APPROXIMATING A PARALLEL TASK SCHEDULE USING LONGEST PATH

Daniel Wespetal
Computer Science Department
University of Minnesota-Morris
wesp0006@mrs.umn.edu

Joel Nelson
Computer Science Department
University of Minnesota-Morris
nels0354@mrs.umn.edu

Dian Lopez (Advisor)
Computer Science Department
University of Minnesota-Morris
lopezdr@mrs.umn.edu

Abstract

This paper focuses on an NP-hard parallel task-scheduling problem. The problem receives input in the form of a precedence graph representation of a job where the objective is a task-to-processor allocation that minimizes the total computation time of the job. The sub-tasks can be executed in parallel on a network of identical computing entities. Communication and latency constraints are also imposed making the problem more difficult to solve. Presented is an approximation algorithm for this problem that runs in O(mn) time, where n is the number of nodes (or tasks) in the job and m is the number of edges. The algorithm utilizes the fact that the longest path problem can be solved to optimality, in O(m) steps, on graphs containing no directed cycles - which is consistent with precedence graphs. Presented are results of the performance of the algorithm through extensive testing on random graphs.

Introduction

This paper involves an approximation algorithm that uses multiple computer processors to solve an NP-hard parallel task scheduling problem. Instead of having a single computer do all the computations, the work is divided among a network of computers. The goal is lower overall computation time by running separate tasks in parallel on identical workstations. Since computer processor speeds will not improve forever and the physical limits of processors are being approached, we must look for new ways to improve computation speed. The algorithm works on allocating workload to various processors to improve computation time and efficiency.

In cases where there is more than one available processor it is essential to have an algorithm that makes use of the given hardware. Dividing the workload amount among multiple processors allows for each processor to handle only part of the problem, verses having a single processor do all of the computations.

Graph Representation of the Problem

Precedence graphs are used to model the real world situation of executing a job composed of multiple (sub) tasks on a network of multiple processing entities. A precedence graph is a rooted directed acyclic graph (DAG) such that there exists a directed path between the root node and every other node in the graph. The entire graph is considered as the overall job to be computed, and the nodes are the individual tasks. The problem dealt with in this paper constrains the precedence graphs to those with exactly one leaf node such that every node in the graph has a directed path to the leaf node. Precedence graphs are used because a rooted graph equates to the real world arrangement of having a program originate from a single computer. Also, all valid graphs contain exactly one leaf node; this node represents the results of the job's execution returning to the point of origin. Note that it is desirable to maintain the property of single-input/single-output. The nodes also each have an integer value associated with them, this number represents the respective execution time of that task on any processor in the network. In a job it is often the case that one task may require data from a separate task before it can run. In this case, the second task must wait until the first has completed before it can run and the data must then be transmitted from the first task to the second. This model reflects this real world idea by having directed arcs between the nodes that have this data dependency. The parent task(s) of any given task are the tasks that must be completed before the child task can begin execution.

Sending information from one task to another is called communication. Communication can take a significant amount time when two tasks with a data dependency (arc) between them are executed on different processors. Varied communication time can be represented in the graphs by associating an integer value on the arcs representing the time it takes for the parent node to send the required data to the child node. However, this paper restricts communication to a constant for the entire graph. Arcs in the graph are all weighted the same. If the tasks are executed on the same processor, the time to communicate is considered negligible or zero since no data needs to be transmitted over the relatively slow network. In addition to communication time, a latency constraint is added to the model to simulate the delay experienced between the time that one processor starts sending a message and the time another processor begins receiving that message. The latency value can also be accurately described as the amount of time it takes to send one bit across the network. It is also assumed that a processor can "spool" communication data while it is executing tasks or doing other communication, delaying the time needed to read the data until after the processor becomes free. This is important because both task execution and communication is considered as exclusive busy time for the processor.

