SOFTENG306

Using AI and processing power to solve task scheduling problem

Dept. of Electrical and Computer Engineering

Amy Liu

Kelvin Lau

Vincent Nio

Jack Wong

Johnny Pham

1.0 Algorithm

1.1 Iterative Deepening A* Algorithm (IDAStar)

The chosen algorithm is Iterative Deepening A* (IDA*). IDA* was chosen for its advantages of not only having speed as fast (or faster) as A*, but also would perform under limited memory conditions. As a result of this, a drawback of the algorithm is that it would have expanded already explored states. The pseudo-code for this algorithm is shown below:

1.1.1 Pseudo-code

```
procedure IDAStar
    for all available nodes
        get initial fCutOff of current node
        while not solved
            isSolved = buildTree(availableNode, 1)
            set next fCutOff value

boolean function buildTree(node, procNumber)

    f = maximum of cost functions

    if f is greater than currentCutOff
        if f is greater than nextCutOff
            nextCutOff = f
        return false

    else
        push node to processor stack
        set node start time and procNumber
        set node availability of node to false
        set children's availability to true if all dependencies are resolved

        if node is leaf AND no more available
            return true

        else
            for all available nodes
                for i from 1 to procNum
                    isSolved = buildTree (availableNode, procNum)
                    break out of loops if isSolved

            pop node from processor stack
            set children's availability to false
            set node's availability to true

            return isSolved
```

1.1.2 Cost Function

The cost function for a node was taking as the max from two of the following (where both are underestimates):
a) Start time plus the bottom level (longest path from that current node to leaf) b) Load balance which

is the total computation time (weights) of all nodes plus the total idle time divided by the number of processors
The reason for using the maximum of the cost functions is the same reason as the A* algorithm - an underestimate guarantees the optimal solution, and an accurate heuristic/cost function (i.e. close to the actual cost) will reduce the search time significantly. By taking the maximum of the underestimates, we guarantee the optimality, but reduce the search time as much as possible.

1.1.3 Data Structures

One of the main classes used was the Node class which stored information about the node such as its parent, children, weight, start time and finish time. A hash map was used to store communication costs between a node and its parent. An ArrayList was used to store all the nodes which effectively represented the input; a Directed Acyclic Graph (DAG) of tasks. ArrayLists were also used to store vital information like the list of nodes already scheduled, and the next reachable nodes. In addition, a stack of nodes were also stored in an ArrayList for each processor so that we could keep track of nodes and get the current latest finish times for each processor. ArrayList were chosen for their ease of implementation and use (easy to traverse). Nodes in the ArrayList were matched via their respective indices.

1.1.4 Pruning Techniques

One of the main classes used was the Node class which stored information about the node such as its parent, children, weight, start time and finish time. A hash map was used to store communication costs between a node and its parent. An ArrayList was used to store all the nodes which effectively represented the input; a Directed Acyclic Graph (DAG) of tasks. ArrayLists were also used to store vital information like the list of nodes already scheduled, and the next reachable nodes. In addition, a stack of nodes were also stored in an ArrayList for each processor so that we could keep track of nodes and get the current latest finish times for each processor. ArrayList were chosen for their ease of implementation and use (easy to traverse). Nodes in the ArrayList were matched via their respective indices.

1.2 Brute Force

1.2.1 Pseudo-Code

We decided to develop another algorithm other than ida* in order to make sure the schedule produced from ida* is optimal. Because it is just used for testing purpose, the performance of this algorithm is worse than ida* algorithm. This algorithm is consisted in three parts: Order of the node, allocation of the node to the processor, and calculation of the finish time of a specific path.

1. Order of the node: The valid order of the nodes are pre-determined in the InputProcessor class. The InputProcessor class handles the input file and generate an ArrayList that keeps the order of each node in the input file. Node class is created to store the information of a node including node name, node weight, and communication cost between nodes. The output ArrayList is used for allocation.

2. Allocation of the node to the processor. This method is created to generate the position of a node. At the beginning, every node in the list set the processor number to 1. An allocation path is generated from this method and store in an array called currProcAlloc. For example, if the list is consisted of 5 nodes, then a potential path would be 11211. If currProcAlloc[index] > number of processor, currProcAlloc[index] is set to 1 and currProcAlloc[index-1]+=1. The currProcAlloc is then used to set the processor of a node inside the list.

3. Calculation of the finish time of the current path. At the beginning, set the starting time of the node to the finishing time of the processor allocated. Check to see for any dependency and for every parent node, calculate the latest time the node could start based on the communication cost. If (currentNode.processor != parent.processor), currentNode.startTime = parent.finishTime + communication cost If the finishing time of the node exceeds the final finish time, break the loop and start another path. Continue until all possible path is consided and the final finish time will be optimal.

