Distributed Batch Processing System

Soham Desai
School of Electrical and Computer Engineering
Georgia Institute of Technology
Atlanta, Georgia, USA
sjdesai16@gmail.com

Apurv Vivek
School of Electrical and Computer Engineering
Georgia Institute of Technology
Atlanta, Georgia, USA
avivek6@gatech.edu

***Abstract*— This report describes the project completed as partial fulfillment of the course Parallel and Distributed Computer Architecture (ECE 6101).**

Keywords*— *Batch processing, fault tolerance, fair scheduling, java Thread class, synchronized, non-blocking job handling and task-scheduling, round robin.

I. INTRODUCTION (*Heading 1*)

Batch processing is the execution of a series of jobs on a computer without manual intervention. This being distributed the jobs submitted by various computer nodes ("clients) are to be carried on different set of computer nodes ('workers'). The batch processing system comprises of three components client, scheduler and workers. The features implemented in the project and described ahead are Non-blocking Request Handling, Parallel Job Execution, Fair Job Scheduling and Fault Tolerance. A distributed system is a software system in which components located on networked computers communicate and coordinate their actions bypassing messages

II. IMPORTANT TERMS

Node: A node refers to the individual computer system as part of the distributed system. In the project a node is simulated using a separate terminal in the Linux servers.
Scheduler: A scheduler is a node that registers new clients and new workers, it is also responsible for scheduling the jobs.
Worker: A worker is a node that actually carries out individual tasks.
Client: A client is a node that submits new jobs for the scheduler to schedule the tasks onto the workers.
Cluster Data Structure: This maintains a record of all the worker nodes in the system. It also maintains a list of workers which are available and ready to work.
JobCluster Data Structure: Maintains the record of all the jobs registered by the client.
Connector Thread: The thread responsible for accepting new connections from clients or workers.
Helper Thread: The thread created from the Connector Thread which actually performs the communication with the worker or client node.
Task Assigner Thread: The thread responsible for scheduling and assigning tasks and worker threads to each worker node.
Worker Thread : The Thread spawned from the task assigner thread responsible for communicating with the worker node in order to complete a task.
Report: A message passed from the worker thread to the helper thread in order to communicated the completion of tasks.
Pre-commit: History of tasks, of a particular job, already assigned to a worker thread, but may or may not yet complete.
Post-commit: History of tasks, of a particular job, completed because of response from the worker node.

III. DETAILS REGARDING IMPLEMENTATION

The implementation typically relies on the Thread class in Java. The main thread is responsible for creating 2 primary threads which handle the connection and the scheduling portion.

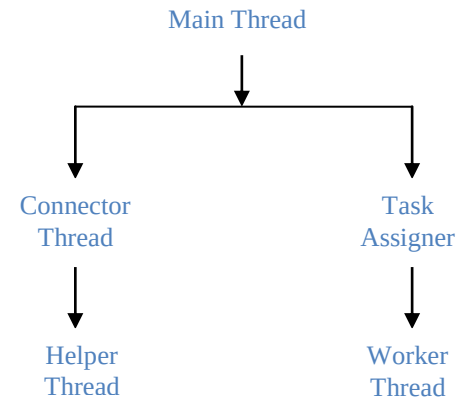

Fig 1. Classification of threads used in the program

A. *Handling parallel connection requests*

The scheduler is required to respond to and handle multiple connection requests from the client or the worker. After the main thread creates and starts the connector thread, the connector executes

```
Socket socket = serverSocket.accept();
```

in an infinite while loop, continuously accepting new connection requests. Each time a new connection is accepted, it also spawns new helper threads which actually perform the communication with worker or client.

The helper thread is responsible for the communication with the worker registering or the client submitting a job. Consider the new node reporting as a worker. The worker node is added and registered part of the Cluster data structure. After this is completed the socket may be closed and the thread terminated. If the new node is a client, the new job is registered into the job cluster data structure, which holds the number of jobs. The client thread after initially communicating 'job start' to the client, continuously accepts reports from the worker thread of which task has been completed and on what worker id. As soon as all the tasks have been completed the client thread passes 'job finish' to the respective client, closes the socket and terminates.

