Final Report

Team 16: Outsourced to Pakistan

1 Team Members

- Adam Sinclair (asin473)
- Henry Man (hman845)
- Osama Kashif (okas406)
- Remus Courtenay (rcou199)
- Syed Ahmad Kazmi (skaz269)

2 Algorithms

Our approach uses the A* scheduling algorithm, which is a version of a branch-and-bound depth-first search that explores the best option first. It achieves this by using a priority queue, where each potential schedule holds some heuristic value.

2.1 Pseudo-code

```
create a priority queue Q of schedules
insert initial (empty) partial schedule into Q

while true:
    pop top schedule S off priority queue
    if S has no free nodes:
        return S, the optimal schedule
    else:
        create all permutations scheduling a free node on S, List(SNew)
        Prune List(SNew)
        Add all unpruned schedules remaining in List(SNew) to Q
```

2.2 Heuristic Function

The sequential algorithm employs a non-blocking priority queue. For the A* algorithm to work effectively, a priority value is needed for ordering in this queue. In the scheduling case, this heuristic is an estimation of the final schedule length. This must be a lower bound for A* to guarantee optimality. The tighter the bound, the more effective is A*. Our heuristic is derived from Oliver Sinnen's 2014 paper. The heuristic is the maximum of the idle time, bottom level, and data ready functions described below.

$$f(s) = max\{f_{idle-time}(s), f_{bl}(s), f_{DRT}(s)\}$$

2.2.1 Idle Time

The idle time of a partial schedule, s, is here defined as the sum of the weights of all nodes on the input graph, plus the current idle time (time a processor is not working on a task) for all processors, divided by the number of processors.

2.2.2 Bottom Level

The bottom level of a node is the sum of computation costs for the current node and all nodes on the critical path from that node to the exit. The max bottom level of a partial schedule, s, is defined as the maximum bottom level of any of its nodes.

2.2.3 Data Ready Time

The data ready time of a task is the earliest time a task could be scheduled on any of the processors, given the communication costs of its parent tasks, as applicable. The max data ready time of a partial schedule, *s*, is defined as the maximum data ready time of its free nodes.

2.3 Data Structures

As we decided to utilise the A* algorithm to generate the required optimal schedules, some thought was put in regarding what would be the best way to store the generated data. A* sources much of its search speed from checking generated partial schedules against a set of already checked ('closed') schedules. This is done to ensure the algorithm only has to check each possible state once. Storing the closed schedules takes a lot of space as the search space for the task scheduling problem is exponential on the number of tasks/processors. The trick is to try to store as little data as possible without increasing the algorithm's time complexity. We decided to split up each partial schedule generated by the algorithm into four separate data structures, explained below.

2.3.1 Tasks

Tasks follow the standard format used by all four objects with a split between Original task state objects and Changed task state objects.

2.3.2 Original Tasks

Original tasks store static data and describe the state of the task as it exists before any tasks are scheduled. These data fields are:

1. Task ID: A short value unique amongst the set of tasks.
2. Task Cost: The amount of time it would take for the task to complete once it has been scheduled.
3. Child Links: A set of outgoing communication costs representing how long it would take for the data from this task to reach each child task, assuming they were to be scheduled on a different processor.
4. Max Communication Cost: The longest communication cost from the set above.
5. Bottom Level: The sum cost of each task that exists in the critical path that starts from this task.

Original tasks store all this data, including duplicate data in the form of the Max Communication Cost because their space complexity is linear on the number of tasks, each task can only have one original state.

2.3.3 Changed Tasks

Changed tasks represent the state of a task after one or more of its parent tasks have been scheduled. They store none of the static data from the Originals, instead, they simply store a reference to their corresponding Original and pass any calls for static data onto it. The data they do store is just the set of processor locations where all data from currently scheduled parent tasks are first available, and an integer describing how many unscheduled parents it currently has left. Once that number hits zero, the task is added to the list of free tasks and can be placed into a processor with respect to its stored processor locations.

