Optimisation and Constraint Programming

World Problems, Problem 4: Load Distribution

Name - Jack Murley, Zach Nicoll

Student ID - 218294029, 218274079

Table of Contents

1. Introduction

For the high distinction task in SIT316 (Optimisation and Constraint Programming), we decided to tackle problem 4. Problem 4 is best described as a load balancing problem. Load Balancing is currently a real-world problem that many companies face on a daily basis and is most commonly handed off to companies who specialise in either cluster or server farm operations as at times it can be significantly complex when you have 100+ devices to distribute over and millions of operations to compute.

1.1. What Exactly is Load Balancing?

Load balancing can be best described as the methodical and efficient distribution of tasks/jobs over a set of specified resources which in our case are computing units/nodes. The overall aim in load balancing is making the overall processing of data/jobs more efficient in terms of time taken to compute all data/jobs.

The main benefit behind load balancing algorithms/techniques is that they can optimize the response time for each task while at the same time avoiding unevenly overloading nodes when others are idle and at times can increase efficiency by up to 100% when the original solution was sequential [3].

Due to this problem having such a real-world application that many companies face day to day; this is not only an optimisation problem but one that is more concerned in parallel computing and the distribution of work across multiple server farms/clusters.

1.2. What Are the Main Techniques in Load Balancing?

When tackling the problem of load balancing, we have two main approach's which exist in a real-world scenario:

1. Static Algorithms, which don't take into account different clusters and are best suited to be used when you only have a single server or cluster in use
2. Dynamic Algorithms, which is the optimal route to take when you have multiple servers or clusters because it takes into account different clusters and are generally more efficient but require the exchange of information between different computing units. Due to this, they can be at the loss of efficiency especially when the communication delay is large

Due to these two distinct paths, many companies choose to use dynamic algorithms when tackling load balancing as they are often distributing the work across multiple nodes, often in different machines. However, many smaller companies implement static algorithms because they can only afford a single cluster.

2. Problem Formulation

As described above, the problem we face is load balancing and due to the nature of load balancing it's come as a minimisation problem to minimise the maximum time taken to process all data. However due to the fact that there are also a variety of constraints that come with load balancing such as a machine/node can only process a single task/job at once. Due to knowing this, we now know that we have a constraint optimization programming problem to solve which can be solved through a variety of different heuristic and metaheuristic methods that come under the umbrella of static scheduling techniques. Through viewing this problem as a constraint optimization problem, it allows us to easily pivot our solution to also include a priority queue on which jobs to process first.

2.1. Ideal Load Balancing Problem

When reviewing the problem more closely, we can see that it is rather an ideal load balancing problem. This is because we know exactly how long each task tasks and this often isn't the case in the real world. Furthermore, we also know that we are only dealing with a single cluster/server and as a result a static load balancing algorithm will work well because of this specific scenario.

Continuing on, the problem itself is relatively simple to solve and this is for a majority of reasons. The first being that there is no priority in which the jobs need to be completed and as a result it allows us to implement a much simpler solution than one which has priority ratings on specific jobs as this greatly reduces the number of constraints we actually have on our problem.

Due to this problem being an ideal load balancing problem, a variety of different approaches could be applied to achieve an optimal runtime to complete all jobs. Nevertheless, we also need to take into consideration that due to the problem being rather straightforward, the complexity of this specific problem greatly increases as does the number of nodes and tasks, especially when the number of jobs greatly outweigh the number of nodes. For this reason, we need to ensure that the solution we design to solve the problem is not only highly scalable but also one that introduces additional constraints to reduce the computation time and reduce the likelihood of a brute force approach.

2.2. Required Constraints

As described above, we have very few constraints that are actually required on our problem. The only required constraints in our problem are the following:

1. A node can only process a single job at a time and the job has to finish before it can start processing another
2. All jobs need to be processed before the problem can be considered completed
3. A job once started has to be completed and cannot be stopped half-way through and offloaded to a different node

Due to these being the only required constraints in our problem, it allows us to experiment with a variety of different models and algorithms as almost any technique could be applied to find an optimal time. Because of this exact reason, we will be looking for the most scalable solution as well as optimal when evaluating the performance of different approaches.

3. Analysis of the Problem

When analysing the problem more closely in relation to the sample data provided, we have to assume that there will always be more jobs to compute than nodes and when this isn't true, we should apply a different method to reduce the computation time to find the optimal solution. Furthermore, as described above, the approach that should be taken should be in relation to either a heuristic or metaheuristic algorithm that falls under static load balancing.

