DEVELOPMENT OF A TEST BED TO DETERMINE PERFORMANCE OF SCHEDULING ALGORITHMS

Adeola Adewola
Computer Science Department
University of Minnesota-Morris
adew0003@mrs.umn.edu

Jon Quarfoth
Computer Science Department
University of Minnesota-Morris
quar0020@mrs.umn.edu

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

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

Abstract: The authors of this paper developed a test bed to test new approximation algorithms for a parallel task-scheduling problem. The test bed consists of graphs and an optimal and worst case solution for each graph. The problem being researched involves taking a job that is divided into tasks and returning a way to schedule these tasks on a network of identical workstations with the goal of having all the tasks and thus the job completed in the least amount of time. The problem is an NP-hard problem. Precedence graphs are used to model the problem. Using a brute force method to find the optimal solution takes an unrealistic amount of time, thus the focus is on coming up with an approximation algorithm that obtains a solution close to optimal in a polynomial time. We are working on creating new approximation algorithms and improving existing approximation algorithms by using the test bed to compare performance results of these algorithms.

Introduction

The idea behind a parallel task scheduling is to execute a very large task by using more than one processor. If a large job or computation is given to one computer, it could take a long time to finish the computation. Time could be saved by dividing the job into smaller tasks and then assigning them to identical processors. These processors could then complete the tasks assigned and then send the result back to the computer where the job originated on. The objective is to complete the entire computation in the least amount of time. Our problem design takes into consideration communication and latency. Communication is the time it takes a processor to upload or download information to the network. Thus when one processor sends information to another, communication time is incurred for the processor loading the data unto the network and a second communication time is incurred for the receiving processor, which must download the information from the network. Latency is the time a processor starts sending information until the time another processor starts receiving the information. In other words, latency is the time it takes one bit of information to travel across the network. Communication and latency times are fixed for this research because the jobs are being divided into smaller workloads using identical processors stationed close to each other. The problem of scheduling even a two layer graph in this manner is NP-hard. This means that an optimal solution to this problem will take an exponential amount of time to generate. Approximation algorithms can be created, however, which will give us a solution close to optimal and will run in polynomial time. To judge the performance of our approximation algorithms, a brute force algorithm is used which finds all possible combinations and obtains the optimal solution for a test case (job).

The next section explains how our problem is represented and then we show how a test bed can help us test new algorithms fast and measure their performance. After this, we proceed to show some results for approximation algorithms that are tested on a newly created test bed. We conclude with a summary and discuss future research.

Representation of the Problem

We use a precedence graph as our representation for our real world problem of scheduling tasks on parallel processors. A graph corresponds to a job and can be divided into nodes, which are the tasks. A precedence graph is a directed acyclic graph (DAG) with a single root node. We add a sync node to the graph as a constraint in order to represent the result of the parallel computation returning to the computer where the job originated. Each node has an associated positive integer, which represents the execution time of each task. The directed arcs in our graph represent dependency - a task might have to wait for information from another task before it can run. The parent task, therefore, will have to complete its execution before the child task can run. Sending information from one task to another can incur latency and communication costs if the tasks aren't on the same processor. Communication and latency are considered constant values for each graph. If a task is assigned the same processor as another task, the communication and latency incurred is zero. Latency is a constant because of the close

proximity of the processors. Using our definitions from above, we have shown how a precedence graph can be used to represent the task-scheduling problem. We write algorithms that take in a precedence graph as input and then produce an approximate solution. This solution is the assignment of each task to a processor. The goal is to obtain a completion time close to optimal time. For example, five tasks might be scheduled in this manner: task 0 on processor 0, task 1 on processor 0, task 2 on processor 1, task 3 on processor 2 and task 4 on processor 0. The last task is always assigned the same processor as the first task because we want to return the solution to the computer it started on. Below is a diagram of what a four-node precedence graph would look like.

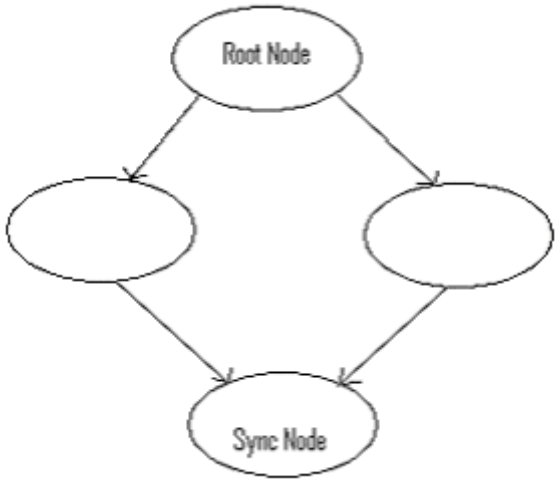

Figure 1
Precedence Graph with 4 nodes

Note that the root node must execute first and then send information to its children before they can execute their computations. The sync node must wait for all processes to complete before it can compute.