2.3.4 Processors

Processors objects store a set of Processor objects each representing the current state of a single processor. Processors is a pretty simple class, containing methods allowing for it to be used as the key in a hashmap, and existing to ensure that multiple Schedule states can reuse unchanged Processor objects without causing issues.

2.3.5 Processor

Similar to Task, Processor is divided into Original and Changed. This time the purpose is mostly just to ensure that Processor State objects remain immutable by limiting access to their constructors. Original also has the benefit of short-cutting many of its methods, useful as the Original objects are still used in a large number of partial schedules. Changed Processors store a set of spaces, where each space represents a 0 to x length space between two scheduled tasks. The data regarding the location of inserted tasks is stored this way, rather than storing the task locations themselves because it expedites the calculations required for finding the earliest location to insert a new task. As we decided to implement insertion as a pruning tactic (rather than simply placing new tasks on the end each time), this was the most efficient solution.

2.3.6 Schedules

Schedules are also divided into Original and Changed, for schedules the Original version, effectively a singleton class describing the state of all processors and tasks before any scheduling has happened, stores all the sets of Original Processors/Processor/Task objects. This is used to allow for Changed Schedules, of which an exponential number of are created, to store only the data which differs from the original. Changed Schedules store a set of tasks that are currently free and a set of tasks that aren't free but at least one of their parents has been scheduled. If data regarding any other task is required, then the Changed schedule will retrieve it from the original. As the algorithm doesn't query data regarding tasks that have already been scheduled, Changed schedules also drop these changed tasks once they have been inserted in order to save on space.

2.4 Pruning Techniques

The A* algorithm, on an NP hard problem, explores exponential solutions. In order to reduce the search space and memory consumption and thus increase performance, optimisations are crucial. The following outlines the pruning techniques employed. These help to cut down both the search space, and the memory usage of A*, leading to improvements in performance.

2.4.1 List Scheduling

In order to prune the search space, we employ a list scheduling algorithm. This algorithm gives a very good, but not guaranteed to be optimal, solution. This solution can then be used to prune the partial schedules that are created; if the heuristic value for these schedules is greater than or equal to the list scheduling's final value, they can safely be pruned. This works extremely well when the list scheduling is optimal, which A* proves. Otherwise, it still aids in cutting down unnecessary memory usage which also improves A*'s performance.

2.4.2 Equivalent Children Task Ordering

Equivalent tasks are those defined as having the same:

1. Task weight
2. Parent tasks
3. Children tasks
4. Parent communication weights
5. Children communication weights

If two tasks are equivalent, they can be completely interchanged without affecting the length of the final schedule. However, A* will exhaustively search through all these options. To avoid this, we can force an ordering between these nodes (adding an edge with 0 communication cost), which will reduce these to only being scheduled in one order, thus cutting the search space.

2.4.3 Processor Normalisation

As we are dealing with homogeneous processors, it can be seen that equivalent schedules could differ only by their processor numbers, whilst the final schedule length is the same. For example, (T_1 *on* P_1 & T_2 *on* P_2) or (T_2 *on* $P1$ & T_1 *on* P_2) would be equivalent. To avoid duplicating the expansion of such schedules, we normalise our processor names. This means that only one schedule, say the first in our previous example, would be further expanded, again reducing the search space.

2.4.4 Closed Set Pruning with Hashing

The closed set is the set of partial schedules that we have already explored. In order to check we are not placing a schedule we have already checked into the priority queue, the set is first checked. To make sure this is a constant time operation, schedules are hashed before being added to the set. In the case of a collision, a deep equals is performed before the schedule is added to the closed set, or pruned accordingly.

2.5 Libraries

The hashing function used was Sandeep Gupta's MurmurHash.