The real world problem of task allocation on multiple processors now translates into assigning each node in a graph to a specific processor. The algorithm takes any graph constrained to the graph properties discussed above as input and returns an approximate solution schedule. The schedule contains a task to processor assignment for each task in the graph. For example, if the algorithm were given a graph with 4 nodes, it might return a schedule that designates the following: "run task 1 on processor 1, run task 2 and 3 on processor 2 and run task 4 on processor 1." This is the goal of the algorithm.

Previous Work

Other researchers in the field currently work on similar problems to this, however they do not constrain themselves with a communication and latency model. Many deem them to be redundant or superfluous, meaning that their existence and consideration do not affect the optimal solution to any reasonable degree. This, however, is false in many practical situations. When the communication and/or latency times are comparable to the execution times of the tasks, these constraints greatly affect the optimal task scheduling. This is especially the case when considering a network of computers in an office building connected through a relatively slow network. For example, a 10 Mbps LAN is relatively slow compared to the on-board wires used for message passing between processors in a super computer. Many other researchers have only considered the problem assuming that super computers or networks with almost zero communication and latency values are the reality. Since this is not always the case, this research considers what happens when communication and latency are far from zero. In addition, the model being considered by this problem also imposes busy time for communication on both the sending and receiving processor. Many other researchers only consider communication from the sending processor. Additional background information and more precise problem definitions for these similar problems may be found in [2].

Precise Problem Definition

Let G = (V,A) be a graph defined by a set of vertices and arcs such that G is a precedence graph with exactly one leaf node. C_{ij} is the communication time required to upload and download data from the network when transmitting from task i to task j. and L is the latency constant for the network. The optimal solution must constrain to the following:

1) If there exists an arc in A from task i to task j in V, then task i must finish computation before task j can begin.
2) If there exists an arc in A from task i to task j in V and they are scheduled to be executed on different processors then the processor that task i is scheduled on must spend C_{ij} amount of time uploading data to the network starting some time after task i has finished executing. The processor that task j is on must spend C_{ij} amount of time downloading data from he network starting some time that is at least L amount of time after the corresponding upload started and the download must finish before task j can start executing.
3) Two tasks cannot execute simultaneously on the same processor.
4) Processors cannot communicate more than one task transmission at one time.
5) Processors cannot communicate and execute tasks at the same time.
6) The end time of the execution of the task represented by the leaf node in G is minimized. (Equivalent to minimizing the total computation time of the schedule).

It is no trivial task to find a way to assign tasks to processors. In fact, it has been proven that even finding the optimal solution on a 2-level tree graph is an NP-hard problem [1]. Using an exponential time algorithm it is possible to find an optimal solution for any given graph input. However, computing the optimal solution using an exponential number of computations in cases of moderate to large input size takes an unrealistic amount of time. The goal is therefore to find an approximation algorithm that can generate a scheduling close to the optimal solution within a realistic amount of time. Basically, the approximation algorithm sacrifices a certain amount of accuracy for a very large amount of time savings.

Multi-Longest Path Approximation Algorithm

For a non-directed graph, the longest path problem is NP-hard. It is an important fact for this algorithm that the longest path can be calculated to optimality in a polynomial number of steps on graphs containing no directed cycles, which is consistent with all valid input to the problem, because precedence graphs contain no directed cycles. The algorithm repeatedly finds longest paths in the graph and schedules all the nodes in each path to a separate processor. Pseudo code for this algorithm follows:

```
Step 0)  Let NOP : = 0
Step 1)  Let $S_i$ : = -1, $\forall$ i $\in$ V
Step 2)  Let lvl  : = minimum{lev | lev = original level of node i, $\forall$ i $\in$ V}
Step 3)  Let LongestPath : = the empty path
Step 4) $\forall$ node n on level lvl,
         Step A)  Let path : = PathWithMaxLength{$P_a$ | $P_a$ = LongestPath(n,p) such that
                  $\exists$ a path from n to p where p $\in$ V}
         Step B)  If PathLenght(path) > PathLength(LongestPath)
                  Then Let LongestPath : = path
Step 5)  Let $S_i$ : = NOP, $\forall$ i $\in$ V, such that i $\in$ LongestPath
Step 6)  Let V : = V \ {i | i $\in$ LongestPath}
Step 7)  Let NOP : = NOP + 1
```

```
    Step 8)  If V != {} Then Goto Step 2
```

