ECE 6101 Course Project
Distributed Batch Processing System - Option 1

Abhishek Jain
ajain333@gatech.edu

Akar Shah
akar.j.shah@gatech.edu

System Overview

Fig. 1

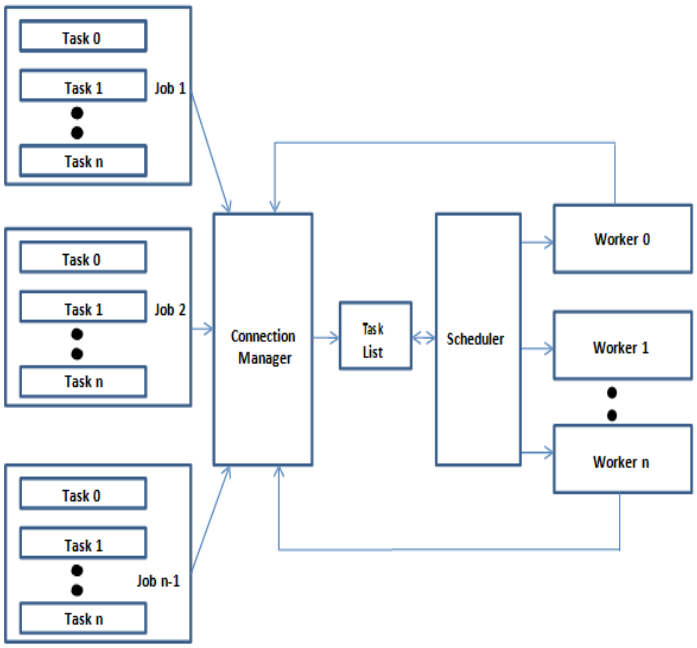


Fig. 2

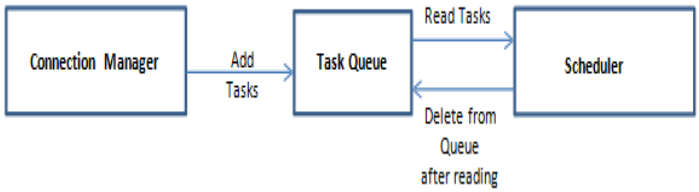
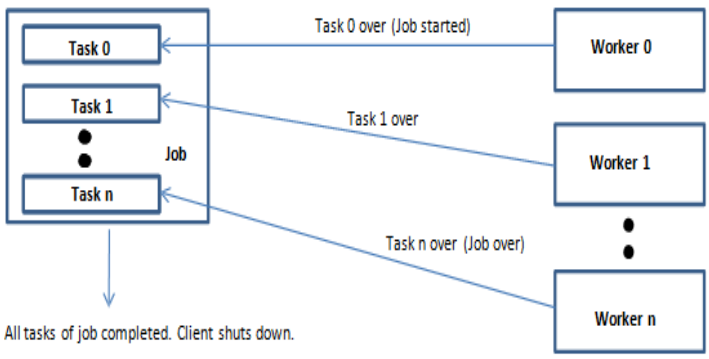


Fig. 3



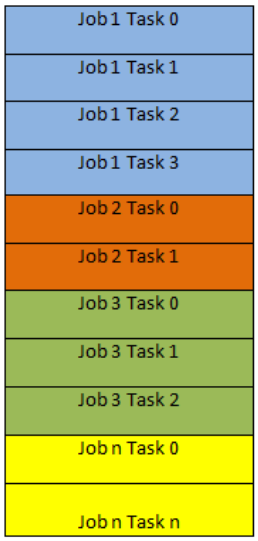
The figures show our implementation of the distributed batch processing system with the following features:

- Non-blocking request handling
- Parallel job execution
- Fair job scheduling
- Fault tolerance