In the following example, we show a simple problem of summing the numbers from 1 to 4000. This computation could be divided among four identical processors for a faster completion time of the summation. This is an example of the type of real world problem we are working to solve. It is important to keep in mind that our research doesn't actually solve a problem like this; rather we would work on solving a precedence graph that modeled this situation. Also note that we aren't concerned with the actual computation of this summation, rather we are working to find the best way to split this computation up in order to have our result in the least amount of time.

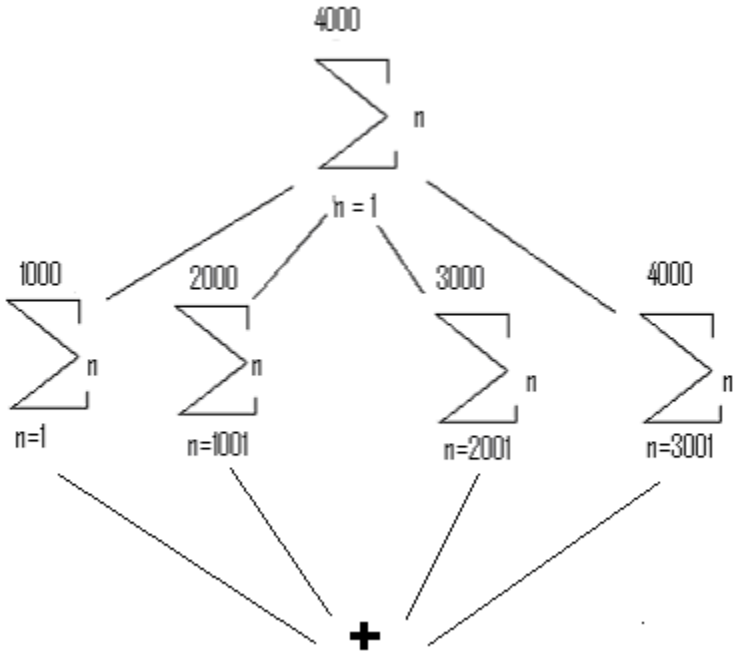

Figure 2
Sample problem of a summation of 4000 terms

How a Test Bed can help us with this problem

The brute force algorithm looks for all possible combinations of task to processor schedulings. After it finds all combinations, it chooses the one with the lowest overall computation time to be the optimal solution. When we have large graphs with many nodes, the brute force method encounters some difficulties. The exponential number of possible assignments causes the brute force algorithm to run for a long time before providing the optimal solution. Approximation algorithms, however, don't have to find the optimal solution. They return a solution close to optimal in a reasonable amount of time. Essentially, we are sacrificing accuracy for a large time savings when returning results. To find the optimal solutions for ten graphs with twelve nodes each takes about a month using the brute force algorithm. In contrast, it takes an approximation algorithm about 30 seconds to generate approximate solutions for the same graphs. As another example, five hundred eleven-node graphs will take about three weeks for the brute force algorithm to finish and only about three minutes for the approximation algorithm. In order for us to determine how well our approximation algorithm performs, we need to have the optimal solution results to compare with. Previously this created a problem because each time we needed to test a new approximation algorithm, we would have to run the brute force algorithm. We also desired to test our algorithm on larger graphs to obtain a better sense of how close it was to optimal for graphs with more significant size. The problem we encountered is we couldn't run different approximation algorithms on the same graph data because each time we tested a new algorithm, we had to generate new sets of graph data. To solve this problem a test bed was created. The test bed is a directory/file system in which we can save all of data from our tests. The test bed

provides a standard set of test cases and their associated optimal and worst case solution. Once that data is stored, new approximation algorithms can be tested by running the algorithms on the test cases and comparing the results to the known optimal and worse cases. We can run the brute force algorithm on all of these different graphs and store their optimal and worst case solutions. With our new test bed, we can run approximation algorithms without worrying about having to run the brute force algorithm. We also store the results from each approximation algorithm so that we can compare results from different approximation algorithms run over the same set of test cases. It now takes less than half an hour to test a new approximation algorithm using the test bed. This is a big difference compared with the several weeks it used to take to test new algorithms. The test bed makes testing algorithms so much easier and faster, especially when comparing results. We can write algorithms and, by just running them and comparing the results with the stored brute force data, know how well they work. Using this method, it is easy to know if the approximation algorithm produces results close to or far from optimal. With the test bed in place, we can write algorithms quickly and then see how they perform against the brute force and other approximation algorithms.