The motivation for this algorithm is centered on the property that a lower bound for the optimal solution value is the length of the longest path of the graph. This is because there exists no faster way to compute the tasks on the longest path than to schedule them on the same processor. Also, the amount of time required for this computation is equal to the sum of the execution times of the nodes on the longest path. Therefore the entire job will certainly take at least that amount of time to finish computation, no matter what the rest of the schedule is. The reason that all the nodes on the longest path are scheduled on the same processor is that the longest path can also be considered as a critical path, to add communication time between these tasks (by scheduling some of them to other processors) would consequently raise the lower bound of the overall computation time. Since the longest path is critical to compute in the lowest amount of time possible, the entire path is scheduled onto the same processor, since there is no faster way to compute the tasks on that path. This property holds for all of the paths in the graph, but since the longest paths are the ones that are the most critical, the algorithm assigns each of the longest paths to different processors.

Algorithm Analysis

The multi-longest path algorithm generates solutions in O(mn) time, where m is the number of edges in the graph and n is the number of nodes. This is significantly less time than the exponential NP-hard brute force algorithm takes.

Lemma: For any precedence graph with n nodes, the multi-longest path algorithm could call the longest path algorithm a maximum of O(n) times.

Proof: The algorithm requires n-2 longest path computations when the graph consists of a single root and leaf node and all the remaining nodes are placed on a single level between the root and leaf. For this case, the longest path algorithm must be called for each node that isn’t the root or leaf node before all the nodes have been scheduled to a processor.

Therefore, the multi-longest path algorithm calls the longest path algorithm at most n-2 times, hence O(n) calls to longest path. Since the longest path algorithm completes in O(m) time, due to a breadth first arc traversal computation, the resulting overall complexity is O(mn).

The performance of the multi-longest path algorithm was obtained through extensive testing on randomly generated precedence graphs. A possible method of testing would have been to generate graphs tailored with the algorithm in mind. Testing done in this manner may have produced better results for the algorithm, but it would not accurately represent all scheduling problems. To more fully test the algorithm over a variety of cases, random graphs were used. The test cases were generated to reflect all types of precedence graphs to better simulate a real world situation.

The random precedence graphs were generated with a specific set of constraints, namely a range of execution times, communication times and latency times. Each node in the graphs was generated with a random execution time in the range of 1 to 1000. The graphs were also each generated with random communication and latency time in the range of 1 to 100. The fixed communication and latency times reflect a network of workstations where the communication and latency times between each pair of processors is comparable. These values were chosen in order to have an average approximate ratio of communication/latency time to execution time of 1 to 10. This type of ratio is required for testing because if the ratio becomes too large, the overall computation time will be dominated solely by the execution times. In these cases, the optimal solution trivially becomes to schedule every task to the same processor. This is because scheduling a task on a different processor would result in too much time spent communicating and make concurrent execution of tasks not worth the communication time entailed. Conversely, if the communication/latency to execution time ratio becomes very small then the communication and latency times are insignificant and have little effect on the optimal scheduling. Again, the optimal solution becomes trivial; simply schedule every task, i, to processor i. These cases are not of interest to this research because the optimal solution, in these cases, can be determined in O(1) time, thus there is no reason to test an approximation algorithm on these types of input.