2.0 Parallelisation

2.1 Our Approach

Our approach: The algorithms considered for parallelisation is Distributed Tree Search and Parallel Window Search. Distributed Tree search is a cooperative relationship between threads. The idea is to minimise the idle time of the threads and it starts by one thread doing all the work. As it performs expansions, it communicates to the other processes that it has too much work, which is when the other idle threads would step in. This approach would have been much harder to implement given our initial implementation: to delegate the workload, a thread would have to keep on providing an instance of the current DAG and its related scheduling at that point in the search. Therefore, the final algorithm used for parallelisation is Parallel Window Search for its simplicity. The idea is that each processor is assigned its own search tree, but they each explore up to different bounds. A disadvantage of this approach is that in conjunction with the IDA* algorithm (where memory is conserved), threads exploring high bounds would have essentially repeated the same search already done by threads exploring lower bounds.
There were no changes to data structures apart from the addition of the priority queue of cost function cut off values (fCutOff values, a data structure to be shared to communicate by all threads, further explained in the implementation section).
The pseudocode for Parallel Window Search is below:

```
procedure IDAStarParallel
    for all available nodes
        get initial fCutOff of current node
        while not solved
            isSolved = buildTreeParallel(availableNode, 1)
            get unique and new nextCutOff from fCutOffQueue

function buildTreeParallel(node, procNumber)
    check if thread should stop this search
    the rest is similar to buildTree but add all fCutOffs to fCutOffQueue

main
    spawn number of threads using IDAStarParallel
    run threads
    wait for threads to finish
    get thread with best schedule
```

2.2 Technology

Parallelisation was done by using Java threads. Java threads were decided over the the suggested libraries (ParaTask, Pyjama) as there is a larger support base.

2.3 Implementation

The IDA* class implements the scheduler interface which extends the Runnable interface. The override run method calls schedule, similarly to the single threaded version. For the number of threads specified, each thread is assigned its own copy of the IDA* class. There is a priority (blocking) queue of cost function cut off values (fCutOff) shared and accessed by all threads. Each thread takes a fCutOff value from the queue and performs the IDA* search up to that assigned value. The fCutOff values are unique and only ever searched once and only once, and that is checked with a history of fCutOffs

3.0 Visualisation

3.1 Concept

Using visualization helps us to test our algorithm and shows representative information to the user to better understand our algorithm. There are two main components in the visualization: animated dag and processor graph.

3.2 The display

Animated Dag: This graph shows a dag representation of the input file. The root node is set to green, which gives the user a reference point of where the start of the schedule is. During the scheduling process, whenever a node is visited, the frequency would increase and hence the colour of the node would become darker. The current visiting node would change to yellow which would help showing the visiting path.

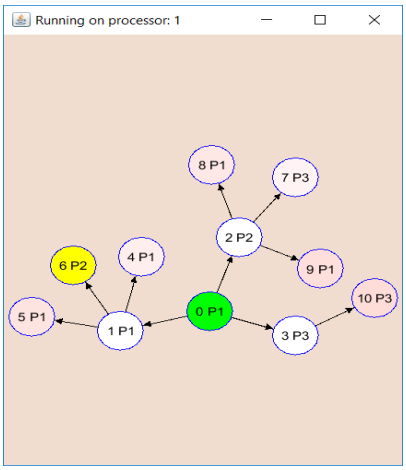

Figure 1. Visualisation

Processor graph: After the scheduling is finished, a processor graph would shows the optimal schedule. There are two main components shown on the processor graph: allocation of the node and the finishing time of the schedule. User can easily see how the nodes are allocated to different processor. User could also toggle each nodes to check out the starting time and finishing time of each nodes.

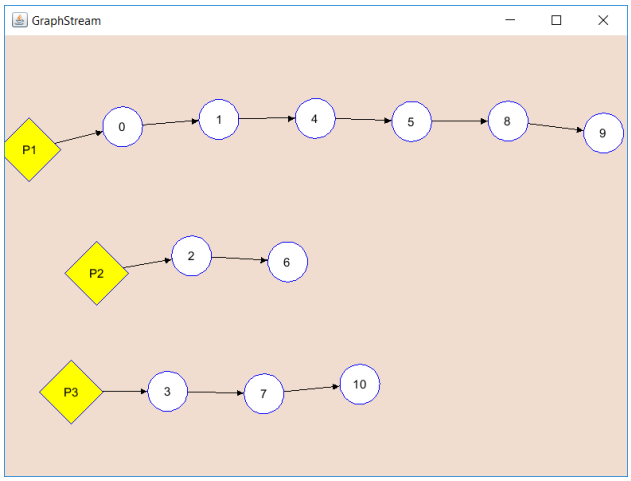

Figure 2. Processor allocation visualisation

3.3 Implementation

We decided to use GraphStream library to develop our visualization because it is a well-supported library for visualization purpose. It is easy to use and reduced the time cost of developing our own GUI. We define the structure and behaviour of the graph in java and define the style of the graph such as node shape and colour in a cascading style sheet.