Fig. 1 shows the basic components of the system. The clients can submit any number of jobs simultaneously at any point of time to the connection manager. The connection manager is responsible for accepting connections simultaneously from both workers and clients and updating the task list as well as worker list. The scheduler is responsible for fairly scheduling the tasks within different jobs and allocating a worker to each task. This is performed using the task queue, where each worker gets a task from the queue using a fair-scheduling algorithm and deletes the entry of that particular task in the queue. Thus both scheduler and the connection manager are responsible for updating the task queue which has been shown in Fig. 2. A different approach is followed in our system where the worker communicates with the client instead of the scheduler upon being assigned a task. A worker on receiving the first task of the job informs the client that the job it had assigned to the scheduler has started. This has been shown in Fig. 3. Each worker on completing a task informs the client of the task completion (using *Opcode.task_finish*). The worker that finishes the last task of that particular job which the client requested informs the client of job completion (using *Opcode.job_finish*) on which the client closes the socket that it had initially established. Once a worker finishes its task it is then added to the free worker list. This process is repeated till all tasks are complete.

Design of Features

Note: For all experiments, the sleep duration for each task of the *Hello* Job was changed from 1sec to 5secs and for the *Mvm* Job, the sleep duration for each task was changed from 0sec to 1sec. New job files were also created for different experiments such as *Hello8* (8 tasks), *Hello3* (3 tasks), *Hello2* (2 tasks). The only difference between these jobs is the number of tasks in each job.



1) Non-Blocking Request Handling

In order to have non-blocking request handling, it is essential that *socket.accept* command (which is a blocking command) be out of the continuously looping scheduler *run* function. In order to achieve this, a new thread was launched upon initializing the scheduler called the connection manager. As explained above, the connection manager is responsible for accepting connection requests simultaneously from any number of workers and clients. Upon receiving a connection request from a new worker, this new worker is added to the free worker list. Upon receiving a connection request from client (which also includes a job servicing request), all the tasks of that particular job are pushed one after another into Task List. The task list is later read by the worker nodes. The task list consists of JobID and Task ID.

As seen in the table, the tasks of different jobs are added one after another as and when a new job is requested to the serviced by the client. The tasks are added to the task list in increasing order of Job ID and Task ID. Each new incoming job is assigned a Job ID in such a way that this new job has a Job ID greater than the ID of the job that is already present in the list.

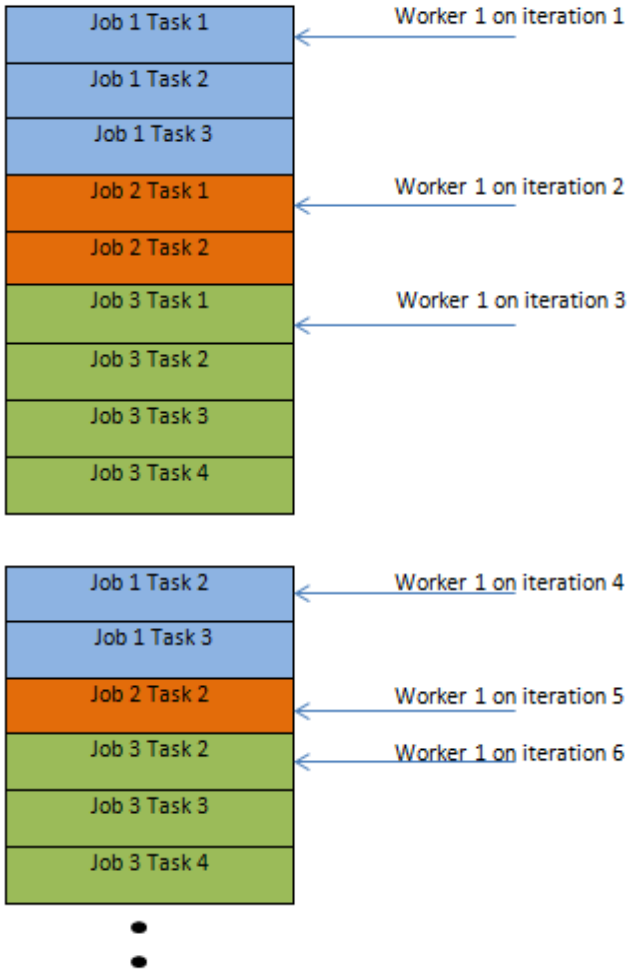
2) & 3) Parallel Job Execution and Fair Scheduling

Parallel job execution and fair scheduling are done using the following piece of code. The foundations for this are established as soon as the task list is made. The while loop of the scheduler contains the following small piece of code and repeats it as long as the scheduler is working.