Testing was done on many different graphs with a variable number of nodes, execution, communication, and latency times. The adjacency structure is also randomized, (ie- the placement of the edges). The graphs were generated and tested in groups. The graphs were grouped together based on the number of nodes in the graph. Results were generated for 20 4-node graphs, 320 5-node graphs, 810 6-node graphs, 1024 7-node graphs, 625 8-node graphs, 259 9-node graphs, 32 10-node graphs and 16 11-node graphs. The reason for the decreasing number of tests executed is because the brute force optimal solution algorithm takes a very long time to run on 10-node and higher graphs. Therefore these graphs cannot be tested as thoroughly. It should be noted that the data is less reliable or representative of the performance of the algorithm when fewer numbers of random tests are performed. Also, testing begins with 4-node graphs because there only exists one 3-node graph that is valid input to this problem, and the optimal solution is to schedule all three tasks on the same processor. Similarly, due to the property that there exist few graphs of n nodes when n is small, for example 4 or 5, not as many graphs of that size were tested because there is no reason to repeat tests. This property exists because of a limited amount of combinations for adjacency structure for small valid precedence graphs with exactly one leaf node.

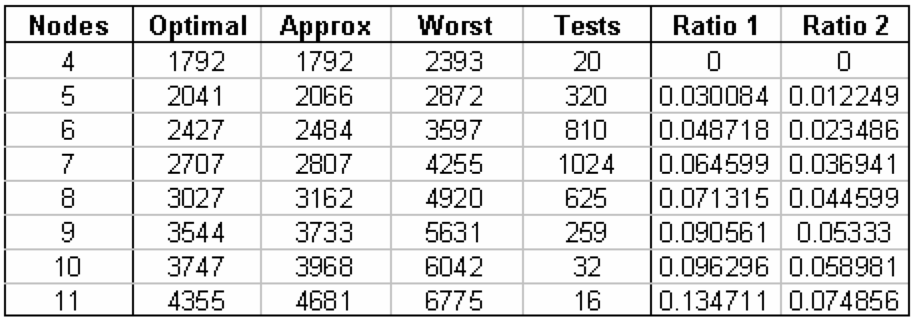
The procedure for testing the graphs is as follows: run the brute force algorithm on the graph to find the worst and optimal solutions and then run the multi-longest path algorithm to find an approximate solution. After each group of tests is completed, a file is generated containing the averages of the worst, optimal, and approximated solutions for the set of graphs. The range of possible performances for the approximation algorithm is somewhere between the optimal solution and the worst solution. The multi-longest path algorithm's performance is evaluated by how close its solution value is to the optimal solution compared to the range of possible solution values. This comparison is expressed in the following ratio: (Average Approximate Solution – Average Optimal Solution) / (Average Worst Solution –

Average Optimal Solution). These values are the average solution values for the entire set of n-node graphs that were tested for n = [4,11]. This is referred to as Ratio 1 in Table 1.

This method of algorithm analysis is ideal because comparing approximated solutions to only the optimal solution can be misleading. For example, say the optimal solution is 90 time units for a given graph and the worst solution is 100 times units for the same graph. If the approximate solution returns 99 time units, there is a temptation to conclude that this is favorable performance because it is only 9 time units or 10% above the optimal solution. However, when the range is considered, a result 99 can be accurately seen to be poor performance because it is far closer to the worst case than the optimal solution. Also presented is the percent of the average approximate solution value over the average optimal solution for each set of n-node graphs that were tested for n = [4,11]. This is referred to as Ratio 2 in Table 1.

Depicted in Figure 1 are the first three columns of Table 1 drawn as lines with the number of nodes on the x-axis and the total computation time on the y-axis.

Table 1: Approx. solution in relation to the optimal



Nodes	Optimal	Approx	Worst	Tests	Ratio 1	Ratio 2
4	1792	1792	2393	20	0	0
5	2041	2066	2872	320	0.030084	0.012249
6	2427	2484	3597	810	0.048718	0.023486
7	2707	2807	4255	1024	0.064599	0.036941
8	3027	3162	4920	625	0.071315	0.044599
9	3544	3733	5631	259	0.090561	0.05333
10	3747	3968	6042	32	0.096296	0.058981
11	4355	4681	6775	16	0.134711	0.074856

