TASK ALLOCATION ALGORITHMS WITH COMMUNICATION COSTS CONSIDERED

Jon Quarfoth, Andy Korth, and Dian Lopez (Advisor)
Computer Science Department
University of Minnesota, Morris
Morris, MN 55112
quar0020@morris.umn.edu
kort0061@morris.umn.edu
lopezdr@morris.umn.edu

Abstract

This paper studies the effect of variable communication times for a parallel task scheduling problem. The problem involves taking a large job that is divided into dependent tasks and finding a way to schedule the tasks among a group of networked computers with the goal of completing all of the tasks in the least amount of time. We model the problem using precedence graphs to represent tasks which depend on one another. The problem is an NP-hard problem, meaning that finding optimal solutions for cases in which the job is split into many tasks requires trying all combinations and takes an unrealistic amount of time. This research focuses on finding approximation algorithms that will solve the problem quickly and still produce results that are close to optimal. This year, we developed 2 approximation algorithms which are returning very promising results on this problem.

Introduction

Many computational tasks may be divided into smaller tasks which can be distributed to multiple processors to reduce the time needed to solve the given task. Thus, there exists a need for methods and algorithms to *schedule* these tasks across multiple processors. However, certain tasks need to be preceded by other tasks whose results will need to be known before beginning the computation on that given task. To further complicate the scheduling, each smaller job could take a different amount of time to complete and have a different amount of data, thereby affecting the time required to send the task to another processor.

A simple example of a computation that may be split up into smaller parts and distributed across multiple processors (or machines) is the summation of a large set of numbers. Given ten thousand numbers and four processors, one can simply send two thousand five hundred numbers to each of the four processors. Each processor would be responsible for calculating the sum of their set and then returning a single answer to the processor that delegated that task (see Fig. 1). Of course, the total sum cannot be computed until each processor has completed their task. This is a very simple example of a problem which our algorithms could schedule.

$$\begin{tikzcd}
& \sum_{n=1}^{10000} n \arrow[dl] \arrow[d] \arrow[dr] \arrow[drr] & & \\
\sum_{n=1}^{2500} n \arrow[dr] & \sum_{n=2501}^{5000} n \arrow[d] & \sum_{n=5001}^{7500} n \arrow[dl] & \sum_{n=7501}^{10000} n \arrow[dll] \\
& \sum_{n=1}^{10000} n & &
\end{tikzcd}$$

Figure 1
Example of a large summation being split into smaller tasks that can run independently

When scheduling a series of tasks which are dependent on each other, there are a number of issues that need to be taken into consideration. For a given set of these jobs, we will need to determine which tasks must be scheduled before others. This research does not concern itself with the actual determination of precedence for real-life problems. Instead, we are only concerned with the scheduling of a job and not the job itself. The problem has been generalized so that any problem that can be split into a series of tasks where the time taken to execute a given task is known can have a solution approximated. The algorithms have been improved from previous cited research [1] to take into consideration a communication time that will differ for each task. This more closely approximates real life scheduling problems because each task will need a different amount of information before the task can be run. The amount of information a task needs to be sent before it can run determines the size of a particular communication time in the problem.

The problem of scheduling tasks is NP-hard. The number of possible solutions that must be investigated to find the best-case grows very rapidly and exponentially. This research included the development of a brute force algorithm which would try every possible method of scheduling in order to find the optimal solution. Because this amount of time is completely unreasonable for actual task scheduling, approximation algorithms have been developed. While they do not provide the optimal solution, they provide an excellent approximation of the optimal schedule in polynomial time.

Constraints on the Problem

The first constraint on the problem of parallel task scheduling is communication time. Communication time is the focus of this research. Communication time is incurred when the processor is spending time either sending information out to another processor or receiving information from another processor. Previous research simplified the problem by assuming communication time was constant for all tasks. This is clearly not the case in real-life situations. With the addition of communication time to scheduling algorithms, a better approximation of the optimal solution is expected. Sending an enormous array of integers for summation will take significantly more time than sending a single integer.

Very few practical tasks can be modeled using a constant communication time. In order to make this research practical, communication must be modeled as a variable. Storing communication time as a constant severely limits the sort of problems that can be modeled. A constant communication time would imply that every task in the node needs the same piece of data before it can begin working. In a practical scenario, a task might hand off many pieces of data to many different processors. For example, perhaps a company has an online job application process and the data produced is sent through a parallel system. When an application comes in, it is parsed by a processor, which then hands off relevant bits to other parts of the system. One processor might only need a social security number to run a background check, whereas another part of the system might need everything in the file to put into the company's applicant database.

