Assignment 3
MapReduce Implementation using MPI

CS3210 – 2021/22 Semester 1

28 Oct 2021

Learning Outcomes

This assignment aims to provide an introduction to programming in a distributed systems environment. In particular, we will be examining the Message Passing Interface (MPI).

1 Problem Scenario

In this assignment, we implement our own variant of the MapReduce programming model, customised for MPI.

1.1 MapReduce Overview

Many resources are available on how MapReduce works and how it can be implemented. This section aims to provide an introduction to the ideas and mechanism underlying the MapReduce framework. The description here is simplified and contains only the key ideas required for this assignment (the actual MapReduce framework is more complicated).

MapReduce was inspired by the eponymous ”map” and ”reduce” functions commonly used in functional programming. It is designed to break up a large batch processing task into several smaller “map” and “reduce” tasks, each of which can be run in parallel.

There are two types of processes involved in a MapReduce task: 1 master process and several worker processes:

- The master process is in charge of coordinating the map/reduce tasks and handling any book-keeping
- The worker processes are in charge of running map/reduce tasks allocated by the master

In the original MapReduce framework, it is assumed that there is a distributed file system that all processes (master/worker) can access. This means that if a process writes a file to the distributed file system, the other processes (which are running on other physically-separated nodes) will also be able to read the file.
Note: You might find it easier to read the following description while comparing it with the Example Walkthrough in the following section.
MapReduce has 3 main phases:

1. Initialisation Phase
   The master process starts up and identifies available worker processes.

2. Map phase

   (a) The master allocates map tasks to the available worker processes by informing the worker process what input file to run the map task on. Note that in a given MapReduce program, the same map task is run by each worker process - it's the input file that is different. The master will continue allocating map tasks to worker processes until all files have been processed.
   (b) The worker process will read in the allocated input file from a distributed file system and run the map task on the contents of the input file.
   (c) The output of the map task is an array of Key-Value pairs. Each Key-Value pair is then put into 1 of M partitions according to its key. Each partition is stored in the distributed file system as a separate file. Why is partitioning necessary? See the reduce phase.
   (d) Once each worker is done with its allocated map task, it informs the master process. This means the worker is now idle, and the master process can allocate another map task to it.

3. Reduce phase

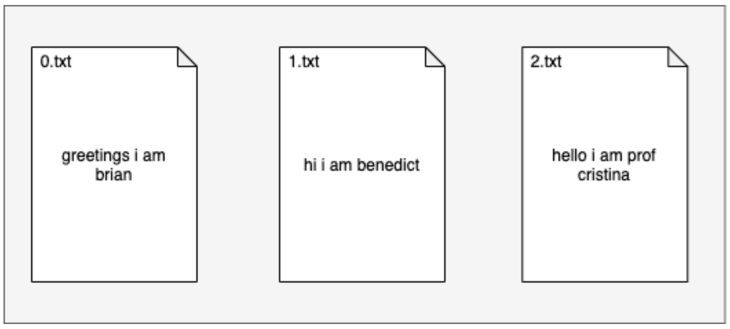
   (a) Once the map phase is over, the workers can begin executing reduce tasks.
   (b) The master process will allocate reduce tasks to each worker process by informing the worker process which partition (there are M of these, as described in the Map phase) it is supposed to work on.
   (c) The worker process then reads from the distributed file system all the files associated with its allocated partition number.
   (d) Having read all these files, the worker process will aggregate the Key-Value pairs for each key and run the reduce task on each key and its aggregated values.
   (e) The output of the reduce task is then stored in the underlying file system.

The outcome of the MapReduce task can then be determined by reading all the reduce phase output files from the distributed file system.

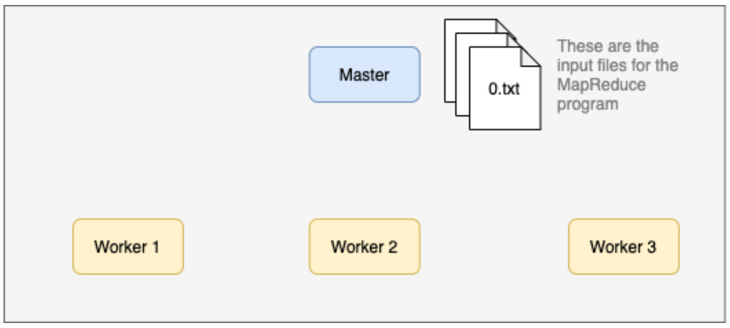
1.2 Example Walkthrough

The above description can be a little convoluted for those who are learning about the MapReduce framework for the first time. What follows is an example of a word counting MapReduce program - we are counting the number of occurrences of each unique word in a set of 3 documents (0.txt, 1.txt and 2.txt).

