Multiprocess Scheduling Application

Team 16 Stonks

Sean Spires
Software Engineering
The University of Auckland
Auckland, New Zealand
sspi841@aucklanduni.ac.nz

Ryan Lim
Software Engineering
The University of Auckland
Auckland, New Zealand
rlim083@aucklanduni.ac.nz

Tyger Kong
Software Engineering
The University of Auckland
Auckland, New Zealand
tkon583@aucklanduni.ac.nz

Mario Sinovcic
Software Engineering
The University of Auckland
Auckland, New Zealand
msin595@aucklanduni.ac.nz

Joshua Fu
Software Engineering
The University of Auckland
Auckland, New Zealand
jfu047@aucklanduni.ac.nz

***Abstract*—This report evaluates the development of the solution to the Parallel Scheduling Problem and overviews the effectiveness of the final product. Specifically, the algorithm, parallelisation, and user interface that Team 16 (Joshua Fu, Tyger Kong Ryan Lim, Mario Sinovcic, and Sean Spires) built via the waterfall development process. Utilisations of software practices and integration of various software architecture types will be discussed and analysed. The intended user base for the system includes the client's whose specifications the project was built from. As a result, the features and design decisions all take into consideration the specific needs of the clients. Implementation, design, and justification for each critical section of the project will be presented along with the corresponding evaluation and testing methods, as they were crucial to justifying the final product's functionality and fitness for purpose. Design choices were made to the best of the team's ability and focused heavily on efficiency and functionality while letting elements, like aesthetics, took a lower priority.**

***Index Terms*—mp-hard, task, scheduling, algorithm, branch-and-bound, A*, multithreading, parallelisation**

I. Introduction

This report covers the thinking and development process of consolidating a potential solution to the well-known computing conundrum, multiprocessor scheduling problem. More specifically, a waterfall process was implemented to ensure timely updates and efficient progression. A team of five penultimate software engineering students were tasked to explore prospective resolutions. Technology has been improving at a steady rate over the past century, and modern processors have become more capable and powerful than they have ever been. However, due to physical constraints, the advancement of processing unit development is reaching a plateau as processor speeds no longer increase along with Moore's Law. Utilising parallel computation is the future. Hence, the development team Stonks were encouraged to investigate the possibilities of using multiprocessor machines to solve the task scheduling problem. The problem lies within the option of scheduling a task in multiple time slots and possible processors while maintaining any dependency constraints from predecessor tasks. Henceforth, this report will discuss the problem statement, developmental process, and execution of this project in-depth, with the aid of graphical representations.

II. Development Process

A. The Waterfall Model

The overarching development process was the Waterfall method specified by the client. This development process was a linear, sequential approach to the development of the final product. To visualise the waterfall structure, Team Stonks used an online planning tool called PlanHammer. This tool was particularly useful as it allowed for quick collaboration during the initial planning process amongst teammates.

Crucial models that the team always referred back to during the development of the product was the Gantt chart and the Project Network diagram. These diagrams reminded the group of the plan, functional/temporal dependencies, and progression of each task. A useful aspect to using PlanHammer was the ability to set the progress of the tasks that were currently worked on, and this enabled each group member to quickly see if the team was on schedule for the next tasks.

The requirement of utilising the Waterfall developmental process posed as a challenge to the team as this is the first time the group embarked on a project that strictly follows the progression as mentioned earlier style. Therefore, initially, it was strenuous for the team to abide by the plan and carry it out. One aspect of the difficulty was the development of the visualisation of the solution. Usually, a team member would begin drawing up designs and plans of the user interface and work in parallel with the development of the algorithm, but we had to ensure that this did not occur and that we followed

the waterfall method. (Please see figure one for the diagram of work progression).

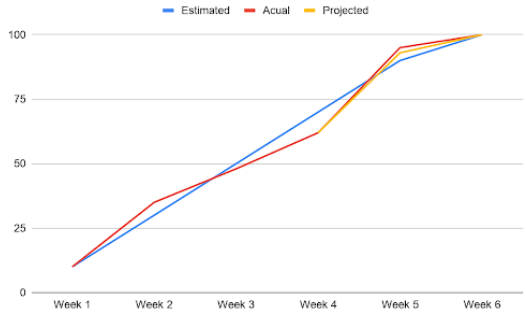

Fig. 1. General work progression diagram.