The second constraint on our problem is latency. Latency is a measure of the time from when a processor begins sending a communication and the receiving processor begins receiving the communication. Latency is an important consideration because a processor will not receive information as soon as another processor sends information to it. Unlike communication time, latency time can be 'absorbed'. Consider the situation where CPU 1 has a very long task and CPU 2 had just completed a shorter task and wants to send information to CPU 1. In a normal situation, CPU 1 would wait idle until it began receiving the communication that CPU 2 was sending to it. Only then would it be able to spend some time processing the communication and then execute the task. However, in this situation, CPU 1 is still computing the result of a different task while the latency time would be in effect. Thus CPU 1 could begin processing the communication from CPU 2 immediately (See Fig. 2). Task 1 receives the data, but cannot process it until it finishes executing Task 1. This models how a system buffers incoming data.

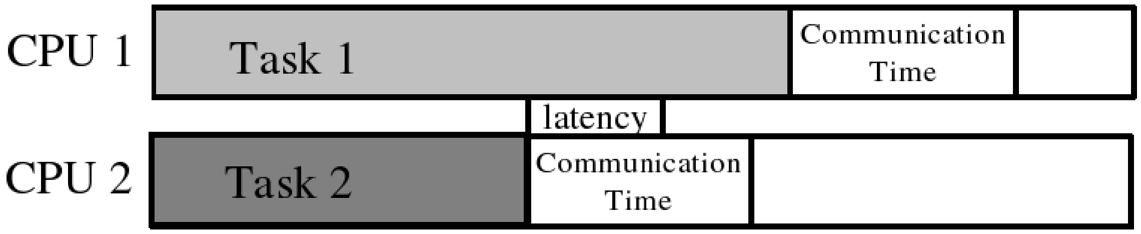

Figure 2
The latency of a communication between CPU 1 and CPU 2 is being absorbed

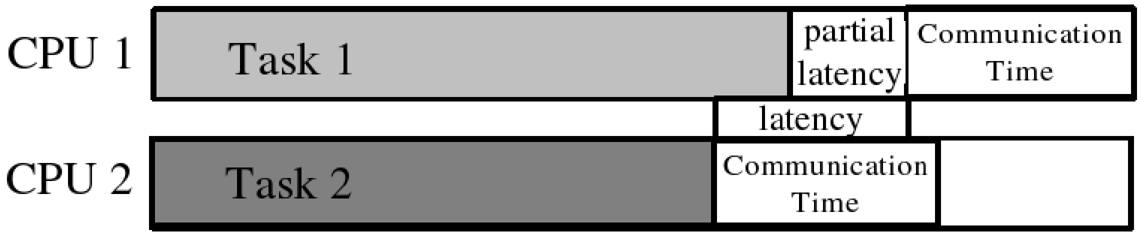

Figure 3
The latency of a communication between CPU 1 and CPU 2 is being partially absorbed

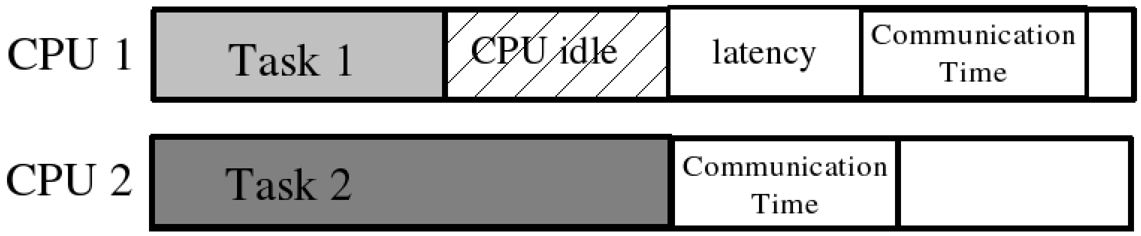

Figure 4
Task 1 takes the full latency of the communication between CPU 1 and CPU 2

Also consider the situation where a processor has completed a task but cannot move onto a new task because it does not have the prerequisite information. This is the case in Fig. 4. CPU 1 has completed its current task, but cannot begin its next task until receiving a communication from CPU 2.

When two consecutive tasks are scheduled on the same processor, no communication or latency time is incurred. That processor already has the information it needs to begin processing the next task. (In some cases, that task would require information from an additional task before beginning.) As a result, certain jobs can be scheduled on a single processor and be completed in less time than if the job was spread over multiple processors. For example, consider our sum of a large number of integers. Sending these numbers over a slow network connection (high communication and latency costs) to be calculated and then waiting for the response would almost certainly take longer than just having the originating processor sum the numbers itself. That being said, scheduling linked tasks on the same processor to eliminate communication and latency is a powerful tactic in our approximation algorithms.