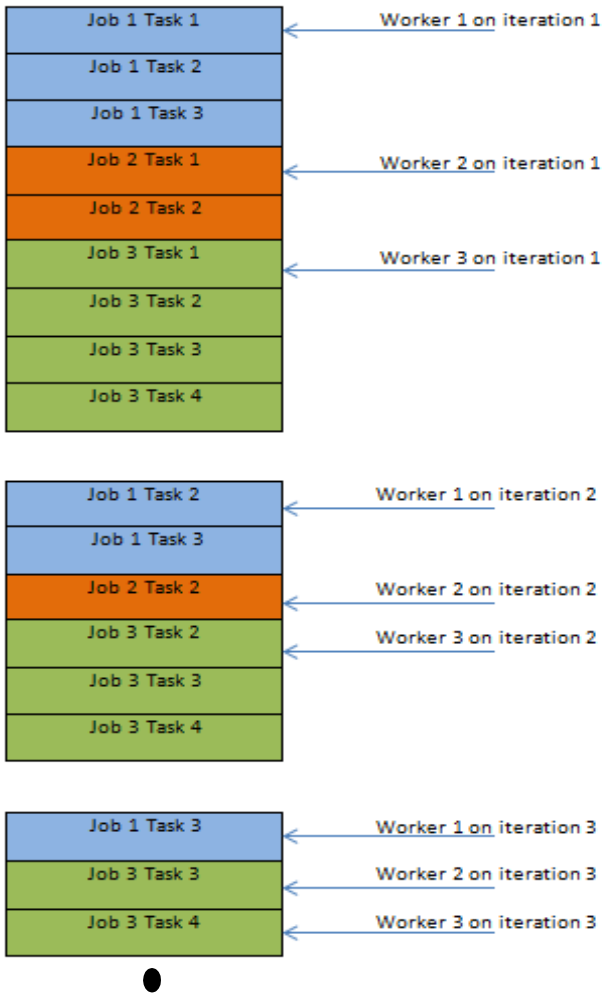
```
if(TaskTable.size()!=0)
    {
        if(TaskTable.size()>=wn.size()) //Number of tasks exceeding the number of free workers
            {
                int i=0;
                int index=0;
                while(wn.size()!=0)
                    {
                        boolean found=false;
                        for(int k=0;k<TaskTable.size();k++)
                            {
                                if(globaltaskid==TaskTable.get(k).jid)
                                    {
                                        found=true;
                                        globaltaskid++; //incrementing so as to get a task of a different job ensuring fairness
                                        index=k;
                                        break;
                                    }
                            }
                        if(!found)
                        {
                            for(int k=0;k<TaskTable.size();k++)
                            {
                                if(globaltaskid<TaskTable.get(k).jid)
                                {
                                    found=true;
                                    globaltaskid=TaskTable.get(k).jid;  //incrementing so as to get a task of a different job ensuring fairness
                                    globaltaskid++;
                                    index=k;
                                    break;
                                }
                            }
                        }
                        if(!found)  //selecting the first task in the task table as there is no new job assigned
                            {
                                globaltaskid=TaskTable.get(0).jid;
                                globaltaskid++;
                                index=0;
                            }

                        new Thread(new work(jobtable.get(TaskTable.get(index).jid),TaskTable.get(index).tid,wn.get(0))).start();
                        jb.modify(index, 0,0,false);    //deleting the task that has been assigned to a worker from the task list
                        wn.remove(0);   //removing a worker from the list of free workers
```

