AUGUST 26, 2016

# TASK SCHEDULER

## SOFTENG 306 PROJECT 1

Jayden Cooke, Juno Jin, Aditya Nair, Priyankit Singh, and Nathan Situ

# Algorithm

## Data Structures

Central to the algorithm was our own custom NodeObject class. Our specific program could be visualised as a tree, but only conceptually. All the data that each node needed was kept in a node object that could be passed into the recursion, and a new node was generated at each step.

The SolutionTree class was a representation of the current state of the solution. It contained the recursive method and several other fields and methods that were accessed by each recursive call. It also has methods which can be used by outside classes, such as the I/O classes or the visualisation. This was also extended to add in the necessary methods for parallel and visual execution.

The AdjacencyList class was our means of storing the input graph once read. It isn't, strictly speaking, a list, as there was no particular need for ordering in our code, and Maps can be interrogated faster. The actual information was stored in a Map which mapped the node as a String to another Map. The inner Map contained the node's parents as Strings, and the edge weight between them as an int. Each node was also mapped in a separate field to an int, corresponding to the weight of that node. Most construction and lookup operations can be done in constant time, and helper methods avoided the complexity of these structures. Finally, this graph was constructed through an adapter, as it went through many iterations, and the code had to be overhauled.

The UserOptions class was a singleton class that served to hold the arguments the user passed in, for easy access by all classes.

## Algorithm Pseudocode

```
Recursive_method ( current_node, list_of_unseen_nodes)
    If list empty
        Check if less than minimum time
        return
    If current_nodes schedule time < min time
        return
    If bounding value < min time
        return
    Loop through list:
        Check valid options for each processor:
            Add to path
            Recursive method(new_node, updated_list_of_unseen_nodes)
```

The method of finding the optimal path was done using a branch-and-bound, depth-first search in a recursive implementation. The tasks were made into NodeObject that contained the relevant information stored inside to check for the next place to be added to the tree. The algorithm had a base case where if the list of tasks were all done, the end time of the schedule was compared with the global minimum time and if it was lower than it, the schedule would be added as the new best and the minimum time value, updated.

If it was not the end case where all the tasks have been completed, but the time to do the current partial schedule was already greater than a previously completed schedule the node on the tree was returned to its parent and the recursive method carried again.

The next step in the branch-and-bound DFS was checking whether a task was valid due to the precedence constraint, and if so would be placed on a processor that is not cut due to the pruning technique (mentioned below). The new task information would be added to a new NodeObject, the previous task added to the path and the recursion repeated.

### Bound and Heuristic Functions/Pruning Techniques

To assist in the traversal of the solution tree, we implemented a pruning technique that checked whether if there were multiple processors left idle when creating the tree, it would lead to duplicate schedules. So for a task on a number of different processors, if there were at least two processors empty, placing it in either of those would lead to an equivalent subtree resulting in duplicate search space. This was one method we reduced the overall time to run the algorithm.

A lower bound is used to prune the tree and look at fewer nodes, leading to a faster execution time. The lower bound is calculated by taking the lowest time of all processors at the current node and adding all tasks not yet executed and dividing it by the number of processors. The processor with the highest time is also subtracted because the time difference between the lowest and highest time processors has already been used. We assume that all future tasks are distributed evenly among all the processors. We take the processor with the lowest time and subtract it from the processor with the highest time as opposed to computing the idle time because calculating the idle time is more computationally intensive than our formula. This method gets a lower bound that is slightly lower than the actual lower bound but involves a lower number of computations.

## Parallelisation

### Technology Used for Parallelisation

ParaTask (http://parallel.auckland.ac.nz/ParallelIT/PT_About.html) was used for parallelisation of tasks. This library was selected because it models tasks rather than threads, which means that it is easier to code compared to threading model. Furthermore, it allows applications to automatically scale when executing on multi-core processors which means it allows better performance. This library also allows you to wait for a group of tasks rather than a single task. This made scheduling very easy as it provided a much better mechanism for handling dependencies. We tried java's threading class but it did not handle transitive dependencies very well which this library had no problem in handling.

