PARALEX

Parallel Task Scheduling

Team: Ammar Bagasrawala, Ben Mitchell, Hanzhi Wang, Jordan Wong, Jun Xu, Kevin Yu

PARALEX - GROUP 1

ALGORITHM

In the initial stages of development, three methods of finding an optimal solution to the problem were identified, Branch and Bound, A* and an exhaustive search. All of these methods were implemented within the first few weeks of working on the problem, however problems quickly became apparent. The A* implementation which was implemented in a way which would use as little memory as possible but was unfortunately still running into memory problems on some inputs, namely fork and join style graphs. The exhaustive search proved useful in validating the output of other methods but was obviously too slow, and did not work on larger inputs. The Branch and Bound had run times on par with the memory light A* that had been implemented, while simultaneously avoiding the memory issues the A* implementation had. We decided to use Branch and Bound as our implementation as it was fast and would not run out of memory on larger inputs.

INPUT PARSING

The GraphStream library was used to parse the dot file directly into a Graph object. GraphStream was used for this process as it has a relatively simple API, and by opting to store the underlying input graph in GraphStream's inbuilt Graph class, GraphStream's visualization capabilities were able to be harnessed. At the end of parsing the dot file a Graph object containing Node and Edge objects would be created.

```
// creating the Adjacency list of the input file graph
ArrayList temp1 = new ArrayList();
for (int i = 0; i < numNodes; i++) {
        ArrayList temp = new ArrayList();
        temp1.add(temp);
        for (int j = 0; j < numNodes; j++) {
                if (adjacencyMatrix[i][j] != -1){
                        temp.add(j);
                }
        }
}
this.setAdjList(temp1);
```

Figure 1- An example of the Adjacency List being created

From this point onwards as much relevant graph data as possible was pre-processed in order to save memory and simplify the following branch and bound algorithm. An adjacency matrix, adjacency list and dependency list were all created in a straightforward process of analysing the relationships between the graph's nodes and edges to allow fast access to any graph information the algorithm would need

Other data structures such as a list of node weights were also calculated. The most important piece of pre-processed information was the bottom levels of each of the graph's nodes. A bottom level of a given node is the value of the maximum path from the given node to any leaf node (excluding communication costs). The bottom levels for each node were calculated in a recursive way starting from all of the sources of the input graph. When a leaf node is encountered it returns its own bottom level (its own weight) to all of its parents who have each previously asked for it. Once a parent has received bottom level values from all of its children it takes the maximum of all its children bottom levels, adds its own weight and sets that number to be its bottom level. This process recursively happens until all nodes have been assigned bottom levels.

```
createBottomLevels(ArrayList<Nodes> sourceNodes){
        for (Node i: sourceNodes){
        ArrayList children = i.getChildren();
        if(!children.isEmpty()){
                        this.selfBottomLevel(selfweight +
                        max(children.createBottomLevels()));
                        return selfBottomLevel;
        } else {
                this.setBotLevels(selfweight);
                return selfBottomLevel;
        }
        }
}
```

Figure 2 - Pseudo code for creating bottom levels

BRANCH AND BOUND

As per mentioned above a branch and bound style algorithm was implemented in order to find the optimal schedule of the input graph. The algorithm held a stack of partial schedules that were to be looked at. When a node was looked at, any possible children of the partial schedule were generated and placed on the stack. Processing ended when the stack was empty. The implemented algorithm consisted of 2 key sections: generating feasible partial schedules in the form of a partial schedule tree, and pruning unpromising branches of said schedule tree.

Two forms of heuristics were used in an attempt to remove as many unpromising branches from the partial schedule tree as possible.

The first heuristic used idle time and total task time along with the amount of processors being scheduled on to produce an underestimated of the total time a given partial schedule would use. The equation used is as follows:

Estimate = (ScheduleIdle_Time + Sum_of_all_Node_Weights) / Total_Number_Of_Processors_To_Schedule_On

The calculated estimate above is an underestimate as it assumes that tasks in the future can perfectly equally be placed across all processors, not only that but it does not consider communication costs in any way.

