CS3210 Assignment 3

Aaron Sng (A0242334N)

15 November 2021

1 Machines Used

All CPU types were tested. The list below sums up the list of machines used.

- Intel i7-7700: `soctf-pdc-013` - `soctf-pdc-016`
- Intel i7-9700: `soctf-pdc-021`
- Intel Xeon Silver 4114: `soctf-pdc-005` - `soctf-pdc-009`
- Intel Dual Socket Xeon Silver 4114: `soctf-pdc-019`
- Intel Xeon Gold 2245: `soctf-pdc-023`

2 Program Design

2.1 Parallel Programming Pattern

The parallel programming pattern has been stipulated in the assignment requirements. In this implementation of the MapReduce algorithm, the master-worker programming pattern has been chosen. Each worker will receive an individual file, from which the MapReduce will be executed on the specified file. There are two different types of workers - one map worker, and another reduce worker. The map worker performs the mapping task specified before runtime, the reduce worker reduces the key-value pairs received from the map workers into a singular key-value pair representation. The respective reduce processes will send the key-value pairs to the master process. Upon obtaining the key-value pairs from the reduce process, the master collates the obtained key-value pairs and produces an output file.

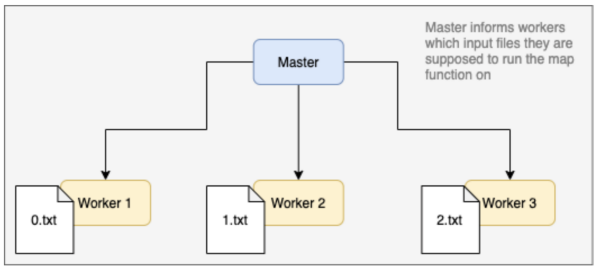

Figure 1: Distribution of Files Across Workers (Adapted from the Assignment Sheet)

2.2 Message Passing Design

MapReduce was implemented using the Message Passing Interface (MPI), a library specification for implementing parallel algorithms on distributed systems. Unlike the original implementation by Google, the map and reduce workers are not able to alternate between map and reduce functions upon process creation; the map and reduce workers are homogeneous upon process creation and will not change function. As such, it makes it hard to use collective communication mechanisms. Therefore, point-to-point communication was heavily relied on in this implementation of MapReduce. In this implementation of the MapReduce worker, the files are passed in a relatively sequential fashion. Figure 2 depicts the flow of a single file through the MapReduce algorithm.

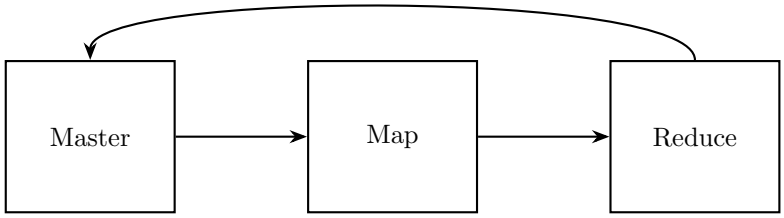

Figure 2: MapReduce Implementation on MPI

2.3 Master-Map Worker Interfacing

To further elaborate on how the different map workers work with the master process, two iterations of design changes will be discussed in further sections. The list below summarises the function of each design iteration.

1. First Design Iteration: Aggregate all the files into a single workload and divide the workload evenly to all map workers
2. Final Design Iteration: Send whole files individually to each map worker

2.3.1 First Design Iteration: Aggregating the Files

In this design iteration, files are aggregated into a single string before the workload is even out across processes. With even workload across processes, the chances of processor starvation will be minimised, improving efficiency of the program. Such a design pattern follows the block distribution scheme. As a result of the even workload across processors, overall communication between processes can be greatly minimised. Communication between the master and map workers only happens once, and the mapping phase concludes. Such a communication paradigm minimises overall communication, and reduces the chance of deadlocks in the system. However, such a design approach is not practical, as the MapReduce requires a whole file instead. This design iteration therefore may not be fully correct. However, such an implementation can help us to fully understand the effect of minimising communications. Figure 3 illustrates this implementation.