Each time the scheduler's loop is entered, the number of tasks in the *tasklist* and the number of free workers at that particular instance of time is calculated. Once that has been done the remainder of the task is divided into 2 cases. The first case handles the scenario where the number of tasks is greater than or equal to the number of free workers. The second case handles the scenario where the number of free workers is greater than the total number of tasks. In the second case, allotment of tasks to workers is fairly straightforward. Each worker gets a task and the task that has been allotted to a worker is removed from the task list as well as the worker that has been allotted the task is removed from the free worker list. The deletion of workers from the free worker list and the removal of the task from the task list are carried out in both cases upon the assignment of a task to a worker. However, it is the first case where things get interesting. In the first case, a global variable called *globaltaskid* is used. The *globaltaskid* is 1 at the start of the program. The scheduler on assigning a task to the worker increments the *globaltaskid*. When the next worker comes in, it checks what the *globaltaskid* is. Incrementing the *globaltaskid* helps us implement the feature of fair scheduling and parallel task execution. It does this in a very simple way. When a task is allotted to a worker, the *globaltaskid* is incremented by 1. When this happens, the new free worker that now checks the *tasklist* for a task to perform, picks up a task of that job that has its *jobid* greater than (or different from) that of the task that is now being serviced by the previous worker (provided that the task table has tasks from different jobs). In this way, at a particular time, each worker is given a task from a different job. This also helps achieve parallelism because if all tasks belong to the same *jobid,* different workers will be given different tasks of the same job. Now one may ask how this is done. The answer is pretty simple. On close observation of the above code, one may observe that the *globaltaskid* is incremented each time a task is assigned to a worker. But it may so happen (and it often does), that this incremented globaltaskid may be more than the maximum *taskid* that is present in the *tasklist*. In such a case, the worker is assigned a task of that *jobid* that is present on the top of the *tasklist*. This is done because the top of the *tasklist* will always have a task that is older than any other task in the remainder of the list. Also, this task will belong to a job that has a *jobid* less than that of any other *jobid* present in the *tasklist*. Now consider the case that the *tasklist* has only one job and all tasks in the *tasklist* belong to this job. In such a case, different workers will all pick tasks of the same job as is evident from the previous explanation, thus ensuring parallelism.

Case 1: 1 worker and number of jobs is greater than 1



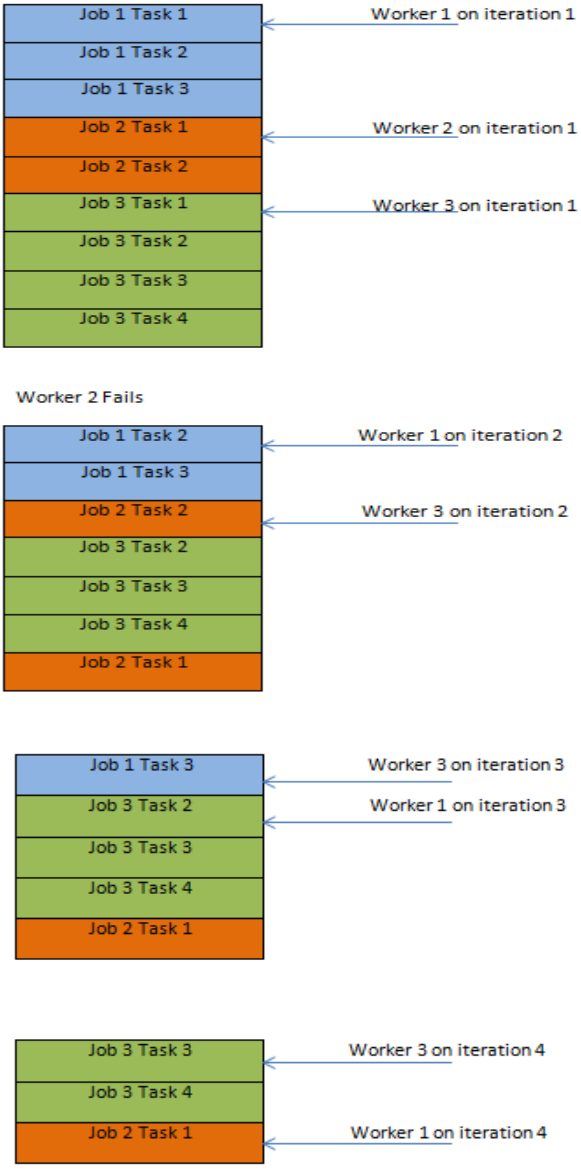
Case 2: Number of workers is less than number of task



