Our implementation has no such fault tolerant mechanism. If one of our workers performs poorly, the master program cannot redelegate the job to another worker.

Location of data

Our implementation assumes no shared memory. This means that all data needs to be passed in full from one worker to another. In order to cut down on unnecessary and costly communications, intermediate data is not communicated to the master program at all.

However, this is different for the google implementation. With shared memory available, the intermediate data are stored in this shared space and the master program has knowledge of these locations. Having shared memory saves on a lot of communication bandwidth since only the addresses of these memory locations need to be communicated.

Conclusion

This assignment has allowed us to experiment with different data distribution methods and node configurations. Unfortunately, and rather surprisingly, our various hypotheses were refuted. We tried our best to run multiple test cases to try and narrow down on the root causes that account for the differences (or lack thereof). Regardless, we have achieved the objective of achieving a good performance speedup throughout our various implementations. Lastly, we have also learned more in detail about the MapReduce framework.

Appendix A

Architecture: Intel Core i7-7700
Nodes: soctf-pdc-012, soctf-pdc-010
Work distribution: Non-dynamic

Map_Reduce_ task_id	Files	No. M_Workers	No. R_Workers	Time taken (s)	Remarks
1	0.txt, 1.txt, 2.txt	1	1	9.924	All workers running on soctf-pdc-012
		3	3	4.464	
	0.txt, 1.txt, 2.txt, 3.txt, 4.txt, 5.txt	1	1	19.235	
		3	3	7.116	
		6	3	5.079	
		3	6	8.288	
		3	3	7.283	M_workers are run on soctf-pdc-012 R_workers are run on soctf-pdc-010
	0.txt, 1.txt, …, 98.txt (custom test case)	3	3	102.316	
		6	3	69.493	All workers running on soctf-pdc-012
		3	6	129.824	
2	0.txt, 1.txt, 2.txt	1	1	10.429	All workers running on soctf-pdc-012
		3	3	4.470	
	0.txt, 1.txt, 2.txt, 3.txt, 4.txt, 5.txt	1	1	19.953	
		3	3	7.095	
		6	3	5.380	
		3	6	8.644	
		3	3	7.465	M_workers are run on soctf-pdc-012 R_workers are run on soctf-pdc-010
	0.txt, 1.txt, …, 98.txt (custom test case)	3	3	106.446	

		6	3	72.052	All workers running on soctf-pdc-012
		3	6	133.855	
3	0.txt, 1.txt, 2.txt	1	1	9.797	All workers running on soctf-pdc-012
		3	3	4.129	
	0.txt, 1.txt, 2.txt, 3.txt, 4.txt, 5.txt	1	1	18.429	
		3	3	7.070	
		6	3	5.182	
		3	6	8.548	
		3	3	7.091	M_workers are run on soctf-pdc-012 R_workers are run on soctf-pdc-010
	0.txt, 1.txt, …, 98.txt (custom test case)	3	3	99.717	
		6	3	70.325	All workers running on soctf-pdc-012
		3	6	130.989	

Appendix B

Architecture: Intel Core i7-7700
Nodes: soctf-pdc-012, soctf-pdc-010
Work distribution: dynamic

Map_Reduce_task_id	Files	No. M_Workers	No. R_Workers	Time taken (s)	Remarks
1	0.txt, 1.txt, 2.txt	1	1	10.019	All workers running on soctf-pdc-012
		3	3	4.548	
	0.txt, 1.txt, 2.txt, 3.txt, 4.txt, 5.txt	1	1	19.868	
		3	3	7.421	
		6	3	5.117	
		3	6	9.141	
		3	3	7.420	M_workers are run on soctf-pdc-012 R_workers are run on soctf-pdc-010
	0.txt, 1.txt, …, 98.txt (custom test case)	3	3	104.068	
		6	3	87.911	All workers running on soctf-pdc-012
		3	6	131.790	
2	0.txt, 1.txt, 2.txt	1	1	9.881	All workers running on soctf-pdc-012
		3	3	4.041	
	0.txt, 1.txt, 2.txt, 3.txt, 4.txt, 5.txt	1	1	18.635	
		3	3	7.128	
		6	3	5.597	
		3	6	9.638	
		3	3	7.012	M_workers are run on soctf-pdc-012 R_workers are run on soctf-pdc-010
	0.txt, 1.txt, …, 98.txt (custom test case)	3	3	101.081	