Usage of Testbed

Now that we have established the reasons for creating the test bed, we are going to discuss how it was implemented in our project. The goal of our research is to design approximation algorithms for our parallel task scheduling problem. When we create a new algorithm, we need a standard to compare it with. This is where our test bed and brute force algorithm come in.

The test bed we created contains basically three different types of information. The first is randomly generated precedence graphs (test cases), the second is associated optimal solutions and worst cases for the graphs and the third is associated approximate solutions for the graphs. The precedence graphs were generated in groups, based on the number of nodes in the graph. These random graphs are to be our test cases. A graph with ten nodes represents a large job split into ten sub tasks. We desire graphs that are as random as possible and not designed in a manner that will make our algorithms perform better. It is for this reason that the random graph generator was created first, before our algorithms were written. We also deem random graphs necessary because they best model real world situations, where we need to solve a variety of problems, not merely cases that are suited for our algorithm.

The random graph generator constructs random precedence graphs with the previously stated additional constraints necessary for our problem. Our random graph generator has a few features that are pseudo-random vs. completely random. One is that we add a certain number of “jump” edges to our graph. A “jump” edge is an edge that goes from the parent node to a child node that isn’t on the level immediately below the parent node. It is necessary to have some “jump” edges in our graph, so we had to make some choices about how we would add these edges. Having no jump edges would be a poor choice, as would including all possible jump edges. Eventually, we settled on adding x number of

jump edges where x is a random number between 0 and 1/3 the number of nodes in our graph. For example, in a 9-node graph, we would add between 0 and 3 jump edges. A jump edge is added by first selecting a parent node at random, then taking all the nodes below this parent and choosing one at random to be the child. Connect these two nodes to make the jump edge.

Our random graph generator allows the user to specify several constraints. Thus we had to make design choices about our bounds for latency time, communication time and execution time. Our graphs were generated with communications times between 1 and 100, latency times between 1 and 100 and execution times (for each node) between 1 and 1000. These ranges where chose because having communication times and latency times that are similar and on average 1/10 of the execution times best models real world situations.

As stated before, graphs were generated in groups based on the number of nodes in the graphs. Below in Figure 3 is the distribution of the number of graphs (test cases) that were generated in our test bed.

Number of Nodes	4	5	6	7	8	9	10	11	12
Number of Graphs	50	100	500	1000	1000	500	500	500	10

Figure 3

Distribution of Test Cases

As you can see from above, we only generated fifty 4-node graphs. The reason this number is lower than the others is there isn't a lot of variation of 4-node graphs. Thus generating more than fifty 4-node graphs would lead to graphs that are essentially the same. One other abnormality to note is that only ten 12-node graphs were generated. This is due to the fact discussed before that running the brute algorithm takes substantial amount of time to obtain solutions on graphs with a large number of nodes. It is unrealistic to attempt to have more graphs due to time constraints.

Our test bed is designed for flexibility for future users. In addition to separating graphs into groups based on number of nodes, our test bed also allows for the creation of multiple sets of graphs with the same number of nodes. Thus, in the future if we have more powerful computers that can handle more 12-node graphs, we can add a set of 12-node graphs that contains 100 graphs or more. Figure 4 below shows the file structure used in our test bed.