3 Parallelisation

3.1 Parellelisation Approach

3.1.1 Work Splitting and Synchronisation

A* expects each self contained partial schedule to be expanded in order, however, with some handling of race conditions, it is possible to use a pool of threads to greedily expand all available schedules. We utilised this method to parallelise the A* algorithm as it causes the program to only explore schedules in the vicinity of the path that a non-parallelised would take. This method has two points of synchronisation, the map of closed schedules, and the queue of fringe schedules. In order to handle race conditions the fringe queue needs to block on both push and pull, whereas the closed map needs to block on write. Finally, to handle the retrieval of the optimal schedule after a complete schedule has been found, all complete schedules currently being expanded must also be collected and then searched through to retrieve the actual optimal solution.

3.1.2 Change in Data Structures

To handle insertion into the thread pool executor our Task objects were made to expand Runnable as well as Comparable. This way Tasks could be polled from the priority queue and sent straight to a waiting thread. To make the rest of the algorithm synchronised we also swapped out our priority queue for a blocking priority queue, the hashmap for a synchronised hash map, and finally a synchronised list was added to handle the final race condition. As all of our basic objects were already immutable no further changes were needed.

3.1.3 Pseudo-code

```
Generate Schedule:
    create a thread pool T
    create a list of schedules L
    create a priority queue Q of tasks
    wrap initial (empty) partial schedule into task O
    insert O into Q

    while true:
        retrieve thread TH from T
        pop top task TA off Q
        Execute task TA on thread T
        if T is shutdown:
            return S, the best schedule stored in L
```

```
Execute:
    retrieve schedule S from task TA
    if S has no free nodes:
        return null
    else:
        create all permutations scheduling a free node on S, List(SNew)
        return SNew

After Execute:
    if return is null:
        retrieve schedule S from task TA
        add schedule S to list L
        initiate shutdown of T
        clear priority queue Q
    else:
        prune returned list (SNew)
        wrap all remaining schedules in SNew into tasks (TNew)
        add all tasks TNew into Q
```

3.2 Parallelisation Technology

As our sequential implementation of A* was already using an implementation of Java's PriorityQueue class we decided to utilise a fairly standard ThreadPoolExecutor. Thread pool executors utilise a priority queue to store tasks for a set number of threads to attempt. We simply used our own priority queue from the sequential implementation and extended the standard Java thread pool executor class to allow for the use of the in built hook methods. Partial states generated by the algorithm were wrapped in objects implementing comparable and runnable and inserted into the task pool. The same objects were then captured by the overridden afterExecute method once analysed by a thread, the generated schedules retrieved and the cycle starting anew.

3.3 Issues

A* relies on the fact that the states generated are expanded in a specific order for it's optimality, this is an issue when using a thread pool as there is no guarantee that a thread which retrieves a partial schedule will finish expanding it before a later thread. In order to solve this issue we collect all completed schedules who were being expanded by threads at the moment the first complete schedule was found, and retrieve the best out of those available. This retains optimality as, while there is no guarantee the early optimal thread will complete first, it is guaranteed that it will have at least started by the time a worse complete schedule is found. This problem was somewhat difficult due to the limited methods provided by the Thread Pool Executor class. None of the available methods for stopping the executor both retained the currently executing threads and removed all other partial schedules. To get around this we implemented our own variation of the shutdown method that correctly changed the executors state to completed without interrupting the current threads.