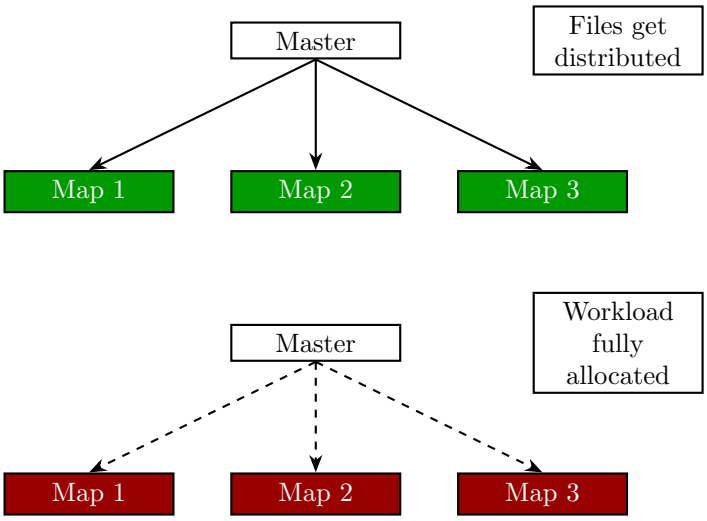

Figure 3: Aggregating Files with MapReduce

2.3.2 Final Design Iteration: A File to Each Worker

Files are not aggregated in this design iteration. This distribution scheme follows the block-cyclic distribution. This implementation is block-cyclic in the sense that each map worker receives one file each, but each file may not have the same file size. Hence, the chances of processor starvation increases, affecting the overall program's efficiency. To mitigate this, a thread-pool is used. Upon the completion of any file, the corresponding worker will ping the master and the next file will be sent to that free map worker. Figure 4 illustrates this thread-pool in action, and figure 5 illustrates the communication between the master and map reduce workers. A result of this implies that there is overall increased communication in the whole program. Effect of increased communication will be further discussed in the discussion section of this report.

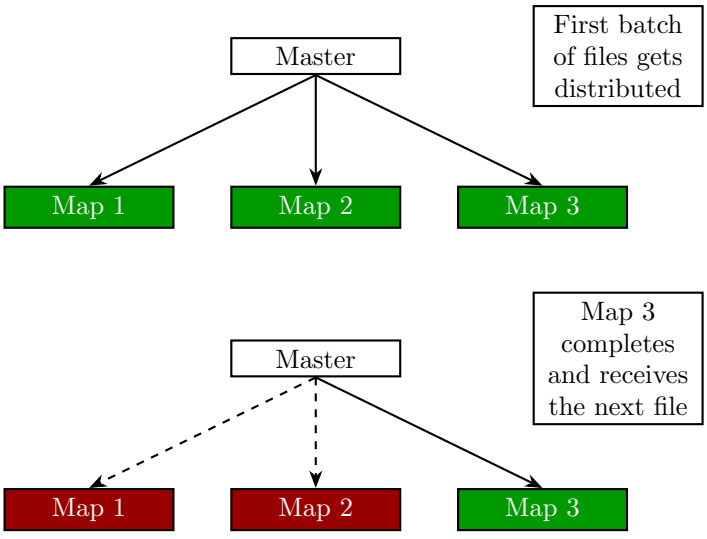

Figure 4: A File to Each Worker in MapReduce

Master

```
// Map phase
while (files not sent) {
    MPI_Recv(..., MPI_ANY_SOURCE,...,
        &status)
    MPI_Send(file, status.MPI_SOURCE,...)
}

while (map workers not aware to stop) {
    MPI_Send(-1, ..., MAP_AND_WORKERS);
}
```

Map Worker

```
MPI_Send(..., MASTER,...)
MPI_Recv(file, MASTER,..., &status)
if (file == -1)
   break;
Map_Process();
```

Figure 5: Interfacing between Master and Map Worker Threads

2.3.3 Deadlocks and Race Conditions

Figure 5 also illustrates clearly the communication interface between the map workers and master. In the interfacing between processes, every send process requires a corresponding receive process. Having the communication between these processes designed in such a manner eliminates the need of a system buffer. The possibility of a deadlock is therefore greatly diminished. In addition, this communication interface between map workers and master is configured such that it happens sequentially. Thus, this reduces the risk of memory contention.

2.3.4 Termination of Master-Map Worker Interfacing

The master worker would stop sending files when all files have been sent. Upon sending all files, a value of -1 would be sent to all map workers. The value of -1 will let all map workers know to stop mapping.

2.4 Map Workers-Reduce Workers Interfacing

After all map workers are completed, the generated key-value pairs are partitioned. The number of partitions are decided based on the number of reduce workers spawned. This follows a pseudo-blockwise distribution of key-value pairs across the processors, where each processor take a similar set of key-value pairs. Subsequently, each map worker will send a partition of key-value pairs to its corresponding reduce worker. Overall, the blockwise distribution reduces the communication needed by all processes.