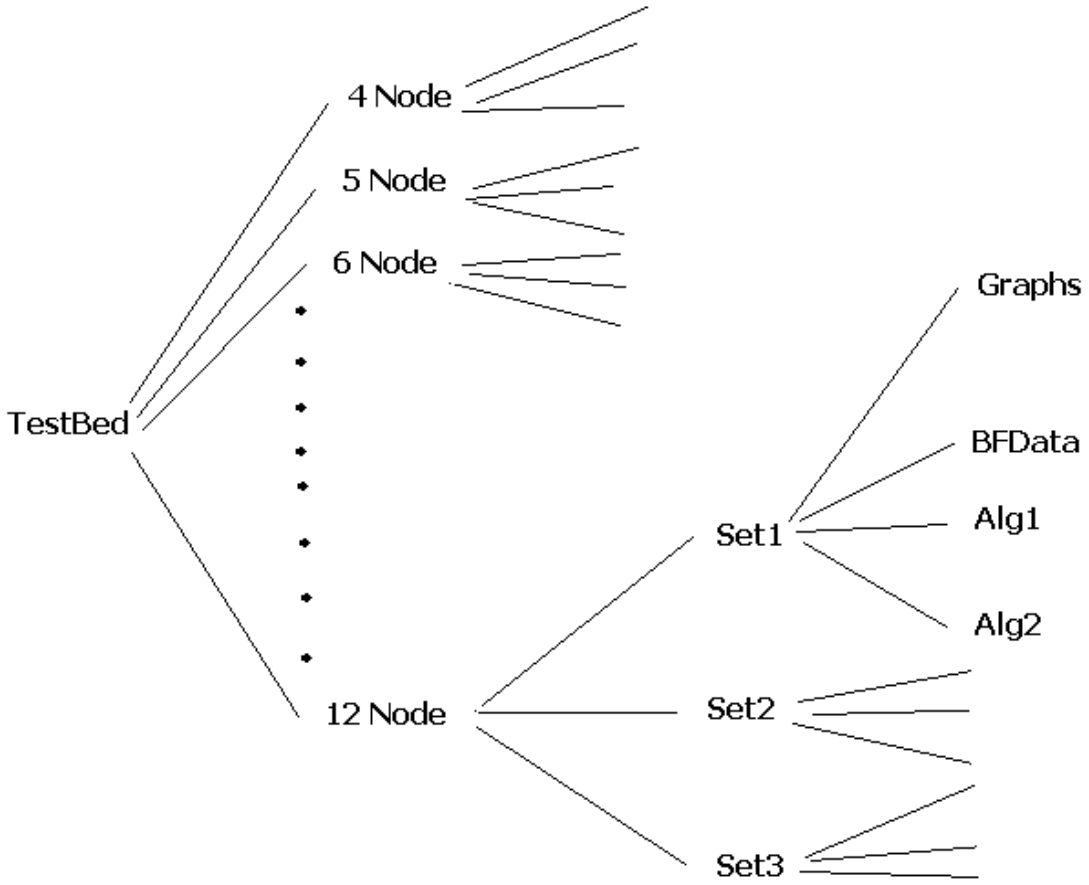

Figure 4
File Structure of TestBed

This diagram doesn't show all the subdirectories. The subdirectories that aren't shown are identical to the ones that are shown for the twelve-node graphs. For example, the set2 and set3 directories found under the 12 Node directory are identical to the 12Node/Set1 directory. Each of these set directories contains graphs, along with their brute force data, and then a directory for each existing approximation algorithm. The 4 through 11 Node directories all are identical to the one shown for the 12 Node directory, thus they all contain 3 set directories of data.

After our graphs were generated, the next step was to run our brute force algorithm on the graphs and output the results to the associated BFData directories. We used a brute force algorithm that was written by previous researchers at UMM [2]. In short, this algorithm generates all possible schedulings and then determines which returns the lowest and the highest overall execution time for each graph. The data is stored in our brute force data directory as the optimal solution and the worst case for each graph. This data will be used later for comparison with approximation algorithms.

We were now ready to run our approximation algorithms on the graphs. The results from these runs are stored in the approximate Alg directories. Therefore, for each new algorithm we create, we simply add a directory in each set of test cases. For example,

after the creation of a new algorithm called LongestPath (it will be discussed in more detail in the next section) a new directory is created: /Testbed/4Node/Set1/LPAlgorithm, along with similar directories for each set of test cases used. Then we simply stored the results for this new algorithm when it was executed on the associated set of graphs. In the case above, the new directory would contain the information from the Longest Path Algorithm being run on the first set of 4 node graphs. Using this design allows us to quickly and easily compare the results for each approximation algorithm, with both the brute force data and also with other approximation algorithms.

Approximation Algorithms

The following section discusses the different approximation algorithms that have currently been tested on our test bed. Previous researchers have written some of the algorithms [3] and some are results of this year's research. One of the main advantages of our test bed is that new approximation algorithms can be tested quickly. Therefore our strategy was to write many algorithms without trying to discern at how well they would perform. The performance of the algorithm was determined by how it performed on the test cases. In this way we save time analyzing how the algorithm will perform and rely mostly on numerical data for analysis.

The algorithm written during last year's research is known as the longest path algorithm (LPA) [3]. The basic idea for this algorithm is as follows. Begin by finding the longest path in the input graph. The longest path is the path from the source node to the sync that has the largest sum of the execution times of the nodes along that path. After the longest path has been determined, schedule all the nodes along this path on a single processor. Next remove these nodes from the graph because they have now been scheduled. Then repeat the process on the nodes remaining in the graph, starting again by finding the longest path among the remaining nodes. Continue this process until all the nodes have been scheduled. The basic theory behind this algorithm is that the overall execution time for the graph is going to be at least the total time along the longest path of the graph. This follows because each successive node along the path must wait for the previous node to complete its work before it can start its own computation. Therefore, it makes sense that we don't want to add any additional time to this longest path. We can avoid adding additional computation time by having all the tasks on the same processor. This eliminates all communication and latency time between the tasks. For a formal definition of the algorithm and its motivation, see [3].