Our system achieves fairness by using an approach similar to the bucket filling phenomenon where each job is considered to be a bucket and each bucket is filled with water(here workers) from time to time. More the workers, faster the job execution. Each time a worker is made free, it is added to one of the (buckets) jobs and execution of the job is carried out. The last two experiments are proof to the fact that our system ensures fairness and even distribution of resources. The first experiment (in this section) has 4 jobs (each with 4 tasks) and one worker. Following the analogy of the bucket and water, each time only one bucket is filled and only one job is allowed to move forward. Then the bucket of a different job is filled in the next iteration. None of the jobs is allowed to complete all its tasks in one go and each time a task of a new job is chosen. Thus all jobs finish nearly at the same time. The results appear to be the same even if different jobs are thrown at the scheduler. In the second experiment, there are 4 jobs (each with 4 tasks) and 3 workers (workers 2, 3 and 4). Following the same bucket-water analogy again, one sees that at any point of time, (2 buckets) 2 jobs are allowed to move forward (i.e. 2 buckets are filled with 2 workers and execution is allowed to continue). As can be seen from the final result, all jobs share the workers evenly between them. In this way it is shown how our system achieves fairness.

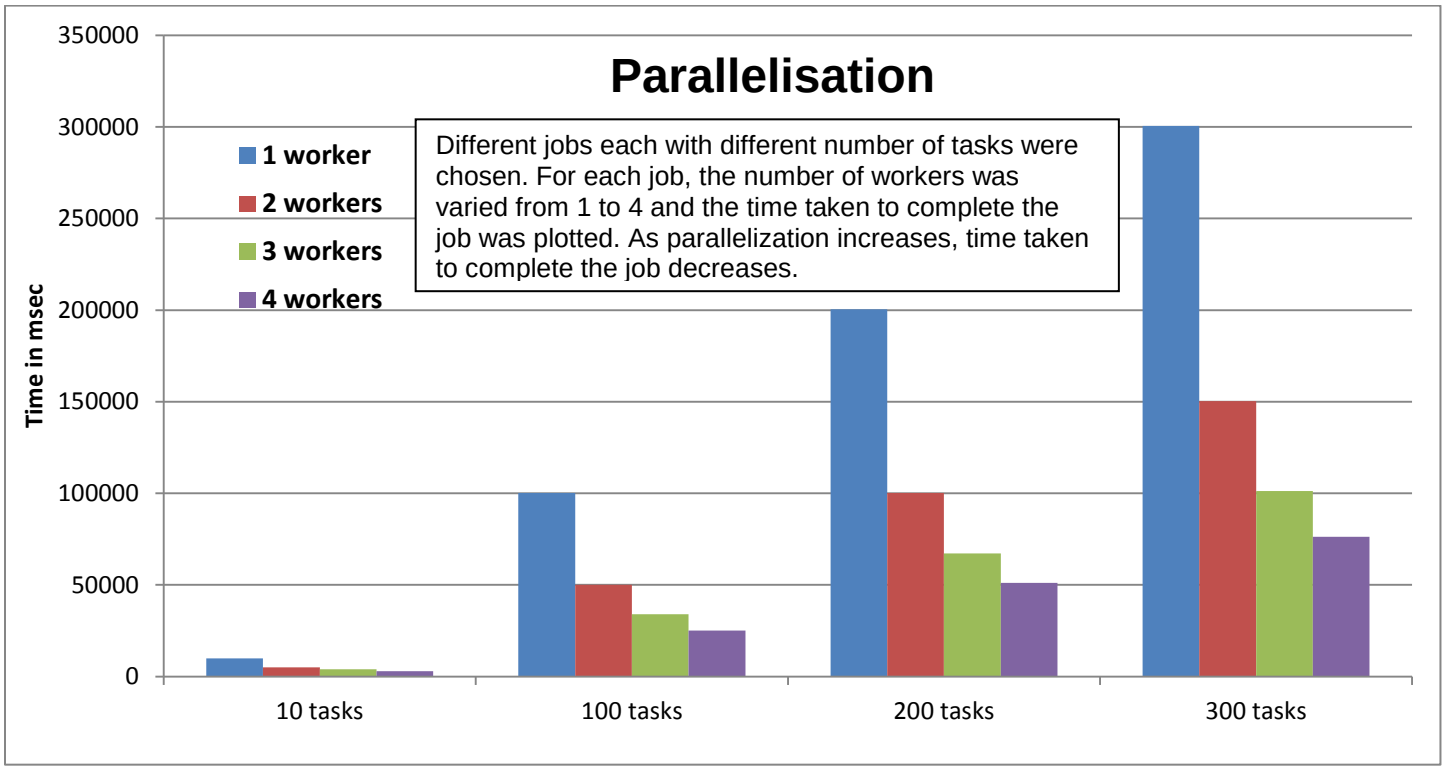
4) Fault-Tolerance

Fault tolerance is achieved by making our system robust against worker failures. In our system, when a worker fails, the task it is executing is pushed to the bottom of the task list along with its Job ID. This pushing of the failed task has been done keeping in mind cases where the task may need some time for the stabilization