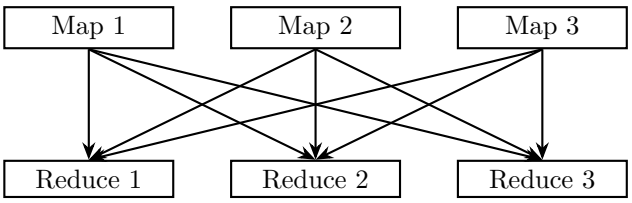

Figure 6: Communication Between Map Workers and Reduce Workers

In a similar fashion to how the Map Workers and Master thread interfaces, the Map Worker would only send its respective partition to the Reduce Worker when the Reduce Worker is available. This

communication instance assumes the absence of a system buffer, which reduces the chances of a deadlock. Like the Map Process, the Reduce Process happens almost sequentially. This communication will also continue despite not all files having completed its mapping process. Figure 7 illustrates the implementation in pseudocode.

Map Process	**Reduce Process**

`// Mapping of KeyValue Pairs`
`// Serialising it into a communication message`
`MPI_Send(KEY_VALUE_PAIRS, PARTITION_ID, ...);`
	`MPI_Recv(KEY_VALUE_PAIRS, ANY_MAP_WORKER,..., &status);`
	`// Reduce process happens here`

Figure 7: Interfacing between Map and Reduce Threads

2.5 Reduce Workers-Master Interfacing

The reduce worker would terminate upon receiving the KeyValue pairs from all Map Workers. It tracks this through a tracker which ensures that each reduce worker has received information from all files. It will collate the partition that has been allocated to it and send it back to the master. The collated key-value pairs will only be sent to master if there's a matching receive process for it. As with the case for the master-map worker and map-reduce worker interfaces, this communication assumes that there is no system buffer. This reduces the chances of a deadlock.

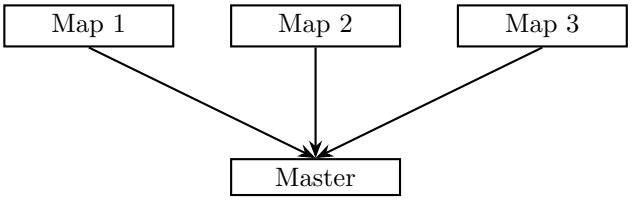

Figure 8: Communication Between Reduce Workers and Master

Reduce Process	**Master**

`// Completed KeyValue Pairs`
`// Serialising it into a communication message`
`MPI_Send(KEY_VALUE_PAIRS, MASTER, ...);`
	`// Reduce Phase`
	`while (all files are not received)`
	`MPI_Recv(KEY_VALUE_PAIRS, ANY_REDUCE_WORKER,..., &status);`
	`// Create and store into output file`

Figure 9: Interfacing between Reduce Threads and Master

2.6 Special Considerations

2.6.1 Endianness of Compute Cluster

The serialising of key-value pairs for communication between workers takes into assumption that the system is little endian. A proprietary byte configuration has been created for communication between the master, map and reduce workers. The first 8 bytes stores the key, while the remaining 4 bytes store the integer. Assuming that the system is little endian, the logical shift left of 24 bits would be made to the smallest address to be stored into this byte configuration. This shift of bits continue for the next 3 bytes, with each shift size reducing by 8 bits each iteration. Figure 10 illustrates this algorithm. Figure 11 depicts the organisation for a key-value pair in the character stream.

```
/* Function to convert KeyValue pair to a char array of 12 bytes */
void keyvalue_to_char_stream(KeyValue * kvs, unsigned char * output) {
    strncpy(output, kvs->key, KEY_LEN);
    unsigned long val = kvs->val;
    *(output + KEY_LEN) = (val >> 24) & 0xFF;
    *(output + KEY_LEN + 1) = (val >> 16) & 0xFF;
    *(output + KEY_LEN + 2) = (val >> 8) & 0xFF;
    *(output + KEY_LEN + 3) = val & 0xFF;
}
```

Figure 10: Converting Key into the Char Stream

Key[0]	Key[1]	Key[2]	Key[3]	Key[4]	Key[5]	Key[6]	Key[7]	Val[0]	Val[1]	Val[2]	Val[3]

Figure 11: Character Stream Organisation for a Key-Value Pair

2.7 Differences from Google's Implementation

- Google's implementation uses a distributed file system while the current implementation file system is localised in the master's file system.
- Each worker thread spawned in Google's implementation are able to alternate between map and reduce processes, while for the current implementation, each worker process is only able to be either a map or reduce process.
- When key-value pairs are generated from the map process or are reduced, these key-value pairs are stored as a text file in the distributed file system and to be retrieved by the next stage for further processing. The current implementation on the other hand strictly uses point-to-point communication between processes due to the absence of a distributed file system.
- Google implements backup operations to curtail inefficiencies from 'stragglers'. 'Stragglers' are nodes or processors within the distributed system that hog up the overall processing of the inputs, due to faults or bandwidth problems. These backup operations ensure that the system is able to compute the problem quicker, as they run in parallel and ensure that the program completes faster.
- The order of the output in the current implementation is arbitrarily and depends largely on 'First Come First Serve' basis from the processors. Google's implementation, however, orders the keys by processing keys in increasing order.

