# Job Scheduler Using Best Fit Algorithm w/ Rescheduling

Luke Glover, 4595587

## Introduction

A distributed system is a computing system where multiple nodes are coordinated so as to appear as a single system to the end user. Where different servers have different capabilities in regard to CPU cores, memory and disk capacity, scheduling becomes a key problem. Scheduling algorithms can optimise for different variables, including server rental cost, turnaround time or resource utilisation.

For stage 2, a new scheduling algorithm was to be developed that optimised for one of the three variables. This project proposes and implements a new scheduling algorithm, a modified best fit algorithm that is capable of rescheduling queued jobs when space becomes available on other servers, to minimise the renal.

## Problem Definition

The previous scheduling algorithm, all to largest, did not perform intelligent schedules and resulted in poor performance in regard to turnaround time and rental cost. It did, however, perform well in regard to resource utilisation.

The aim of this project is to implement a smarter scheduling algorithm that minimises the rental cost of server resources for any given set of jobs, without significantly sacrificing turnaround time or resource utilisation as compared to all to largest, and the three baseline algorithms: first fit, best fit, and worst fit. Total rental cost is determined by the rental cost of each server multiplied by the time the server is running.

## Algorithm Description

The algorithm consists of two parts: a scheduler and a balancer. The scheduler is called in response to a JOBN message, and will schedule the new job to a server. The balancer is called in response to a JCPL message, and attempts to migrate queued jobs to any servers that have become idle.

#### Scheduling

Scheduling uses a modified best fit algorithm. Whereas the standard best fit algorithm only considers servers without waiting jobs, this modified algorithm ignores the waiting job queue, and instead considers the number of unused cores on a server when determining if a job fits on a server.

The algorithm iterates over a list of servers, S, sorted in ascending order of core count. The list is sorted as a performance improvement, because if a server is found with a score of 0.0 (see step 2 in process below for score calculation), the loop can break before reaching the end of the list. The first server in the list that has available cores greater than or equal to the core requirement of a job, *j,* and meets the memory and disk requirement, is returned. More formally:

**Input**: a job *j*; a list of servers, *S*, in ascending order of core count  
**Process**: iterate over *S*, returning the first server that satisfies the requirements:

1. Sufficient memory and disk for *j.*
2. Available cores on *s* - core requirement of *j* is 0.0.

Or, returning any server that satisfies the requirements:

1. Sufficient memory and disk for *j.*
2. Available cores on *s* - core requirement of *j* is the lowest.

**Output**: A server, *s*, for job *j* to be scheduled on, or no server.

If no server if found with the above process, the initial scheduler performs a second search which returns the first server that has total cores (not available cores) greater than or equal to the core requirement of *j*. The server returned by this search cannot run the job immediately, and the job will be queued on the server. More formally:

**Input:** a job *j*; a list of servers, *S*, in ascending order of core count **Process:** iterate over *S*, returning the first server that satisfies the requirements:

1. Sufficient memory and disk for *j*
2. Total cores on s – core requirement for j is the lowest

**Output:** A server, *s*, for job *j* to be scheduled on.

#### Balancing

The balancing algorithm also uses a modified best fit algorithm, similar to the scheduling algorithm, but without performing the second search if no server is found. Instead, in these cases, the job is not migrated. More formally:

**Input**: a job *j,* that is queued on a server; a list of servers, *S*, in ascending order of core count, excluding the server that *j* is currently scheduled on.  
**Process**: iterate over *S*, returning the first server that satisfies the requirements:

1. Sufficient memory and disk for *j.*
2. Available cores on *s* - core requirement of *j* is 0.0.

Or, returning any server that satisfies the requirements:

1. Sufficient memory and disk for *j.*
2. Available cores on *s* - core requirement of *j* is the lowest.

**Output**: A server, *s*, for job *j* to be scheduled on, or no server.

#### Example Schedule

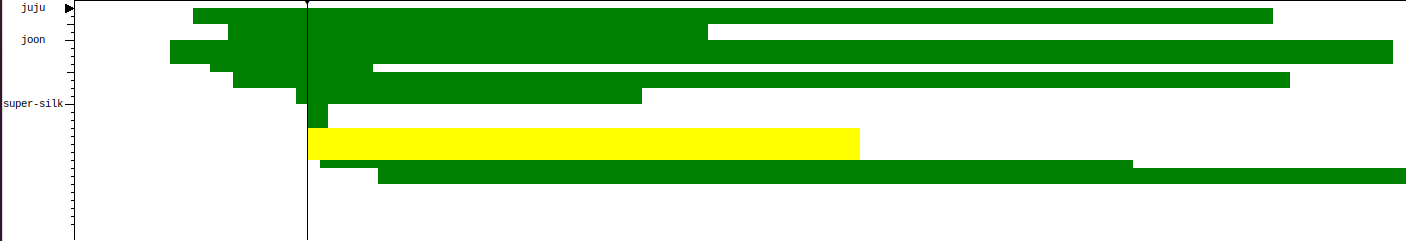
For this example, the server was run with the ds-sample-config01.xml configuration file provided in the ds-sim repository. An excerpt from the ds-system.xml file is as follows:

|  |
| --- |
| <server type="juju" limit="2" bootupTime="60" hourlyRate="0.2" coreCount="2" memory="4000" disk="16000" />  <server type="joon" limit="2" bootupTime="60" hourlyRate="0.4" coreCount="4" memory="16000" disk="64000" />  <server type="super-silk" limit="1" bootupTime="80" hourlyRate="0.8" coreCount="16" memory="64000" disk="512000"  /> |