### Paratask

ParaTask creates a lightweight task object which encapsulates independent code within itself.[1] In our program, this task object contained the recursive calls that carried out the depth-first search. This leads to the subtree to be searched independently of the rest of the tree.

The scheduling was done by using ParaTask's TaskIdGroup object which lets you wait for multiple tasks. We used this object to wait for all tasks that a particular node called once that node has finished its recursive search.

### Approach to Parallelisation

When parallelising our search algorithm, we decided to follow the basic structure of the recursive depth-first search algorithm. The final parallel algorithm works the same way as the sequential tree traversal algorithm except that it delegates the traversal of subtrees to separate tasks, in essence becoming a hybrid of depth- and breadth-first search. At any given node, if the maximum number of tasks which can be created by the process has not been created, the program creates a new task which traverses the subtree of one of the nodes children.

---

[1] *http://homepages.engineering.auckland.ac.nz/~ngia003/files/thesis_giacaman.pdf*

The Paratask library was used to create subtasks which worked in parallel to the main thread. The tasks were dynamically created and allocated using a semaphore which kept track of how many new tasks could be produced. The semaphore value was decremented whenever a new task was created and incremented at the end of a task. No new tasks were created when the semaphore was zero to avoid creating too many tasks. This also ensured that the program would create a new task whenever possible.

When implementing our algorithm, no data structures needed to be changed as the ParallelSearchTree class extended our sequential class. This meant that it used the variables from its parent class, which were sufficiently thread-safe.

## Pseudocode

```
//Same as beginning of sequential recursive
    // create new TaskIDGroup
    TaskIDGroup group
    Loop through list:
        Check valid options for each processor:
            If semaphore = 0
                // Do subtree in new task
                Create new task
                // Add task to TaskIDGroup
                Recursive_method_parallel (node, list of new unseen nodes);
            Else
                // Do subtree in current task
                Recursive_method(node, list of new unseen nodes);
            Add to path
// Wait for all tasks in the task group
group.wait
TASK Recursive method parallel(current_node, list_of_unseen_nodes)
    Semaphore -- // Decrease semaphore value
    RecursiveMethod(current_node, list_of_unseen_nodes)
    Semaphore ++ // Increase semaphore back to original value when task finishes
```

## Semaphore Explanation

The parallel system makes use of a semaphore to control the generation of threads. A semaphore is a variable that is used to control access to a resource in a concurrent system. In this program, the semaphore is a simple global variable in the in the parallel tree class and it is used to control the number of tasks created by the program. The value of this variable is set to the number of cores in the host system. Whenever a new task is created, the value of the global semaphore is decreased by one and no new tasks are created if the semaphore has a value of 0. When a task finishes execution, the semaphore value is incremented by 1 to allow the creation of more tasks dynamically. By using this approach, at each point in time, a maximal number of tasks are running leading to faster execution.

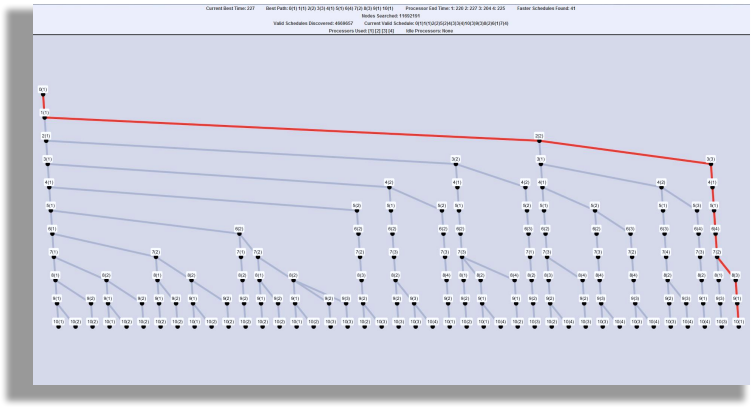
## About the overhead

Creating new tasks using ParaTask has an overhead involved. This leads to a small amount of speedup using parallel for smaller input graphs because the ratio of time saved using multiple processors and the time wasted on excess computation is small. This ratio increases for larger graphs which yields a significantly faster time compared to sequential search. This object contains a

waitTillFinish() method which allows us to wait for a Task nested within another Task to finish doing its recursive search. This is to prevent race conditions which would break our visual and parallel search.