3 Methods

3.1 Running Experiments

With the `DEBUG` parameter in `utils.h` set to 1, the program will print to `stdout` debug statements and the time taken to run MapReduce. After compiling the program with `make build`, Slurm was used to benchmark the program. Within the submission, there is a `batch_script.sh`. This file runs in the user home directory. Upon copying the `testcases` and compiled executable to the nfs partition, the Slurm script can run with the following commands. The Slurm script will check with the `correct_outputs` folder at the end of running the experiment. If a mismatch occurs, the line with the wrong output will be printed.

```
# Profile the performance of the program.
chmod +x batch_script.sh
sbatch -p locked --constraint="[<CPU-type>*<Number-of-Machines>]" batch_script.sh
```

`CPU-type` and `Number-of-Machines` will be varied according to the machine being tested on. Machine types are listed in section 1 of this report.

3.2 Additional Testcases

Number of input files across all Slurm jobs are set at 9. These testcases will be used as input and will be fed into to the MapReduce algorithm in an ascending order (a filename 0.txt will be fed into MapReduce before a filename 1.txt). Three additional `testcases` have been downloaded from Project Gutenberg from gutenberg.org. Below is the list of additional input files used. Kindly note that these files are used purely for academic purposes and have been chosen due to factors such as the file size and the different encoding sets with different alphabets used.

- The King James Bible
- Atsumono by Junichiro Tanizaki
- Han Shu by Gu Ban

4 Results

Unless otherwise specified, the program input and its corresponding execution time would be on 9 testcases.

4.1 Assumptions

- No other programs in the background were running
- Slurm uses little computation resources

4.2 Execution Time on the Same Number of Map and Reduce Workers

In this section, number of map workers equal to the number of reduce workers. The speedups computed are taken by dividing execution time when 1 map / reduce workers are used by when 3 map / reduce workers are used.

4.2.1 Intel i7-7700

A speedup of 2.93x was observed. The plot as shown below depicts computed for the three different map functions were used. The execution times were averaged.

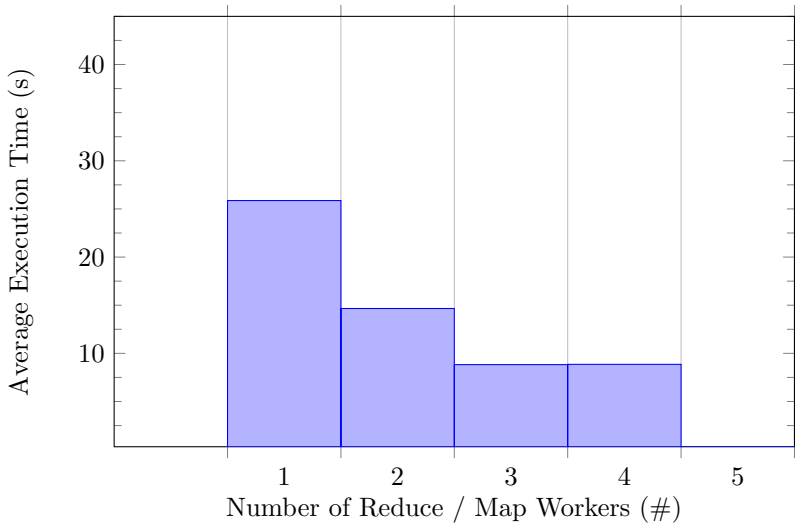

Figure 12: Execution Time against Map / Reduce workers on the Intel 7700

4.2.2 Intel i7-9700

A speedup of 2.84x was observed. The plot as shown below depicts computed for the three different map functions were used. The execution times were averaged.