B. *Assigning Taks to Workers:*

The task assigner thread is responsible for allocating threads to worker threads. The worker thread are implemented as pool of threads using the Executor Service. The number of workers in the thread pool can be incremented or changed even dynamically if required, but is kept fixed at maximum of 8 based on the project description. The task assigner is mainly responsible for finding an incomplete job, a free worker and allocating the task to this worker. Each worker thread receives a worker object from the task-assigner thread. It uses this to establish connection with the worker.

In order to implement a fair policy, the following options were implemented:

First option: (Round Robin) was to allocate a remaining task per job to a free worker and proceed to the next job in queue. This ensure fairness, as the granularity of the assignment is the smallest possible, that of a single task. Since each task is completed per job, for each incomplete job, every job has the opportunity to proceed ahead and not face starvation. The disadvantages of this design is the inherent creation of connections between the worker node and scheduler worker thread for each and every task.

Second option: (Profiling and Pre-emption) The objective was to allocate multiple number of tasks to a single worker. This follows the FAIR scheduler mentioned in [1]. A min-share variable indicates the number of workers available per job. Number of tasks remaining in the job divided by the number of the workers indicates the number of tasks to be allotted in bulk to the worker. But, this must hold only until a new job or new worker is added or removed, after which the work needs to be re-distributed. The redistribution can be done after a certain period of time 'heartbeat' or immediately. In order for this to happen an in-flight task group assigned to a worker may need to be cancelled. But, care needs to be taken that at least a single task is completed before pre-emption or cancelling, else if the new events (job or working removal or addition) are very frequent, the job may face starvation. A way out was to use profiling in which the task completion time of the very first task is recorded and notified to the task assigner thread. The task-assigner thread pre-empts the worker thread only after waiting for this task completion time. The disadvantages of this design are mainly the overhead due to cancellation, notification of the date collected by profiling and the added complexity. Another possible way was to make some jobs non-cancellable if they have seen no progress until a certain fixed number of heartbeats.

After implementation, it was seen that round robin provides comparable performance unless the number of jobs are much less than the number of workers.

C. *Worker thread to Helper thread communication*

The helper thread always remain in communication with the client, until the job is completed. The worker thread needs to notify the helper thread after completion of each task. This is done by passing 'reports' using the global data structure 'job cluster'.

report = jobCluster.receiveReport(jobId); (in helper thread)

jobCluster.addReport(report, jobId); (in worker thread)

D. Fault tolerance

This mainly required recoverability if one of the workers died in the middle. Fault Tolerance has mainly two components

- Maintaining a history of tasks

Two arrays were created, pre-commit and post-commit. The length of these data-structures equals the number of tasks in the job. A 1 or 0 in this array indicates whether a job has been completed or not respectively. The pre-commit is set when the tasks are assigned by the task-assigner thread. The number of ones describes the number of tasks have already been assigned to a worker thread. The post-commit is set only when confirmation of completion is received from the worker. Thus, the post-commit refers to the tasks which have been completed for sure. In the task-assigner thread, while assigning tasks the first zero in the pre-commit indicates the next task to be assigned.

- Creating an Executor Thread Pool

An Executor Thread Pool was created so as use the automatic thread management. The threads on completion or on interrupts are automatically added to the pool, avoiding the overhead due to thread creation.

- Catching Exceptions

Using the try-catch block allows us to catch exceptions. There are two exception scenarios encountered. One if when the worker is killed before the task is completed. (ConnectException). Another occurs a new worker is registered, but is killed before it is assigned a task (EOFException).When these faults occurred, the pre-commit is reset and the entire post-commit is copied into the pre-commit data-structure, so as to reflect the actual number of tasks still left to be done.

```
}
catch (EOFException e){
cluster.removeWorkerNode(n);
```

```
jobCluster.resetPrecommit(taskIdStart,
tasksAssigned, jobId);
}
catch (ConnectException e)
{
cluster.removeWorkerNode(n);
jobCluster.resetPrecommit(taskIdStart,
tasksAssigned, jobId);
}
catch (Exception e)
{
System.out.println("Exception ??? \n "+e);
}
finally {
}
```

E. Worker.java alterations

The Worker node needs to be able to handle requests from different ports on the scheduler. The worker node also spawns a new thread on every connection from the scheduler after it has been registered initially. Each thread completes one (or more) task, communicates with the server and terminates.