The second heuristic calculates another underestimate of the total time a given partial schedule would use. This underestimate uses the bottom level of the most recent node added to the partial schedule along with the start time of that node, once again the equation is as follows:

Estimate = Node_BottomLevel + Node_startTime

Both of the heuristics were calculated when a given partial schedule generated children, essentially creating a new partial schedule by adding a task. The maximum of these two estimates was then taken and set as the newly created partial schedules' under-estimate finish time. As each partial schedule underwent this process every partial schedule would have its own under-estimate. Each schedule's under-estimate is considered and compared against the current best schedule before and after it is placed/removed from the stack. If a schedule's under estimate is worse than the current best schedule time then it must be that any complete schedule generated from that partial schedule cannot be better than the current best schedule, and hence it is discarded. The act of discarding essentially removes unpromising partial schedules from the schedule tree.

The schedule class represented a node in the schedule tree. Each node contains how each of the currently assigned tasks are scheduled according to the node's partial schedule. Other fields such as current idle time, current estimated finish time and the number of occupied processors, are also contained within the Schedule class. There is also a method that generates all valid child schedules by assigning a doable task to a processor. This method produces valid schedules as it keeps track of which tasks have their dependencies satisfied, and only assigns these tasks to processors. For each child schedule generated, all the values that the parent has is copied over to the child, and then each of its fields are recalculated. The method returns the list of valid child schedules.

```
function generateChildren
    for i = 0 ; i < doableTasks ; i++
        for j = 0 ; j < processorCount ; j++
            recalculateChildFields
            childList.add(new Schedule)
    return childList
```

PARALLELIZATION

HOW WORK WAS SPLIT

The parallelised implementation first generates the upper section of the tree using Breadth First Search (BFS) and adds the resulting nodes to a queue. These nodes, each of which represent a partial schedule, would then be expanded by the workers to find the possible ways which these partial schedules may be completed.

```
function initialise
    for i = 1 ; i < X ; i++
        node = queue.pop()
        children = node.generateChildren
        queue.addAll children
```

Firstly, a 'phantom' node, representing an empty schedule, is added to the queue as the root of the search tree. After that, the first element of the queue is taken off, the generateChildren method is run on it to get a list of schedules which can be created given the node as the base, and all of these are added to the queue. This is repeated several times in order to obtain a queue of partial state tree nodes to be expanded by the workers

DATA STRUCTURES

Our implementation did not require any changes to our schedule's data structure, as we were able to use the same implementation as before. The traversal algorithm needed some changes to the storage data structures, as each worker needed to keep track of their own progress.

A global variable for the best schedule found so far is kept (the global current best schedule). Each worker keeps their own local current best schedule as well, and this is periodically synchronised with the global one. This is done periodically to reduce the amount of locking and unlocking required which incurs additional overhead.

PARALLELIZATION TECHNOLOGIES

The technologies that were investigated were Pyjama and ParaTask. Although Pyjama seemed to allow for more control over the load balancing of the parallel scheduling, ParaTask was used for the parallelization of our search algorithm. Since this was our first experience creating a multi-core application, the technology used must have the functionality needed, while also avoiding any unnecessary complexity. Both Pyjama and ParaTask had the required functionality, but Paratask abstracted a lot of unneeded detail, which is why ParaTask was chosen. ParaTask's simple plug-and-play interface allowed the parallelization of the search algorithm to be implemented more simply. Pyjama's learning curve, and the risk of not being able to complete the parallelization due to time constraints, were the main reasons why ParaTask was chosen.