3.1. Static Scheduling Techniques

As mentioned above, this problem can be solved by a variety of different techniques however this specific problem has a direct relation to static load balancing techniques. Within static load balancing techniques there are two main approaches that are taken [2]:

1. Opportunistic Load Balancing (OLD)
2. Minimum Completion Time (MCT)

For our problem we will be mainly considering MCT approaches rather than OLD due to MCT generally outperforming OLD. Nevertheless, when MCT is expanded we can see that it encompasses a variety of different techniques as seen below:

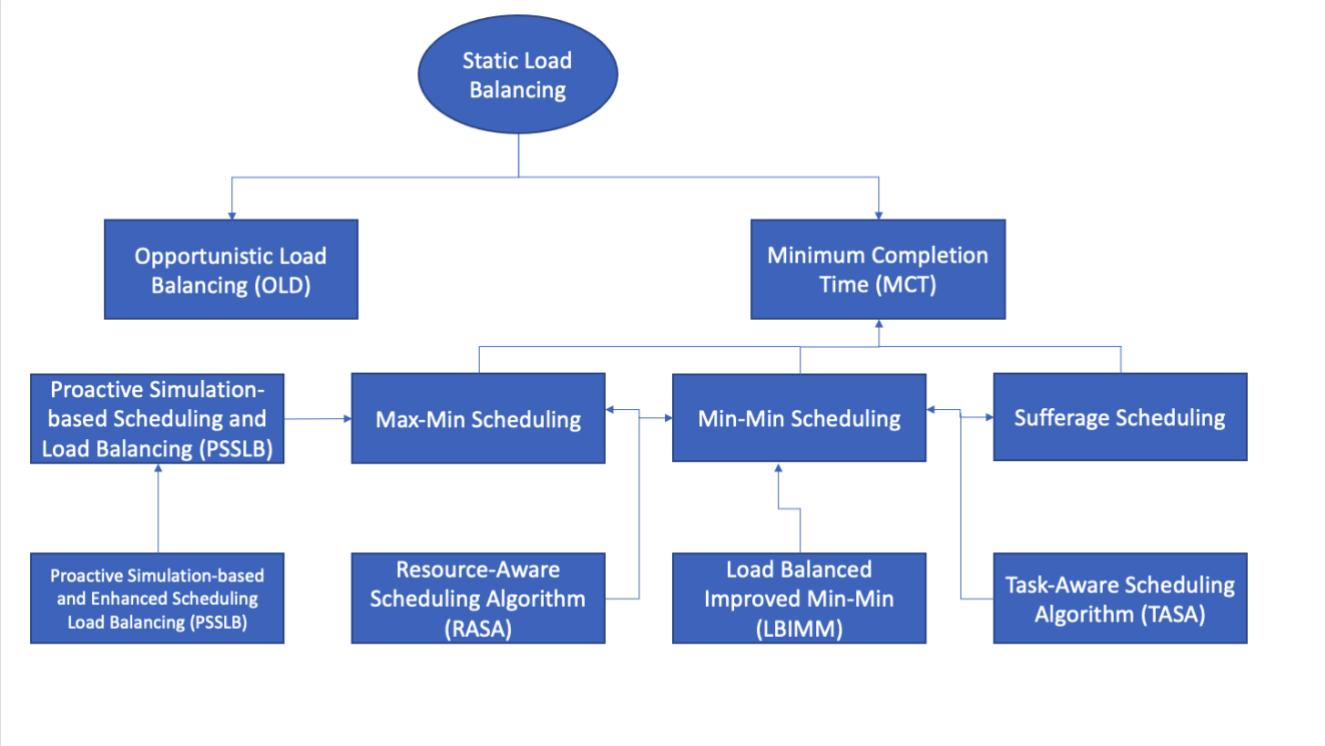

Figure 1: Minimum Completion Time Techniques

Due to their being a variety of different techniques that use different approaches, their individual runtime/analysis is unique to that specific algorithm and as a result this is described in section 4, whereas this section will focus on a more generic runtime analysis.

3.2. Generic Worst Possible Runtime Analysis

When looking a generic worst possible runtime analysis there are two different scenarios that can occur. These scenarios are:

1. Where only a single node is used to compute all jobs
2. Where all nodes are used to compute all jobs

3.2.1 Generic Worst Possible Runtime, Single Node

When looking at the worst possible Runtime for a single node we can see that it will be rather straightforward. The worst possible runtime for a single node can be calculated by first calculating the sum of all jobs assigned to a single node and then selecting the highest sum from all nodes. In terms of the sample data provided, we can see that it will be 386 operators on Node 6.