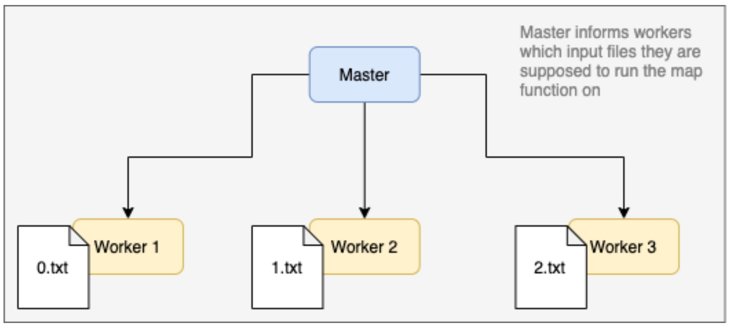
1. The contents of the 3 documents are shown below. These 3 documents are in a distributed file system, accessible by the master process and worker processes.



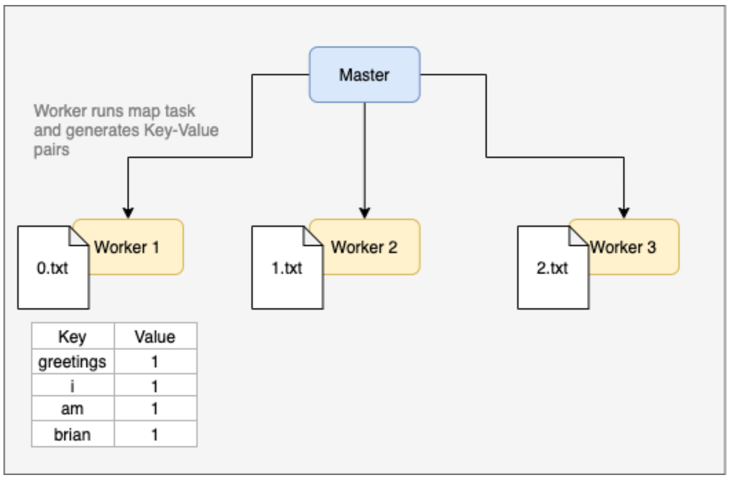
2. (Initialisation Phase) Here, the master process starts up and takes note of which worker processes are available. In this case, there are 3 worker processes.



3. (Map Phase) The master informs each worker which input file they are to run the map function on. Since there are 3 workers, the master informs workers 1, 2 and 3 that they are in charge of running the map task on files 0.txt, 1.txt and 2.txt respectively.

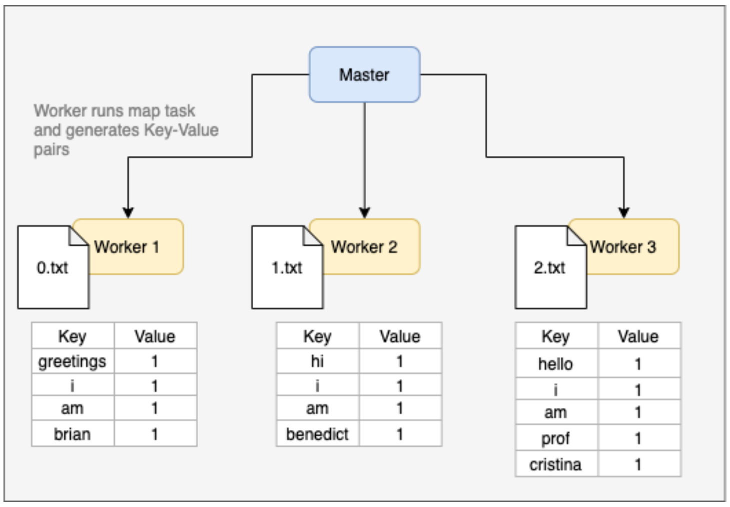

Master
Master informs workers which input files they are supposed to run the map function on
0.txt
Worker 1
1.txt
Worker 2
2.txt
Worker 3

4. (Map Phase) Let's focus our attention on Worker 1. Worker 1 reads the contents of 0.txt and runs the map function on it. Since this is a word counting MapReduce program, the map function will generate a list of Key-Value pairs, with the Key being a word, and the value being the number of occurrences of the word in the file.