		6	3	69.634	All workers running on soctf-pdc-012
		3	6	134.559	
3	0.txt, 1.txt, 2.txt	1	1	9.733	All workers running on soctf-pdc-012
		3	3	4.142	
	0.txt, 1.txt, 2.txt, 3.txt, 4.txt, 5.txt	1	1	18.619	
		3	3	7.897	
		6	3	5.130	
		3	6	8.695	
		3	3	7.167	M_workers are run on soctf-pdc-012 R_workers are run on soctf-pdc-010
	0.txt, 1.txt, …, 98.txt (custom test case)	3	3	100.440	
		6	3	82.373	All workers running on soctf-pdc-012
		3	6	133.111	

Appendix C

Architecture: Intel Core Xeon Silver 4114
Nodes: soctf-pdc-002, soctf-pdc-003
Work distribution: Non-dynamic

Map_Reduce_ task_id	Files	No. M_Workers	No. R_Workers	Time taken (s)	Remarks
1	0.txt, 1.txt, 2.txt	1	1	14.769	All workers running on soctf-pdc-002
		3	3	6.121	
	0.txt, 1.txt, 2.txt, 3.txt, 4.txt, 5.txt	1	1	27.997	
		3	3	10.689	
		6	3	6.010	
		3	6	10.761	
		3	3	10.540	M_workers are run on soctf-pdc-002 R_workers are run on soctf-pdc-003
	0.txt, 1.txt, …, 98.txt (custom test case)	3	3	159.830	
		6	3	84.662	All workers running on soctf-pdc-002
		3	6	160.124	
2	0.txt, 1.txt, 2.txt	1	1	15.649	All workers running on soctf-pdc-002
		3	3	6.561	
	0.txt, 1.txt, 2.txt, 3.txt, 4.txt, 5.txt	1	1	29.972	
		3	3	11.308	
		6	3	6.333	
		3	6	11.509	
		3	3	11.136	M_workers are run on soctf-pdc-002 R_workers are run on soctf-pdc-003
	0.txt, 1.txt, …, 98.txt (custom test case)	3	3	172.234	

		6	3	90.628	All workers running on soctf-pdc-002
		3	6	171.662	
3	0.txt, 1.txt, 2.txt	1	1	14.664	All workers running on soctf-pdc-002
		3	3	5.990	
	0.txt, 1.txt, 2.txt, 3.txt, 4.txt, 5.txt	1	1	27.543	
		3	3	10.520	
		6	3	5.916	
		3	6	10.708	
		3	3	10.560	M_workers are run on soctf-pdc-002 R_workers are run on soctf-pdc-003
	0.txt, 1.txt, …, 98.txt (custom test case)	3	3	158.739	
		6	3	70.606	All workers running on soctf-pdc-002
		3	6	159.617	

Appendix D

Architecture: Intel Core Xeon Silver 4114
Nodes: soctf-pdc-002, soctf-pdc-003
Work distribution: Dynamic

Map_Reduce_ task_id	Files	No. M_Workers	No. R_Workers	Time taken (s)	Remarks
1	0.txt, 1.txt, 2.txt	1	1	16.191	All workers running on soctf-pdc-002
		3	3	6.349	
	0.txt, 1.txt, 2.txt, 3.txt, 4.txt, 5.txt	1	1	30.506	
		3	3	11.182	
		6	3	6.013	
		3	6	10.919	
		3	3	11.464	M_workers are run on soctf-pdc-002 R_workers are run on soctf-pdc-003
	0.txt, 1.txt, …, 98.txt (custom test case)	3	3	163.042	
		6	3	84.003	All workers running on soctf-pdc-002
		3	6	163.199	
2	0.txt, 1.txt, 2.txt	1	1	15.671	All workers running on soctf-pdc-002
		3	3	6.411	
	0.txt, 1.txt, 2.txt, 3.txt, 4.txt, 5.txt	1	1	29.895	
		3	3	10.961	
		6	3	5.807	
		3	6	10.725	
		3	3	11.182	M_workers are run on soctf-pdc-002 R_workers are run on soctf-pdc-003
	0.txt, 1.txt, …, 98.txt (custom test case)	3	3	160.818	