IV. TESTING AND EXPERIMENTS

A. Testing for Non Blocking request handling:

To test this feature we used the two jobs: Mvm and Hello with slight modifications. A delay of 1 second was added to each (using Thread.sleep(1000)) to give us time to better analyze what was going on in the program execution. The number of tasks and the number of workers registering were varied. In order to submit requests in parallel System.nanotime was compared to a fixed predetermined value before proceeding with the code. This helped achieve the closest possible concurrent requests. Later more number of jobs were tested and the program ran successfully so we can clearly say that we can handle non-blocking requests.

B. Testing for Parallel Job Execution:

To test this feature we changed the Hello job to have ten tasks. Keeping the job constant we varied the number of workers. The number of tasks were evenly distributed amongst the workers, instead of the original implementation which had tasks assigned to a single worker. For example we made eight simultaneous workers and ran the Hello job on them. Initially the first eight tasks got allocated to eight free workers. Also we had put a one second sleep in each task of the job and so after approximately one second all the workers finished their tasks and were free to do the remaining two tasks in the job. The two workers which finish their initial tasks the fastest, pick up the two unfinished tasks required for the completion of the job. Later experiments were carried out with more number of jobs.

C. Testing for Fair Job Scheduling:

To test for fair job scheduling we need to make sure that each job get its chance without having to wait too long. That is we have to ensure that a single job cannot hog all the resources i.e. each job should have nearly the same share of CPU. To test this we gave parallel jobs and ensured that each one was getting a chance without having to wait too long. We passed five parallel jobs to our program, with the following configurations:-

1. Job Hello: 10 tasks, 1 second delay (sleep) for each task
2. Job MVM: 4 tasks, 10 second delay for each task
3. Job Hello: 10 tasks, 1 second delay for each task
4. Job MVM: 4 tasks, 10 second delay for each task
5. Job Hello: 10 tasks, 1 second delay for each task

The delay of MVM was kept high to check that it does not block Hello for too long. It was seen that each workers would initially start executing one task from the Hello job, so 8 out of 10 tasks of Hello were getting executed. After that the tasks are assigned in a fair manner as per the availability of workers and min-share share basis, tasks for different jobs would be allocated to the workers and it was seen that at any point in time no one job could block all workers. For the round-robin scheduler, one task per job is assigned while looping over the number of the jobs. Advantages of this implementation are the simple implementation complexity. The problem with this approach is mainly when the duration of tasks are very long and all workers are busy. A new job of short duration faces a latency equaling the task time of the job with the long tasks. Preemption based scheduler solved the problem, but the execution time observed for the pre-emption based scheduler was higher as compared to the round robin scheduler when the frequency of events occurring was high.

D. Testing for Fault Tolerance:

The following situations were tested for correctness:

The worker after being registered is immediately closed.

This tests the case that the worker has already been registered in the cluster and the connection for registering has already been closed. So, closing the worker node at this moment does not immediately give rise to an exception. This exception is caught when the task is assigned to the worker thread and it tries connecting to the assigned worker node. The observed behavior matched requirements.

The workers are all closed simultaneously

This is done by creating a separate thread which continuously monitors the system time. When a pre-determined time is reached all the worker java programs execute System.exit so as to terminate the program simultaneously

Individual worker node randomly started and shut down

A script was written which creates and shuts off the worker nodes.

```
worker node command &
PID=$!
# Wait
sleep 2
# Kill
kill $PID
```

This was looped with increasing sleep times.

E. Performance based testing

We varied number of jobs keeping the number of workers constant at 8 and observed the execution time. Then we kept the job and number of workers constant and varied the number of tasks per job and observed the execution time. Also, we varied the number of workers from 1 to 8 and recorded the execution time. (sleep time = 1 second per task)

V. RESULTS

Note: We used multiple instances of the Hello job with a latency of 1 second for each task in all the performance analysis.