# Visualisation

## Concept



The aim of visualizing the program was to display something that was easy to understand for the client, while also displaying that the program is calculating the paths of many different schedules live. While the full solution tree was an option, in large input graphs the solution tree became so large that valuable information was hard to recognize. Showing the input DAG was an option, but was quickly found to be dull as what was shown were just changes in the states of the nodes (tasks). The idea was to use a view similar to a pruned tree of the solution tree that only showed key and useful information to the client. The pruned solution tree solves this issue by displaying the full path in an easy to understand format while keeping some of the important parts of the graph visible. This was further assisted through the use of statistics at the top of the graph visualisation.

## Implementation of GraphStream Library

GraphStream - http://graphstream-project.org/

This Java library is used for modeling dynamic graphs and used for all of the graphs generated in visualisation. The GraphStream library was used to display the best time tree. The graph object bestTimeTree was created in GraphVisualizer.java, where it can be added into a swing JPanel to be displayed. The nodes are created in the updateGraph method which is called when the tree finds a new best time. The program retrieves the best path, and iterates through it, creating nodes (graph.addNode()) for bestTimeTree each time (skipping the root node). Then each node that was created is connected with a red edge (graph.addEdge()) to indicate it as the best path. When updateGraph is called again, the edge colors of the graph are reset.

The statistics were implemented simply by retrieving information from the program as it runs and updating the labels using JLabel.setText().

## Displayed Information

Statistics: Information pertaining to the program and how it runs through valid solutions in the graph were displayed here. The statistics section of the visualisation shows:

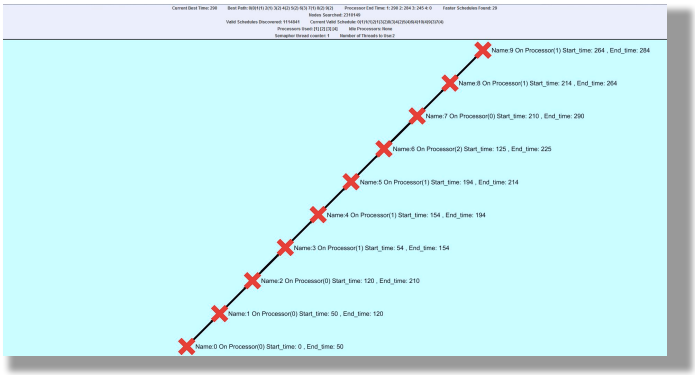
- Counters for nodes, valid schedules, and schedules that beat the previous best.
- The most recently discovered valid and best schedule.
- The time to run through, and the times on each processor for the current best schedule.
- The processors that are used and the processors that are idle for the current schedule.

Best Time Tree: A partial solution tree that only shows the schedules that result in a faster time than the previous best schedule. This means that relevant information, such as the previous best times can be seen in an easy to read manner. Each node of this tree is labeled by the name of the node on the best current path, followed by the processor number. The current best path is highlighted with a red line. The tree splits off to the right every time the program discovers a better schedule. When the

program runs through the entire solution tree, the background changes to green to indicate that the program has completed.

## Changes from Sequential to Parallel

The statistics used in sequential worked well in parallel as well as they showed relevant information on the paths that are being traversed in the tree. Additionally, two labels were added, which were the number of threads to use based on the user input, and the number of idle threads. This number was based on the semaphore value which decrements whenever a new thread is assigned. If the value was 0, it meant that we were using the correct number of threads for parallelisation.




Changing from sequential to parallel meant that we had to simplify the visualisation of the traversal. This was due to the fact that the GraphStream library is not thread-safe. The original design concept was to use sequential implementation for visual and let it run in parallel but include the root node in case there were multiple source nodes. What would be displayed is multiple points leaving at different times and creating a best possible schedule based on the section of the tree a thread was working on. This worked sometimes, but due to graph streams own exceptions and not being thread-safe we had to change this close to the deadline.