Key	Value
greetings	1
i	1
am	1
brian	1

5. (Map Phase) The same thing happens to the other workers.

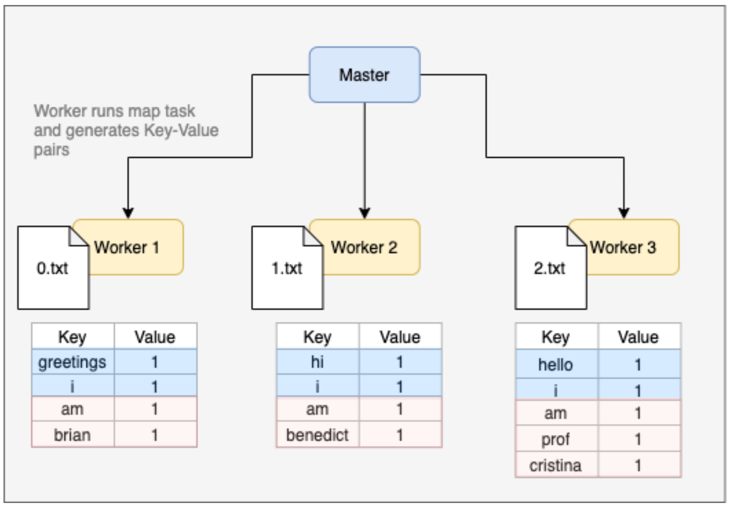


Key	Value
greetings	1
i	1
am	1
brian	1

Key	Value
hi	1
i	1
am	1
benedict	1

Key	Value
hello	1
i	1
am	1
prof	1
cristina	1

6. (Map Phase) Now, the map workers partition the Key-Value pairs generated by the map tasks. In this case, let's assume that we generate 2 partitions and that the partition function will allocate a Key-Value pair to partition 1 (in red) if the Key starts with "a", "b", "c" or "p", and to partition 2 (in blue) if the Key starts with any other character. These partitioned Key-Value pairs are then stored in separate files on the distributed file system (i.e. for each worker, the blue Key-Value pairs are stored in a separate file from the red Key-Value pairs). At this point, the map task is done and the worker process informs the master process about this. If the master still has more map tasks that need to be completed, the now-idle worker will be assigned a new map task.

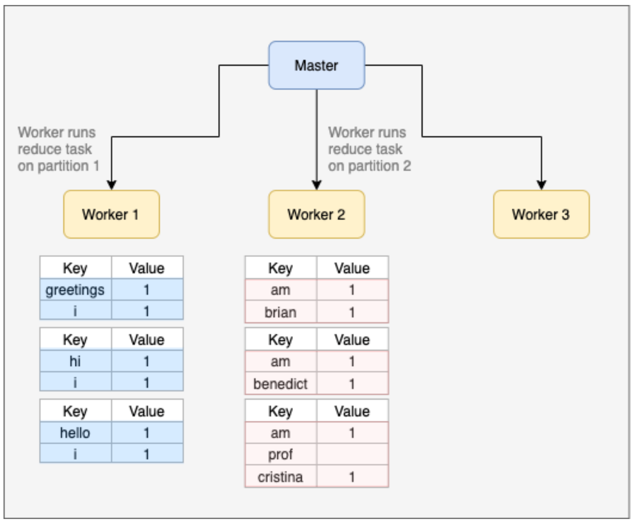


Key	Value
greetings	1
i	1
am	1
brian	1

Key	Value
hi	1
i	1
am	1
benedict	1

Key	Value
hello	1
i	1
am	1
prof	1
cristina	1

7. (Reduce Phase) Now that the Map Phase is complete, the Reduce Phase can now begin. There are only 2 partitions, so the master will inform 2 workers to each be in charge of running the reduce task for 1 partition. In this case, worker 1 is in charge of partition 2 (in blue), and worker 2 is in charge of partition 1 (in red). Each worker will read the partitions' files from the distributed file system.