And, the ds-jobs.xml file is as follows:

|  |
| --- |
| <jobs>  <type name="short" minRunTime="1" maxRunTime="300" populationRate="60" />  <type name="medium" minRunTime="301" maxRunTime="1800" populationRate="30" />  <type name="long" minRunTime="1801" maxRunTime="100000" populationRate="10" />  <job id="0" type="medium" submitTime="37" estRunTime="653" cores="3" memory="700" disk="3800" />  <job id="1" type="medium" submitTime="60" estRunTime="2025" cores="2" memory="1500" disk="2900" />  <job id="2" type="medium" submitTime="96" estRunTime="343" cores="2" memory="1500" disk="2100" />  <job id="3" type="medium" submitTime="101" estRunTime="380" cores="2" memory="900" disk="2500" />  <job id="4" type="short" submitTime="137" estRunTime="111" cores="1" memory="100" disk="2000" />  <job id="5" type="short" submitTime="156" estRunTime="8" cores="3" memory="2700" disk="2600" />  <job id="6" type="medium" submitTime="198" estRunTime="1074" cores="4" memory="4000" disk="7600" />  <job id="7" type="medium" submitTime="225" estRunTime="442" cores="2" memory="500" disk="2100" />  <job id="8" type="medium" submitTime="249" estRunTime="926" cores="1" memory="100" disk="800" />  <job id="9" type="medium" submitTime="308" estRunTime="2010" cores="2" memory="600" disk="1500" />  </jobs> |

The resulting scheduling diagram is as follows:



In this example, job 6, highlighted in yellow, is scheduled on Super-Silk 0 alongside job 5. Further, job 7 and job 8 are later also scheduled on Super-Silk 0. These scheduling decisions demonstrate the difference between the baseline best fit algorithm and the modified best fit algorithm used here. The baseline algorithm would have queued job 6 on a Joon type server, whereas this algorithm found an opportunity to execute the jobs in parallel.

This example, however, does not demonstrate any balancing, as all jobs begin executing immediately after scheduling, and balancing is only performed on queued jobs.

## Implementation

The algorithm was implemented as a class implementing a *SchedulingAlgorithm* interface. This interface describes two required methods: *schedule* and *balance*. Depending on the value of the command line argument -a, the main method creates an instance of a *SchedulingAlgorithm*. After receiving a JOBN message, the *schedule* method is called, and after receiving a JCPL message, the *balance* method is called.

The *Protocol* class handles translation between the text based communication protocol and the object based internal state used by the client, updating the static *SystemStatus* class with the latest information from DS-Server. A *SchedulingAlgorithm* can access information about the simulation through the *SystemStatus* class, which provides methods such as *getJobById*, *getJobsByState*, and *getServers*. These methods return a *Job* object, and *ArrayList* of *Job* objects, and an *ArrayList* of *Server* objects, respectively. The algorithm can make scheduling decisions by calling the *scheduleJob* and *migrateJob* methods on a *Server* and passing a *Job* as a parameter.

This

## Evaluation

The algorithm was evaluated using the provided test\_results script. The script used the configuration files in the configs/other directory from the distsys-MQ/ds-sim git repository. For each metric, the average of all test runs was calculated, and compared against ATL and the three baseline algorithms.

The table below shows the average result of each algorithm for each of the three metrics:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Metric | ATL | FF | BF | WF | Modified Best Fit |
| Turnaround Time | 254086.33 | 1473.33 | 1462.83 | 6240.72 | 52357.44 |
| Resource Utilisation | 100.00 | 66.79 | 64.94 | 72.85 | 70.29 |
| Rental Cost | 256.05 | 417.90 | 414.42 | 443.03 | 382.94 |

From these results, the modified best fit algorithm performs comparably to the baseline algorithms with respect to resource utilisation. This is considered an acceptable result for the purposes of this project. Less acceptable, however, is the turnaround time, which is significantly higher than the baseline algorithms. The exact reason for this is unknown, as the algorithm attempts to maximise parallelisation (running jobs simultaneously on the same server), which should, in theory, result in a reasonable turnaround time. More investigation would be needed to determine the reason for this result. The algorithm performs better than all three baseline algorithms with respect to rental cost, which is the metric that this algorithm was designed to minimise. However, it does not outperform all to largest, as it runs all jobs on a single server, which is difficult to exceed in terms of cost effectiveness.

The modified best fit algorithm should be used when rental costs is to be minimised, but turnaround time should still be reasonable. The cost associated with the algorithm is approximately 150% of all to largest, but completes all jobs in approximately 20% the time, i.e. it is 5 times faster for 1.5 times the cost.

For example, a set of jobs that is run overnight, but where all to largest would not complete, such as data backup, may be a suitable task for the algorithm.

## Conclusion

The project is considered a success by the authors. The new algorithm successfully reduced rental cost when compared to the baseline algorithms and presents an interesting and useful alternative to the all to largest algorithm. With further development, the algorithm could perform better. For example, consolidating running jobs into the fewest number of servers when space become available, terminating the now unneeded servers, would further reduce rental cost. Code quality in some parts of the project are also less than satisfactory, and should be rewritten to be more reliable before using the algorithm in any production environments.

## References

GitHub Repo: <https://github.com/Luke-Glover/COMP3100_Assignment_Stage_2>

[1] Y. Lee, Y. Kim, and J. King, ‘ds-sim: A Distributed Systems Simulator User Guide’. [Online]. Available: https://github.com/distsys-MQ/ds-sim.

[2] E. Gamma, *Design patterns: elements of reusable object-oriented software*, 37th printing. Reading, Mass: Addison-Wesley, 1995.