The first of our new algorithms was titled the Parallel Processing Algorithm, or PPA, for short. This algorithm is extremely simple and straightforward. The goal of any of our algorithms is to lower overall computation time through parallel processing. Therefore we tried to formulate a simple method that maximizes parallel processing time, in hopes that increasing the amount of processes running in parallel would lower the overall computation time. We came up with the following method for doing so. Begin with all the nodes on the first level of the input graph. Then schedule each node/task on a separate processor, beginning with processor zero. Repeat this process for the nodes on

the next level below the current level. Continue down the graph until the sync node has been reached. For clarity, say we encounter a level with three nodes on it. We want to schedule these three nodes on separate processors. So we would schedule one node on processor 0, one on processor 1 and one on processor 2. We determine which task goes with which processor by keeping track of the sum of the execution times of the tasks that have been previously scheduled on each processor. The node with the highest execution time goes with the processor with the lowest summed time. Similarly, the node with the lowest execution time goes with the processor with the highest summed time.

A formal definition of the PPA is given below. Let G be our input graph with n nodes. Let ExecutionTimes[n] be an array of size n that keeps track of the sum of execution times scheduled on each processor.

Parallel Processing Algorithm (PPA)

1) For int i ← 0 to maxLevelOfGraph;
    a. CurrentNodes ← G.getNodesOnLevel(i);
    b. While (CurrentNodes.hasMoreNodes())
        i. LowestNode ← CurrentNodes.getLowestExectionTime();
        ii. HighestProcessor ← getHighProcessTime(ExecutionTimes[]);
        iii. Schedule LowestNode on HighestProcessor;
        iv. Remove LowestNode from CurrentNodes;
2) Return generated scheduling.

Note on line ii we are getting the highest processor time for the processor 0 up to the number of CurrentNodes we are working with (at the beginning of the while loop).

For now we are going to hold off from giving the performance of this algorithm until after all the algorithms have been introduced. The PPA is very simple and was used as a starting point for thinking about different algorithms.

We continued to study the problem by examining how we, as humans, would go about scheduling several connected tasks on a network of computers. One observation we made is that after a processor has completed a task, its next step should be to send out the information needed by other tasks. It is advantageous to send out the information as soon as possible as to not delay the overall computation of the job. We also observed that the processor becomes free after completing a task; therefore it makes sense to always schedule at least one of its child tasks on the same processor. Always scheduling at least one child process on the same processor makes sense because this strategy will help eliminate unnecessary communication and latency time.

These two observations lead to our next strategy, entitled the Child Scheduling Algorithm (CSA). This title was given because the algorithm works its way from the top of the graph down by scheduling the children of each node encountered. The algorithm works its way down the graph level by level. For each node on a level, we schedule all its children in the following manner. First, find the child node with the highest execution time and schedule it on a different open processor. Then find the node with the next highest execution time and schedule it on a second open processor. Continue this process

until only a single child node remains and schedule this node on the same processor as the current parent node. Following this method insures our last node will be the child node with the lowest execution time. Thus the parent's processor has the child with the lowest execution time also scheduled on it. An important feature of this algorithm is the order of our scheduling of the child nodes. In this algorithm, we are always scheduling the processes that are on a different processor first. This is significant because if we schedule the tasks on the parent (same) processor first, the other tasks will have to wait for the execution to finish before they can run because they are waiting on information from the parent processor. Figures 5 gives the makespan pictures of the two different cases we are referring to.

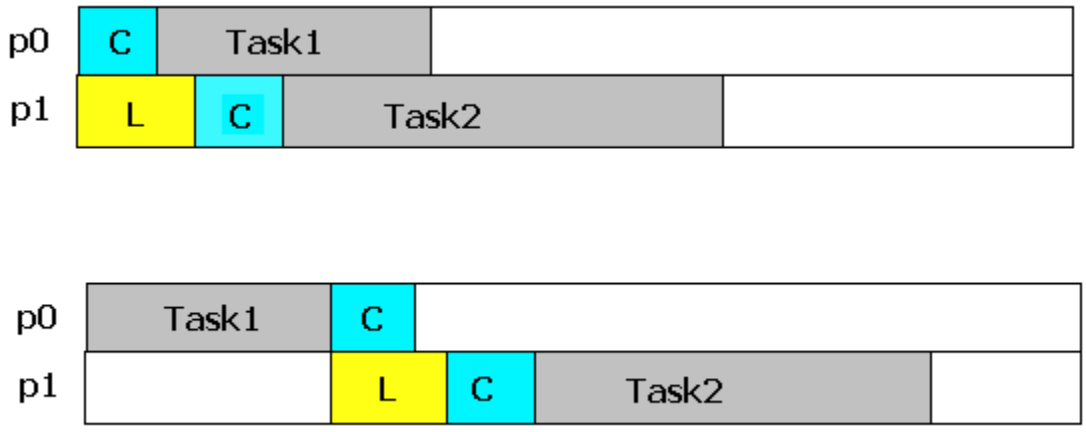

Figure 5
The top picture shows communication followed by execution.
The bottom picture shows execution followed by communication.