Master
Worker runs reduce task on partition 1
Worker runs reduce task on partition 2
Worker 1
Worker 2
Worker 3

Key	Value
greetings	1
i	1

Key	Value
hi	1
i	1

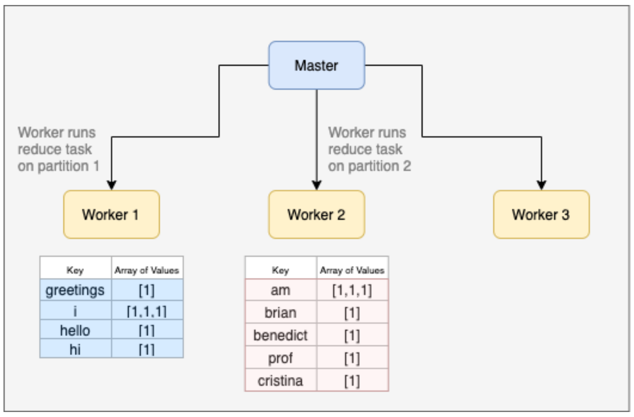
Key	Value
hello	1
i	1

Key	Value
am	1
brian	1

Key	Value
am	1
benedict	1

Key	Value
am	1
prof	
cristina	1

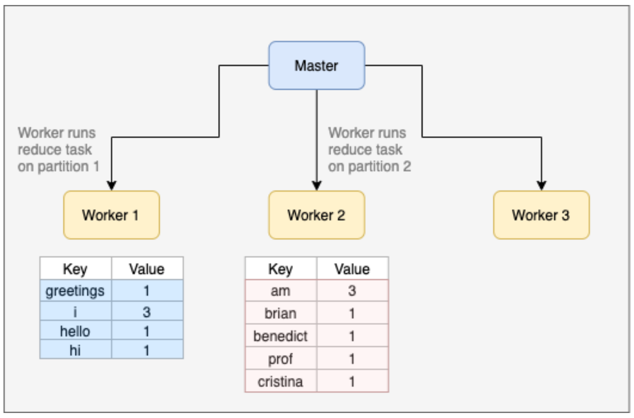
8. (Reduce Phase) Having read the partition's data from the distributed file system, the reduce worker will now aggregate the values for each key into an array


Master
Worker runs reduce task on partition 1
Worker runs reduce task on partition 2
Worker 1
Worker 2
Worker 3

Key	Array of Values
greetings	[1]
i	[1,1,1]
hello	[1]
hi	[1]

Key	Array of Values
am	[1,1,1]
brian	[1]
benedict	[1]
prof	[1]
cristina	[1]

9. (Reduce Phase) The worker will now run the reduce function on each key and its aggregated array of values. Since this is a word counter program, the reduce function simply sums up the numbers in each array of values.


Master
Worker runs reduce task on partition 1
Worker runs reduce task on partition 2
Worker 1
Worker 2
Worker 3

Key	Value
greetings	1
i	3
hello	1
hi	1

Key	Value
am	3
brian	1
benedict	1
prof	1
cristina	1

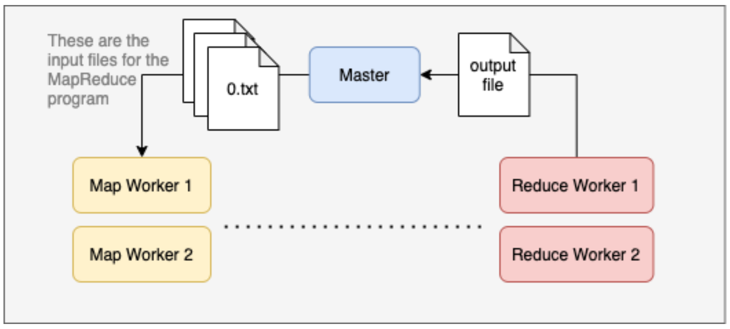
At this point. The reduce phase is done. The reduce output (i.e. the reduced Key-Value pairs) for each partition is then stored in the distributed file system. If someone wants to retrieve the reduce output, he/she simply queries the distributed file system for all the reduce output files (there are 2, because there are 2 partitions).

If you are interested in learning more about MapReduce, there are tonnes of good resources available online. In particular, the original research paper is insightful, easy to read, and can be found here.

