Task Scheduling for 2 Processors with Weighted Nodes

Ryan Rosiak and Grant Dawson

May 2020

1 Abstract

As a finished program, we achieved a class that would find the most efficient path/order of doing jobs that would take the least amount of time as possible. This program will display the order of jobs that need to be completed at specified time intervals for a job to be completed in the smallest amount of time possible. We tailored our solution to work within the means of task scheduling for computer processors and we restricted ourselves to two processors to work with. Our program looks at the weights of the nodes and schedules them to be in a minimum order. Our processor 1 normally takes more data than processor 2 because that is what we default to when a choice between either processor can be made. Overall, our results show that there are a lot of complexities to this problem. There are many edge cases to be looked at as well as many variations to how the data can be displayed and sorted.

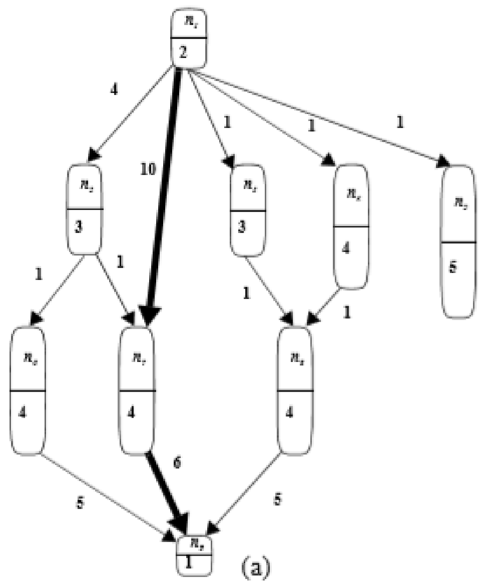
2 Introduction

Our motivation for this developmental research project was to create a program that would find the most efficient order of doing jobs that would take the least amount of time as possible. This, in the end, will give a detailed description of what each processor/worker should do. A great example of job scheduling is scheduling jobs for a CPU to complete. This allows for the CPU to efficiently run through all listed jobs. Another application that this could be used for is construction. The construction of a house can be split up into multiple jobs that all rely on one another. These dependencies form a graph. The graph cannot have any loops of dependencies because it must be a DAG (Directed Acyclic Graph). An example of a graph that would have a cyclic element would be if all the walls on a house being built required the other one's walls to be up. This is an infinite loop because you can never fulfill all the dependencies to do one wall. But, assuming that we have a DAG graph, we can find the most efficient path to completing all nodes based on a given number of "workers/processors." Workers refers to the number of people on a construction site.

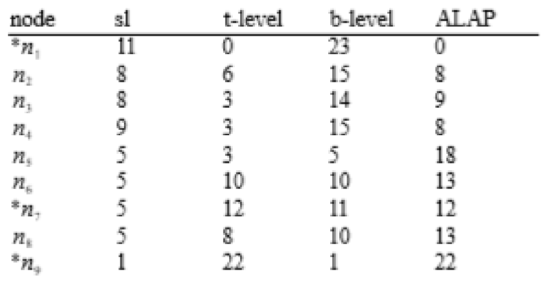
Processors refers to the number of threads accessible in a CPU. The program will take the given graph and tell each processor what job it will do in what specific order which will be the fastest ordering of the tasks. The analytics will be displayed on the screen. So far we have gotten everything we have needed from three major research papers, “Static Scheduling Algorithms for Allocating Directed Task Graphs to Multiprocessors”[2], “How to Design a Job Scheduling Algorithm”[3] and ”Low Performance Effective Task Scheduling Algorithm for Heterogeneous Computing Environments”[1]. All three papers have allowed us to gain a basic knowledge of the parameters and specifications that are involved in creating a task scheduling algorithm as well as some functions that can be used to implement basic forms of the overlying problem. Modern applications of the task scheduling problem involve finding very fast and reproducible algorithms for very intricate types of graphs such as weighted/non-weighted or tree structured versus a DAG. Most of these algorithms focus on cutting out a lot of nasty edge cases and focusing on very specific applications that are much easier to solve[3]. We are doing a general approach in hopes to cover a lot of the common edge cases in one single run.