From this picture it is easy to see why it is best to communicate first and then execute, instead of doing it the other way around. Executing first doesn't make use of parallel processing.

The Child Scheduling Algorithm can actually be implemented in two separate ways. The first way is as discussed above. The second would be to leave the node with the highest execution time to be scheduled on the same processor as the parent. Thus the algorithm would start by scheduling the nodes with lowest execution times on different processors and then end up with the node of highest execution time and schedule it on the same processor. Given below is the formal algorithm, with the highest execution time scheduled on the same processor as the parent. Note NOP stands for the next open (free) processor.

Child Scheduling Algorithm (CSA)

1) Schedule Task 0 on processor 0
2) For int i ← 1 to maxLevelOfGraph
    a. CurrentLevel ← getNodesAtLevel(i)
    b. For int j ← 0 to CurrentLevel.size()
        i. CurrentParent ← CurrentLevel.get(j)
        ii. CurrentChildren ← CurrentParent.getChildren()
            1. while(CurrentChildren != 1)
                a. lowestNode ← getLowestNode(CurrentChildren)
                b. Schedule lowestNode on NOP
                c. ++NOP
                d. Remove lowestNode from CurrentChildren
            2. Schedule last Child in CurrentChildren on same processor as CurrentParent
3) return Scheduling

Notice that this algorithm is easy to alter if we wish to schedule the child node with the lowest execution time on the same processor as the parent. All we must change is line with lowestNode ← getLowestNode(CurrentChildren) to highestNode ← getHighestNode(CurrentChildren). Once the getLowestNode() and the getHighestNode() methods were written, it was easy switch between the different implementations.

An approximation algorithm related to CSA is known as the Summed Execution Times Algorithm (SETA). This algorithm follows the same pattern of traversing down the graph level by level and scheduling the children of each node on the current level. However, SETA differs from CSA in the fact that CSA schedules child nodes based on their execution times, while SETA schedules based on a child node's summed execution time. We define a node's summed execution time as its execution time plus the execution time of all its descendents. In other words, a node's summed execution time is the sum of its execution time plus all the execution times of the nodes below it that are connected to it (directly or indirectly). This summed execution time is essentially a priority value for each node. The higher the sum is, the more important it is to schedule this node next because more tasks are waiting for the information from it. Thus in SETA,

the child with the highest summed execution time is scheduled first and on a different processor than the parent. This leaves the child node with the lowest summed execution time to be scheduled on the same processor as the parent node.

Like stated before, SETA is just an alteration of the CSA algorithm given above. The code is mostly identical; the parts that are different are given below. The following lines are to be used in place of line 1. and below.

Changes in CSA that yields SETA

1. while(CurrentChildren != 1)
    a. highestNode ← getHighestNode(CurrentChildren)
    b. Schedule highestNode on NOP
    c. ++NOP
    d. Remove highestNode from CurrentChildren
2. Schedule last Child in CurrentChildren on same processor as CurrentParent

3) return Scheduling

Note that the method getHighestNode(CurrentChildren) from line a. returns the child node with the highest summed execution time. It is easy to see that this code is very similar to the original CSA, except the getHighestNode() method is based on summed execution times instead of just each nodes execution time.

The CSA algorithm idea turns out to fit well with our plan of utilizing the test bed for writing algorithms. It is a good fit because there are several slight adjustments that can be made to the algorithm in hopes of improving our results. We tried several different versions of this algorithm, three of which were listed above. Thus the creation of this algorithm contained the following steps:

(1) Implement the algorithm and test it.
(2) Think of improvements for the algorithm (such as SETA) and alter the code to implement the improvements.
(3) Test again and see if the results improve.
(4) If the results improved, keep the changes, otherwise discard them.

This method wouldn't even be possible without the test bed, which allowed for quick comparisons of algorithms on a standard set of test cases.

Before beginning to talk about performance evaluation for our specific algorithms, we are going to mention how we analyze approximation algorithms in general. The general method of evaluation is to compare the approximation algorithm's solutions in the range of feasible solutions. The range of feasible solutions is computed by subtracting the worst case solution from the optimal solution. The approximate solution, worst case solution and optimal solution are the timings that are a result of the schedulings generated by each. The usual method of comparison for problems like ours is to compare the approximate solution only to the optimal solution. In other words, the normal method is to see what percent above optimal our approximate solutions generate. However, we feel comparing over the range of feasible solutions is a better method as evident in the

following example. Say that the optimal solution for a given graph is 100 time units. Then suppose our approximation algorithm returned a scheduling of 109 time units. Under the standard method of evaluation, we would be tempted to conclude that is an acceptable result. However, say the worst case for the graph was 110 time units, making our solution of 109 comparatively poor. Using the range of feasible solutions, we would easily see our solution is far closer to the worst case than optimal, and thus correctly identifying it as a poor solution.