2 Your Task: Implementing MapReduce Customised for MPI

For this assignment, we aim to re-create the MapReduce framework, but this time using MPI.
Note that:

- We do not have a distributed file system. Hence, the distribution of data (i.e. file data, map task output, reduce task output) needs to be achieved using MPI.
- In the original MapReduce framework, worker processes can run both map and reduce tasks. To simplify the implementation for this assignment, we will have dedicated worker processes just for running map tasks (we will call these map workers) and dedicated worker processes just for running reduce tasks (we will call these reduce workers).
- The MapReduce framework can run any kind of map and reduce tasks - it's up to the programmer's imagination to implement their own map and reduce functions. However, for the sake of simplicity for this assignment, we have provided you with the implementations of 3 different map functions and 1 reduce function (i.e. all 3 map functions use the same reduce function). The reason we did this is because the focus of the assignment is on MPI, not the definition of individual map and reduce functions. Of course, feel free to define your own map and reduce functions when testing, if you think it would be helpful. This assignment also provides the guarantee that the Key-Value pairs can always be represented by the KeyValue struct provided in the starter code. Together, these guarantees simplify the passing of Key-Value pairs between processes via MPI.
- In MapReduce, the output is scattered across multiple partition files (the output from each reduce task). For this assignment, you will need to gather the output into a single file and store it in the master process' local file system. The order of the KeyValue pairs in the final output file does not matter.



The main challenges for this assignment are:

1. To figure out what data needs to be passed between master, map workers and reduce workers, and how that can be done efficiently via MPI primitives.
2. How to ensure that your MPI implementation of MapReduce can never deadlock
3. How to ensure that your MPI implementation of MapReduce is not vulnerable to race conditions

2.1 Inputs and Outputs

Input Files. As mentioned in the previous sections, the input files provide the data on which the map function is run.
We have established a naming convention for the input files (see the input files in the `sample_input_files` directory in the starter code):

1. Input files must have the `.txt` suffix
2. Input files are numbered, starting from 0

It is important to take note of this naming convention when writing your code and when creating your own input files for testing. During grading, we will be replacing the sample input files with another set of files, so it is critical that your code understands this naming convention.

Output Files. The master gathers the reduce task outputs from all reduce workers.
The output files **must** follow this format:

1. Each Key-Value pair occupies 1 line in the output file
2. The Key is separated from the Value by a space

See the output files in the `sample_output_files` directory in the starter code for an example.

Running your Code (Parameters). Your code must compile into an executable named a03. Note that there is no suffix `.out`.
When running the executable, several parameters will be specified. In combination, these parameters determine how your MapReduce program will be run. During grading, we will specify various combinations of these parameters and your output will be checked for correctness.

```
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>
```

The handling of the CLI parameters has already been done for you in the starter code. Table 1 describes what each CLI parameter is meant to do.

2.2 Checking for Correctness

After finishing your implementation, you can check the correctness of your output file with the following command:

```
sort <YOUR_OUTPUT_FILE> | diff sample_output_files/<NUM.output -
```

CLI Param	Explanation
NUM_MPI_PROCESSES	Number of MPI Processes to run. This is equal to 1 + NUM_MAP_WORKERS + NUM_REDUCE_WORKERS. Note: The 1 refers to the master process. Note: You must ensure that your code can run with different numbers of map and reduce workers. While testing your implementation, you should test with 5-7 MPI processes. If testing with higher numbers, please make sure that you do not hog other machines' resources. Check that your processes were killed properly, otherwise it might affect other students.
PATH_TO_HOSTFILE	This is the hostfile, which determines the nodes your MPI processes will run on. Note: do not include this parameter when running your MPI program under Slurm.
INPUT_FILES_DIR	This is the directory where the executable is supposed to look for the input files When running against the input files provided in the starter code, this should be set to `sample_input_files`. You can create your own input files in a separate directory and run your program against that directory.

NUM_INPUT_FILES	This is the total number of input files that map workers must work on.
	The input files are found in the INPUT_FILES_DIR
	The input files will be read in order, starting from 0.txt.
	For instance, if NUM_INPUT_FILES is set to 4 and INPUT_FILES_DIR is set to `sample_input_files`, then your MapReduce program should process the following files:
	`sample_input_files/0.txt`
	`sample_input_files/1.txt`
	`sample_input_files/2.txt`
	`sample_input_files/3.txt`

NUM_MAP_WORKERS	In this MPI assignment, we assume that we have dedicated worker processes just for running map tasks. NUM_MAP_WORKERS is the number of dedicated map workers you have.
NUM_REDUCE_WORKERS	Similar to NUM_MAP_WORKERS, this is the number of dedicated reduce workers you have.
OUTPUT_FILE_NAME	This is the name of the output file that the master process will write the final result of the map reduce program to.
MAP_REDUCE_TASK_ID	As mentioned in a previous section, there are 3 map tasks provided. We identify which map task to run by using MAP_REDUCE_TASK_ID. There are only 3 possible values for MAP_REDUCE_TASK_ID: 1, 2 or 3.