3.2.2 Generic Worst Possible Runtime, Multiple Nodes

When looking at the worst possible runtime for multiple nodes it becomes rather more complex. Assuming that either the worst or best possible solution path has been followed, the worst possible runtime should exist in the jobs left over that could not be evenly divided among nodes. This subsequently then means that the worst runtime should exist on the nodes who have to pick up the extra jobs as they are assigned an extra job.

In terms of this in relation to the sample data provided, 4 nodes should technically have the possibility to hold the worst possible runtime. However, this can drastically change based upon the data in each job as well as the distribution over each job in terms of how long it takes to calculate the job.

This specific worst possible runtime analysis is more prevalent where the computation between each node for jobs is relatively close and where jobs take extended period of times to calculate. Whereas when the computation between nodes is not relatively close, the worst possible runtime will exist in the node who has the lowest total time to compute all of their assigned jobs, but this only seen in algorithms where all nodes are considered at every step of the process.

Once again, it needs to be heavily emphasised that the worst possible runtime is heavily dependent upon the method applied to the data as well as the data itself as shown in section 5.

3.3. Generic Best Possible Runtime Analysis

When the examining the generic best possible runtime analysis, just like the worst possible runtime analysis we can break it up into two sections, however we won't be explaining the generic best possible runtime on a single node as it is just the reverse of the worst possible runtime on a single node.

3.3.1 Generic Best Possible Runtime, Multiples Nodes

When taking an in-depth look at the generic best possible runtime for multiple nodes, once again it will be heavily dependent on the method used as well as the data. However, in an ideal scenario it would be the minimum time to complete all jobs that can be evenly divided between the nodes plus the job which will take the longest to complete onto the most available node with the lowest current operations.

3.4. Possible Edge Cases Analysis

When looking at possible edge cases for our problem and how they can affect our results, their remains one direct edge case. This edge case is where a single or multiple nodes are never used because the amount of time they will take to complete the job will only result in a poorer runtime. This is a very interesting edge case as this is seen quite commonly in real world when there is a large gap between processing units in terms of power.

Outside of the aforementioned edge case, a variety of different edge cases can come up such as negative values for operations, string values and null values for jobs but this can all be prevented through exception handling; however, one other main edge case still remains.

This specific edge case is a single job taking significantly longer than any other job. For example, the sum of all jobs except one could be 253 operations to complete whereas this single job could take over 1000 operations to complete. This can also be seen in a scenario where the data is split up between some jobs taking very small operations to complete such as 3 and others taking 100's.

While this edge case can easily be overcome by prioritising jobs who take significantly longer than others, it is still an edge case which needs to be heavily investigated and avoided as it could drastically increase the runtime to complete all jobs if not appropriately dealt with.

4. Discussion of Various Approaches to Solve the Problem

When looking at the variety of different approaches to solve this problem, due to it being such a widely known problem in the ICT industry there exists a variety of different solutions to solve this problem that range from population based heuristic methods to simple mathematical solutions such as the round robin load balancing algorithm. Inside this section we will not only explain a variety of different approaches but also their strengths, weaknesses and time complexity.

4.1. Round-Robin Load Balancing Algorithm

The round-robin load balancing algorithm uses the round-robin scheme for allocating jobs. It works by selecting the first node randomly and then allocates jobs to all other nodes in a round robin fashion. Nodes are assigned each job in a circular order without any sort of priority and hence there is no starvation. This algorithm serves as an advantage because of the fast response in the case of equal workload distribution amongst processes/nodes [1].

However, different processes have different processing times and as a result at any point in time some nodes may be heavily loaded while others remain idle and under-utilized. Because of this significant draw back, this is not the ideal algorithm to use for our problem as it is highly unlikely to yield an optimal time.

4.2. Weighted Round-Robin Load Balancing Algorithm

Weighted round-robin was initially developed to improve the critical issues with the round-robin algorithm as described above such as some nodes being heavily loaded while others remain idle.

Within the weighted round robin algorithm, each node is assigned a weight and according to the values of the weights, jobs are distributed. Nodes with greater capacities are assigned a larger value. Hence the highest weighted nodes will receive more jobs and in scenario's where all weights become equal, the server received balanced jobs.