4 Visualisation

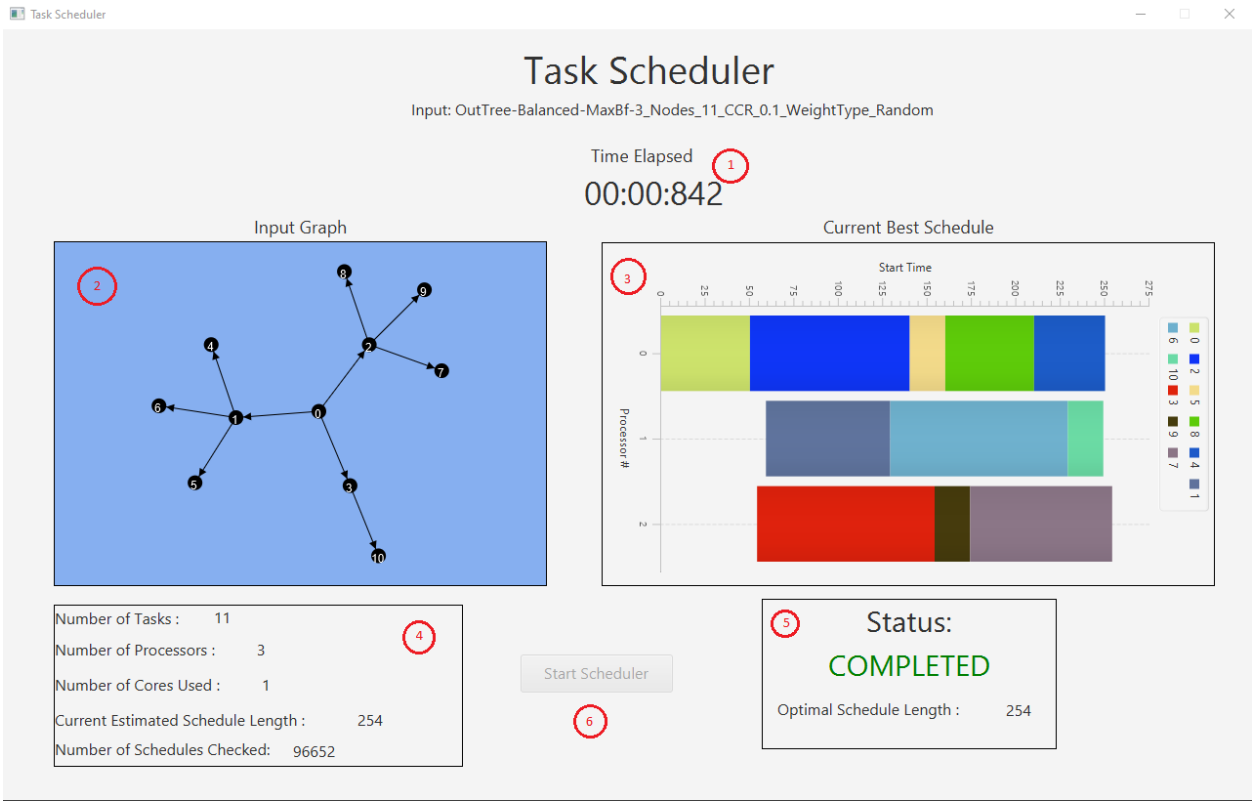
4.1 Concept

In designing the visualisation, we went for a minimalist concept as to try and avoid unnecessary elements from cluttering up the interface. We also decided to apply a sort of 'widget' approach, whereby most elements were put into containers. This makes the interface look and feel more like a 'dashboard' of sorts, which we thought was most appropriate for this sort of application.

4.2 User Interface

The user interface of the scheduler visualisation comes with a number of elements as shown in Figure 1.

The elements of the visualisation, as labeled in Figure 1, are as follows:



Figur 1: Task Scheduler User Interface

1. Timer
2. Input graph visualisation
3. Current best schedule visualisation
4. Scheduler parameters
5. Current algorithm status indicator
6. Start Scheduler Button

Each element will be described in further detail below:

4.2.1 Timer

The timer provides a live timer which shows the current running time of the algorithm. Stops once the algorithm has completed and a complete optimal schedule has been found.

4.2.2 Input graph visualisation

A visualisation of the input graph provided in the DOT file. Each node is labeled with its corresponding name provided in the DOT file. Node and edge weights were omitted from the visualisation as we found that it would often clutter up the display and make it difficult to read the more complex graphs. The nodes automatically spread out to make it easier to read.