of the resources it might be using. Worker failures have been tested by stopping the worker which is executing a particular task. The workers were stopped by pressing the key Ctrl+C. Our system has the capability to recover from any number of worker losses and can continue executing jobs or tasks of different jobs as long as there is even 1 worker alive.



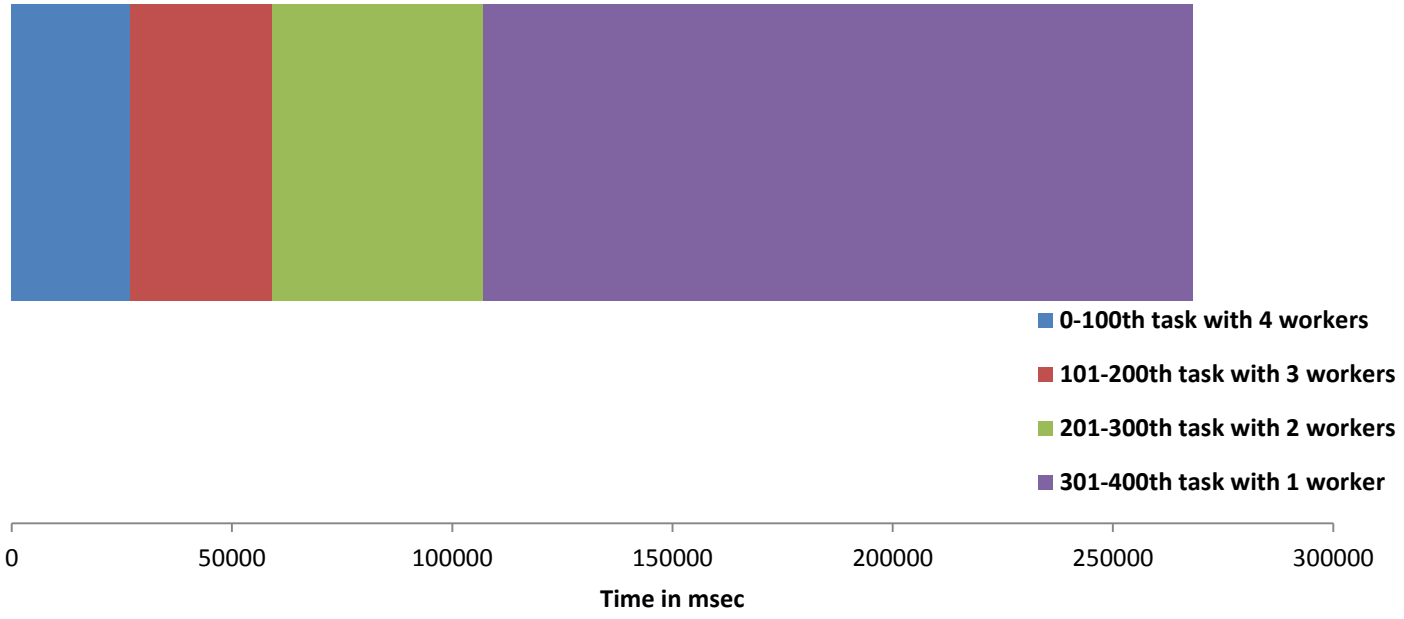
The diagram explains a scenario where there are 3 jobs (Job 1 with 3 tasks, job 2 with 2 tasks and job 3 with 4 tasks) and 3 workers. In the first iteration each worker gets a task of a different job. Now assuming worker 2 fails, since it had task 1 of job 2, task 1 of job 2 is pushed at the bottom of the tasklist. However since worker 1 and 3 are still alive they complete the tasks that have been assigned to them and become free for iteration 2. Now worker 1 and 3 choose the tasks in iteration 2 as described in the diagrams below. By the end of iteration 3 only one task of job 1 and 3 tasks of job 3 remain along with the failed task of job 2. Now let us analyze iteration 3 in detail. Worker 1 takes task 2 of job 3 and increments globaltaskid by 1. Since there is no task with jobid 4 worker 3 takes task 3 of job 1 and first sets the globaltaskid to 1 and then increments it to 2 once it takes the task 3 of job 1. Since, now globaltaskid is 2, worker 1 comes in, searches for a task with jobid 2 and finds the previously failed task 1 of job 2, takes that task for execution and increments the globaltaskid to 3 thus asking the next free worker to choose a job with id 3 and then execution is carried out normally. Thus in this case it is seen that the failed task is executed as soon as possible and does not have to wait for all remaining tasks in the tasklist to complete in spite of being pushed to the last of the tasklist.