IMPLEMENTATION

Each worker thread was given the same algorithm to run. The worker takes a node from the global queue of partial schedules and expands it using branch-and-bound. When the worker has explored the whole subtree of the node it took from the queue (reaching every leaf which was not bounded off by estimations), it takes another node from the global queue to expand. When the global queue becomes empty, the worker exits. When all workers have exited, the global current best schedule is returned as the optimal schedule.

```
function runWorkers
        initialise
        startAllWorkers
        waitForAllWorkersToFinish
        return globalBestSchedule

function run //startAllWorkers starts calls this
        while true
                while localStack not empty
                        node = localStack.pop
                        if node.estimate < localBestSchedule.totalTime
                                children = node.generateChildren
                                localStack.addAll children
                        if node is leaf and node.totalTime < localBestSchedule.totalTime
                                localBestSchedule = node
                                push
                        schedulesTraversed++
                        if schedulesTraversed multiple of 10000
                                pull
                if globalQueue is not empty
                        nextNode = globalQueue.pop
                        localStack.push nextNode
                        push
                        return
function push
        lock
        if localBestSchedule.totalTime < globalBestSchedule.totalTime
                globalBestSchedule = localBestSchedule
        unlock
```

```
function pull
        lock
        if globalBestSchedule.totalTime < localBestSchedule.totalTime
                localBestSchedule = globalBestSchedule
        unlock
```

Each worker keeps track of its own 'best' complete schedule found so far and uses it for bounding. When a worker encounters a complete schedule which is better than their local current best schedule, they will update their local current best schedule and synchronise it with the global one (push). While this will incur a performance cost in needing to lock the global variable for writing, we decided that this would be negligible as during testing we discovered that the best schedule is only updated a few times, even for medium to large inputs.

Each worker will also synchronise their local current best schedule with the global one (pull) every time it traverses 10,000 nodes. This ensures that each worker has a relatively recent current best schedule for bounding, while still minimising the amount of time wasted by needing to lock and unlock the global best schedule variable.