Number of Jobs	Execution Time (ms)
1	1967
2	2964
3	3934
4	4868
5	6721
6	8156
7	8989
8	9832
9	11694
10	12733

Table 1. Variation of execution time wrt number of jobs

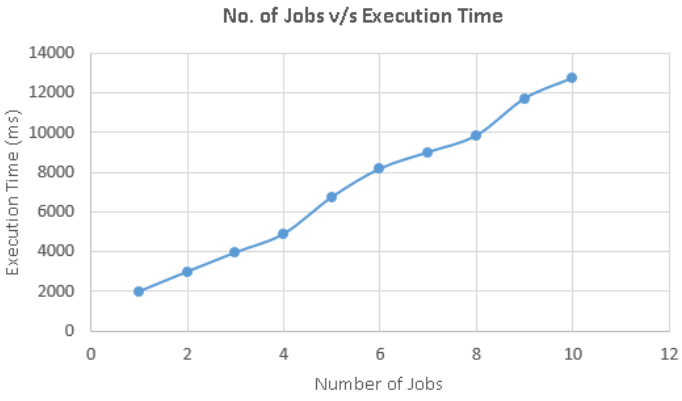

Fig 2. Variation of execution time with increasing no. of jobs

In this analysis, we kept the number of tasks for each job to be equal to 10. The graph shows an expected linear increase in execution time as the number of jobs increase. Another point to note is that each job is finished in about 1 second,which corresponds to the delay in each task.

Number of Tasks	Execution Time (ms)
1	963
2	963
3	963
4	963
5	965
6	965
7	965
8	966
9	1966
10	1966
16	1968
17	2969

Table 2. Variation of execution time wrt number of tasks

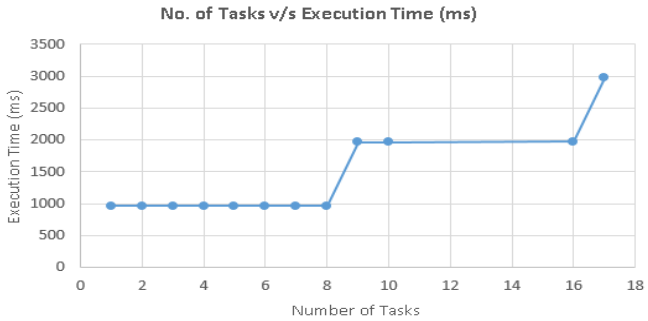

Fig 3. Variation of execution time with increasing no. of tasks

We see from the graph that the execution time increases in steps with respect to the number of tasks. Specifically this increase is seen every ninth task. This is expected as we have eight workers which do work in parallel but as soon as a ninth task comes it has to wait for about 1 second for the 8 tasks ahead of it to finish.

Number of worker Nodes	Execution Time for 20 tasks	Execution Time for 40 tasks
1	20053	40069
2	10026	20019
3	7017	14005
4	5030	9998
5	4015	7992

6	4008	6989
7	3010	5986
8	3009	5021

Table 3. Variation of execution time wrt no. of workers

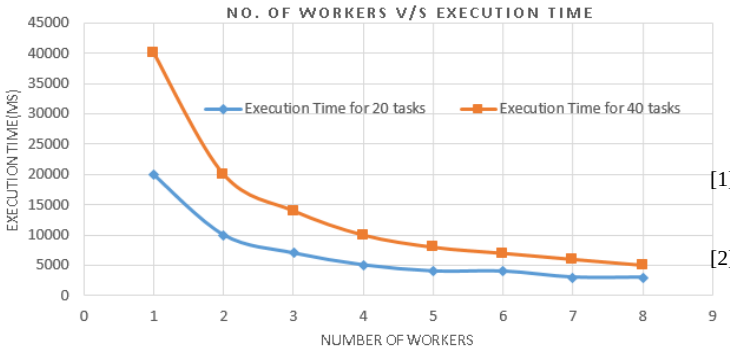

Fig 4. Variation of execution time with increasing no of workers

In this analysis we used 2 instances of the Hello job with 20 and 40 tasks respectively aand we increased the number of worker nodes from 1 to 8. As can be expected we got a decrease I the execution time, since the more number of workers we have, more work can be done simultaneously by employing the concept of parallel execution.

REFERENCES

[1] Job Scheduling for Multi-User MapReduce Clusters Matei Zaharia et. all., ECE Berkeley Technical Report UCB/ EECS-2009-55

[2] An Optimised Round Robin Scheduling Algorithm for CPU scheduling, Ajit Singh et. all, IJCSE international Journal on Computer Science and Engineering Vol. 02, No., 07, 2383-2385, 2010

[3] A Priority based Round Robin CPU Scheduling Algorithm for Real Time Systems, Ishwari Singh Rajput et.all, IJIET