		6	3	85.151	All workers running on soctf-pdc-002
		3	6	164.705	
3	0.txt, 1.txt, 2.txt	1	1	15.883	All workers running on soctf-pdc-002
		3	3	6.283	
	0.txt, 1.txt, 2.txt, 3.txt, 4.txt, 5.txt	1	1	30.182	
		3	3	11.247	
		6	3	5.826	
		3	6	10.591	
		3	3	11.239	M_workers are run on soctf-pdc-002 R_workers are run on soctf-pdc-003
	0.txt, 1.txt, …, 98.txt (custom test case)	3	3	159.139	
		6	3	82.368	All workers running on soctf-pdc-002
		3	6	162.105	

Appendix E

Architecture: Intel Core Xeon Silver 4114
Nodes: soctf-pdc-002, soctf-pdc-003
Work distribution: Non-Dynamic

Map_Reduce_task_id	Files	No. M_Workers	No. R_Workers	Time taken (s)	Remarks
3	0.txt, 1.txt, 2.txt, 3.txt, 4.txt, 5.txt	3	3	9.202	rank 0=soctf-pdc-002 slot=0:0 rank 1=soctf-pdc-002 slot=0:1 rank 2=soctf-pdc-002 slot-0:2 rank 3=soctf-pdc-002 slot=0:3 rank 4=soctf-pdc-002 slot=0:1 rank 5=soctf-pdc-002 slot=0:2 rank 6=soctf-pdc-002 slot=0:3
				11.373	rank 0=soctf-pdc-002 slot=0:0 rank 1=soctf-pdc-002 slot=0:1 rank 2=soctf-pdc-002 slot-0:2 rank 3=soctf-pdc-002 slot=0:3 rank 4=soctf-pdc-002 slot=0:0 rank 5=soctf-pdc-002 slot=0:0 rank 6=soctf-pdc-002 slot=0:0
				9.695	rank 0=soctf-pdc-002 slot=0:0 rank 1=soctf-pdc-003 slot=0:1 rank 2=soctf-pdc-003 slot-0:2 rank 3=soctf-pdc-003 slot=0:3 rank 4=soctf-pdc-002 slot=0:1

					rank 5=soctf-pdc-002 slot=0:2 rank 6=soctf-pdc-002 slot=0:3
				9.925	rank 0=soctf-pdc-002 slot=0:0 rank 1=soctf-pdc-003 slot=0:1 rank 2=soctf-pdc-003 slot-0:2 rank 3=soctf-pdc-003 slot=0:3 rank 4=soctf-pdc-003 slot=0:1 rank 5=soctf-pdc-003 slot=0:2 rank 6=soctf-pdc-003 slot=0:3

Architecture: Intel Core Xeon Silver 4114
Nodes: soctf-pdc-002, soctf-pdc-003
Work distribution: Dynamic

Map_Reduce_ task_id	**Files**	**No. M_Workers**	**No. R_Workers**	**Time taken (s)**	**Remarks**
3	0.txt, 1.txt, 2.txt, 3.txt, 4.txt, 5.txt	3	3	9.53	rank 0=soctf-pdc-002 slot=0:0 rank 1=soctf-pdc-002 slot=0:1 rank 2=soctf-pdc-002 slot-0:2 rank 3=soctf-pdc-002 slot=0:3 rank 4=soctf-pdc-002 slot=0:1 rank 5=soctf-pdc-002 slot=0:2 rank 6=soctf-pdc-002 slot=0:3
				11.701	rank 0=soctf-pdc-002 slot=0:0

					rank 1=soctf-pdc-002 slot=0:1 rank 2=soctf-pdc-002 slot-0:2 rank 3=soctf-pdc-002 slot=0:3 rank 4=soctf-pdc-002 slot=0:0 rank 5=soctf-pdc-002 slot=0:0 rank 6=soctf-pdc-002 slot=0:0
				9.873	rank 0=soctf-pdc-002 slot=0:0 rank 1=soctf-pdc-003 slot=0:1 rank 2=soctf-pdc-003 slot-0:2 rank 3=soctf-pdc-003 slot=0:3 rank 4=soctf-pdc-002 slot=0:1 rank 5=soctf-pdc-002 slot=0:2 rank 6=soctf-pdc-002 slot=0:3
				9.522	rank 0=soctf-pdc-002 slot=0:0 rank 1=soctf-pdc-003 slot=0:1 rank 2=soctf-pdc-003 slot-0:2 rank 3=soctf-pdc-003 slot=0:3 rank 4=soctf-pdc-003 slot=0:1 rank 5=soctf-pdc-003 slot=0:2 rank 6=soctf-pdc-003 slot=0:3

