COMP3100

Simulated Distributed System Job Scheduler

Cheap-Fit resource utilisation algorithm

Name	Student ID
Max Williams	45200556

Introduction

This project (Stage 2) is focusing on a very relevant subject which is creating and analysing scheduling algorithms by strict performance metrics. Job scheduling revolves around assigning a specific job to a server that has the required resources to complete the work. Many algorithms exist to do this that have different positives and negatives such as First-fit which is fast but inefficient, Best-fit which is memory efficient but slow and Worst-fit which is slow but can put small jobs in the fragmentation gaps on the server.

A scheduler is usually created to load balance resources efficiently, allow multiple users to share resources and to achieve a target quality of service. Scheduling is fundamental to computation as it is a key concept of multitasking.

The overall goal of Stage 2 is to code a new Scheduling algorithm that will optimise one or more of the provided metrics and to create a report that will outline the algorithm, justify why I chose to create it how I did, clearly indicate the performance objectives, and compare the results to the three baseline algorithms (FF, BF, WF). This algorithm must also work with any provided simulation configuration.

Problem Definition

For Stage 2, we will be creating our own custom algorithm that must optimise one of more of the following objectives:

- Minimise average turnaround time
- Maximise average resource utilisation
- Minimise total server rental cost

These objectives are often conflicting so therefore, optimising in one area may lead to sacrificing performance in another metric.

The main metric I focused my algorithm on was to **minimise the total server rental cost**. But I also maximised average resource utilisation. These two metrics came at the cost of having a increase in average turnaround time.

I chose to create my algorithm in this way as many individuals and companies prefer to keeps costs (overhead) as low as possible even if it results in a longer turn-around time. This algorithm would also be useful if the jobs were not on a strict time-schedule to be allocated such as making system backups.

Algorithm Description

Configuration file used: ds-config01—wk9.xml

```
<servers>
<server type="tiny" limit="1" bootupTime="40" hourlyRate="0.4" coreCount="1" memory="4000" disk="32000"/>
<server type="small" limit="1" bootupTime="40" hourlyRate="0.4" coreCount="2" memory="8000" disk="64000"/>
<server type="medium" limit="1" bootupTime="60" hourlyRate="0.8" coreCount="4" memory="16000" disk="128000"/>
</servers>
<jobs>
<job type="short" minRunTime="1" maxRunTime="60" populationRate="30"/>
<job type="medium" minRunTime="61" maxRunTime="600" populationRate="40"/>
<job type="long" minRunTime="601" maxRunTime="3600" populationRate="30"/>
</jobs>
<workload type="unknown" minLoad="20" maxLoad="60"/>
<termination>
<condition type="endtime" value="86400"/>
<condition type="jobcount" value="5"/>
```

Figure A- Configuration file

Figure B- Example Scheduling

```
SENT OK
RCVD REDY
SENT JOBN 54 1 1144 1 400 800
RCVD SCHD 1 tiny 0
t:         54 job     1 (waiting) on # 0 of server tiny (booting) SCHEDULED
SENT OK
RCVD REDY
SENT JOBN 55 2 260 2 900 1600
RCVD SCHD 2 small 0
t:         55 job     2 (waiting) on # 0 of server small (booting) SCHEDULED
SENT OK
RCVD REDY
```

Figure C- Final Schedule Results

```
# ------------------------------------------------
# 1 tiny servers used with a utilisation of 100.00 at the cost of $0.45
# 1 small servers used with a utilisation of 100.00 at the cost of $0.03
# 1 medium servers used with a utilisation of 100.00 at the cost of $0.56
# ================================== [ Summary ] ==================================
# actual simulation end time: 4095, #jobs: 5 (failed 0 times)
# total #servers used: 3, avg util: 100.00% (ef. usage: 100.00%), total cost: $1.05
# avg waiting time: 430, avg exec time: 1375, avg turnaround time: 1805
```

Description & Discussion:

My algorithm works off a very simple principle: If the job requires **X** number of cores, **Y** amount of memory and **Z** amount of disk then it will be sent to a server that has the same **X** cores as long as the jobs **Y** (mem) and **Z** (disk) are lower than the servers **Y** and **Z**. If the job does not fit the above specifications it will be compared to the next largest server.