B. Communication and Decision Making

Communication within a team is paramount to the success of the project. Team Stonks felt that proper communication channels needed to be set up for the smooth running of the project. The main communication tool that the team used was Facebook Messenger. However, Messenger was only used for the planning of the meeting times, locations and small decision making as the team decided that large decisions should be made in-person so as to avoid any miscommunication that might arise from online messaging.

The team adopted a democratic stance when making decisions. No team member had more influence when it came to voting on a decision. Voting was only carried out when the team was split in views, and we felt that voting was most fair in carrying out decision making.

C. Conflict Resolutions

Throughout the development of the final product, the team ran into multiple issues that were amicably resolved. Being in a group of five conflicts were bound to happen, but being able to address these issues, cordially was what made the development process go smoothly.

As the team consisted of five good friends, each team member understood that any difference in opinion of design decisions was not taken personally. Team Stonk's group leader, Sean Spires, was always there to mediate conflict resolution and ensured that any conflict was to be resolved harmoniously. In cases where a quick debate could not resolve disputes, the outcome was decided upon a vote.

Overall, through the conflicts the team faced, working towards a common goal was essential for getting past conflicts/disputes. Additionally, having an attitude that respected individual team member's views and opinions allowed the team to effectively and efficiently resolve any conflicts faced.

D. Tools and Technologies

As multiple people were working on the project, a GitHub repository was created with all the team members as collaborators. Utilising the different branches of the repository, the team was able to work on various aspects of the project at the same time without interfering with each other. Only when these aspects were completed could they be merged and implemented into the master branch.

Using a repository is also advantageous in that it provides version control. If any part of the code was wrong for whatever reason, there was always the option to fall back on a previous iteration of the system. This feature was crucial when during a merge from the master branch into a development branch, an error occurred, and the repository had to be rebased to effectively what it was half a week ago. During the process, although much of the commit history was lost, the repository was able to have the local working iteration pushed onto it.

In terms of the external libraries, JavaFX was used for the visualisation. JavaFX provided a component system that enabled the creation of custom components such as the Gantt Chart and the Zoomable Scroll Pane and supported CSS styling. The end product was a user interface that was both simple to implement, and intuitive to use.

E. Work Culture

Team Stonks kept a high level of positivity and always encourage one another when the stressed started to set in. Each member was ready to lend a hand when pair programming activities were carried out and being each other's "Rubber Duck". Please refer to table one for the work contribution by each team member.

TABLE I
PERCENTAGES OF TASK CONTRIBUTIONS BY EACH MEMBER

Task	*Sean*	*Mario*	*Ryan*	*Josh*	*Tyger*
File I/O	-	20	35	10	35
Algorithm	30	-	10	50	10
Advanced Algorithm	50	50	-	-	-
Parallelisation	35	10	5	40	10
Visualisaiton	5	25	30	5	35
Documentation	10	15	35	10	30

F. Fit For Purpose

The solution provided by the development team is fit for purpose. The waterfall process and design specifications given by the client were strictly followed. This ensured the team from diverging their focus and prevented possibilities of scope-creep. Furthermore, rigorous testings ensured that each dependency is valid, preserved, and reliable.

III. Implementation

A. Background

The client of this project suggested that the team should explore the solution space via two methodologies, branch-and-bound and A* (pronounced A Star). The branch-and-bound algorithm is a design paradigm made for creating solutions for combinatorial optimisation problems such as the scheduling problem. The idea behind the algorithm is to exhaustively search through every possible permutation of schedules with the given tasks and pre-defined number of processors. Each initial partial schedule introduces multiple subsequent schedules which could derive from the parent schedule. These subsequent schedules should be explored as they may contain the desired optimal solution; hence, branching. However, blindly searching the entire problem space poses a problem of inefficiency when a noticeably non-optimal partial schedule is explored. Therefore, in addition to branching into potentially viable child schedules, the algorithm also keeps track of the current best solution and compares it before expanding a partial schedule. This additional layer of checking reduces the solution space significantly; which both improves memory usage and overall performance.

The A* algorithm is the amalgamation of several design iterations built on top of Edsger Dijkstra's best path algorithm with the incorporation of a heuristic calculation that suggests a general direction. (Cs.indstate.edu. 2019). The general idea behind the algorithm is that A* will traverse the problem space by calculating an estimation of the potential schedules it has explored, and go along a path which seems to be the most promising and viable combinations. However, the calculation of the heuristic could either drastically improve the algorithm's speed of finding the optimal schedule or waste time extending schedules which are unlikely to be optimal. Therefore, there are advantages and disadvantages to both algorithms. The team rigorously researched both algorithms and performed several team meetings to arrive at the end algorithm of choice. Furthermore, during meetings the team deemed it unnecessary to employ external libraries for the purposes of implementing the scheduling algorithm.