This specific algorithm is a very easy way to implement a solution in relation to the current problem we have, this is because we know how long each job takes on each node and because of this we can then rank the nodes and assign jobs accordingly. However, a downfall to this method, is while it is likely to receive better results than the standard round robin algorthim for calculating all processes, it is highly unlikely that it will achieve the best possible time due to possibly overloading either a specific or multiple nodes [1].

4.3. Opportunistic Load Balancing Algorithm

Opportunistic load balancing (OLB) focuses on keeping each node busy. However, because of this it does not consider the present workload of each computer. OLB dispatches unexecuted tasks to currently available nodes in a random order regardless of the node's current workload. Since OLD does not calculate or take into consideration the execution time of the node, the task to be processed will be processed in a solver manner, resulting in bottlenecks despite some of the nodes being free. Due to this very reason, we won't be implementing this algorithm as it will result in a poor makespan because it is not resource aware [1] [2].

4.4. Min-Min Load Balancing Algorithm

Min-min load balancing works by first finding the minimum completion time for all tasks and then among these minimum times, the minimum value is selected which is the minimum time amongst all tasks on any resource. According to that minimum time, the task is then scheduled on the corresponding node. The execution time for all other tasks is updated and then that task is removed from the list.

The procedure is then followed until all tasks are assigned a specified node to run on. In scenarios where the number of small tasks is more than the number of large tasks, this algorithm achieves better performance. However, this approach can lead to starvation which is allowed as long as it still ensures we achieve the most optimal computation time [1] [2].

4.5. Max-Min Load Balancing Algorithm

Max-min load balancing is similar to the min-min algorithm with the following exception: after finding out the minimum execution times, the maximum value is selected which is the maximum time amongst all jobs on the resources. Then according to the maximum time, the task is scheduled on the node. The execution time for all other tasks is then updated and the assigned job is removed from the list of jobs that are to be assigned to nodes. Since the requirements are known beforehand, this algorithm is expected to perform well [1] [2].

4.6. Brute Force Algorithm

Within a brute force algorithm/approach, we calculate every single possibility to the problem and then select the lowest found time. While this method is guaranteed to find the best possible time, this is not a feasible solution as the size of jobs and nodes increase so does the computation time and as a result, we won't consider this approach.

4.7. Minimum Completion Time (MCT) Algorithm

The MCT technique assigns a candidate job to a node that consumes the minimum time for the job. The MCT heuristic examines the current load of the nodes to find a suitable node for the job assignment. At each scheduling step, the MCT heuristic has to scan all the available nodes to find the most appropriate computing resource (i.e., the node producing the minimum completion time for the job) [2].

However due to the expensive search mechanism employed by this algorithm, it causes a significant overhead which could impact scalability.

4.8. Sufferage Algorithm

The sufferage scheduling algorithm calculates the sufferage value for each job. To calculate the sufferage value (i.e., a penalty in terms of longer execution time), the minimum completion time and the second-best minimum time producing nodes are determined for each job (in each iteration).

Afterward, the job experiencing the highest sufferage value is assigned to the node which produces the minimum competition time for that job. The Sufferage heuristic produces good results often with a reduced makespan; however, this scheduling mechanism causes higher overhead due to the calculation of the sufferage value for each job as compared to MCT, Min-Min, OLB and Max-Min [2].

4.9. Resource Aware Scheduling Algorithm (RASA)

The RASA technique uses both Min-Min and Max-Min heuristics in the alternate scheduling decisions until all jobs are scheduled. RASA exploits the merits of both Min-Min and Max-Min to evade the corresponding limitations of these two scheduling algorithms.

Most generally RASA will result in a lower makespan when it considers smaller and larger jobs in its alternate scheduling steps. However, a downside with RASA is that it penalizes smaller sized jobs when there are more larger jobs in the workload [2].

4.10. Task Aware Scheduling Algorithm (TASA)

TASA Favours smaller sized jobs in its first scheduling step which is based on the Min-Min heuristic and then it finds an appropriate node for the job using the sufferage heuristic in the second scheduling step. In most cases, TASA will produce a better makespan as compared to other scheduling heuristics such as Min-Min, Max-Min and OLD [2].

4.11. Load Balanced Improved Min-Min (LBIMM) Algorithm

The LBIMM algorithms works by assigning jobs to nodes using the Min-Min scheduling technique in its first phase then in the subsequent phase, LBIMM finds the smallest job on the most loaded node and determines its completion time on the other nodes.