3 Topic Details

The overall goal for this topic is to find the minimal ordering of tasks to complete a job. Our overall approach to the issue is to use the path-finding algorithm Breadth-First-Search (O(V + E)) to compute the the shortest distances from the starting tasks. We group these distances into levels that we will go through from lowest to highest, scheduling each node in its specified spots. Since we are restricted to two processors, we need to figure out when and where it is applicable to move a node to processor 2 as well as when nodes can be completed alongside each other. These cases are accounted for by the use of metadata that is calculated for each node before the algorithm even starts it’s first loop. This metadata includes the T-level, B-level, and ALAP of a node. The T-level of a node is the shortest path weight to a start node not including the weight of the node we are calculating the T-level for. This gives us what is known as the ”As Soon As Possible”[2] time which is used to tell when is the earliest time a node in the graph can be started in the scheduling process. The B-level of a node is the shortest path to an exit node in the graph including the weight of the node we are calculating the B-level for[1]. The is one of the most important pieces of information because it provides us with a constant piece of data. As the nodes in the graph are scheduled, the B-level of a node will never change. This provides us some structure when moving down the tree. The ALAP of the node is known as the ”As Late As Possible”[2] time which is used to tell when the latest a node can be scheduled without changing the increasing the minimum time for the ordering of the scheduled tasks. This data is very important in choosing between certain nodes to be completed before one another. For example, if we have two nodes that need to be scheduled but one node has a greater ALAP than the other, than we will schedule the one with the lower ALAP because we have more time before the higher ALAP node has to be scheduled.



(a)



node	sl	t-level	b-level	ALAP
$*n_1$	11	0	23	0
n_2	8	6	15	8
n_3	8	3	14	9
n_4	9	3	15	8
n_5	5	3	5	18
n_6	5	10	10	13
$*n_7$	5	12	11	12
n_8	5	8	10	13
$*n_9$	1	22	1	22

(b)

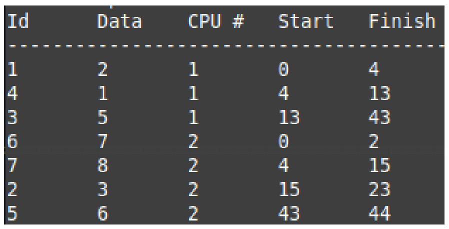
Overall, all three of the algorithms for calculating the metadata for each node run O(V + E).[2]

This is because a path must be calculated either going up or down the tree looking at the node weights on the way. All of this metadata is then used in our calculations as we schedule node by node to each processor. In order for us to figure out the choice of processor, we default the first job to always go to processor 1. We then find the next processor that will be available to start a new job first which can be either processor depending on what tasks in the beginning are scheduled to each processor. We then assign that next task to the processor that is available and repeat the process going through each node. Last, we keep a global time that allows us to keep track of what is being done at what times during the scheduling process. The times on each processor are evened out when moving from level to level because we know that whatever nodes are placed next, must be scheduled when the level above it is completely done. We use this metadata, the BFS algorithm, and a global "clock" to keep track of what time each node will be processed and in the correct minimal ordering. Our complete algorithm takes time O(L * 4 * (V + E)), L equal to the amount of levels in the graph that we need to scan to assign suitable processors to, because we have to BFS the graph and acquire all of the metadata for the graph which will take 4 * (V + E). We then will traverse V + E times the amount of levels that we have in the graph. This is because we have to BFS and recalculate some metadata going forward as we start scheduling and removing nodes from the graph.

