MapReduce-like Framework Documentation and Performance Evaluation

Srisaipavan Valluri
vallu006@umn.edu

David Schoder
schod005@umn.edu

April 21, 2016

1 Components of System

1.1 Client Implementation

The Client will simply send a sort request to the Server using the getSorted() interface function provided by the Server. The filenames that the Client will use as input are located in **input_files**. It will then wait for the sort request to be processed. After the sorted filename has been returned, the Client will print out the filename which will be located **output_files**. It will also print out the execution time of the sort request.

1.2 Server Implementation

The Server is responsible for accepting client requests to sort files. Given an input filename, the Server will read all integers from the filename and split the dataset based on the chunk size, which is given as a command line parameter. Each subset of integers is then used for each sort task. Afterwards, each sort task is assigned to a Compute Node using Sort(), which is an interface function provided by the Compute Node interface.

Assigning sort tasks: The assignment of sort tasks occurs in reassignSortTasks(vector¡Task¿, redundancy). This function takes a vector of Task structs and a redundancy parameter. Each Task struct contains information about the task such as the filenames, completion flag, offset, size, and merge_task_number. Some of these attributes arent used for the sort procedure and are left uninitialized. The redundancy parameter will be used to implement proactive fault tolerance. For each sort task in the vector, a suitable Compute Node is found. This

determined by checking whether the Compute is alive, its currently executing, or it has already finished the same task. If any of these criteria is met by the Compute Node, then the task isnt assigned. Otherwise, the task is assigned to the Compute Node which doesnt meet ANY of these criteria. If the redundancy parameter is greater than or equal to 2, then each task in the vector will be assigned to multiple nodes. Thus, multiple Compute Nodes will be executing the same task. This achieved proactive fault tolerance because we are making sure ahead of time that if one Compute Node fails, there are other Compute Nodes executing the same task. If a task cannot be assigned to any Compute Nodes, its simply skipped. However, this typically happens when most of the Compute Nodes are dead and its impossible to assign redundant tasks. When a result is returned from multiple Compute Nodes responsible for the same task (redundant task), the result which came first is accepted and the rest are ignored. If the result wasnt accepted by the Server, the Compute Node deletes the file it created.

Once all sort tasks have been assigned, the Server waits until these tasks have been completed. This is because the merge functionality depends directly on the produced sort files. These sorted intermediate files will be stored in **intermediary_files/**. Once all sorted filenames have been received by the Server, the merge procedure begins. The first round of merge tasks are assembled. The number of merge tasks required for the first round is determined by the number of intermediate files for each merge task, which is passed as a parameter to the system. Next, the merge tasks are assigned to Compute Nodes.

Assigning merge tasks: The assignment of merge tasks happens in assignMergeTasks(iterator) and reassignMergeTasks(vector¡Task¿, redundancy). The procedure for assigning is very similar to assigning sort tasks. However, there are a few differences. One is that a single merge task is responsible for multiple files, whereas a single sort task is responsible for a single file. Another difference is that assignMergeTasks(iterator) initially assigns a single merge task using the iterator to pick the Compute Node. It then calls reassignMergeTasks() to assign the rest redundant tasks. The proactive fault tolerance for merge tasks is handled exactly the same way as its for the sort tasks.

After the first round of merge tasks is done, the second round of merge tasks is assembled and assigned to Compute Nodes. Only after the second round is complete, the third round begins. This pattern is continued until only one merge file is left to process. This file is renamed and stored in the **/output_files/** folder. This filename is also returned to the Client. After execution of getSorted() has completed, statistics like total computation time, total faults, and total redundant tasks are printed. All files in the **intermediary_files/** folder are then deleted to save space.

The Server is also responsible for providing **proactive fault tolerance** in the entire system. It keeps track of heartbeats sent by the Compute Node by providing an interface function to Compute Nodes called sendHeartBeat(). Every time a Compute Node sends a heartbeat,

the Server generates a new heartbeat and stores it. Also, when the Server is first started, it creates a new thread which is responsible for checking whether or not a Compute Node has died. This timer is run every 100ms. If the Server detects that the Compute Node hasnt responded within a timeout interval (600 ms), then it declares it as dead. It will then reassign all the uncompleted tasks. At this point, no future tasks will be assigned to this Compute Node until the next job.

If the Compute Node was terminated, then the Server will ping the Compute Node to try and establish a connection. If an exception was caught, then the Compute Node is removed from the Servers list of Compute Nodes and will not be used in future tasks or jobs.

1.3 Compute Node Implementation

The Compute Node provides two functions: Merge() and Sort().