Architecture: Intel Core i7-7700
Nodes: soctf-pdc-012, soctf-pdc-010
Work distribution: Non-dynamic

Map_Reduce_ task_id	Files	No. M_Workers	No. R_Workers	Time taken (s)	Remarks
3	0.txt, 1.txt, 2.txt, 3.txt, 4.txt, 5.txt	3	3	6.270	rank 0=soctf-pdc-012 slot=0:0 rank 1=soctf-pdc-012 slot=0:1 rank 2=soctf-pdc-012 slot-0:2 rank 3=soctf-pdc-012 slot=0:3 rank 4=soctf-pdc-012 slot=0:1 rank 5=soctf-pdc-012 slot=0:2 rank 6=soctf-pdc-012 slot=0:3
				8.345	rank 0=soctf-pdc-012 slot=0:0 rank 1=soctf-pdc-012 slot=0:1 rank 2=soctf-pdc-012 slot-0:2 rank 3=soctf-pdc-012 slot=0:3 rank 4=soctf-pdc-012 slot=0:0 rank 5=soctf-pdc-012 slot=0:0 rank 6=soctf-pdc-012 slot=0:0
				6.6413	rank 0=soctf-pdc-012 slot=0:0 rank 1=soctf-pdc-010 slot=0:1 rank 2=soctf-pdc-010 slot-0:2 rank 3=soctf-pdc-010 slot=0:3 rank 4=soctf-pdc-012 slot=0:1 rank 5=soctf-pdc-012 slot=0:2 rank 6=soctf-pdc-012 slot=0:3
				6.778	rank 0=soctf-pdc-012 slot=0:0

					rank 1=soctf-pdc-010 slot=0:1 rank 2=soctf-pdc-010 slot-0:2 rank 3=soctf-pdc-010 slot=0:3 rank 4=soctf-pdc-010 slot=0:1 rank 5=soctf-pdc-010 slot=0:2 rank 6=soctf-pdc-010 slot=0:3

Architecture: Intel Core i7-7700
Nodes: soctf-pdc-012, soctf-pdc-010
Work distribution: dynamic

Map_Reduce_ task_id	**Files**	**No. M_Workers**	**No. R_Workers**	**Time taken (s)**	**Remarks**
3	0.txt, 1.txt, 2.txt, 3.txt, 4.txt, 5.txt	3	3	6.781	rank 0=soctf-pdc-012 slot=0:0 rank 1=soctf-pdc-012 slot=0:1 rank 2=soctf-pdc-012 slot-0:2 rank 3=soctf-pdc-012 slot=0:3 rank 4=soctf-pdc-012 slot=0:1 rank 5=soctf-pdc-012 slot=0:2 rank 6=soctf-pdc-012 slot=0:3
				9.042	rank 0=soctf-pdc-012 slot=0:0 rank 1=soctf-pdc-012 slot=0:1 rank 2=soctf-pdc-012 slot-0:2 rank 3=soctf-pdc-012 slot=0:3 rank 4=soctf-pdc-012 slot=0:0 rank 5=soctf-pdc-012 slot=0:0

					rank 6=soctf-pdc-012 slot=0:0
				6.6843	rank 0=soctf-pdc-012 slot=0:0 rank 1=soctf-pdc-010 slot=0:1 rank 2=soctf-pdc-010 slot-0:2 rank 3=soctf-pdc-010 slot=0:3 rank 4=soctf-pdc-012 slot=0:1 rank 5=soctf-pdc-012 slot=0:2 rank 6=soctf-pdc-012 slot=0:3
				6.75	rank 0=soctf-pdc-012 slot=0:0 rank 1=soctf-pdc-010 slot=0:1 rank 2=soctf-pdc-010 slot-0:2 rank 3=soctf-pdc-010 slot=0:3 rank 4=soctf-pdc-010 slot=0:1 rank 5=soctf-pdc-010 slot=0:2 rank 6=soctf-pdc-010 slot=0:3