B. Algorithm

The final implementation of the algorithm has been dubbed "Branch-and-Bound-A*". The reasoning for this is the algorithm combines the work of Rahman's branch-and-bound algorithm, with one the key A* heuristics of Sinnen. The final algorithm creates the search space like a normal branch-and-bound algorithm, but traverses the space using heuristics akin to A*. The branch-and-bound logic of Rahman was mostly kept. However, Sinnen's heuristics were added in favour of Rahman's.

Rahman's pseudo-code (please refer to figure two) is a general branch-and-bound algorithm for finding optimal schedules. Being a general combinatorial optimisation

BEGIN

Step 1: (Initialization)

1.1. Set C_{ii} = cost of all the tasks for $i = 1$ to n
1.2. Set $CC_{ij} = (t_i, t_j) \in E$ for $i = 1$ to n *and* for $j = 1$ to n
1.3. $\mu(\alpha) = \{\}$
1.4. Create a root node α of the search tree Set $\alpha = (S_{\alpha 1}, \ldots, S_{\alpha m}) = [\ ]$
1.5. Set $T_\alpha = t_i$, i = 1….n, $k_{ai} = -1$, i = 1….n
1.6. add the root node α as open list in the first place of $\mu(\alpha)$
1.7. go to step 2.1

Step 2: (Branching from a node)

2.1. start branching
2.2. for each pair of (j, i) do the followings, where $j \in T_\alpha$, $i \in P = 1,\ldots m$
2.2.1. Create a child node β by copying all the information form the parent node α.
2.2.2. **If** all the predecessors of the task j is already been scheduled **Then**
2.2.2.1. The new schedule of β, can be obtain by appending a new task j at the tail of sequence of $S_{\beta i}$.
2.2.2.2. Compute the start time by following the rule of the communication cost of the task j for the processor i.
2.2.2.3. set the status of the task j, $k_{\beta j} = C_j +$ start time
2.2.2.4. compute the partial Make Span of the node β
2.2.2.5. compute LB_β
2.2.2.6. compute UB_β
2.2.2.7. **IF** $UB_\beta <$ BestUB **THEN**
2.2.2.7.1. set BestUB = UB_β
2.2.2.7.2. delete all the node form $\mu(\alpha)$ that has $LB >$ BestUB
2.2.2.8. **END** IF
2.2.2.9. **IF** $LB_\beta > UB_\beta$ **THEN**
2.2.2.9.1. delete the node β
2.2.2.10. ELSE
2.2.2.10.1. add the node as a open list in $\mu(\alpha)$

2.2 **END** of **FOR**
2.3 Select a node β from the open nodes list μ (α) for which the LB is the minimum among all the nodes in the list.
2.4 Set $\alpha = \beta$

2.5 **IF** $LB_\alpha = UB_\alpha$ **AND** $T_{\alpha=}[\ ]$ **THEN**
2.5.3 **STOP** the algorithm
2.6 **ELSE**
2.6.3 **Go to** step 2.2

END

Fig. 2. Branch and Bound pseudocode Rahman, M.M. (2009).

algorithm, Rahman's algorithm considers all possible solutions to find an optimal schedule. However, the algorithm, using lower and upper bounds, discards subsets of solutions that are abortive.
Initially, Rahman's algorithm was implemented in its entirety. However, upon validating the algorithm through JUnit tests, problems were discovered with Rahman's method of calculating the upper bound of a schedule. The algorithm always produced a valid schedule, but only produced an optimal schedule for smaller graphs.

To get Rahman's algorithm to work in all possible cases, the upper and lower bounds would have to be modified. Rahman's upper bound heuristic was replaced with Sinnen's bottom level of a schedule as the upper bound. Rahman's lower bound heuristic was removed entirely. Alongside this, some conditional logic as to which schedules to prune and which to explore was also changed. The final algorithm, Branch-and-Bound-A* pseudocode is as follows (note: The terms node and schedule are interchangable):

```
Create a closed list of schedules already
    visited
Use a priority queue of schedules ordered by
    lowest upper bound cost
Create empty root schedule and add it to
    priority queue
node = rootnode
Best upper bound = infinity

While(true)
 IF node is a complete schedule, RETURN node
```

```
ELSE
   Add item to CLOSED (nodes already seen)
       list
END IF
   For all unscheduled tasks t of the current
        node
      For all processors p
      Create child node = node
         Schedule task t on child node on
             processor p
         Calculate upper bound of new node

         IF node is contained in CLOSED list
            DELETE child node
      END IF

         IF upper bound > Best upper bound
            DELETE child node
     END IF

         IF upper bound < best upper bound &&
              child node is finished
            Best upper bound = upper bound
     END IF

     Add new child node to the priority queue
         (unless deleted)

END IF
    node = Poll lowest cost item from the
        queue
    Add node to closed list
END WHILE
```