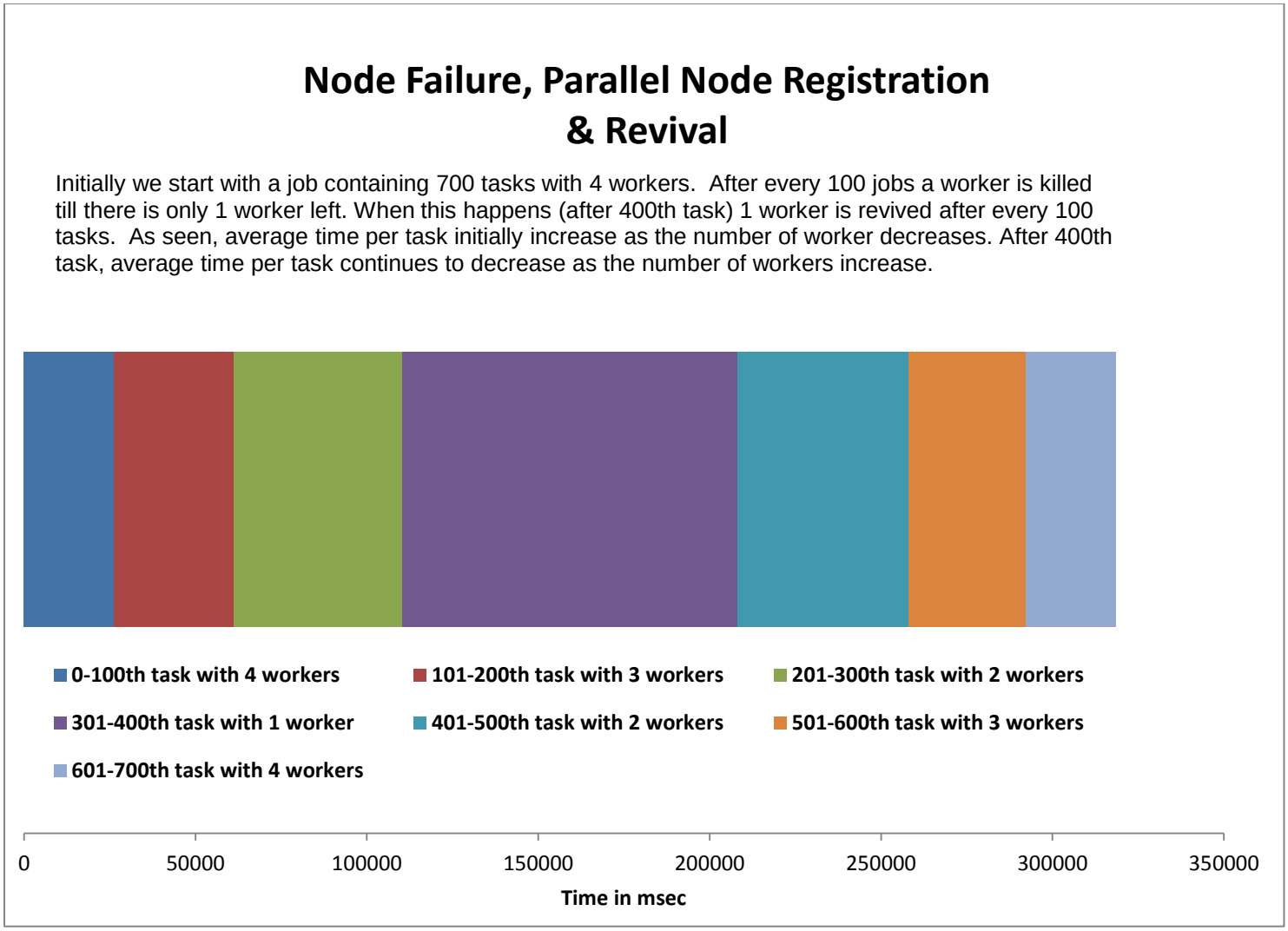
Experiments



Node Failure

Experiment starts with 400 tasks and 4 workers. After completion of every 100 tasks, a single worker is made to fail. As can be seen, on node/worker failure, the remaining tasks are allotted to the existing workers. As the number of worker decreases, the time taken to execute same number of tasks increases. Thus in spite of node failure, all tasks is eventually completed.





Experiment for Fair-Scheduling:

For carrying out this experiment, different jobs each with different number of tasks were chosen. The length of each task was hard-coded to 1 sec. The number of workers for each test case was kept as 4. As can be seen from the table below, in the presence of fair-scheduling all jobs in a particular test case complete at the same instance with the same execution time. However, in the absence of fair-scheduling the execution time of all jobs in a particular test case remains the same but jobs complete at different instances. E.g. for test case 1 with 4 jobs, job 1 completes at 25 secs, job 2 completes at 50 secs, job 3 completes at 75 secs and job 4 completes at 100 secs. This happens in the absence of fair-scheduling. When fair scheduling is applied, all jobs complete at the same instance i.e. at 100 secs. Thus allocating resources fairly over time to all jobs.

	Execution Time of each job (msec)			
Test case	Without Fair-Scheduling (Similar to FIFO)	With Fair-Scheduling	Number of Tasks/Job	Number of Workers
Test Case 1	25038	99170	100	4
Test Case 2	51121	199409	200	4