After that, the minimum completion time of that job is compared with its makespan. If this value is less than the makespan, the job is assigned to a new node and the ready time of both nodes are modified. This procedure is repeated for the next smallest job on the most loaded node as well.

This process is repeated until there is no other node that can produce the minimum completion time for the smallest job on the heavily loaded node than the makespan on another node. This technique shares a load of heavy nodes with the idle or lighter nodes. LBIMM produces a better makespan and load balancing than the Min-Min heuristic [2].

4.12. Proactive Simulation-Based Scheduling and Load Balancing (PSSLB) Algorithm

The PSSLB and PSSELB algorithms are both proposed to assign large-sized jobs to the nodes that can execute them faster than the other nodes. PSSLB finds the matrix (i.e, each row has a completion time of a specified job on all nodes) of completion time of each job on each node.

The matrix is then sorted in a way that the last column stores minimum completion time for each job. Therefore, the longest job on the last column is selected and assigned to the node producing the minimum completion time for it [2].

4.13. Proactive Simulation-Based Scheduling and Enhanced Load Balancing (PSSELB) Algorithm

The PSSELB Algorithm is the modified version of the PSSLB that produces a load balanced schedule. The largest job among the unallocated jobs is assigned to the node using PSSLB, and the completion time of this job is considered as a pivot.

After that, the jobs that produce the completion time (on other nodes) equal to or are less than the pivot are iteratively determined and assigned to the concerned nodes (i.e., producing MCT for a job equal to or less than the pivot value).

Next, the largest job is assigned to the concerned node using PSSLB, and the pivot is updated with the completion time of the largest job. Again, the jobs with MCT on other nodes (i.e, except the node with the last largest job assigned) that is equal to or less than the pivot are determined and assigned to the concerned node.

This scheduling procedure is repeated till all the unallocated jobs are assigned to nodes in the same way [2].

4.14. Strengths, Weaknesses and Time-Complexity of Aforementioned Methods

Assuming N number of jobs to be scheduled on M nodes, Table 1 presents the summary of the strengths, weaknesses and time complexity of the aforementioned methods.

Method	**Strengths**	**Weaknesses**	**Complexity**
OLB and Round Robin	Low Complexity, Minimal overhead and keeps nodes busy	Load-imbalance and no-fairness in scheduling	O(M)
Weighted Round Robin	Improved makespan than round robin and low complexity	Load imbalance, overloads faster nodes	O(M*N)
MCT	Improved makespan than OLD, node aware scheduling	Load imbalance and overloads faster nodes	O(M*N)
MinMin	Favors smaller jobs, Reduced makespan for smaller jobs	Overloads faster nodes with smaller jobs, Penalizes larger jobs	O(M*(N*N))
LBIMM	Improved makespan than Min-Min, improved resource utilization than Min-Min	A few smaller jobs are penalized while rescheduled	O(M*(N*N))
MaxMin	Favours larger jobs, reduced makespan for larger jobs	Penalizes smaller jobs, Load-imbalance for job pool with more larger jobs	O(M*(N*N))
RASA	Fair treatment of larger and smaller jobs	Penalizes smaller jobs in dataset with more larger jobs	O(M*(N*N))
Sufferage	Improved makespan than MCT, Min-Min and Max-Min, Job allocation to appropriate node	High Scheduling overhead due to sufferage value calculation	O(M*(N*N))
Brute Force Approach	Guaranteed to find the most optimal solution	Time complexity and infeasible	O(M*(N squared N))

TASA	Improved makespan than Max-Min, Min-Min and RASA, favors smaller jobs	Load Balancing is not considered	O(M*(N*N))
PSSLB	Reduces completion and response time for larger jobs	Penalizes smaller jobs	O(M*(N*N))
PSSELB	Improved makespan than PSSLB	Results in load imbalance compared to PSSLB	O((N*N)/2)

Table 1: Summary of Explained Scheduling Algorithms [2]

5. Solutions

From the algorithms discussed previously, a select few were chosen to find a solution to the given load balancing problem. The chosen algorithms were weighted round robin, min-min load balancing (simple, complex and reverse complex) and max-min load balancing. The reasoning behind selecting these algorithms were primarily focused around what we would be able to develop in the current time period as well as because they are the building blocks for more complex algorithms. It's also key to note that for our project we built a framework in C# that you can use to experiment with different algorithms and methods.