Sort(): Given an input filename, offset, and chunk size, the Compute Node will read from the input filename using offset and chunk size. Once it has read all the values, it will call the std::sort() function to sort the values. Next, the Compute Node will output the result to a file which is named using offset, chunk size, IP Address, and port number. To avoid creating duplicate sort files (redundancy and proactive fault tolerance), the Compute Node will invoke a callback function provided by the Server interface (sendFilename). This functions returns true if the Server accepts its result or false if the Server cannot accept the result. If the function returns false, the file created will be deleted.

Merge(): Given a vector of sorted files and a task number, the Compute Node will read each sorted file into a different array. It will then merge all these arrays by merging two arrays at a time using the MergeHelper() function. Next, the Compute Node will output the result to a file named using the task_number, IP Address, and port. To avoid creating duplicate merge files (redundancy and proactive fault tolerance), the Compute Node will invoke a callback function provided by the Server interface (callback_merge). This function will return true if the Server accepts its result or false if the Server cannot accept the result. If the function returns false, the file created will be deleted.

Failure Injection: Each Compute Node will also be given a failure probability which it will use to fail itself (NOT TERMINATE). At the beginning of each Merge/Sort task, a fail probability is computed using rand(). If the probability generated is less than the fail probability, it will set its heartbeat to false and not execute the task. Eventually, the Server will notice this and also stop sending requests to the Compute Node. This is because the Compute Node sends its heartbeat every 100ms only if heartbeat is true. Once heartbeat is set to false, it will stop sending its heartbeat. In this case, the Compute Node wont start handling tasks until the next job/client request as specified in the project description.

Manual Termination: You can also fail the Compute Node by manually terminating

the process. In this case, the Compute Node will never receive requests until it registers itself with the Server again.

2 Running the System

The server source code is located in /gen-cpp-server/. The compute node source code is located in /gen-cpp-node/. The client source file is called Client.cpp and is located at the root directory of the project folder.

2.1 Here is how to start and run system:

- Compile the Server using the make all command in the /gen-cpp-server/ folder.
- Run the Server (look at Running the Server)
- The Server can be run on any VM. We used csel-kh1250-12.
- Compile the Compute Node using the make all command in the /gen-cpp-node/ folder.
- Run the Compute Nodes (we ran 7 Compute Nodes on machines csel-kh1250-02 through 08) (look at Running the Compute Nodes)
- Compile the Client using the make all command in the directory (root) which contains the Client.cpp file
- Run the Client (Look at Running the Client).
- If you want to sort a file, type 1 and press enter. Then type the input filename and press enter. Dont use the relative file path, we already append that for you. Just the file basename. For example, if the file abc.txt is located in /input_files/, dont type ./input_files/abc.txt. Just do abc.txt. If you want to quit, type 2 and press enter.

2.2 Running the Client

```
./client <server-ip-address> <server-port>
```

2.3 Running the Server

Once the Coordinator is running, the File Servers must be run next. Here is the command to start 10 local File Servers, each assigned a different port number:

```
./server <my-port> <chunk\_size> <num\_merge\_files> <num\_redundant\
```

The chunk size is the size of each sort task, the num_merge_files parameter is the number of intermediate files used for every merge task, and the num_redundant_tasks parameter is the number of redundant tasks generated for each unique task. This parameter should be less than Number of Compute Nodes - 1. So if 7 Compute Nodes were running, then the greatest possible integer for num_redundant_tasks should be 5. We used 3 when testing the system.

2.4 Running the Compute Node

```
./compute_node <my-port> <server_ip> <server_port> <fail_probability 0-100>
```

The fail probability parameter is a double between 0 and 100 and represents a percentage. However, if the fail probability is above 10, the Compute Node servers are failing too fast. If all Compute Nodes fail, then the client request cannot be processed. We would like it if you keep the failure probability 3 or less. We will also use this range of numbers for our test cases.

If you want to run the Compute Nodes locally, i.e. on the same machine, you can use the startup_nodes.sh script. This automatically starts 7 Compute Nodes on the same machine, but with different ports. Here is the command line for that (should be run in root directory of project folder):

```
./startup\_nodes.sh <fail-probability 0-100>
```

NOTE: You will need to set the IP Address to the IP Address of the Server in the script file if you want to use it.