VISUALIZATION

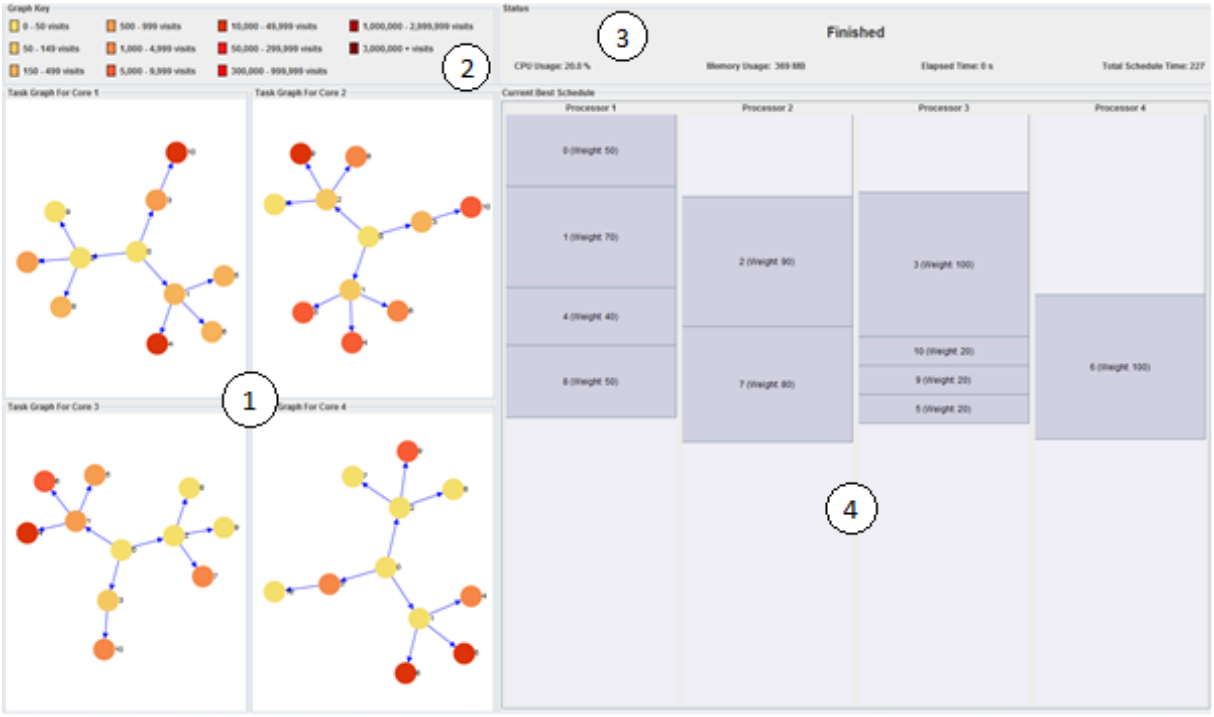
CONCEPT

In the early stages of the project, namely the planning phase, the initial concept of the visualization was discussed as a group. From this discussion, it was decided that the major components to be shown on the user interface would include the schedules that were being generated by the algorithm and a tree representing the search tree of partial schedules. This was believed to be a respectable idea due to the fact that any user would clearly be able to see how the algorithm worked, for example the pruning would be clearly visible. Parallelization was also taken into account, and the plan was to initialise the GUI with more than one search tree, so each core had its own tree. It was not until the algorithm itself was completed that the visualization of this concept could be created, however. Before the implementation of this idea, it was noticed that the number of schedules produced could reach levels where millions of nodes would be generated for certain input graphs. Seeing as generating this tree was not feasible due to the enormous amount of nodes that would be created, the concept was modified.

The schedules were still included in this new concept, but the difference was what the information the graph would show. Instead of showing schedules, a decision was made to display the graph provided as an input and colour the nodes based on the number of times the algorithm tried to schedule those tasks. This showed the tasks which were being scheduled the most, which implied how the algorithm is trying to schedule the tasks. In order to represent the parallelization, multiple instances of the input graph were produced where one core would modify the colours of its own graph. This was believed to clearly show how each core worked with others in order to produce the best schedule. Throughout the project development lifecycle, the visualization was modified to best represent any crucial information. This included both small and large changes which impacted the overall visual appeal of the user interface and its informativeness.

DISPLAYED COMPONENTS AND JUSTIFICATION

The final visualization implementation included four major components: the graphs represented by the input dot files (1), a key indicating what the colours of the nodes represent (2), a panel where the current best schedule is shown (4) and a status panel which includes the CPU usage, memory usage, the time taken for the program to find the best schedule and the total time of the best schedule (3). Out of these four major components, the graphs and best schedule were considered as the most significant and thus on the GUI they were given the most space.



During the group discussions, it was determined that the input graphs would be shown, and if multiple cores were working, each core would have its own graph. The colour of the nodes in each of these graphs would be indicative of the number of times that core has tried to schedule that task. This was implemented as such as it clearly showed points where most of the work was going on, as a cluster of red nodes for example would indicate these tasks have been visited many times in order to be scheduled. It would also signify other attributes such as those tasks have many dependencies or are leaf nodes.

One aspect of the visualization which was a cause of concern was the responsiveness of the GUI. At times, because the best schedule was found early on and the algorithm had to still verify this was the best schedule, the GUI components wouldn't change as much which caused the interface to look like it was unresponsive. A modification which combatted this issue was the "popping" of the nodes. Every time a node was visited approximately 200,000 times, the node would be enlarged and then shrunk.

Another feature which was added to aid usability was the highlighting of nodes on the graphs depending on where the mouse was hovered in the best schedules panel. This was implemented so users could easily and efficiently map between tasks located on the schedule and where they were in the input graph. Being able to link these components aided in the understanding of the best schedule. The colours chosen for the nodes were based on the idea of having a heat map, so colours in the range of yellow to red were used to indicate the level of which nodes were visited, where yellow meant those nodes were not visited often and dark red meant they have been visited many times. The partitions for these colours follow a logarithmic curve, where initially the colours represent very ranges of numbers, and these ranges get larger as the colours get redder.

Since the initial phases of the project, it was determined that a schedule would be shown on the GUI. This was because it was the best representation of the output. However, it was slightly modified so that only the best schedules would show rather than every schedule. This was due to the fact that the schedule would update too many times and thus be unreadable, by instead having the current best schedule this solved the issue and also presented the user with more information.