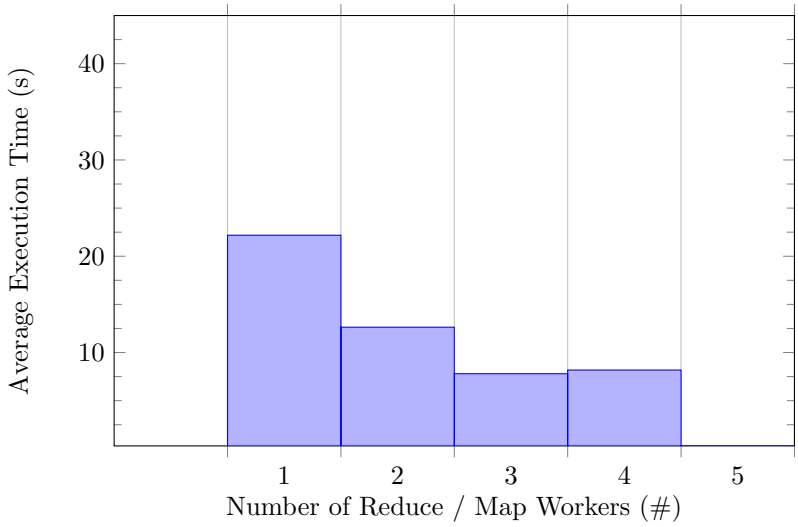

Figure 13: Execution Time against Map / Reduce workers on the Intel 9700

4.2.3 Intel Xeon Silver 4414

A speedup of 2.80x was observed. The plot as shown below depicts computed for the three different map functions were used. The execution times were averaged.

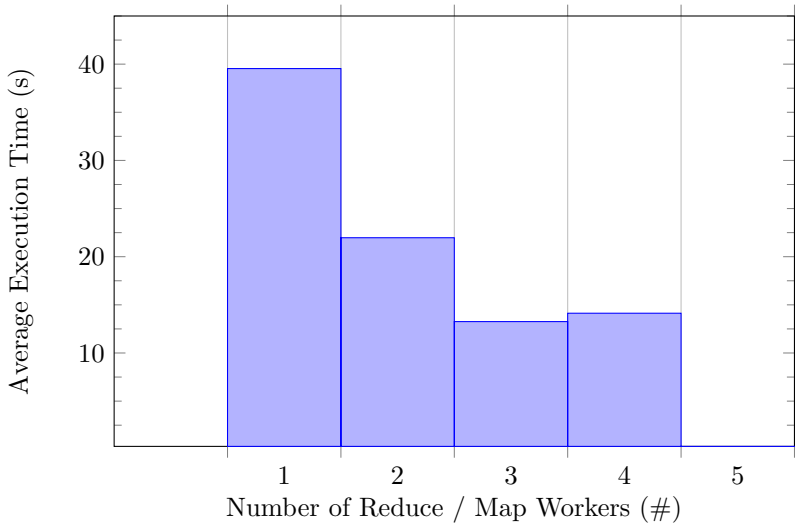

Figure 14: Execution Time against Map / Reduce workers on the Intel Xeon Silver 4114

4.2.4 Intel Dual-Socket Xeon Silver 4414

A speedup of 2.76x was observed. The plot as shown below depicts computed for the three different map functions were used. The execution times were averaged.

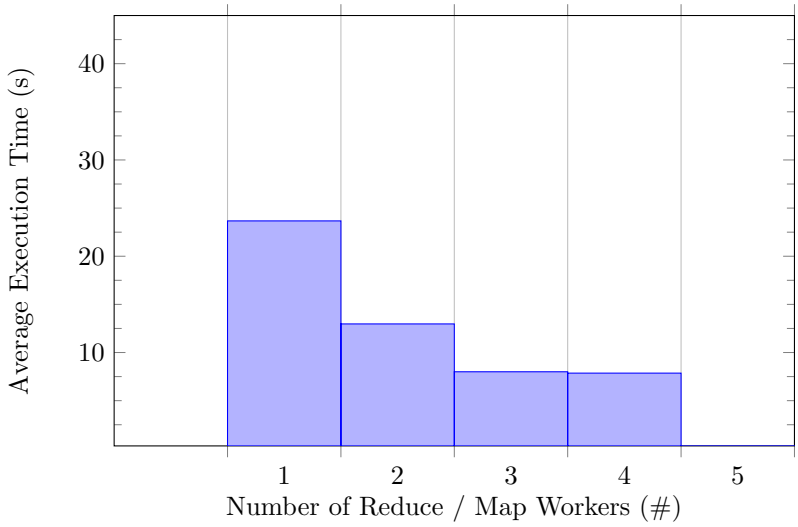

Figure 15: Execution Time against Map / Reduce workers on the Intel Dual-Socket Xeon Silver 4414

4.2.5 Intel Xeon Gold 2245

A speedup of 2.80x was observed. The plot as shown below depicts computed for the three different map functions were used. The execution times were averaged.