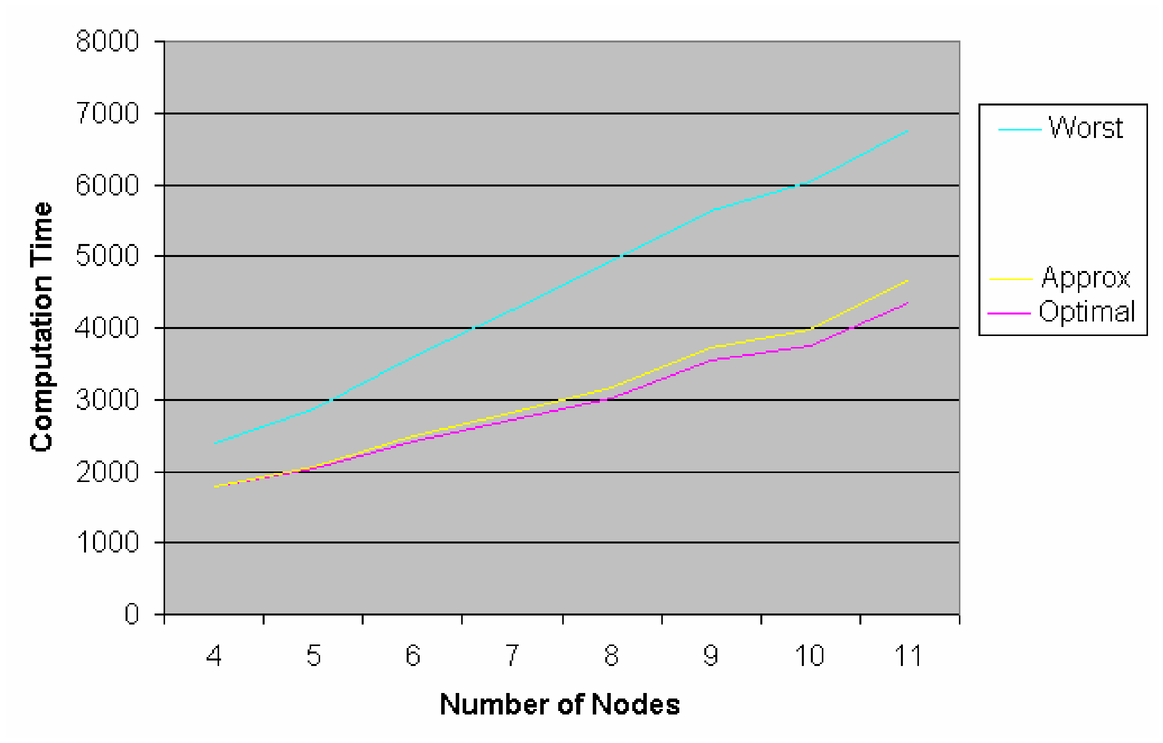


Figure 1: Average Solution Values

Future Work

The most important step to take with this algorithm now is to prove a bound on the performance, no matter what graph is given to it. This means that, for example, it may be the case that for any valid graph, the approximate solution value will never be worse than twice the value of the optimal solution. If no bound is proven then testing should continue for larger graphs to determine if the trend above can be extended.

It is also always important to continue to look for other approximation algorithms that may approximate solutions to the problem in a fast polynomial amount of time with good performance. Many graph problems are closely related to each other, meaning that one graph problem can often help solve another. Recall that this algorithm utilized the longest path algorithm for directed acyclic graphs. Keeping this in mind, a good method of developing new approximation algorithms is to look into other solvable (ie- polynomial time) graph problems.

References

[1] Hsu, T.s., Lee, Lopez, D.R., and Royce, W., "Task Allocation on a Network of Processors," IEEE Transactions on Computers, Vol. 49, No. 12, pp. 1339-1353, December 2000.

[2] I. Ahmad and Y.-K. Kwok, “On paralleling the Multiprocessor Scheduling Problem,” *IEEE Trans. Parallel and Distributed Systems*, vol. 10, pp. 414-432, 1999.

[3] Lopez, D.R. et al. “Design and Testing of Parallel/Distributed Precedence Tree Scheduling Algorithms.” Proceedings at Midwest Instructional Computing Symposium, 2002, Cedar Falls, IA.