Table 1: CLI parameters

As mentioned in a previous section, we **do not** require an order for the Key-Value pairs in the output file: this is because we will first `sort` your output file before `diff`-ing against the benchmark output file (which has already been sorted).

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

- 1.output
  `MAP_REDUCE_TASK_ID: 1`
  `NUM_INPUT_FILES: 1`
  `INPUT_FILES_DIR: sample_input_files`

- 2.output
  ```
  MAP_REDUCE_TASK_ID: 2
  NUM_INPUT_FILES: 5
  INPUT_FILES_DIR: sample_input_files
  ```
- 3.output
  ```
  MAP_REDUCE_TASK_ID: 3
  NUM_INPUT_FILES: 6
  INPUT_FILES_DIR: sample_input_files
  ```

3 Bonus

As described in the earlier sections, there are some differences between the actual MapReduce framework and the MPI implementation you made for this assignment.

You may obtain up to 2 bonus marks for identifying 5 differences between the actual MapReduce framework (as described in the original paper by Google) and your MPI implementation.

The differences you identify must be substantial. We are looking for a good understanding of the MapReduce framework and its intended usage.

This bonus section is meant to motivate you to learn more about a distributed programming framework that has been battle-tested in industry. Of course, there are other newer paradigms in existence, but MapReduce is a great place to start!

4 Admin Matters

Your code should successfully compile and run on the lab machines. Run your programs with different input parameters. Investigate how different parameters and input files affect the runtime of your program.
When taking measurements, you should use the following computers:

1. SoC computer Cluster (not managed with Slurm):
   - xgpd0-7
   - xgpc0-7
2. Lab machines (managed with Slurm):
   - Direct login enabled: soctf-pdc-001->004, soctf-pdc-010->012, soctf-pdc-018, soctf-pdc-020, soctf-pdc-022, soctf-pdc-023
   - Direct login disabled (managed with Slurm only, `authorized_keys` file disabled): soctf-pdc-005->009, soctf-pdc-013->016, soctf-pdc-019, soctf-pdc-021, soctf-pdc-024

Updated Slurm guide for MPI can be found here. When running your MPI program with Slurm you are advised to not use a `HOSTFILE` as the mismatch between the `HOSTFILE` and the nodes used by Slurm might make your run fail.
To view the usage of the lab machines, you can use this Telegram bot: `@cs3210_machine_bot`. Simply type `/start` to get a real time status update for all machines.

4.1 Hints and Suggestions

This assignment can be quite challenging and a little hard to debug (that is almost always the case for parallel and distributed programming problems). Here are some tips and suggestions that might make the implementation process easier for you:

1. Start off by planning your implementation. Ask yourself where deadlocks/race conditions could occur. Try to "attack" your implementation in as many ways you can conceive.
2. Look at the function signatures of the "map" and "reduce" functions. Understand how they fit into your implementation. Plan how you will move data across the master, map workers and reduce workers using MPI.
3. Simple printf statements can be helpful, but you need a way to distinguish the process that printed to stdout. The rank of the process is a good way to do so. But do note that your code in your final submission should not print to stdout/stderr.
4. The input files from 0.txt to 5.txt are increasingly larger and more complex. When debugging, start with the simple input files, and maybe restrict yourself to just 1 map worker and 1 reduce worker. This may help you identify the source of problems more quickly.

FAQ
All FAQs for this assignment will be answered here. The most recent questions will be added to the beginning of the file, proceeded by the date label. Check this file before asking your questions. If there are any questions you have that are not covered by the FAQ document, please post on the LumiNUS Forum or email Brian (e0310531@u.nus.edu) or Benedict (benedictkhoo.mw@u.nus.edu).