The other two components of the GUI; the key for the graphs and the status panel aided in providing information to the user to understand how the program works and its current status. CPU and memory usage were added due to the lag associated with certain input graphs, by seeing that the usage is at its maximum the user can understand the reason for any lag or delay occurring. It was also to show the difference between the sequential and parallel algorithms as parallel would use all of the CPU, while sequential would use only 25%.

IMPLEMENTATION

In order to visualise the input graphs an external library, GraphStream was used. Methods in the library allowed us to read the input dot file and draw the graph automatically. This initial graph was added to a list of graphs depending on how many cores were selected to perform the scheduling so that each core mapped to one graph. Due to each core having an ID ranging from 0 to the number of cores selected, a core would access its own graph by using its ID as the index value. This also ensured that no core would access another's graph and thus provide incorrect output in terms of visualization. In order to show how many times each node in the input graph was visited by the scheduler, two main methods were used; incrementTask() and setNodeColour() along with a CSS file which held all the styling properties for each level of visits. Each core calls the incrementTask() method and passes in arguments such as its ID and the task which was just visited. This then increments the value for that task and that core and calls the setNodeColour() method which sets the colour of that node accordingly by changing its CSS class. The "popping" or "blinking" of the nodes was configured so that the size of a node increases every 100,000 visits and shrinks after the next 100,000. This added responsiveness improves the visual feedback of the program as it clearly identifies whether the program is still running.

In order to add the graphs to the GUI in a manner which still allowed users to clearly identify important information, a GridLayout was used for the overarching panel and each graph was added to this. The number of rows and columns were set based on the number of cores supplied in the input so each graph could be easily added to a row or column and be resized automatically depending on the total size of the panel. The section of the GUI showing the current best schedule was implemented through the use of JTables and a GridLayout for the panel which holds all of these components. The main reason for using multiple JTables was due to the fact that the tasks scheduled on each processor ends at different times. This meant that each task's cell could be resized individually. The functionality for the highlighting of nodes was implemented through a mouse listener and was added to each JTable. In order to update the current best schedule, a SwingWorker was used which was branched of the main thread. This queried the scheduler algorithm at regular intervals to obtain the data stored and update the GUI accordingly.

DIFFERENCE BETWEEN SEQUENTIAL AND PARALLEL VISUALIZATION

The major difference between the sequential and parallel implementation of our visualization is that each core has its own graph to display. The visual frame itself will determine and provide the space required to display each graph. Each core will update their own graph on the visual frame. The likelihood of a graph having the same characteristics as another is slim on large inputs and this will be obviously indicated by same nodes having different colours on each graph. Using this method, we can get a general estimate of what tasks cores have been working on and identify the critical tasks some cores have worked on. The statistics will reflect the amount of cores being utilised with such examples being CPU usage reaching up to 100% with 4 cores.

TESTING

Thorough testing and validation was conducted intensively throughout the timeline of our project. Each aspect of our project was tested and validated in some form to ensure correctness and reliability. When components from different areas came together, we had to ensure that the flow of the program matches the functionality and rigorous testing was conducted to ensure this correctness. Performance testing was performed mainly during the optimisation stage. The tests help measure the degree of improvement in our code and was a major asset in optimisation. Stability of our algorithm was also tested and necessary fine tuning was conducted to ensure a correct solution. Edge cases such as fork and join graphs as well as empty and independent node graphs were tested against our algorithm to measure the stability of our program. Performance testing would also include the integration of parallelization. Runtime differences when running in parallel was a good indicator of the parallelization working. We personally developed test cases to ensure our program was running but we also used Oliver's provided input graphs to add an extra layer of verification and confidence in our algorithm. Visualization would also be another component to test as the graphical interface would have to reflect the algorithm's activities and data that is present in this interface would have to be legitimate.

In the early development phase, testing was mainly conducted manually as we had limited time and resources to develop some form of automatic testing. Our priorities being milestones and implementation were of greater importance and manually running certain inputs, and looking at their respective output provided enough

assurance to complete earlier milestones. Later on in the development phase, we were able to incorporate automatic testing.