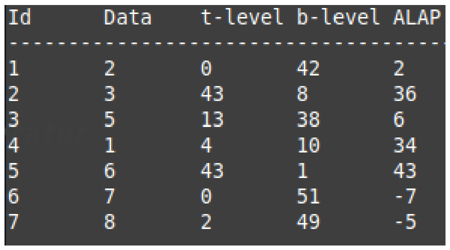
4 Conclusion

As per our results, the task scheduling problem was not easily completed. Some special edge cases arose when implementing our algorithm that we were not able to solve. The big edge case of this algorithm is that when we zero out the times of both of the processors, we run the risk that there will be a large gap of empty space on a processor when it could potentially be working on another task from the next level. This edge case could not be solved because we had to make sure that every graph satisfied the fact that a level before another must have been completed before we started the next level. This is not always a 100 percent true statement all the time, but this was the logic that our algorithm followed. This, along with many other weird edge cases and possibilities are why the task scheduling problem is so hard to complete. There is no known polynomial time algorithm that can solve the task scheduling problem for any situation. This is why the problem is broken down into several instances where these special unsolved edge cases do not exist. We limited ourselves to two processors, only weighted nodes, and strictly DAG graphs. There are algorithms that have solved special situations in polynomial time such as two processor scheduling on an unweighted graph or multiprocessor scheduling for tree structured DAGs[2]. This leads to the conclusion that the overall task scheduling problem is NP-Hard. This is because no one has found an algorithm that is NP-complete for the general case and the general problem may very well be harder than all of the NP problems that exist[1]. Nevertheless, our results proved to be accurate when our one large edge case does not occur. Further improvements to this algorithm are to ensure that no special edge case is found so that a proper minimum is always found.

5 Appendix

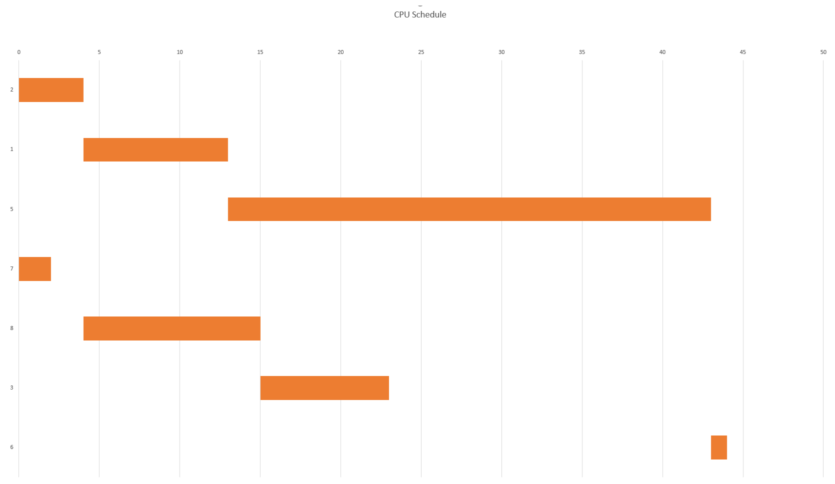


```
Id      Data    CPU #   Start   Finish
--------------------------------------
1       2       1       0       4
4       1       1       4       13
3       5       1       13      43
6       7       2       0       2
7       8       2       4       15
2       3       2       15      23
5       6       2       43      44
```



```
Id      Data    t-level b-level ALAP
------------------------------------
1       2       0       42      2
2       3       43      8       36
3       5       13      38      6
4       1       4       10      34
5       6       43      1       43
6       7       0       51      -7
7       8       2       49      -5
```

CPU data and metadata results from one of our test runs



Graph of our CPU's executing the sample above

References

[1] E Ilavarasan and P Thambidurai. Low complexity performance effective task scheduling algorithm for heterogeneous computing environments. *Journal of Computer sciences*, 3(2):94–103, 2007.

[2] Yu-Kwong Kwok and Ishfaq Ahmad. Static scheduling algorithms for allocating directed task graphs to multiprocessors. *ACM Computing Surveys (CSUR)*, 31(4):406–471, 1999.

[3] Uwe Schwiegelshohn. How to design a job scheduling algorithm. In *Workshop on Job Scheduling Strategies for Parallel Processing*, pages 147–167. Springer, 2014.