Optimised scheduling algorithm

Student Info:

- Jack Davenport (45946396)

Optimising for:
Rental Cost

Links:
GitHub: https://github.com/mq45946396/comp3100-project

Introduction:
This project is a modified version of the stage one project code, with a new scheduling algorithm implemented in the project. The goal of the new algorithm is to improve the metrics over the all-to-largest (ATL) algorithm which was implemented in the stage one implementation (specifically, the rental cost of the algorithm).

In practice, compared to the previous algorithm it was able to optimize turnaround time and rental cost pretty consistently. However, it was not able to efficiently improve turnaround time or resource utilization compared to the three baseline algorithms (BF, FF and WF), and was only able to improve the rental cost in some configurations of the simulator compared to the baseline algorithms.

This is achieved because the algorithm attempts to allocate each job to the server which has the closest number of resources available for the job, which increases the efficiency of job cost by picking the most appropriate server based on the job's CPU requirements, and reduce the amount of time a job must wait before running on any given server.

Jack Davenport
45946396

Problem Definition

The primary issue with the current ATL algorithm is that it is very inefficient and does not use the server resources in the most effective manner, especially when compared to the three baseline algorithms. It performs the worst when looking at turnaround time. As shown below, when compared to the three baseline algorithms (FF, BF and WF respectively), ATL turnaround time performance is several times worse than the other baseline algorithms, meaning jobs will take much longer to complete when using the algorithm. This makes sense, as the largest server's cores will be quickly consumed and result in most jobs needing to queue for extensive periods of time before running.

Turnaround Time

Average	254086.33	1473.33	1462.83	6240.72

Despite this, ATL performs fairly well in terms of cost and resource utilisation. Since only one server is used, its cost per hour will be consistent, which would not happen with multiple servers being deployed. And since only one server is performing tasks, it is active at all times during the process, and therefore it's resource utilisation is always 100%.

Resource Util

Average	100.00	66.79	64.94	72.85

Cost

Average	256.05	417.90	414.42	443.03

Algorithm Details

My new algorithm attempts to improve the scheduling efficiency by checking the core count of the servers which are available to be scheduled. It does this by calculating the difference between the number of cores required by the job and the number of cores which are available on the server. It then picks the server with the smallest difference and schedules the job to that. This means that each job goes to the server which is most appropriate for each given job. In an effort to promote preloading jobs while booting, the algorithm will also provide a bias to the difference value, halving it if the server is booting, ensuring that currently booting servers are more likely to get the jobs.

In pseudo code, the algorithm works as follows:

```
FOR EACH Job j:
   GETS All AS servers
   FOR EACH Server s:
      let bootBias = 2 if s.state == booting, else 1
      let difference = abs(s.cores - j.cores) / bootBias
   SORT servers BY difference DESCENDING
   SCHD j ON servers[0]
```

There may be situations where either:

- There are no servers with enough cores to complete the job
- There are no servers with sufficient memory or disk requirements

In this situation, the algorithm will schedule the job to the server which has the highest number of total cores (i.e. the largest server). However after some jobs have completed executing, the algorithm can continue allocating jobs to the most appropriately sized server.