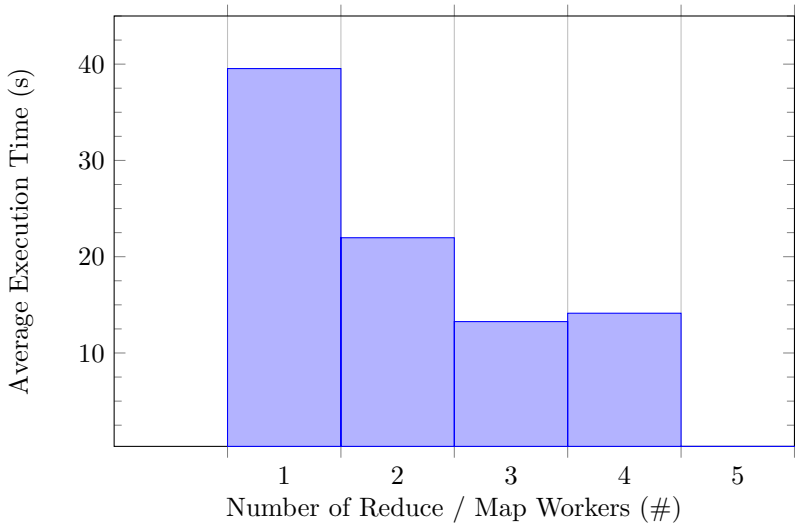

Figure 16: Execution Time against Map / Reduce workers on the Intel Xeon Gold 2245

4.3 Execution Time on the Different Number of Map and Reduce Workers

In this section, number of map workers are not equal to the number of reduce workers. The purpose of this section is to stress test for potential deadlocks or race conditions that can lead to an incorrect output. This stress test is only done on 2 Intel Xeon Silver 4114 nodes.

Figure 17: Execution Time against Reduce workers on the Intel Xeon Silver 4114

4.4 Execution Time on Different Machines

In this section, number of map workers are equal to the number of reduce workers. The purpose of this section is to understand the communication overhead caused by having a distributed algorithm operating on different machines. All machine types in section 1 were used. Appendix A details the machines used at the different configurations mentioned here.

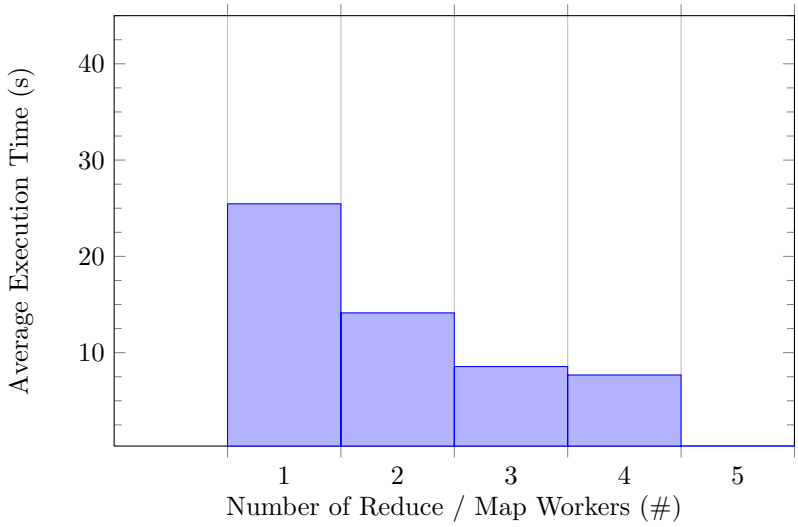

Figure 18: Execution Time against Reduce workers on Different Machines

4.5 Execution Time on First Design Iteration's Distribution Scheme

In this section, number of map workers are equal to the number of reduce workers. The purpose of this section is to understand the communication overhead savings caused by a blockwise distribution scheme. For comparison, the Intel Xeon Silver 4114 was used.

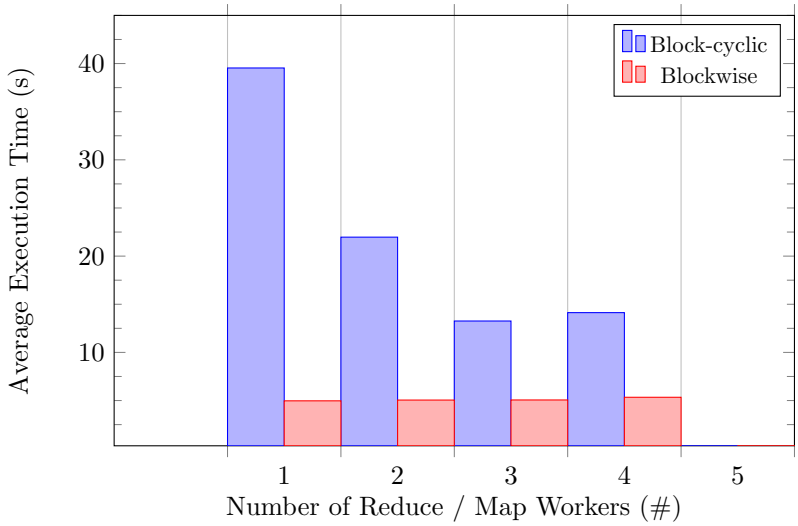

Figure 19: Execution Time between Blockwise and Block-Cyclic Distribution

4.6 Comparison of Execution Time Between Processors

In this section, the execution time when 1 map worker and 1 reduce worker will be compared between all the processors in section 1. This comparison will provide some insights to the single core performance of each processor.