For the final algorithm, only one bound was used- an upper bound for a schedule calculated using Sinnen's bottom level algorithm. The bottom level of a schedule is the longest potential path in a schedule based on the weights of its scheduled tasks and a particular task that has not been scheduled yet. So the bottom level of a schedule is the length of time of the longest path starting in a particular task.

When compared to Rahman's upper bound greedy function as our upper bound, as seen below, Sinnen's bottom level heuristic provides a much tighter upper bound and always leads to the optimal solution. The reason for Sinnen's bottom level being superior over Rahman's greedy algorithm, in this case, is due to the bottom level function being a more advanced means of estimating the final length of schedules.

The branch and bound A* algorithm also differs from Rahman's in the sense that the concept of a lower bound is removed entirely. Instead, all pruning is done using the upper bounds of schedules solely. This eliminates the need for calculating two bounds for each partial schedule.

C. Data Structures

Data structures were of vital importance in the implementation of the algorithm. As an enormous search space is produced. Storing and traversing this space using inadequate data structures would lead to a massive slow down in computation time. As speed was a priority in the implementation of our final algorithm, various data structures were needed and selected to speed up computation.

UB with Greedy SJF:

1. **while** (number of unscheduled task >0)
   a. for each processor $P_i \in P$ do the followings
      i. select a task that has a minimum cost and all the predecessors are already scheduled
      ii. get the start time of the task to be scheduled to P_i
         1. start time = MAX [end time of the predecessors + communication cost]
      iii. Schedule the task to P_i in the sequence of S_β
      iv. Update k_β
   b. End While
2. compute make span
3. return make span as the upper bound of the node

Fig. 3. Greedy upper bound function Rahman, M.M. (2009).

Hashmaps were used whenever that needed to be a data structure in which the algorithm would need to make a lookup call. The reason hashmaps were chosen for this purpose was that the lookup time for a hashmap query is constant (O(1)). With a large search space, the algorithm is continuously making lookup calls, and not having constant lookup time would slow down the algorithm considerably.

As seen in the pseudo-code of the final algorithm, the algorithm keeps a list of closed nodes, that is, nodes which have already been visited. The algorithm checks for the existence of nodes in this list numerous times during computation. Rather than use a generic java list, which has a time complexity of O(n) when checking if a certain object exists in the list, a hashset was used to store the closed nodes. A hashcode algorithm was written to generate a hashcode for any particular node, and this hashcode integer value would be used to keep track of closed nodes. Now when the algorithm goes to make a lookup call, this lookup call can be done in O(1) time due to the nature of hash sets.

D. Pruning Techniques

The primary pruning technique used in the algorithm was based on the upper bound heuristic of schedules. This heuristic was used to determine which schedules were worth exploring and which were not.

Throughout the branch-and-bound-A* algorithm, a priority queue of schedules is used to determine which schedule to explore next. This priority queue is ordered by the upper bounds of schedules, with schedules with the lowest upper bounds being at the head of the queue.

With the upper bound heuristic as the bottom level of a schedule being a good estimate of the schedules final length, the schedules at the head of the queue are the ones which are the ones most likely to lead to the optimal schedule.

Moreover, by combining the priority queue with the upper bound heuristic, the algorithm ignores many subpar potential paths are simply focuses on paths that are the most likely to lead to the optimal solution. With this pruning technique, it is highly unlikely that the algorithm will need to traverse the entire search space to find the optimal schedule.

IV. Parallelisation

Even with optimal data structures and pruning, the computation speed of the algorithm still left a lot to be desired on larger inputs. This is simply because the search space, even for smaller graphs, is extremely large. As such, parallelisation, that is splitting up work among multiple threads, was implemented to reduce the computation time needed for the algorithm to find an optimal solution.

The parallelised algorithm utilises multiple threads to find an optimal solution. The reason threads were used was because data had to be shared between different threads. Moreover, threads are lightweight, are easy to create, and use relatively low resources.