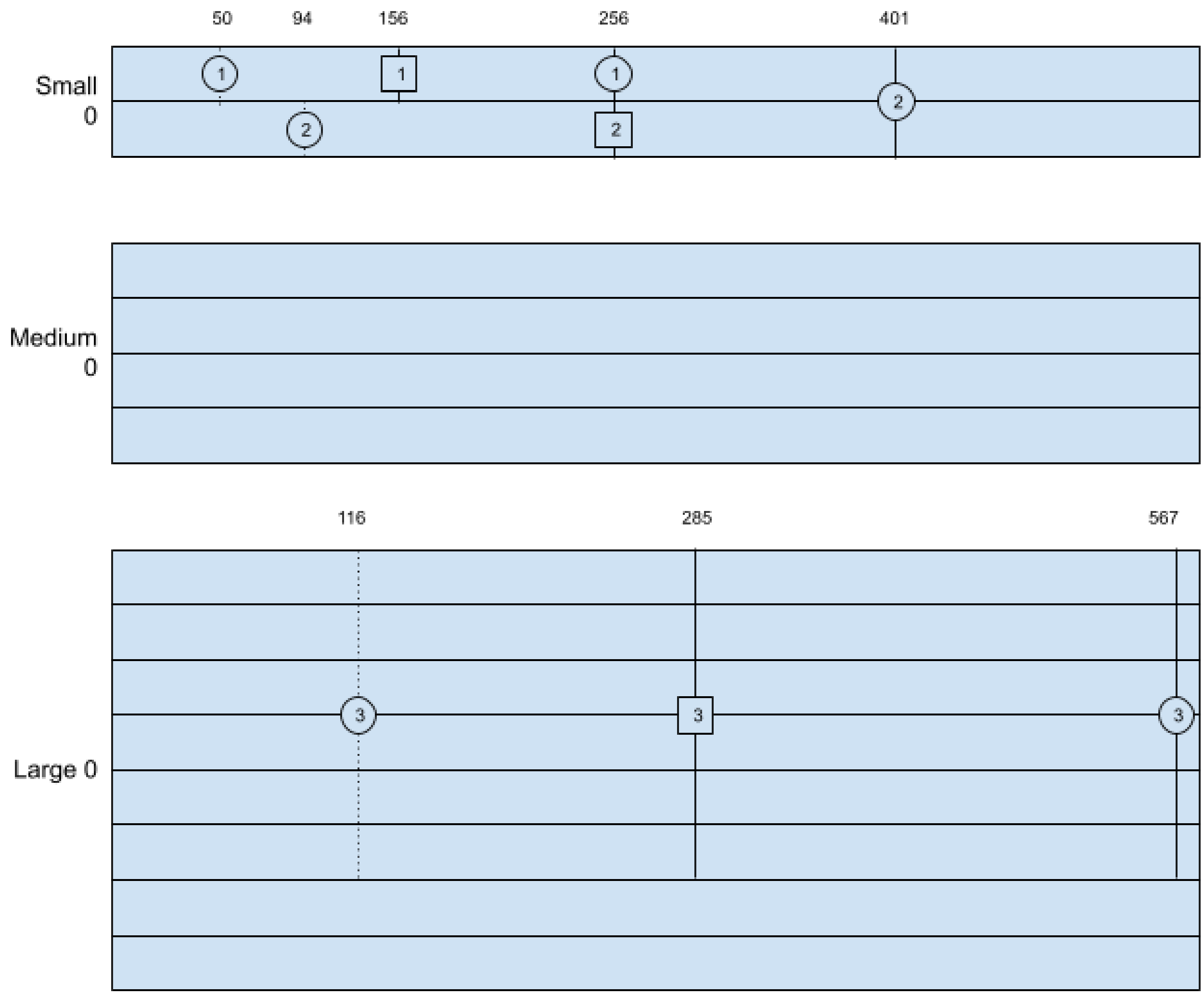
Scheduling Example

For this scenario, the following job and server configurations will be used:

Job	Cores Required	Server	# Cores
1	1	Small 0	2
2	2	Medium 0	4
3	6	Large 0	8

The resulting scheduling should result in the first job being allocated to small 0, since it's core count is the closest to job 1's requirements. The second job will be allocated to small 0, since it will be currently booting and has the smallest difference. Finally, job 3 will be allocated to large 0, since it is the only server large enough to handle job 3 and has the smallest core difference.

In a diagram form, the schedule may look like this.



Performance

When running with the stage 2 configurations, the performance is unfortunately not as good as I hoped. While it is effective in reducing the rental cost of scheduling compared to the baseline algorithms, ATL is still able to achieve a cheaper rental cost. I believe that this is mostly caused by ATL's design using only a single server to run all jobs.

While it dramatically improves turnaround time compared to ATL, the algorithm's turnaround time is at least twice as slow as BF and FF, but is half as slow as WF. It is quite similar in performance to the baseline algorithms in terms of utilization, but is still slightly worse compared to the baseline algorithms, and is significantly worse than ATL which achieves 100% utilization, since it only uses one server.

Turnaround time

Config	ATL	FF	BF	WF	Yours
config100-long-high.xml	672786	2428	2450	29714	8571
config100-long-low.xml	316359	2458	2458	2613	3674
config100-long-med.xml	679829	2356	2362	10244	5753
config100-med-high.xml	331382	1184	1198	12882	3853
config100-med-low.xml	283701	1205	1205	1245	2785
config100-med-med.xml	342754	1153	1154	4387	3993
config100-short-high.xml	244404	693	670	10424	2941
config100-short-low.xml	224174	673	673	746	1252
config100-short-med.xml	256797	645	644	5197	2706
config20-long-high.xml	240984	2852	2820	10768	6326
config20-long-low.xml	55746	2493	2494	2523	2757
config20-long-med.xml	139467	2491	2485	2803	3380
config20-med-high.xml	247673	1393	1254	8743	4953
config20-med-low.xml	52096	1209	1209	1230	1493
config20-med-med.xml	139670	1205	1205	1829	2242
config20-short-high.xml	145298	768	736	5403	3694
config20-short-low.xml	49299	665	665	704	763
config20-short-med.xml	151135	649	649	878	1248
Average	**254086.33**	**1473.33**	**1462.83**	**6240.72**	**3465.78**
Normalised (FF)	172.4568	1.0000	0.9929	4.2358	2.3523
Normalised (BF)	173.6947	1.0072	1.0000	4.2662	2.3692
Normalised (WF)	40.7143	0.2361	0.2344	1.0000	0.5553

Improvement: -13.30%

Pros

- More cost effective than ATL, and is much more efficient in terms of turnaround time.
- Attempts to allocate jobs to the most appropriate server.

Cons

- In most situations, it is less efficient than the three baseline algorithms.
- When no suitable servers are present, it resorts to the inefficient ATL algorithm.

Conclusion

While the algorithm achieves its primary goal of reducing costs compared to the three baseline algorithms, it sacrifices its performance in turnaround time and resource utilization when compared to the other algorithms. That being said, it still performs significantly better than ATL in terms of turnaround time, making it a more efficient algorithm in terms of the time taken to run jobs.

In retrospect, I would have liked to better analyse the cause and effect of different scheduling techniques to derive an algorithm which performs better than all four existing algorithms.