5.1 Weighted Round Robin Solution

Out of all the implemented solutions the weighted round robin gave us the worst results at 70 for the sample data provided. This algorithm takes each node and gives it a weight and this weight is used to select the best node for each job. This weight is found by finding the GCD (Greatest common divisor) of the first two nodes and then finding the GCD of the rest of the nodes and the GCD of the first two nodes which was found earlier. This gives us the weights for each of the nodes and they are then ordered highest to lowest. Due to how the weight is calculated and how many nodes are in the dataset the weight for each node ends up as score out of 200.

To then calculate the runtime the algorithm selects the lowest weight to the selected node to do the job. The main issue with this is as it goes through each node in a linear fashion if two nodes have the same operation time the one iterated through later will get the lower weight and be selected. This is a big problem as the dataset used has its most selected node share the same operation time as other nodes on many jobs and this means that nodes with little to no load with the same operation time our overlooked as they are earlier in the list of nodes. This led to node 5 being overloaded in this case and the whole operation time ballooning to 70 when it could have been much lower.

Due to the poor performance of this algorithm, it was decided that we would not make it scalable due our current development timeframe.

5.2 Min-Min Load Balance Solutions

When developing the Min-Min algorithm, we went through a variety of different implementations to not only experiment with how it reacted to the data but also to see if we could improve it. Currently we have 3 different Min-Min algorithms developed.

5.2.1 Basic Min-Min Algorithm

Inside our basic min-min algorithm it is purely focused around selecting the minimum value from the current job and then assigning it to the node with the smallest processing time for that specified job. While this algorithm is rather basic and has direct downfalls such as the high probability of overloading nodes and creating a large makespan, in specific scenarios it is able to find the most optimal solution as shown below in figure 2 while having a low time complexity.

5.2.2 Complex Min-Min Algorithm

Inside our complex min-min algorithm is the academically accepted version of the min-min algorithm as described above in section 4.4. This algorithm/function works by first receiving the 2D array of data as well as how many jobs exist in the data. We then loop through the 2D array and find the index location of the minimum value for each job and then add it to its own specified list.

From there, we then create another list which holds in the index location of each job from their minimum values in relation to the highest minimum values for each job. We now then actually perform the min-min algorithm, we first loop through every job and then every node and then an array which holds the total run time currently assigned to each node.

Inside this specific for loop on the total run time for each node, we then check to see if the current job + current run time will be less than the current job total. We then jump outside of this loop and at the end of every loop for the job, we then assign that lowest value to the node in the run time currently assigned array.

As you can see below in Figure 2, the min-min algorithm was able to tie the best-found result for the sample data found but was ultimately the 2nd worst algorithm in comparison to the 4 that we made and tested.

5.2.3 Reverse Complex Min-Min Algorithm

Our reverse complex min-min algorithm implementation is the exact same as the complex min-min algorithm with the only change being that instead of starting from the smallest index of all jobs and going from small to large, it instead goes from large too small.

This is an interesting implementation, as such a small change such as reversing the order you assign jobs is able to have such drastic change in the results received as seen specifically in the tests surrounding the supplied data, min node as well as the Fibonacci sequence

5.3 Max-Min Algorithm

Our max-min load algorithm implementation is the academically accepted version of the max-min algorithm as described above in section 4.5. It follows the exact same implementation of the min-min algorithm with the exception of finding the maximum values for each job and looping their index's first rather than the minimum. Once again, this is a very interesting algorithm that performs quite well however there are specific scenarios as shown in figure 2 where it falls behind other implementations as well as its main downfalls as described in table 1.

5.4 Other Areas the Algorithms Could Be Used

Due to these algorithms being developed around a generic scheduling problem, the uses for these algorithms are not only limited to load balancing problems. Rather these algorithms can be used anywhere you need to find an optimal time to complete all jobs/tasks over a set number of processors. One direct application this can be applied to is task allocation for staff at work, where the job is a job that needs to be completed in the office and you can select from a variety of staff and you approximately know how long it will take each staff to complete the task.

Through using this method, not only will you be able to complete all jobs in an optimal time, but also ensure that you don't overload specific staff members when using specific algorithms.

6. Results

For the final results, the performance of our 4 scalable implementations to solve this specific problem were tested in various ways including the initial test file given to be solved as well as some other tests to test the scalability and differences between each algorithm used (Round Robin not mentioned due to reasons as described above in section 5.1). Below is a graph showing the results from each test for our 4 scalable implementations.