4.2 Grading

You are advised to work in groups of 2 for this assignment (but you are allowed to work independently as well). You may discuss the assignment with others but in the case of plagiarism, both parties will be severely penalized. Cite your references in your report (what you referenced, where it came from, how much you referenced etc.)
The deadline for submission for this assignment is **Friday, 12 November, 11am**. The grades are allocated as follows:

- 4 marks: MPI implementation correctness (whether your implementation's output is the same as the output of our reference implementation for the grading testcases)
- 2 marks: MPI implementation speedup: your MapReduce program achieves speedup of at least 1.5x when run with 3 map workers and 3 reduce workers as compared to when run with 1 map worker and 1 reduce worker
- 1 mark: additional input files you created to test your MPI implementation. These should be in your submission's `testcases` folder.
- 6 marks: Your report.

You should also be careful about memory leaks in your implementation. For the map tasks, we have provided a `free_map_task_output` function.

4.3 Report Requirements

Your report should include the following:

- A description of your design and implementation. Here, also include an analysis of why your implementation is not vulnerable to deadlocks/race conditions. We don't require a mathematically rigorous proof, but please describe your thought process.
- A description of assumptions you made, if any.
- Any special considerations/implementation details that you consider non-trivial.
- Measurements for your implementation. Here, you decide what factors (e.g. machine type, number of input files etc.) you wish to vary and take the corresponding measurements. You should demonstrate how your distributed implementation scales with an increasing number of MPI processes by looking at different metrics for:
  - Different configurations and numbers of each type of lab machine
  - Different input parameters

  To analyse the improvements in performance, compare the execution time against other prototype implementations you have tried. You should select parameters and inputs that have meaningful execution times when solved by your final implementation.
- Details on how to reproduce your results (e.g. scripts you used, how you did certain measurements etc.)
- Analysis of your implementation and its performance. Include graphs and any other insights/anomalies you find. Describe patterns you observe and explain them using concepts you have learned in class.

> - There are many variables that contribute to performance. It is highly impractical to study every combination. Instead, focus on selecting a couple of key variables and do a careful analysis of your results. You are graded more on the quality of your investigations, not so much on the quantity of things tried or even whether your hypothesis turned out to be correct.
> - The lab machines are shared with the entire class. Distributed programming also necessarily involved multiple machines. Please be considerate and do not hog the machines or leave bad programs running indefinitely. Use `top` or `htop` to check. Start early!

There is no minimum or maximum page length for the report. Be comprehensive, yet concise.

Submit your assignment before the deadline under LumiNUS Files.
If you are working in pairs, each pair should only submit one zip archive named with your student numbers (e.g. A0123456W_A0123692T.zip) to LumiNUS. This means that if your buddy submits the zip archive, then you should not submit one.
If you are working independently, then you should only submit one zip archive named with your student number (e.g. A0187654X.zip) to LumiNUS.
Submit exactly once. If you need to replace an old submission, remember to delete the old submission from LumiNUS. Please include the following files in your zip archive:

1. `main.c` - The main entry point of your code.
2. Any other files required for compilation. These are the files that will be compiled together with `main.c`.

   Take note that `tasks.c`, `tasks.h` and `hostfile`, which were provided to you as part of the sample code, will be replaced during the grading process. These 3 files should not be included in your zip archive. This is checked by the `check_zip.sh` script.

   Be sure to include everything your submission needs such that when `make build` is run on the lab machines, your MapReduce implementation can be built and run without issue.

   Please remember to remove all print statements from your code. When your compiled code is run, it should not print to stdout/stderr.
3. `Makefile` This should have a ‘build‘ recipe that compiles your MPI implementation exactly as you intend it to be graded for correctness and performance.

   The name of the compiled executable must be a03. Do not include any suffix.
4. Report in PDF format (A0123456Z_A0173456T_report.pdf or A0123456Z_report.pdf).
5. A folder, named `testcases`, containing any additional test cases (input and output) that you might have used.
6. An **optional** folder, named `scripts`, containing any additional scripts you used to measure the execution time and extract data for your report.

Once you have the zip file, you will be able to check it by doing:

```
chmod +x ./check_zip.sh
./check_zip.sh A0123456Z_A0173456T.zip (replace with your zip
file name)
```

During execution, the script prints if the checks have been successfully conducted, and which checks failed. Successfully passing the checks ensures that we can grade your assignment.
Note that for submissions made as a group, only the most recent submission (from any of the students) will be graded, and both students receive that grade. A penalty of 10% per day (out of your grade) will be applied for late submissions.