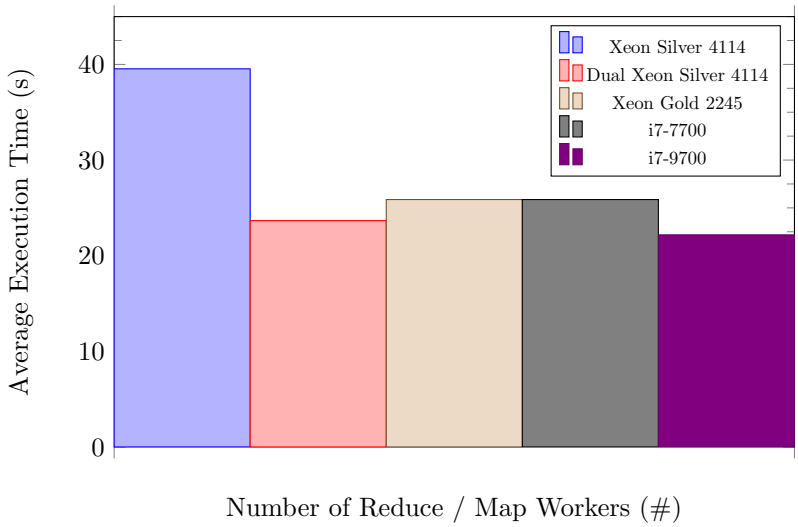

Figure 20: Execution Time between Processors

5 Discussion

Unless otherwise specified, discussions made are when the program input is 9 testcases.

5.1 Absence of Deadlocks and Race Conditions

The purpose of increasing the number of reduce workers disproportionately to the number of map workers in section 4.3 was to observe the programs ability to avoid deadlocks. Following the communication protocols described in the Final Design Iteration, deadlocks and race conditions are generally avoided. The disproportionate increase in the number of reduce workers to map workers was made to increase the likelihood for race conditions. Despite such conditions, the output generated by the program is consistent and correct with the provided sample test cases, and with the ones with an equal number of map and reduce workers.

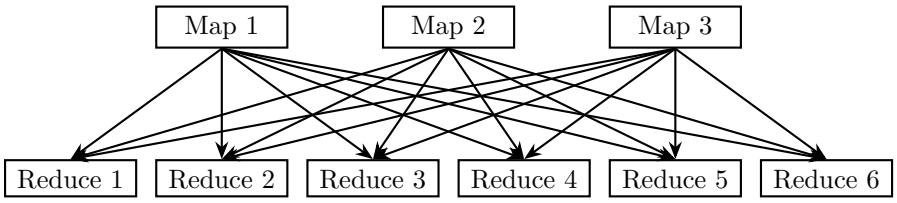

Figure 21: Increased Complexity of Communication Between Map and Reduce Workers

5.2 Processor Starvation

Comparing figures 12 to 16 and figure 20, it becomes apparent that the Intel i7s have better single core performance compared to the Intel Xeons. Assuming that 1 map worker, 1 reduce worker and 1 master thread produces a relatively sequential performance, it can be said that Intel i7s have nearly two times faster single core performance compared to the Intel Xeon Silver 4114 processor. Hence, in a distributed system including both Xeon and i7 processors, if not managed well, may increase the general likelihood of processor starvation.

5.2.1 Testing Intel Xeons and i7s Together

In Figure 18, the heterogeneous use of Intel processors has appeared to generally improve the execution timing of the MapReduce program, particularly when 4 map and 4 reduce workers were used. At 4 map and 4 reduce workers, the execution timing was at 7.68s, while the fastest on 7.84s on a homogeneous test case was on the Intel Dual-Socket Xeon Silver 4114. It appears that the mixed use of Intel Xeon processors and Intel i7 processors have improved the performance, overriding the general communication overhead incurred by the systems. An explanation for such could be due to a right ordering of files. Larger files could be handled by the more efficient i7 cores while smaller files could be handled by the slower Xeon cores. As a result, files get processed faster as the system is being fully utilised, resulting in an overall faster execution time.