3.4 Difference between sequential and parallel visualisation

When the program runs sequentially a single window is created. It contains an animated dag, which continually updates during the scheduling process.
When the program runs in parallel, multiple windows appear during scheduling. Each window contains an animated dag, where each dag is updated by a specific thread. Each thread can be distinguished by the windows name.
After the scheduling process the processor graph is shown, which is the same for both sequential and parallel visualization.

4.0 Testing

4.1 Tested modules

In the designing of the software architecture phase, the implementation was broken down into different modules to be developed in isolation enforced by encapsulation principle. Each module will produce an explicitly specified output. This mean that debugging is easy and tests can be done on each module in isolation. Every module was tested right after its completion to find defects and to ensure that it's functioning.

The modules being tested includes:

- Input processor
- Output processor
- IDAStar algorithm implementation
- Brute force algorithm
- Visualisation
- Parallelisation.

4.2 Testing implementation

We mainly used two methods of testing which are black-box and white-box testing. Black-box testing is used to test a module as a whole to assess its functionalities. All black-box tests are constructed similarly, begin by initiate an object of a module and provide inputs to that object. Invoke the method in that object, get the output and compare it with the expected output. We use Junit to automate the black-box test cases. This also allows every team member to tests the functionalities of the modules that they are not working on.

Example:

```
InputProcessor ip = new InputProcessor(args[]);
ip.processInput();
IDAStar s = new IDAStar(ip.getArg())
s.schedule();
int finishTime = s.getFinishTime();
assertEquals(finishTime, expectedFinishTime);
```

We used white box testing for casual testing as the code being written, that means the testing is interleaved with the implementation. The basic criteria for white-box testing is full statement coverage. This helped us in debugging testing the logic of the code, especially with conditional statements.

5.0 Development Process

5.1 Process

During the start of the week a team meeting is held. At the start of the meeting each member discusses what they have been doing, which includes their implementation(s) or issue(s) that may have occurred.
The Gantt chart is then consulted. This gives the team an indication of how fast they have been working and may require them to increase their work load for the week. The network diagram and work breakdown structure are then used to create tasks. For certain tasks such as scheduling algorithm(s), visualisation and parallelisation the team is broken up into groups to conduct research. The team then meets up later in the week to compare notes and choose the best approach on how to implement a task. The tasks are then allocated to each member, where each member is given an equal amount of work.
At the end of the meeting a deadline is set on when the tasks should be implemented, which will coincide with the next group meeting. This process is then repeated.

5.2 Communication

Team meetings and face to face communication was a large source of communication during the start of the project. This was due to the large amount of cooperative work such as planning and research that was required. During this phase of the project the team would share their research, which would then lead us to choose a solution a member proposed. This would be done as a vote.
During the later stages of the project communication moved online. Team member were confident in the tasks that they were required to implement. This was due to careful planning, which lead each member to understand the program as a whole. Post's on Facebook where used as a source of communication, where each team member would post what they are currently doing and tasks they have finished.
GitHub issues where used to communicate bugs that occurred in our program. The team member who was responsible for the bug would then be required to fix it.

5.3 Conflict resolution

There were very few conflicts within the group. Conflicts mainly arose when the team had to choose a certain method of implementation. These cases were prevalent during our choice of algorithm and choice of visualisation.
The largest conflict within the group was choosing what to visualise during the running of the program. We had many ideas within the group, which included a Gantt chart representation, Dag representation and Search Tree representation. The group was split between a Dag and Gantt chart representation. The conflict was resolved when we could not find any useful libraries for a Gantt chart representation. As a compromise we added a processor graph at the end to show how the nodes are scheduled.

5.4 Tools and technologies

Facebook was used as a source of real time communication. Post where used as announcement and general information of what each team member is currently doing. Even though this might not be a popular communication platform in a professional workplace. We still decided think that it is great for setup meeting and reaching certain member in case of urgency.
GitHub issues where use to communicate bug in the code. The team member who was responsible for the bug would be then responsible for correcting it.
GitHub wiki was used as our source of documentation. The wiki would be regularly updated to show our planning, pseudo code and other admin. Team members were responsible for documenting their tasks.

5.5 Team spirit

The team overall performed amazingly together. Each member was eager and willing to work cooperatively. Each member showed enthusiasm and a willingness to give 100% to the project. Members participated in meetings and group discussions, where each member shared their thoughts and ideas on the program. This created a space for larger more engaging discussions within the team.

6.0 Contribution

	Amy Liu	Kevin Lau	Vincent Nio	Jack Wong	Johnny Pham
Planning	20%	20%	20%	20%	20%
Design and Architecture	20%	20%	20%	20%	20%
Input and output	5%	5%	35%	5%	50%
Branch and bound/ BF	5%	5%	25%	40%	25%
IDASTAR	42.5%	42.5%	5%	5%	5%
Parallelisation	42.5%	42.5%	5%	5%	5%
Visualisation	5%	5%	30%	45%	15%
Testing	20%	20%	20%	20%	20%
Documentation	20%	20%	20%	20%	20%

Table 1. Contribution