In the parallel visualisation, it only shows the current best path and updates the line with stats right next to the node. We displayed the task name, the processor the task is currently running on, the start time for that task on that processor and its end time on that processor. These values updated as a newer and better path was found in the solution tree. This was much more simplified that the sequential display, but if we had additional time and a better understanding of GraphStreams library, the original sequential concept could have been applied and working properly.

# Testing

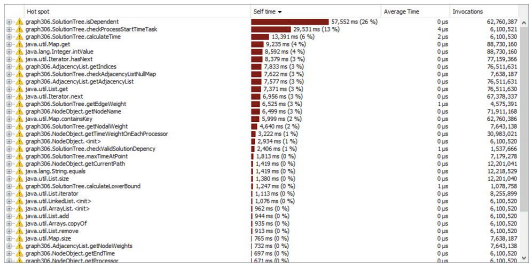
## Data Tested

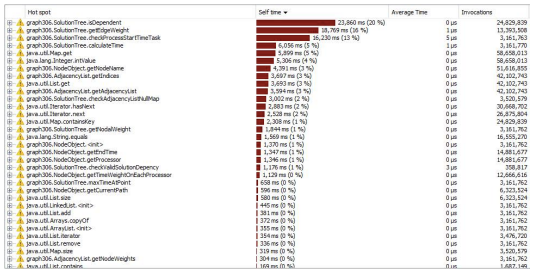
Testing came in two distinct phases: before and after the delivery of the first milestone. Prior to the first milestone, the testing concentrated on generating valid schedules and ensuring schedules are correct. Once we had an implementation that was working, the goal became to ensure speed without losing correctness.

In the first phase, the optimal schedules for a few small graphs were calculated by hand, and these graphs were written into .dot files to be run through the program. During this stage, the state of the tree at all points in the algorithm was observed, and the final schedule. The state of the tree could be used for debugging, and the final could be used to compare and verify validity.

In the second phase, we measured the runtime of a program when it was run on the same computer as an old version, but for more objective testing of techniques, we measured the number of nodes checked, or the number of times a method was called, with respect to the average time it spent running. Additionally, we would output the schedule generated, to verify that it was still optimal. (By this point, we had calculated definite optimal schedules for all of our test graphs using milestone 1 code.)

## Means of Testing





The means of testing during the validity phase were predominantly JUnit tests. At this stage, the code would use the GraphStream library (it was later removed), which has tools for comparing graphs, so by hard coding a correct graph, we could verify that the generated graph was correct. The output schedule could be printed out for visual comparison or tested with JUnit assertion.

During the speed phase, a copy of the latest working version was saved into another package. This allowed us to automatically (Again, using JUnit) test whether the old, correct code generated the same schedule as the new, experimental code. This also allowed us to verify that fewer nodes were checked and that the runtime had decreased. Around this point, we also started making use of JProfiler (output before and after optimisation included in figures) to identify methods that were taking up large amounts of runtime. In some cases, these were constant time methods, and so reducing their runtime was a case of optimising data structures, while algorithmic changes were possible for some of the larger methods. Near to the end of the project, however, we opted to simply print out statistics as changes became smaller and more incremental, so the effort required to update the JUnit tests and backup packages exceeded the benefits of automatically testing using them. Furthermore, at this stage, different branches were being used on Github, so the master branch could be used for comparison.

# Development Process

## Processes

The well-defined scope and limited time frame of this project lent itself well to a waterfall-style development, so initial planning, and then design, implementation, and testing loosely summed up our development process. However, we applied more iterative development processes as well, determining a task and a timeframe (such as “valid schedule algorithm by Sunday”), then building an entire feature, testing it and repeating the process for the next feature.

## Communication and Decision-Making

The size of the group meant that it was feasible for all decisions to be made as a team, face to face, rather than through any external medium. This allowed for atypical group management strategies.