Representation of the Problem

A precedence graph is used to represent the problem of scheduling tasks on parallel processors. A single graph will correspond to a larger job which has been divided into a series of tasks which may be handled separately. Each of these individual tasks is represented by a node on the graph, where each task has an execution time, representing the amount of processor time needed to complete that task. The edges on the graph are used to indicate which tasks depend on the other tasks. The edges between the nodes can also be thought of to represent the communication that occurs between the tasks. Therefore, each edge contains a communication time, which represents the amount of

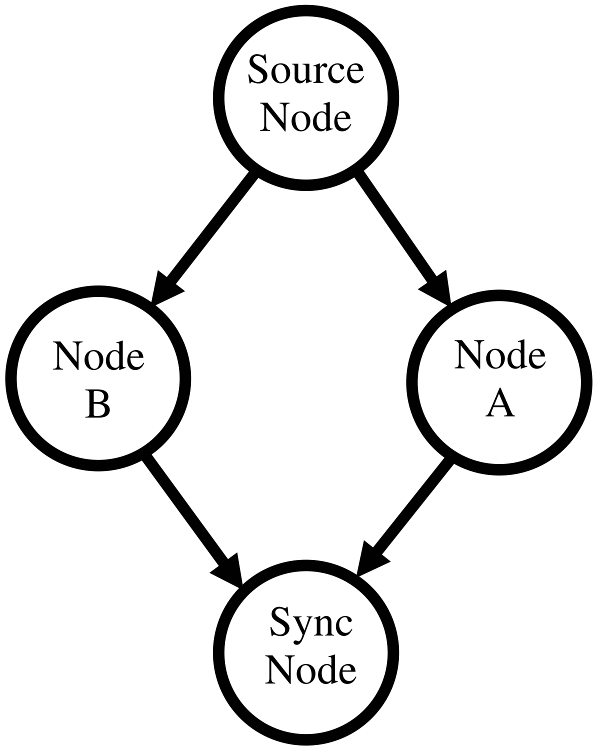

Figure 5
An example of a simple precedence graph.

time that is taken sending the prerequisite information for that task to be run.

The graph itself is a directed acyclic graph; each edge's direction denotes which node needs the other node to be completed before it can be run. For example, in Fig. 5 on the next page, the source node must execute before Nodes A or B can execute. Likewise, the sync node cannot complete until Nodes A and B have completed. A single root node, called the source node, serves as a starting point for each graph. A sync node has been generated at the bottom of each graph. The sync node is where the task completes itself; it is always the last task to be executed and it is always run on the same processor as the source node. These additional constraints make the problem more applicable to actual scheduling situations (We usually want the solution to be returned to the computer which began the calculations). [3] Our approximation algorithms are given one such directed graph and will calculate a schedule for the tasks it represents.

Approximation Algorithms

The approximation algorithms are intended to return a near-optimal solution in a very short amount of time. Since optimal solutions are so time-consuming to find, we sacrifice a small amount of accuracy in our schedules for an enormous time savings. A given schedule includes information on the order in which the processes are to be run and which processor is responsible for each task. The algorithms have to take into consideration when a processor will send out a task, because this process occupies the CPU just as much as actually executing a task.

There are many strategies to making good approximation algorithms. One thing we always take into account is to make sure that each task communicates out to other processors in a timely manner. I'll use Figure 5 as an example, let us say that Node A and the Source Node are scheduled to CPU 1 and Node B is scheduled to CPU 2. Once the source node finishes execution, it needs to communicate information to Node B before Node B can begin execution. Since Node A is on the same CPU as the source node, no communication is necessary before it can run. In cases like this, it is important to send out communications to all waiting nodes before executing things on the same processor. If Node A is executed before Node B, CPU 2 will sit idle waiting for a communication. It was proven [3] that it is *always* faster to communicate tasks out in this fashion. This is illustrated on the next page in Fig. 6