The java framework Fork/Join was used to parallelise the algorithm. The Fork/Join framework is designed to take advantage of multiple processors by adding threads assigned with specific tasks to a thread pool. This thread pool is then invoked, and the parallelised algorithm begins.

With the search space of the algorithm being so large, a sufficiently efficient way of parallelising the algorithm is to use multiple threads to search through this search space in parallel. This can be done by sharing the priority queue of schedules between the threads. Each thread takes the head of the queue at any given time and does the branch and bound A* algorithmic computation as described previously. In effect, with threads, more work is being done, and more the search space is being traversed at the same time. This results in the optimal schedule being found more quickly.

No new data structures had to be created, and most of the sequential algorithm could be kept the same. However, as data had to be shared, great care was taken to ensure data was thread-safe. This is most prevalent with the priority queue shared between threads, as multiple threads could be adding and polling from this queue at the same time. Thus, the priority queue had to be made thread-safe. This was done in practice by using a java object type PriorityBlockingQueue. Other variables that were shared between threads were made atomic to ensure thread-safe nature.

As the Fork/Join framework was used, the parallelised algorithm needed to be split between a parallel task that does the algorithmic computation, and a manager that calls this task and handles its output. The algorithmic logic of the parallel task is almost identical to the sequential algorithmic, with the only difference being that on each iteration the algorithmic checks to see if any other thread has finished, and if it has, the thread will stop computation and return. In the implementation, this task was encapsulated in a class called SchedulerParallelTask, which extended RecursiveTask. Alongside this, a new class, SchedulerParallel, was also implemented. SchedulerParallel's role was to create an optimal schedule using multiple threads. The pseudo-code of how the two classes work together to create an optimal schedule is as follows:

```
createOptimalSchedule(unscheduled tasks,
    number of processors, number of cores,
    menu controller)
Create new fork join pool = pool
Create new priority blocking queue = openNodes
Create new list of SchedulerParallelTasks =
    schedulerTasks
Initialise new atomic long variable
    bestUpperBound
Initialise new atomic boolean variable done =
    false
Initialise empty root node and add it to
    openNodes

For the number of cores
      Created new instance of scheduler
          parallel task = t
Add t to schedulerTasks
Invoke t
End for

Initialise new node variable = out

For each task in schedulerTasks
      Node joined = output of task once thread
           has finished

      If joined == null
         Continue
      End if

      If (joined == null or (upper bound of
          joined < upper bound of out))
            Out = joined
      End If
End for

Return out
```

V. Visualisation

A. Concept

The intent of the visualisation component was to provide the user with meaningful live visual feedback of the search. Furthermore, the specifications indicate that the visualisation needs to update as the algorithm progress and also needs to provide some form of interaction. When designing any user interface, there is a natural and commonplace tradeoff between displaying the complexity of the data and keeping the interface intuitive and easy to understand. In the case of this project, the data complexity is associated with all the different iterations of the algorithm and the continually

changing optimal schedule. As a result, presenting the progress directly would be too difficult to understand due to the exhaustive methodology behind the search functionality. Therefore, our team decided to try to minimise the amount of text-based output. More specifically, the team would only display detailed text information when the user requested it (e.g., details about specific tasks). This allowed the rest of the interface to focus on the main graphical component. This segment of the graphical user interface (GUI) would serve as the focal point for the live visualisation. To further justify this concept, and accompanying layout, research was conducted to ensure that that this was an appropriate choice for the specifications given. Specifically, finding projects with similar interfaces and specifications helped solidify the team's understanding of visualisation solution would be fit for purpose in this project.

B. Implementation

The interface went through multiple iterative stages to get to the final design (please see figure four). Notably, the graphic component was initially implemented as a tree with the intent of displaying the branching structure of the algorithm. The two significant problems that were found with this approach were to do the meaningfulness of the display and the native java tree libraries not suiting the interface specifications. Similar complications would occur during the development of this solution due to this section of the project having comparatively open-ended specifications. Major design decisions and problem-solving sessions were all combatted with in-person group discussions along with individual research, exploration and validation. The waterfall development process lead the team to an application with two major sections driven by user interaction.