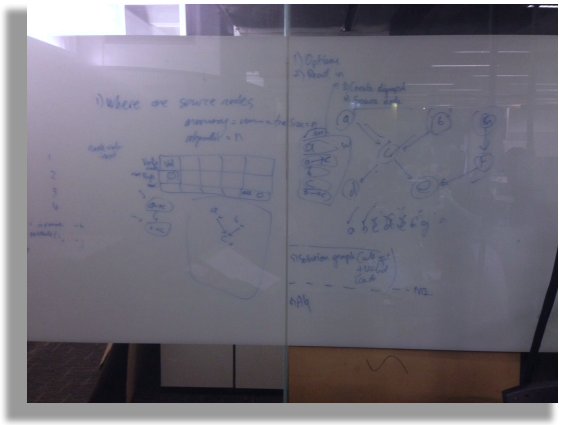
Decisions were made usually by those responsible for the task at hand. A visualisation problem, for example, would be handled by the visualisation team. For more consequential decisions, such as a decision to change the overall project structure, the group would all be called, the situation explained, and a decision made as a team. As the entire group was fairly well informed on all aspects of the code, all decisions were largely group decisions, and there was a fair degree of trust in one another’s abilities.

## Conflict Resolution

Conflicts arose rather frequently near the end of the project. In many cases, they were over code that could be implemented quickly, and so the solution was to build both options and see which worked best.  In the case of larger conflicts, a case-by-case basis was applied, one instance was settled with a vote, but largely, a group member who had not weighed in on the conflict would typically act as a

mediator, ensuring both members were heard and developing a solution from both suggestions, or rejecting one, as appropriate.

## Tools and Technologies Used



Meetings were arranged through Facebook messenger, but all formal code communication was done in person. Planning was always done on a whiteboard before any coding was actually started.

While the early stages considered the use of Github Issues, Trello, and Asana for communication, these proved to take more time to update than they saved in actual communication, as everyone was in one place for most of the project and could coordinate directly. Google Drive was used initially for file storage, but storing was moved to GitHub and Drive was used only to edit Google docs.

As alluded to before, our code was shared using Github on our own various Git apps. (Github desktop was the most common, but IDE integration and GitKraken also were used). Our code was developed on either IntelliJ or Eclipse, the latter proving to have more useful plugins, namely the ParallelTask plugin. Some challenges were posed by the differing IDEs and operating systems (OSX and Windows) that the code was developed on, but as Java is largely system-independent, these were easily worked through.

## Team Cohesion and Spirit

As a group, team spirit was high throughout most of the project. Everyone attended meetings frequently and worked at their hardest to complete each task that lay ahead. Planning and trying to understand what we were about to implement was done together and in front of a whiteboard so that everyone had a good understanding of what they are required to accomplish.

As a result of both pair programming and working closely, a strong team environment was created which encouraged spirits. No individual in the group felt uncomfortable in this group environment and weren't afraid to bounce ideas off of each other, which contributed a lot towards our team cohesion. As the deadline approached, most team members were exhausted and stressed, but through frequent breaks and encouragement from the team we were able to pull off quite a lot of work. There were times near the end where as a group we felt we were unable to deliver a certain functionality, but we managed to get it done near the end of the day. This rise and fall of team spirit built up a solid cohesion among the team, which assisted us greatly in being able to deliver the final project.

## Percentage Contribution

| | Jayden | Juno | Aditya | Priyankit | Nathan | Total % |
|---|---|---|---|---|---|---|
| **Planning** | 20 | 20 | 20 | 20 | 20 | 100 |
| **Administration** | 15 | 30 | 25 | 15 | 15 | 100 |
| **Testing** | 40 | 15 | 15 | 15 | 15 | 100 |
| **Research** | 20 | 20 | 20 | 20 | 20 | 100 |
| **Algorithm** | 15 | 15 | 25 | 25 | 20 | 100 |
| **Optimisation** | 20 | 20 | 20 | 20 | 20 | 100 |
| **I/O** | 15 | 15 | 20 | 25 | 25 | 100 |
| **Sequential** | 20 | 20 | 20 | 20 | 20 | 100 |
| **Parallel** | 25 | 10 | 15 | 25 | 25 | 100 |
| **Visualisation** | 15 | 30 | 25 | 15 | 15 | 100 |
| **Paravisual** | 10 | 20 | 20 | 25 | 25 | 100 |