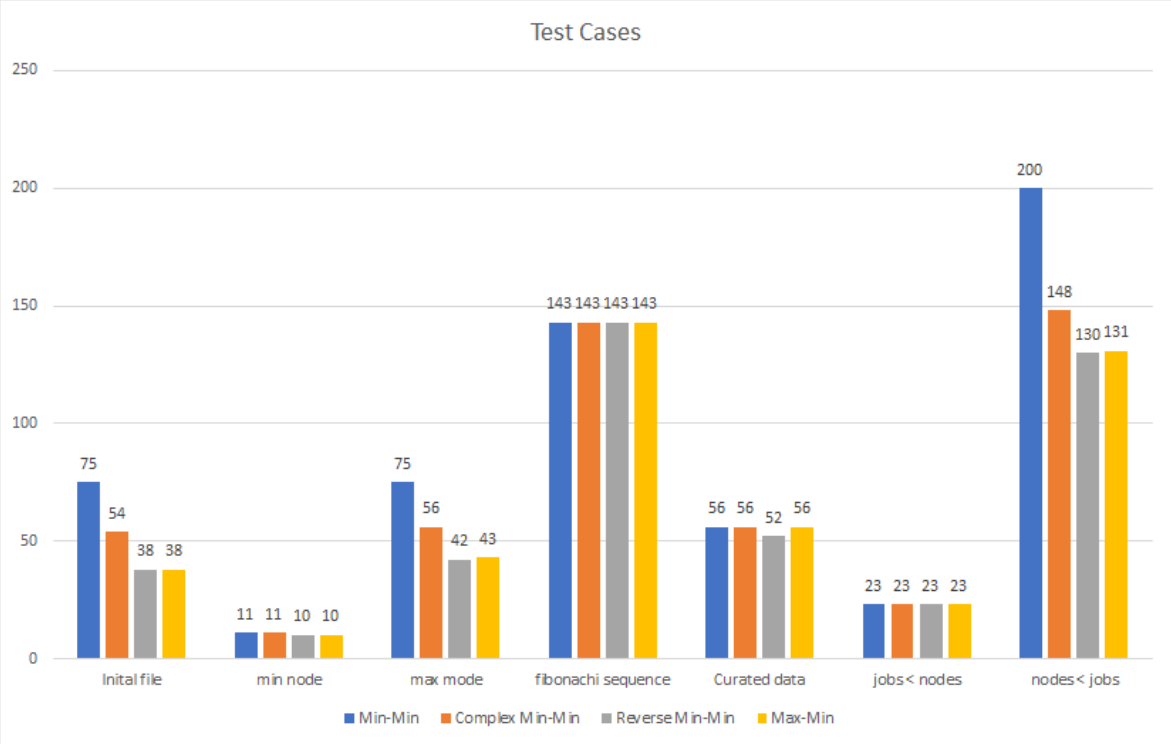

Figure 2: Test Case Results

From the above graph it can be seen that we performed 7 tests. These tests were:

1. The initial code file supplied (initial file)
2. A test where we set a nodes result all to 1 (min Node)
3. A test where we set a single node to drastically high operational numbers (max node)
4. A test where we recreated the Fibonacci sequence in the test data (Fibonacci sequence)
5. A test where we curated a dataset with a best-known time of 52 (curated data)
6. A test where there were significantly more nodes than jobs (jobs < nodes)
7. A test where there were significantly more jobs than nodes (nodes < jobs)

6.1 Results Discussion

When reviewing the results for the initial code file supplied, it can clearly be seen that the Basic min-min and complex min-min gave poor results when compared to our other two methods which were able to both provide the most optimal solution which we were able to discover through our implementation.

However, during our next test which tested the min node, we were able to see that the reverse min-min method was able to find the most optimal solution whereas the other 3 implementations were unable to. This was particularly interesting, especially in examining each of the implementations and how they react and perform when posed with drastically different data. This was only exacerbated during our next test max-min where we were able to see the same pattern in results with the reverse min-min algorithm finding that 1 operation advantage.

While a single operation advantage might not seem that impressive, when put in the perspective that each operation could take a day to compute, that is essentially one less day that you need wait to compute all data and that is crucial in environments where you need that data as soon as possible such as processing COVID-19 models.

For the Fibonacci sequence the node times increased in a Fibonacci sequence and the next node would continue where the last left off. Due to this, the first node would always be the best solution (only 3 nodes were tested) and all algorithms found this to be the best solution as well. This could then help reduce runtime complexity, because if you know what format your data is in, you could potentially then opt for an algorithm which has a better time complexity and ultimately know the most optimal pattern sooner.