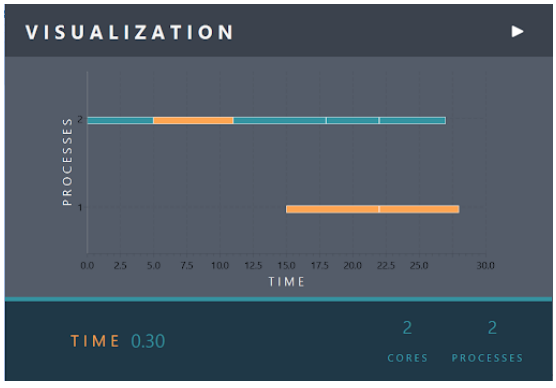

Fig. 4. Final Iteration of the Graphical User Interface.

C. Illustrative Visual Output

The overall layout for the visualisation platform utilised the common rule of thirds design principle. The top two-thirds of the display was dedicated to the primary output, which in the case of our project was a Gantt chart. On initialisation, this section of the interface shows an empty chart prompting the user to press the start button. This design decision was critical in promoting user-centred control and intuitively communicating to the user that the algorithm had not started yet.

The utility of the play/start button was also functional because when the system was tested without it (e.i., would immediately start visualisation after the user pressed enter on the command line) the user would often miss the first few seconds of the live search updates. This simple addition allowed the user to understand the GUI before interacting more intently with it. Once pressed, the algorithm will run, start the visualisation and produce the desired output. The Gantt chart always displays the current optimal schedule. Based on pruning and sorting methods, the algorithm will continuously update what it considers optimal. The y-axis of the Gantt chart refers to which processor each task is scheduled on, the amount of which is specified on input. Whereas, the x-axis displays the cumulative start-time and end-time for each of the tasks in the schedule to give the user a clear indication of what the current best schedule is.

A tradeoff associated with the Gantt chart updating continuously means it can be hard to read initially as it updates frequently, but over time the changes become less drastic. As the algorithm runs, there will be fewer schedules that have the potential to be the optimal schedule. This will be reflected by the slower rate in "flickering". Possible solutions for reducing the flicker would have involved either slowing down the algorithm or to update the chart less. Neither of these solutions is ideal. Slowing down the algorithm would be counter-intuitive as the point of parallelisation was to speed up the algorithm. Updating the chart less would be kept the efficiency of the algorithm, but it would not be a live representation of what the algorithm is doing, which was one of the requirements given to us in the design brief. Due to the specifications indicating the visualisation had to be live and meaningful, the team concluded that it would be more critical for the user to see the data unaltered rather than simplifying the output.

To further enhance the user experience, the Gantt chart was made zoomable via direct user interaction through mouse scrolling and dragging. Zooming allows the user to see the chart in more detail and also lets them check exactly what times each of the tasks finish. The justification for the main visual component and its features originates directly from the specifications and tries to limit any subjective design decisions in favour of professional design standards.

D. Advanced Visual Features

The other part of the visualisation is more detailed and consists of less graphically orientated output. Once the algorithm has stopped, which is indicated by the timer

stopping, the user can interact with the output and gather some more detailed, meaningful information. Tooltips have been implemented so that each Task can be hovered over to reveal more details. This includes the Task's start time, task number, weight and processor. This feature intends to not only solidify the visualisation details but to also provide a direct visual metaphor for the output file with the optimal schedule. Further data output is provided by the bottom bar, which includes a live timer, for immediate real-time feedback, along with some information about the user's input (cores and processors). This was intended to help users identify if they had mistyped anything that was not what they intended to enter but also wouldn't be picked up by the error checking. The whole visualisation part of this project was implemented using the JavaFX library, with Scene Builder used as a utility to create the general layout. As JavaFX did not natively have components that we required, such as the Zoomable ScrollPane and the Gantt chart component, we found resources on Stack Overflow that we modified to suit our uses.

The Zoomable ScrollPane is a class that inherits from the JavaFX ScrollPane component. In this class, scroll event listeners have been added to implement the zoom function. Whenever a scroll is detected, the new positions of the contents of the pane will be calculated and displayed accordingly. As this class is technically a ScrollPane, it means that with the pannable attribute set to true, the user will be able to move the graph around by clicking and dragging the mouse (please refer to figure five).

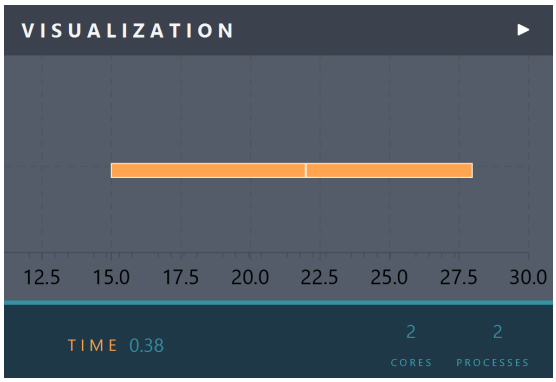

Fig. 5. Zoom and drag feature of the Graphical User Interface.