Since our overall goal is to improve our approximation algorithm to obtain solutions closer to optimal, our goal translates into beating our current best approximation algorithm. At the beginning of this year of research, the algorithm that generated solutions closet to optimal was the Longest Path Algorithm. Therefore our goal was to generate better results than those generated by the Longest Path Algorithm over the set of test cases in our test bed. In addition to this, the Longest Path Algorithm appeared to be diverging from the optimal solution as n (the number of nodes in the graph) was increasing. In our test cases with 12 nodes, the Longest Path Algorithm was only in the lower 13^{th} percent of feasible solutions, compared with being almost at optimal for 4 node graphs. It would be much more desirable to have an algorithm that appeared to be converging to optimal as n increased. Thus a converging algorithm was the secondary, and more difficult, goal for our new algorithms.

The results for all four of the previously mentioned approximation algorithms are given in Figure 6 below, along with the worst cases and optimal solutions.

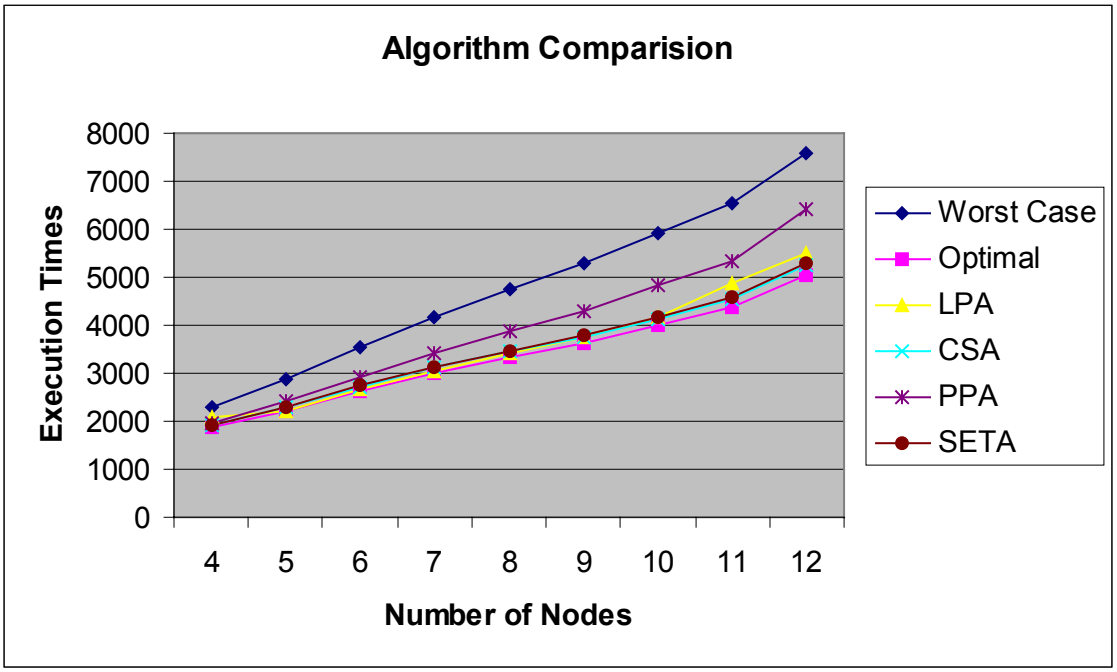

Figure 6
Comparison of all algorithms

In this graph, total time is given in generic time units and is the average of the total times for each graph in the set. Each point on the graph represents the total time for a given algorithm for a given group of test cases. Notice that all approximate solutions fall in our range of feasible solutions. In Figure 3, it is hard to see the lines for SETA and CSA. The reason it is hard to distinguish between these two lines is the execution times for both algorithms are within 50 time units of each other. The important point to consider is that LPA, CSA and SETA all have performances similar to each other.