Test Case 3	76219	299551	300	4

Experiment for Non-Blocking Request Handling (Parallel Job Submission):

For this experiment first a job with 300 tasks was submitted along with a job with 100 tasks. When the 300 task job was still running and the 100 task job had completed another job with 200 tasks was started thus showing parallel job submission (this is depicted clearly in 100with300part1.png). At the end of the 300 task and 200 task job completion another screenshot was taken which shows parallel job submission and completion (200with300part2.png)

The total time of execution for each job is as follows:

Job with 300 tasks - 50 seconds for the first 100 tasks (along with the 100 task job) + 100 seconds for the remaining 200 tasks (along with the 200 task job) = 150 seconds
Job with 100 tasks - 50 seconds for 100 tasks (along with the 300 task job)
Job with 200 tasks - 100 seconds for 200 tasks (along with the 300 task job)

Also shown is a case where a job with 200 tasks is started and at the completion of its 100 tasks another job is started that has 100 tasks. Thus each job gets 2 workers at a time. This can be easily seen from the execution time of each job (which is shown in Jobadding100with200.png). The job with 200 tasks completes in roughly 75 seconds

(100/4 workers =) 25 + (100/2 workers =) 50 = 75 seconds and the job with 100 tasks completes roughly in 50 seconds

NOTE: All experiments that have been mentioned above, their screenshots are present in the folder "Experiment Screenshots" which is included in the submission package.