The Gantt chart is a more complex component that inherits from the XYChart component. After initialising the chart, it waits for update calls which will be handled by the controller. When the algorithm (Scheduler object) finds a new potentially optimal schedule, it will invoke the updateGraph() method on the controller object with the schedule as the parameter. The controller will then retrieve the list of scheduled tasks from that schedule and extract the information it needs. The start time and the process associated with the task is used as the X and Y axis values respectively, with the weight of the task used as the length of the block displayed on the chart. All of this information, as well as the task number, is passed through as parameters to the Gantt Chart component.

To provide the user with more information about the schedule, a JavaFX Tooltip is associated with each task/block. The tooltip pops up whenever a task hovered over, providing the user with all of the information regarding the task. Please see figure six.

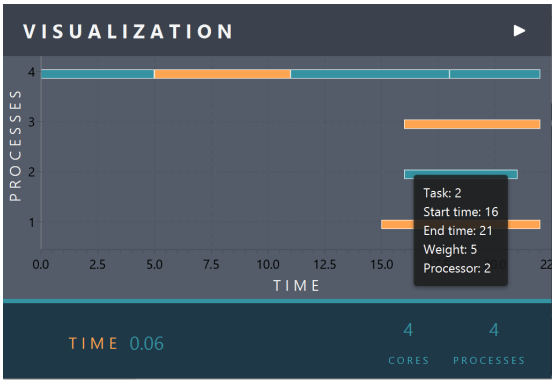

Fig. 6. Intuitive illustration of task information.

E. Comparison Between Sequential and Parallel Visualisation

In practice, there was no difference in visualisation between sequential and parallel visualisation. The only difference was the display letting the user know that the program was being run on n amount of cores. However, the user would notice that the Gantt chart would "flicker" more. This is not a bug, but rather a design decision after considering the trade-offs.

Whenever a new schedule is found that is potentially better than the current best schedule, the chart will update to show that schedule. The issue here is that due to how the algorithm operates, in the beginning, it will find many schedules successively that it believes is the current best, causing the chart to update many times, creating a "flicker" effect. When the program runs in parallel, each of the threads searches down a different branch for an optimal schedule. This increases the speed at which a new potential optimal schedule is found, therefore, increasing the rate at which the chart will flicker.

VI. Testing and Validation

Despite not having a testing phase in the waterfall method of development, the team firmly believes that a feature is not fully implemented if it has not been tested and proven to work. For that reason, JUnit test suites were created to check the main aspects of the project have been implemented correctly.

The first thing that needed to be tested, and also arguably the most crucial, was that the input file was being read correctly. The test suite written for this created a temporary test file with appropriate data, and the FileIO class was tested on it. This test suite verified that the information from the

dot file was correctly read and processed. This included making sure all the nodes were made into Task objects with the correct information, and that the Task objects contained any parent or child tasks associated to it as specified by the transitions.

Another area that was heavily tested was the algorithm. To determine if the algorithm was correct, the output of the scheduler needed to be checked against the correct values. These were either provided along with an example input, or one that was solved by hand. Again, a JUnit test suite was used as it is the most efficient way of checking the algorithm. It also provided a means of calling the algorithm independent of other aspects of the project. By nature, the test suite does not care what the implementation of the algorithm is. As long as the method it calls corresponds to the method that begins the scheduling, it will match the output with the correct answers. This gives a quick indication of whether the algorithm is right or not. At the same time, as part of the test suite, print statements are used to print out the output schedule. These provide an insight into where the algorithm might be going wrong and how to fix it.

VII. Conclusions

Overall, this project was a valuable learning experience for everyone involved. It not only furthered the team's technical capabilities but by trying to solve an NP-hard problem, the team developed a more in-depth understanding of sophisticated software design. Furthermore, the difficulty of individual tasks meant that they took lots of time to complete, and thus, all team members could experience engagement with each of the subsections of the problem. Designing and planning a solution for the parallel scheduling problem is inherently ambiguous, but since the specifications are set, the waterfall approach was a logical choice. As a result, the team was able to recognise the limitations and areas of improvements easily due to the development process's linear structure.