One observation to make is the poor performance of the Parallel Processing Algorithm. It seems to fall about in the middle between the worse case and optimal solutions. This is far from acceptable, and after we tested it we immediately disregarded it. The Longest Path Algorithm and the Child Scheduling Algorithms (all 3 different versions discussed above) return results very close to each other. In general, these 4 different algorithms (LPA, CSA-highest and lowest version, and SETA) return solutions in the lower 10^{th} percentile of feasible solutions. All of results are acceptable, but which algorithm performs the best? Our goal was to perform better than the LPA. In some respects we met this goal. The results show that for middle cases (6-9 nodes) the different versions CSA fail to beat LPA. The LPA beats the CSA by less than 100 times units in all of these cases. However, for the other cases (4-5 and 10+ nodes) the CSA does better. As stated before, one of our concerns was the LPA divergence from optimal as the number of nodes increased. The CSA is a big improvement on this for it appears to remain in the lower 10^{th} of feasible solutions over the whole data set. As a test to see if CSA would continue to beat LPA as n increases, we tested the approximation algorithms on graphs over 12 nodes. We were not able to obtain brute force data for these graphs because the brute force algorithms didn't/won't finish in a reasonable amount of time. Thus we only compared the results of the approximation algorithms. The CSA continued to beat LPA for graphs with 14-18 nodes. See the graph below**.**

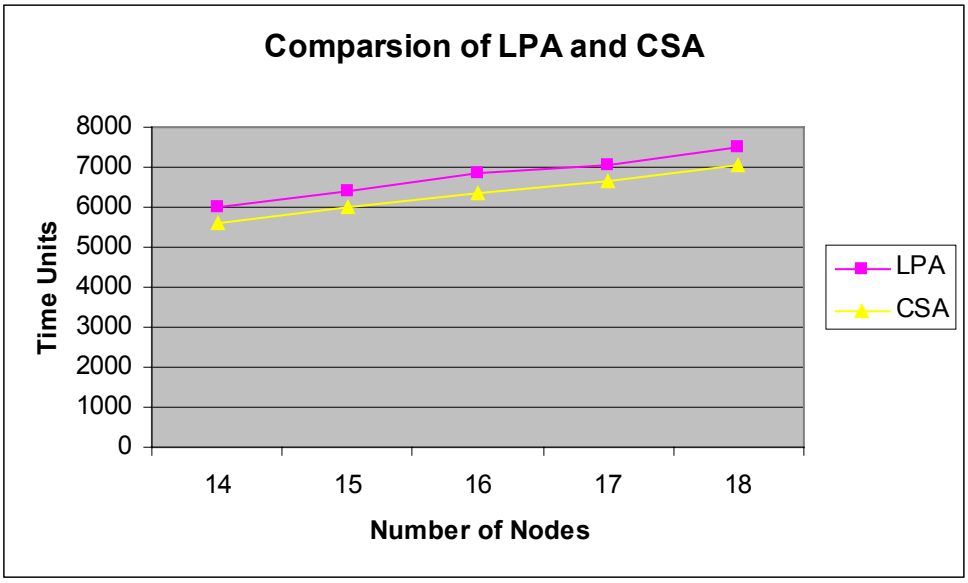

Figure 7
Comparison of LPA and CSA

The difference between the 3 different versions of the CSA is negligible. All of the results generated by these algorithms are within about 50 time units of each other. Though the difference isn't great, just as a note, the CSA in which scheduling is based on execution time and where the node with the lowest execution time is scheduled on the same processor generated the best results.

Future Research

The adding of the 'jump' edges, for instance, to our random graphs has not been proven to generate completely random graphs. Giving a proof of the randomness of our graph generator will ensure our test cases are unbiased and also help us in writing better algorithms. The algorithms we write assume an infinite number of processors, however, in the real world, we are going to have a limited amount of processors. Having a limited number of processors would make the problem more realistic. We also assume that communication is fixed, but in the real world, different processors are going to have different speed when transferring information. Therefore, we are working on writing approximation algorithms that will consider variable communication between tasks and also work with a fixed set of processors. It is also important to continue to look for algorithms that will perform better than the algorithm we have currently.

Conclusions

We began with a need to test approximation algorithms quickly and efficiently. We discovered that using a test bed gives us many advantages for algorithm testing. It gives us a standard for testing because all algorithms are run on the same set of graphs. It also allows for fast testing because the brute force data is stored in the test bed, thus only the new approximation algorithms need to be run on the test cases. This advantage allows us to write many algorithms quickly and to test their usefulness by evaluating their performance on the test cases. We were able to use our test bed to write and test several algorithms this year, as compared to years past where only a single algorithm was written and tested. The goal for our new approximation algorithm was to beat the previous best algorithm. While this wasn't completely achieved, we did make the improvement of not diverging from optimal as n got large. It is our hope that research will continue in this field and that this future research can be aided by our test bed.

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] Lee, J., Lopez, D., Royce, B., "Simulation Studies of a Parallel Scheduling Algorithm," Proceedings, 30th Annual Small College Computing Symposium, pp. 227-234, April 17-19, 1997, Kenosha, WI

[3] Wespetal, Daniel, Nelson, Joel and Lopez, D. R. “Approximating a Parallel Task Schedule Using Longest Path,” Proceedings at Midwest Instructional Computing Symposium, 2003, Duluth, MN