5.2.2 Optimal Number of Files

The optimal number of files is directly affected by the number of map workers used in the system. Ideally, the number of files should be an integer multiple of the number of map workers. Failure to consider this would result in processor starvation. This can be seen from figures 12 to 19. Across all these test cases, the number of test cases stayed at a constant of 9. After a certain point where the number of map workers increase from 3 to 4, the speedup becomes negligible. This observation accentuates the case of processor starvation. Figure 22 illustrates this, where in the final batch, 3/4

processors used in mapping are left idle. This explains the negligible speedup after increasing the number of map workers from 3 to 4.

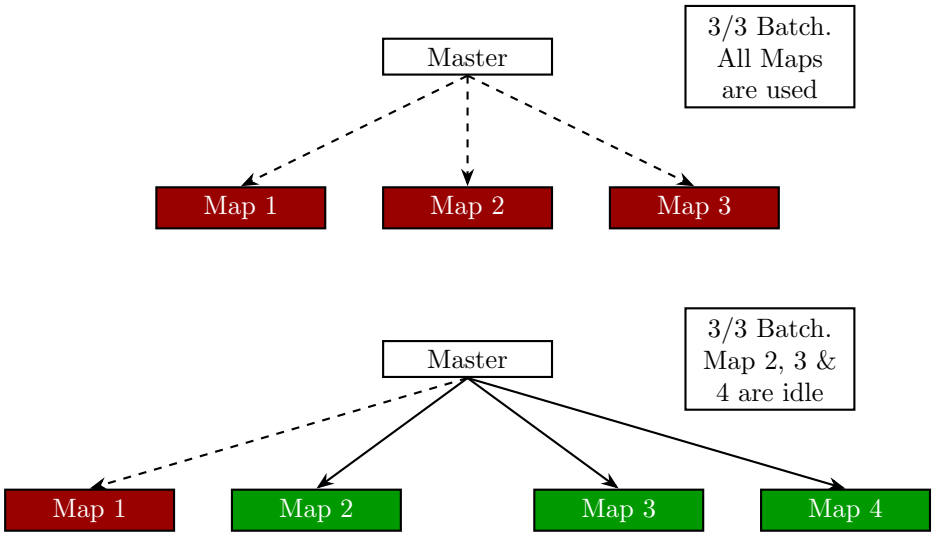

Figure 22: Processor Starvation in MapReduce

5.3 Blockwise Distribution in First Design Iteration

The blockwise distribution is found to be highly efficient compared to the block-cyclic distribution used in the Final Design Iteration. On a single map and reduce worker, the MapReduce took 4.97s, while using the final design iteration yielded a performance of 39.54s. The blockwise distribution resulted in a speedup of 7.96x. This observation is highly anomalous, and there are a couple of deductions that can be made from it.

5.3.1 Communication Overhead Savings

Within the Final Design Iteration on a single map and reduce worker, there are a couple of communication requirements made to ensure the scalability of the algorithm. Such communication requirements include polling, which are used to ensure that the map and master is ready to initiate a send operation. These communication requirements overall ensures no deadlock occurs and reduces the chances of data contention. However, these operations are blocking and asynchronous operations and might cause the program to slow down significantly.

5.3.2 Correctness of Program

However, a blockwise distribution in the current MapReduce implementation is not going to yield correctness in the program. The blockwise distribution of the program will cause files to be divided evenly across processes. Mapping tasks that require the inspection of the whole file will cause the wrong output, and this was observed when task 3 was ran on 6 files (from `0.txt` to `5.txt`). First Design Iteration was also not a complete code implementation overall, and lacked the stage where the reduce workers send back the reduced key-value pairs to the master for generating the output text file.

5.4 Areas of Improvement

5.4.1 Implementation of a Blockwise Distribution Scheme

It has been shown earlier that the blockwise distribution scheme speeds up the program significantly with fewer communication overheads. A key challenge to the correct implementation of such a work distribution scheme is to split up the file such that file-specific information can be preserved and in fact computed in parallel. This improvement will speedup the program significantly, and perhaps more than that values found in section 5.3.

5.4.2 Intelligently Allocate Nodes based on the Local Topography

This area of improvement is aimed to minimise the overall communication overhead by a distributed system. A virtual topography mimicking the local topography may be created, and be used by the system.

Appendix A

Rank / # of Map Workers	0	1	2	3	4	5	6	7	8
1	005	007	008						
2	005	007	008	009	019				
3	005	007	008	009	019	013	014		
4	005	007	008	009	019	013	014	015	021

Table 1: Distributed System Configuration with varying number of Map Workers

- Numbers shown here represent the index of the node (i.e. 005 is soctf-pdc-005)
- Number of Map Workers = Number of Reduce Workers