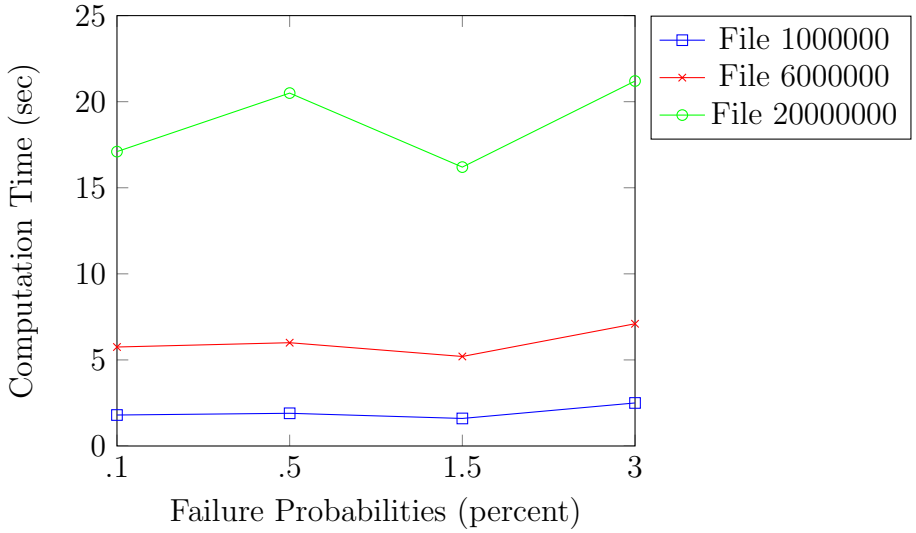
2.5 Test Cases documenation and Performance Evaluation

We tested the system using 7 Compute Nodes, 3 redundant tasks for each sort/merge task, and 8 for number of intermediate files for every merge task. Since the project description only said to vary file size, chunk size, and failure probability, we didnt vary these parameters. Here are the test cases:

Table 1: Number of Tasks Assigned and Tasks Killed

Failure Probability	Filename	Chunk Size (MB)	Total Tasks	Killed Tasks
.1	1000000	1	22	15
.1	1000000	2	14	9
.1	1000000	3	10	6
.1	6000000	1	125	90
.1	6000000	2	63	45
.1	6000000	3	46	30
.1	20000000	1	422	294
.1	20000000	2	212	147
.1	20000000	1	144	99
.5	1000000	1	22	11
.5	1000000	2	18	9
.5	1000000	3	10	4
.5	6000000	1	125	90
.5	6000000	2	63	45
.5	6000000	3	43	30
.5	20000000	1	422	294
.5	20000000	2	212	147
.5	20000000	3	144	99
1.5	1000000	1	22	15
1.5	1000000	2	14	9
1.5	1000000	3	10	6
1.5	6000000	1	125	90
1.5	6000000	2	78	56
1.5	6000000	3	55	34
1.5	20000000	1	530	350
1.5	20000000	2	345	169
1.5	20000000	3	155	112
3	1000000	1	22	15
3	1000000	2	15	10
3	1000000	3	10	6
3	6000000	1	135	84
3	6000000	2	66	45
3	6000000	3	43	30
3	20000000	1	651	345
3	20000000	2	212	147
3	20000000	3	156	105

Average Computation Time for all Chunk Sizes at Different Fail Probabilities



Each data point represents the computation time which was averaged over all chunk sizes.

Negative test cases: If all the Compute Nodes fail, then the system will freeze. This is because the computation cannot go on. In this case, the user must terminate the server. In order to tell if all Compute Nodes have failed, the Server will print the number of alive nodes. If the number of redundant tasks is greater than the number alive nodes, then only the possible assignment will happen, and the rest of the redundant tasks will be assigned. Finally, if the user tries to select an input file which doesn't exist in the input_files directory, then the client will output an error.

2.6 Analysis

Looking at the results, it seems that if we increase the probability of failure and more Compute Nodes fail, then the average computation time for every chunk size or file size increases. This makes sense because when a Compute Node fails, the load is distributed to another node, which decreases the overall throughput at the Compute Node which received the new tasks. As a result, for the same number of sort/merge tasks, the computation time increases. Also, there is overhead in reassigning the tasks which further increases the overall computation time. In terms of the chunk size and file size, if the file size increases, but the chunk size stays constant, then the overall number of tasks increases, which means that the computation time will also increase due to the overhead of increased communication between the Server and Compute Nodes. If the chunk size increases, but the file size stays constant, then there is improvement in the computation time since there are less tasks in the system which decreases communication. However, this increase in performance exhibits diminishing returns because there isnt much improvement from increasing the chunk size from 2 MB to 3 MB. In fact, sometimes a chunk size of 3 MB performs worse. This is probably because the Compute Node spends more time processing 3MB files compared to 2

MB. It seems that a chunk size of 2 MB typically worked the best. If one were to increase the redundancy parameter, then the overall computation time may increase, but it prevents the overhead of reassigning a task if multiple nodes are processing the same task. This can be useful especially in systems where the nodes regularly fail. However, if the chunk size is very large, then assigning redundant tasks and increasing the redundancy parameter can be burdensome since the latency of each task increases substantially.