Throughout the waterfall development process, the team stopped to reflect on organisational standards and process practices. Recognising the advantages and disadvantages of the workflow was a continuous topic of discussion, leading the team to draw their own conclusions about possible improvements. Using the waterfall approach as a platform for in-person project development was successful in establishing a working environment that was organised and well-coordinated, especially when combined with conventional software development concepts like pair programming and version control. The team found that fundamental disadvantage of the standard waterfall model is in its unidirectional nature as this limited the team's ability to reorganise and adapt to problems. Furthermore, the initial requirements/planning phase of the model produced a plan with some significantly optimistic approximations, which lead to the team to being behind schedule during the fourth and fifth weeks of the project. It is felt that if the development model was more iterative (e.g., iterative waterfall process or agile development), then some of the drawbacks leading to the dip in progress could have been handled better.

The final product successfully meets the requirements set by the client, but this does not indicate that there aren't any areas of improvement. The visualisation of the search could have been handled in a variety of different ways, and a result, it is difficult to objectively evaluate the general concept due to its associated trade-offs. However, more extensive testing would have likely lead to the team finding problems earlier. Explicitly, the lagging timer could have been resolved if the team had decided to go with a more rigorous testing approach that included boundary value analysis or other similar ideologies. However, one area of the project that was handled well was the communication and resolving of conflicts within the team. Working together was a rewarding and education-rich experience for all members. This working environment meant the team had a positive attitude towards the project and a real drive to create an efficient and effective solution.

Acknowledgment

It is essential to acknowledge the clients and primary stakeholders, Xuyun Zhang, Henry Li, and Benny Zhang for this project as they provided the detailed specifications for the project and the corresponding milestones to help structure the progress. Furthermore, their willingness to answer questions during the interview sessions allowed the team to get a better understanding of the project standards where necessary. This, along with the specification sheets, was extremely useful for our team because it allowed us to clarify and develop certain features with limited ambiguity and uncertainty.

Developmental tools should be acknowledged as they significantly aided in structuring and documenting the project. PlanHammer was used to create all of the initial planning graphics (i.e., Gantt chart, etc.) and documentation. Github was used as a source for the project Wiki as well as a way to assign group members to individual tasks and keep track of their progress (via the backlog). It should also be noted that the Google cloud suite was used to store presentations, meeting minutes, and other shared data in a secure team drive.

Similarly to developmental tools, other external tools/libraries were used alongside Java 8 to help accelerate the development of the project. Notably, the JavaFX and Junit libraries allowed the group to implement the MVC model effectively with appropriate unit testing. Git was critical for version control due to its branching and pull request features.

Finally, M.Mostafizur Rahman and Oliver Sinnen must be acknowledged for the public documentation they created. Without them, the final product would not be as efficient, as these papers were heavily researched and analysed to try and understand the full scope of the problem domain and solution.

REFERENCES

[1] Cs.indstate.edu. (2019). History of A* Algorithm. Retrieved from: http://cs.indstate.edu/hgopireddy/history2.pdf

[2] Roland. (2015, January 16). Gantt chart from scratch. Retrieved from https://stackoverflow.com/a/27978436

[3] Daniel Hri. (2017, June 17). Scaling / zooming ScrollPane relative to mouse position. Retrieved from https://stackoverflow.com/a/44314455

[4] Sinnen, O. (2014, March). Reducing the solution space of optimal task scheduling. Retrieved from https://researchspace.auckland.ac.nz/handle/2292/30213

[5] Rahman, M.M. (2009). Branch and Bound Algorithm for Multiprocessor Scheduling . Retrieved from https://pdfs.semanticscholar.org/1aa2/b187ee99 abc19250d5dce73f390a7f65d9fe.pdf?fbclid=IwAR 2C-ku Nbi2AtXA_lhsHdzhaMtDefDu7T5e0nwH9ojTAS0f2YVTFcrKqp44

[6] Rahman, M. M., & Chowdhury, M. F. I. (2009, December 21). Examining Branch and Bound Strategy on Multiprocessor Task Scheduling. Retrieved from http://citeseerx.ist.psu.edu/viewdoc/download? doi=10.1.1.210.3983&rep= rep1&type=pdf&fbclid=IwAR1VDW3o mzAIwA_KqqU OrDA2wKrskk3D4kG-CQy-jk_yPiMVEzzspERZa9s

[7] Sinnen, O., & Venugopalan, S. (2014). Schedule length bounds for optimal task scheduling. Retrieved from https://scheduling2014.sciencesconf.org/ conference/scheduling2014/pages/OliverSinnen.pdf