Whereas for the curated data the best possible result was 52 and we deliberately designed this data to ensure that no algorithm found the most optimal solution however this was not the case during testing. This was a simple 3 job 3 node problem and from the results we can see the best algorithm was the Reverse Min-Min as it was the only one to get the correct answer while the other algorithms had an answer of 56.

In the test case where we had more nodes than jobs, all algorithms found the same answer as it is easier to brute force and figure out just using minimum values if there is a very small amount of jobs.

Whereas in the test case where we have more jobs than nodes, we can see a massive disparity in results as there are many jobs that need to be juggled between the limited nodes and this means that the amount of times all nodes are doing a job is the best way to optimise the result and not just finding minimum value and the results reflect this. The results show that the Min-Min and complex Min-Min have the higher operation times with 200 and 148 respectively, the best results were from the reverse Min-Min and Max-Min with 130 and 131.

From the results, the overall best performing algorithm is the reverse Min-Min with the Max-Min closely behind and only ever generally 1 operation slower and as a result further testing needs to be done on a variety of different types of data to find where the reverse min-min algorithm falls short.

6.2 Future Improvements/Extensions

When looking into future improvements and extensions that we could perform on this project, there is a variety of different options that we could do. The most obvious future improvement and extension is further optimising and testing the reverse min-min algorithm and classifying which type of data it best performs in. Outside of further investigating the reverse min-min algorithm, a direct improvement to potentially find more optimal times would be to implement and test more advanced methods as described above in table 1 such as the PSSELB.

7. Final Discussion

Ultimately from our investigation into problem 4: Load Distribution, we were not only able to decipher that the problem was directly focused around load balancing but also develop a framework which you can test load balancing algorithms on easily. Furthermore, through our investigations we were not only able to research and explain a variety of different methods that range from heuristic to multi-state heuristics that you could use to solve our specified problem but also how they could be used on other scheduling problems as well.

Through our investigation, we were able to implement 5 different methods to solve our specified problem and make 4 of those methods completely scalable regardless of node or job size. The first of these methods being Weighted Round Robin which we did not implement scalability for, in order to focus more on the other four algorithms.

These methods were tested with the supplied load distribution data set given and our custom test data and it was found that the Basic Min-Min and Weighted round robin had the worst performance at 75 and 70 respectively in the supplied data and the Complex min-min was an improvement at 54 and lastly the best algorithm was the Reverse Complex min-min and max-min with runtimes of 38. However, we then tested our four scalable algorithms to see how they not only reacted to different types of data but to see how we could separate the Reverse Complex min-min algorithm and the max-min algorithm.

Our further investigation into our scalable algorithms went through a series of 5 additional tests which pushed the algorithms to their limits such as having a node with all 1 processing time or all 10,000 processing time in order to test how the different algorithms reacted in different and unique situations. Through our additional testing we were able to find out that the best overall algorithm not only for the data provided but also over our additional tests was the reverse complex min-min algorithm which always gave the most optimal time without fail in all test cases.

Due to the reverse min-min algorithm always providing the most optimal time without fail in all test cases, more testing needs to be done into this specific algorithm we developed in comparison to more complex methods such as PSSELB and TASA and as a result this is subsequently the future work to be done on the project.

Throughout the duration of the project we not only implemented a variety of different already well known methods which could solve the problem with the supplied data in an optimal time frame but also develop our own custom methods which ultimately gave us the most optimal time out of all other methods implemented over additional test cases.

8. References

[1]. Shah N, Farik M. Static Load Balancing Algorithms in Cloud Computing: Challenges & Solutions. National Journel of Scientific and Technology Research [Internet]. 2015 [cited 6 October 2020];4(10):365-356. Available from: http://www.ijstr.org/final-print/oct2015/-Static-Load-Balancing-Algorithms-In-Cloud-Computing-Challenges-Solutions.pdf

[2]. Hussain A, Aleem M, Iqbal A, Islam A. Investigation of Cloud Scheduling Algorithms for Resource Utilisation using CloudSim. Computing and Informations [Internet]. 2019 [cited 9 October 2020];38:525-545. Available from: http://www.cai.sk/ojs/index.php/cai/article/viewFile/2019_3_525/963

[3]. How Does Load Balancing Work? [Internet]. Docs.vdms.com. [cited 12 October 2020]. Available from: https://docs.vdms.com/cdn/Content/Route/Administration/LB_How_Does_It_Work.htm