4.2.3 Current best schedule visualisation

A visualisation of the current best partial schedule as given by the algorithm. A partial schedule is shown as it is the current 'best' option which the algorithm has found. This partial schedule becomes a complete schedule once the algorithm finds its first complete schedule, and an optimal schedule is displayed at the end once the algorithm terminates. The colour of each 'block' for each task is randomly generated in order to ensure that each block is uniquely coloured to make it easier to identify which tasks are scheduled at what time.

4.2.4 Scheduler parameters

A list of parameters which include the initial input parameters given to the scheduler (Number of Tasks, Number of Processors, Number of Cores Used), along with parameters showing the current progress of the algorithm (Current Estimated Schedule Length, Number of Schedules Checked). The input parameters being displayed helps ensure that the algorithm has been set up to run with the correct parameters, while the current progress parameters help give an idea of the progress and performance of the algorithm.

4.2.5 Current algorithm status indicator

Provides an indication of the current status of the algorithm. The algorithm can be in one of three states:

- WAITING (BLACK): The algorithm has not been started
- SCHEDULING (RED): The algorithm has started, and tasks are being scheduled.
- COMPLETED (GREEN): The algorithm has terminated.

Different colours were used for each state to easily distinguish the three.

4.2.6 Start Scheduler Button

Starts the scheduling algorithm and as a result triggers the other active elements of the interface such as the Current best schedule visualisation, timer, scheduler parameters and the current algorithm status indicator.

4.3 Libraries

The libraries used to implement the visualisation were JavaFX and GraphStream. The bulk of the user interface is implemented using JavaFX. Many of the elements were built using standard JavaFX components and the interface layout was created using SceneBuilder. The current best schedule visualisation uses JavaFX's stacked bar chart with each bar simply being assigned a colour generated using random RGB values to make them distinct. Idle time was included using transparent bars. The only element of the visualization which did not use JavaFX was the input graph visualisation which uses GraphStream's JavaFX plugin instead.

4.4 Difference between sequential and parallel visualizations

The only difference between sequential and parallel visualizations is the number indicated by the 'Number of Cores used' as part of the scheduler parameters. Due to time constraints as well as many issues which were encountered along the way, we did not consider making different views for the two. In hindsight, we would have liked to have included some other ways to differentiate between sequential and parallel visualizations.

5 Testing

Testing was done with two major aspects in mind - Unit Testing and Integration Testing. Due to time constraints everything was not tested, however we identified the key areas of most importance and greatest risks. So, the tests were targeted at those areas.

5.1 Unit Testing

We did Unit Testing for our project using JUnit. Due to the time constraints we did not focus on 100% Branch Coverage, instead focusing on the underlying data structure our application would be built on. In addition to this we tested the Input Output as that is where all the error checking was done. Since this was a standalone application we only added error checking at IO end rather than within different classes due to the time constraint. The testing was done by running the Test Classes on the IDE.

5.1.1 Data Structure Testing

We made our own structure for the graph object used in our project, the algorithm and heuristic both build on top of this. Henceforth, the testing for the graph classes was prioritised. Some of the key classes were tested such as those for TaskNodesänd ScheduleState". We also made some Unit Tests for Processorsänd Processorbut left those out from the final version of the project as some changes were made to the structure towards the end which would be different for those tests.

5.1.2 IO Testing

For the Input Output Testing we tested only the major methods, i.e. the readFile and writeFile methods as they were the most important and would read the data from a file and split it into our graph data structure and vice versa, writing the graph data into a file of the correct format. These were prioritised as they are client requirements, and without them working properly the project is rendered useless.

5.2 Integration Testing

The whole integration testing of the application was done using a bash script after the jar file was built. From the root of the project, the jar was built using:

```
mvn clean package
```

Then, from the root of the project the following command is run:

```
bash testAllGraphs.sh
```

Or

```
./testAllGraphs.sh
```

(Or a variant of the command depending on the shell used).