In Figure B you can see that JOBN 1 was assigned to the server "tiny" as that server has the same Core-count and larger memory and disk than the job requires (Figure A).

I went with this algorithm design because I noticed that the default "all-to-largest" algorithm was often the cheapest, but it could be even cheaper if I assigned jobs to smaller servers that had same amount of cores, mem and disk.

This overall results in a **cheap total cost** and **high average utilisation** at the cost of high turnaround time.

Implementation Details

```java
public void sendToServer(String x) {
    out.write(x + "\n");
    out.flush();
}
```

Figure 1

```java
public boolean newStatus(String x) throws IOException {
    input1 = input.readLine();
    if(input1.equals(x)){
        return true;
    }
    return false;
}

public boolean currentStatus(String x) {
    if(input1.equals(x)){
        return true;
    }
    return false;
}
```

Figure 2

```java
public void serverRecieve() {
    String[] serverInput = input1.split("\n");
    serverType = serverInput[0];
    serverID = Integer.parseInt(serverInput[1]);
    serverState = Integer.parseInt(serverInput[2]);
    serverTime = Integer.parseInt(serverInput[3]);
    serverCpuCores = Integer.parseInt(serverInput[4]);
    serverMemory = Integer.parseInt(serverInput[5]);
    serverDisk = Integer.parseInt(serverInput[6]);
}
```

Figure 3

```java
while (!newStatus("NONE")) {
    if(currentStatus("OK")) {
        sendToServer("REDY");
    }else if(input1.startsWith("JCPL")) {
        sendToServer("REDY");
    } else if (input1.startsWith("JOBN")) {

        String[] jobInput = input1.split("\\s+");
        jobSub = Integer.parseInt(jobInput[1]);
        jobID = Integer.parseInt(jobInput[2]);
        jobTime = Integer.parseInt(jobInput[3]);
        jobCpuCores = Integer.parseInt(jobInput[4]);
        jobMemory = Integer.parseInt(jobInput[5]);
        jobDisk = Integer.parseInt(jobInput[6]);

        if(jobCpuCores == CORE2 && jobMemory < MEM2 && jobDisk < DISK2) {
            sendToServer("SCHD"+ " " + jobID + SERVER2);
        } else if (jobCpuCores == CORE3 && jobMemory < MEM3 && jobDisk < DISK3) {
            sendToServer("SCHD"+ " " + jobID + SERVER3);
```

My algorithm uses multiple different data structures to achieve the goals. I first started by initalising all the variables and java utilities. I then made the constructors for reading the files and opening the connection. I then created functions for communicating to and from the server and client (Figure 1). I also created two functions that parses the job and server information (Figure 2) and splits it so I can compare the relevant data. readSysInfo2() reads all the server information from the ds-system.xml and stores in in a Linked list. Finally my Client() function (Figure 3) goes through the authentication process and compares the jobs with the servers. Finally a job is scheduled when a match is found.

Evaluation

Simulation setup:

- Compile Client
- Run ./test_results “java Client” -o co -n -c other (Figure 2.1, Figure 2.2)