The tests were mainly performed using Junit test cases and the use of Maven. Maven was used to instantiate our test suite and also work in conjunction with our Travis CI implementation on GIT. Personalised tests were also run independently to isolate functionality and ensure that issues were not present. Travis CI was also beneficial in terms of identifying commits which broke the build. We also developed a brute force algorithm which help validate the correctness of our solutions. When we integrated new algorithms, we would verify the new output with the brute force algorithm and this was a perfect safeguard for our project. We also intended to develop a schedule validator which would allow us to define whether a schedule was a valid schedule from a graph but due to time constraints, this feature never progressed to the development stage.

DEVELOPMENT PROCESS

In the meetings after we have decided on our approach to the project, we developed the software architecture of our design and modularized it. The development of each module was assigned to each team member. The advantage of this is that we were not restricted in terms of the implementation of each module. We just needed to ensure that the interfaces handling communication between modules were satisfied. When it came to implementation, we were more waterfall based because we had clearly defined tasks and everyone has agreed on the design. There were minimal occurrences where changes had to be made due to unforeseen aspects of the design. This sped up our development because there was not much time consumed in regular meetings and design changes. The downfall was the larger amount of time spent in planning the design of the solution but everyone in the team thought that this was a good approach since the project requirements were not likely to change.

We were mostly developing in the same room, so whenever there was an issue, we could have quick discussions with each other. These issues were also tracked using the GitHub Issue Tracker, which also sent an email to us and this was helpful when not everyone was present at that time. We found the Issue Tracker extremely helpful in terms of keeping track of changes that needed to be made.

Regarding meetings, we have similar timetables, which made it easy to schedule meetings, and we found a convenient room where we could have our meetings. During meetings, we discussed ideas and assigned tasks to each other in person but after meetings, we used the GitHub Issue Tracker to formally assign tasks. This way, there was a consistent page that defined the tasks that everyone needed to do and this was visible to everyone. When making decisions such as which algorithm to use, instead of ‘majority vote’, we decided to analyse the different options and list the pros and cons of each algorithm. We would then see how this affects other areas of the project such as visualization. From the analysis, we made our decision keeping in mind correctness, speed and good software engineering practices.

Throughout the development process, everyone was worried about the merge conflicts that would occur since everyone is working on a different part of the project. To overcome this, before we started coding, we designed the software architecture of the project and modularized the different components. The requirements for interaction between the modules were well defined and each person was then in charge of developing each module. This made it easier in the development process because we were not restricted in terms of the implementation of the module; only the interface interactions between the modules needed to be satisfied. Because of this, there were minimal merge conflicts.