6 Development Process

6.1 Development Process Description

6.1.1 Initial planning

Our Development Process reflected the Waterfall Model. We initially did a thorough plan of our project, initially debating on different algorithms and deciding what to use. We all agreed that A* was the best choice for the Optimal Schedule; but we could see storage issues occuring by using it. Hence, we then pondered on solutions for that issue. We decided on hashing for storage, however with factors added for uniqueness to avoid duplicate hash values. We also decided to use the List Algorithm to develop the Valid Schedule.

6.1.2 Design

Having decided on our basic approach we worked on the data structure design and algorithm heuristic. We made our own Graph Data Structure and refined it by planning it as a UML class diagram, which allowed us to identify issues early on and rectify them by adjusting the design. Parallel to that we came up with the basic design for our heuristic using A*.

6.1.3 Implementation to Milestone 1

Having done our design we began to implement it. The maven project was setup and put on GitHub (We used git for version control). We worked on 4 major areas, I/O, initial Graph Data Structures, initial A* heuristic, and a valid scheduler using the List Algorithm. We split the work evenly between the different members, usually with one member specialising in a particular area. As we worked we communicated with each other through in-person meetings and online meetings, whilst maintaining good version control practices so that others could understand what we did (where unclear we would just clarify with each other). Working on our individual parts and combining them we managed to get a valid schedule released on time.

6.1.4 Implementation to Milestone 2

We employed a similar approach as before for Milestone 2, however with different areas to focus on. Now we had the finalised A* algorithm (including heuristic and pruning), the finalised Graph Data Structures, the GUI, and testing. We split up these areas like before and worked on different branches. From there with sufficient progress we would merge one branch into another and connect the work and resolve conflicts - finally merging the work into the main branch. After which we made the final release. During this time we also did the Wiki for our respective sections and finished the other necessary documentation for the project.

6.2 Communication and Decision Making

We relied on in person meetings and online platforms for our communication and decision making. When we needed to make a discussion we would set a time for a meeting and discuss then. We used Discord and Messenger to communicate with each other directly. To allocate tasks we used Trello.

6.3 Conflict Resolution

Since we would discuss as a team before making our decisions we did not have as many conflicts. When we did, it would either be when one of us could not understand the other's code, or have merge conflicts when pulling or merging branches. In the former case we would just discuss it, in the latter we would either discuss it, or the person doing the task would take responsibility if they knew the details and manage it, while updating the team - in the case of any issues we would revert and/or recommit.

6.4 Used Tools and Technologies

- TeamGantt - Used for making the Gantt Chart for the project timeline
- Trello - Application used for Task allocation and tracking
- Discord - Application used for communication
- Messenger - Application used for communication
- draw.io - Used for designing the UML Class Diagram
- Java - Development language
- JavaFX - Library used for the GUI
- JUnit - Library used for Unit Testing
- GraphQL - Library used for reading in and printing out the graphs
- Maven - Software Project Management and Comprehension Tool/Built Tool
- Bash - Used to script permutations of the running command for Integration Testing
- IntelliJ IDEA Ultimate Edition - IDE used for development
- GitHub/git - Used for Version Control

6.5 Team Cohesion and Spirit

Overall our team worked very well together and progressed well. The team communicated and coordinated well with each other - we had an understanding of each others time commitments and other responsibilities while also appreciating the fact that we had to communicate with each other and do our individual parts as well. As such, with that mindset our team dynamic was good. One of our members was unwell during the development process and so he could not contribute as much (which is reflected in the contributions section) due to his personal circumstances. This in turn did affect the rest of the team as we were a member down effectively, so 4 people had to do a 5 people job - so it was a stressful situation in both workload and time constraints. However, we managed with each other and worked through it.

7 Contributions

Table of Member Contributions					
Task	**Team Member Contribution (%)**				
	Adam	Henry	Osama	Remus	Syed
Planning	24	24	24	24	4
Algorithm	40	5	5	40	0
Parallelisation	50	0	0	50	0
Visualisation	5	90	0	5	0
Testing	0	0	90	0	10
Percentage of Project	23.8	23.8	23.8	23.8	2.8