Total rental cost

Config	ATL	FF	BF	WF	Yours
config100-long-high.xml	620.01	776.34	784.3	886.06	602.55
config100-long-low.xml	324.81	724.66	713.42	882.02	309.51
config100-long-med.xml	625.5	1095.22	1099.21	1097.78	607.41
config100-med-high.xml	319.7	373.0	371.74	410.09	299.02
config100-med-low.xml	295.86	810.53	778.18	815.88	281.04
config100-med-med.xml	308.7	493.64	510.13	498.65	292.95
config100-short-high.xml	228.75	213.1	210.25	245.96	209.87
config100-short-low.xml	225.85	498.18	474.11	533.92	190.33
config100-short-med.xml	228.07	275.9	272.29	310.88	197.56
config20-long-high.xml	254.81	306.43	307.37	351.72	252.59
config20-long-low.xml	88.06	208.94	211.23	203.32	99.85
config20-long-med.xml	167.04	281.35	283.34	250.3	170.58
config20-med-high.xml	255.58	299.93	297.11	342.98	251.76
config20-med-low.xml	86.62	232.07	232.08	210.08	96.29
config20-med-med.xml	164.01	295.13	276.4	267.84	158.07
config20-short-high.xml	163.69	168.7	168.0	203.66	158.7
config20-short-low.xml	85.52	214.16	212.71	231.67	97.21
config20-short-med.xml	166.24	254.85	257.62	231.69	161.29
Average	**256.05**	**417.90**	**414.42**	**443.03**	**246.48**
Normalised (FF)	0.6127	1.0000	0.9917	1.0601	0.5898
Normalised (BF)	0.6178	1.0084	1.0000	1.0690	0.5948
Normalised (WF)	0.5779	0.9433	0.9354	1.0000	0.5563

Improvement: 42.02%

Final results:
2.1: 1/1
2.2: 0/1
2.3: 0/1
2.4: 6/6

Figure 2.1

Resource utilisation

Config	ATL	FF	BF	WF	Yours
config100-long-high.xml	100.0	83.58	79.03	80.99	100.0
config100-long-low.xml	100.0	50.47	47.52	76.88	100.0
config100-long-med.xml	100.0	62.86	60.25	77.45	100.0
config100-med-high.xml	100.0	83.88	80.64	89.53	100.0
co l	100.0	40.14	38.35	76.37	98.9
co l	100.0	65.69	61.75	81.74	100.0
co .xml	100.0	87.78	85.7	94.69	100.0
co xml	100.0	35.46	37.88	75.65	93.72
co xml	100.0	67.78	66.72	78.12	100.0
config20-long-high.xml	100.0	91.0	88.97	66.89	99.52
config20-long-low.xml	100.0	55.78	56.72	69.98	82.41
config20-long-med.xml	100.0	75.4	73.11	78.18	90.78
config20-med-high.xml	100.0	88.91	86.63	62.53	94.98
config20-med-low.xml	100.0	46.99	46.3	57.27	83.01
config20-med-med.xml	100.0	68.91	66.64	65.38	92.03
config20-short-high.xml	100.0	89.53	87.6	61.97	97.65
config20-short-low.xml	100.0	38.77	38.57	52.52	80.68
config20-short-med.xml	100.0	69.26	66.58	65.21	84.14
Average	**100.00**	**66.79**	**64.94**	**72.85**	**94.32**
Normalised (FF)	1.4973	1.0000	0.9724	1.0908	1.4123
Normalised (BF)	1.5398	1.0284	1.0000	1.1218	1.4524
Normalised (WF)	1.3726	0.9168	0.8914	1.0000	1.2947

Improvement: 38.32%

Figure 2.2

Comparisons:

As seen in Figure 2.1 my algorithm outperforms all 3 baseline algorithms by having a lower total rental cost. This resulted in an improvement of 42.02%. Also in Figure 2.2 it can be seen that my average resource utilisation outperforms the 3 baseline algorithms by 38.32%.

When running my algorithm with just the normal configuration files (e.g. ds-config-01.xml) it also shows a noticeable reduction in total rental cost at the expense of higher turnaround time.

Pro's:

- Low cost
- High utilisation
- Code runs quickly with no errors

Con's:

- High turnaround time
- Custom capability selection instead of using GETS

Conclusion: Overall, my algorithm achieves the problem statement and optimises 2 performance metrics (total rental cost & maximising utilisation). However, this was done at the cost of a greatly increased turnaround time which may stop individuals from using this scheduling algorithm when a more balanced algorithm such as best-fit is available. I found it interesting that by starting up fewer servers it greatly reduced the overall cost which is why all-to-largest is sometimes cheaper than the other algorithms.

References

GitHub Repository: https://github.com/maxiebaddie/Comp3100Assignment2