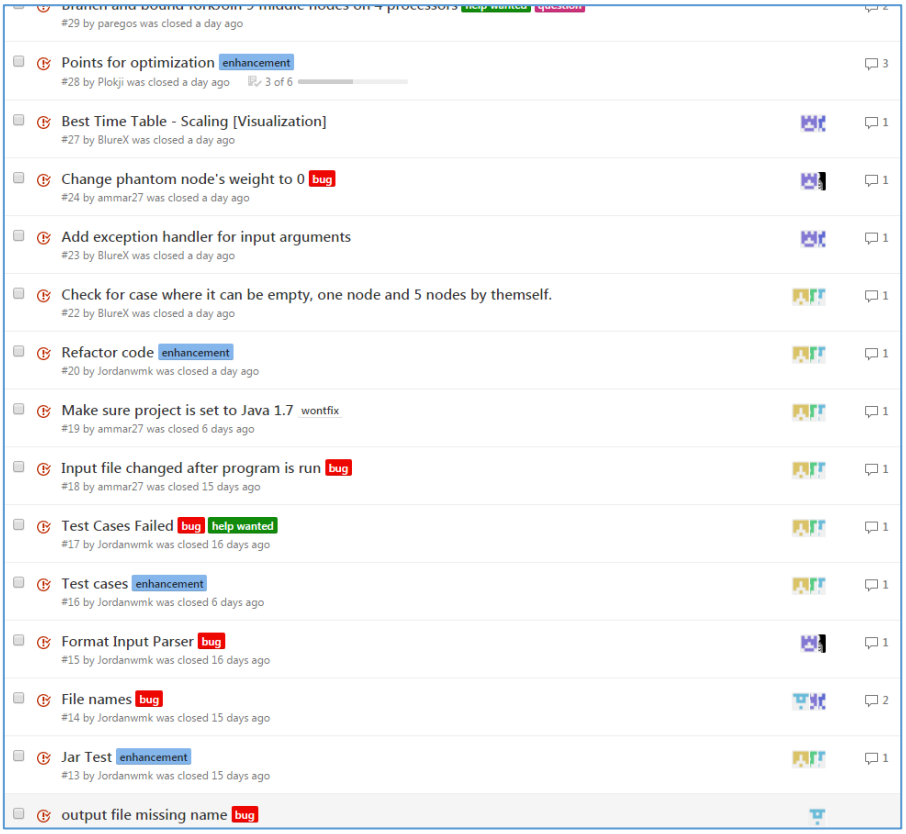
For source control, we used GitHub because it provided free private repositories for students. We integrated TravisCI with our git so that we were informed whenever a commit broke the build. It took some time to set it up as we were new to this technology but it proved to be very helpful in the event that we overlooked an area of code and created a bug. However, after implementing parallelization, we were no longer able to use TravisCI due to being unable to add ParaTask dependencies to Maven. This was not too much of a problem because by this point we have already mostly finalised the algorithm implementation, and further testing was done manually.

Before deciding on which technologies we used, we did research on their capabilities and ease of use by looking at online documentation and making small test applications. For the graph visualization and parsing, we used GraphStream because it provided an extensive API that made development smoother. For the parallelization, we decided to use ParaTask because we found it easier to understand the documentation for it compared to other similar technologies such as Pyjama.

Throughout this project, we have gained a better understand of each other's skill set. When planning and assigning tasks, the work was distributed in a way so that we were able to utilize our strengths. This made the development process much smoother because the people working on each task has had prior knowledge in the area, which allowed us to avoid some pitfalls that would have slowed us down. Being able to work on the area that interests us also motivates us more when we are working and this was important in keeping the team morale high.

We thought that this project has taught us how to work better as a team. In the early stages of the project, meetings took a relatively large amount of time but in the later stages, we have improved the coordination of meetings, which made them more efficient. Moreover, we have also learnt the importance of understanding and accepting the ideas of other people because different people often have different views and hence, different solutions to problems. Throughout this project, our team cohesion has been continuously increasing as we learn more about each other.

Efficient communication of ideas ensured that everyone was on the same page at all times and this made discussions and development smoother as well. Since everyone contributed to all the areas in the final project, we were able to gain some insight as to how we each work. From this knowledge, we self-improved and provided advice whenever we noticed something that could have been done better. This distribution of experience in the team not only made the development faster, but it also provided a valuable learning experience for all of us.



MAJOR TASK CONTRIBUTIONS

Names: **Project Components:**	Jordan Wong	Jun Hao Xu	Hanzhi Wang	Ammar Bagasrawala	Kevin Yu	Ben Mitchell
Code Refactorization	16.67	16.67	16.67	16.67	16.67	16.67
I/O Handler	0	5	0	45	0	50
Estimation Algorithm	0	20	40	0	40	0
Tree Generator and Traversal Algorithm	0	0	40	0	40	20
Documentation	40	12	12	12	12	12
Parallelization	5	5	40	5	40	5
Visualization	10	40	0	40	0	10
Report	16.67	16.67	16.67	16.67	16.67	16.67
Optimisation	50	10	10	10	10	10
Testing	20	30	0	20	0	30
Software Architecture/Planning (WBS/Network Diagram/Gantt Chart)	25	28	8	18	8	13