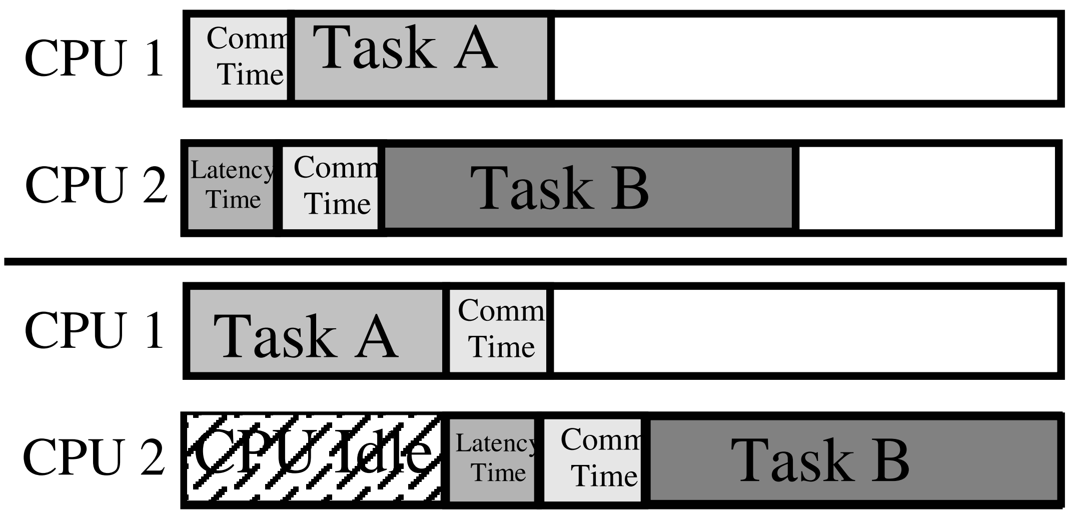

Figure 6
Top: CPU 1 sends out a communication before executing Task 1
Bottom: CPU 1 sends out a communication after it executes Task 1

Another strategy, mentioned earlier, is to keep dependent tasks running on the same processor. By keeping dependent tasks on the same processor, communication and latency times can be nullified for a particular edge of the precedence graph. This is because if a task is being executed on the same processor as its parent, there is no need for communication. The process already has the information it needs. These 2 concepts have become the base rules behind all of our successful approximation algorithms to date. We have developed 2 strong algorithms this year, both based on these strategies. These are the Longest Path 2 algorithm, and the Longest Communication Path algorithm

Longest Path 2 Algorithm

Our first algorithm is a modified version of the Longest Path algorithm developed in 2003 by Joel Nelson and Daniel Wespetal [2]. We call it the "Longest Path 2" algorithm. The original version of the longest path algorithm didn't take our first strategy into account. The CPUs would sometimes run tasks of their own before sending out information to waiting processors, causing processors to sit idle. Our research team spotted this bug and fixed it; we now believe that this is one of our better approximation algorithms. The algorithm will look only at the execution times of tasks, and choose the path down the graph with the highest execution time. Here is some pseudo code for this simple, yet powerful algorithm:

```
1. Until all nodes are scheduled:
    a. Find all the nodes that have no parents, or have only parents that have already
       been scheduled.
    b. Calculate the longest path of execution times down the graph from each of
       these possible start nodes.
    c. Of these calculated paths, take the longest path of these paths calculated from
```

```
        the start node.
    d. Schedule all these nodes to the next unused processor, starting at 0 and
        incrementing each time through.
```

Longest Communication Path Algorithm

Our second algorithm took the concept of the Longest Path Algorithm and applied it to the communication times of the graph. We find the longest *communication* pathway down the graph with this algorithm. By following this process, we are attempting to minimize the communication and latency times, by nullifying the ones that are greatest. Nodes always have to execute, but communication and latency can be nullified, so we try to nullify the greatest sources of communication time with this algorithm. This algorithm is currently our best algorithm for solving graphs with variable communication times. The pseudo code for this algorithm is almost identical to the last algorithm:

```
1. Until all nodes are scheduled:
    a. Find all the nodes that have no parents, or have only parents that have already
        been scheduled.
    b. Calculate the longest path of communication times down the graph from each
        of these possible start nodes.
    c. Of these calculated paths, take the longest path of these paths calculated from
        the start node.
    d. Schedule all these nodes to the next unused processor, starting at 0 and
        incrementing each time through.
```

Generating Results with the Test Bed

Our test bed is where we compare our algorithms. The test bed consists of a large number of randomly generated precedence graphs. These graphs are created and stored so that each algorithm will be tested on the same set of graphs. This makes comparison between algorithms easy and keeps all of the project data well-organized. Previous researchers working on this project populated the test bed with graphs and ran brute force algorithms on them in order to obtain optimal and worst-case times. However, these graphs did not model communication time being variable. The test bed had to be expanded with a new set of graphs for this project, and new brute force data needed to be generated.

Graphs are stored in the test bed as XML data files. These files are human-readable, and can be constructed by hand or by our random graph generator tools. The results that were gathered for this project consist entirely of randomly generated graphs. Our graphs were split into categories based on the number of nodes in each graph. Graphs were generated with 4 to 11 nodes. Edges were added randomly, and communication times and execution times were added randomly. Each graph also has a randomly generated latency. All times were randomly generated within specified ranges. Execution times ranged from 1 to 1000, Communication times ranged from 1 to 250, and latency times ranged 1 to 75.

Algorithm Results

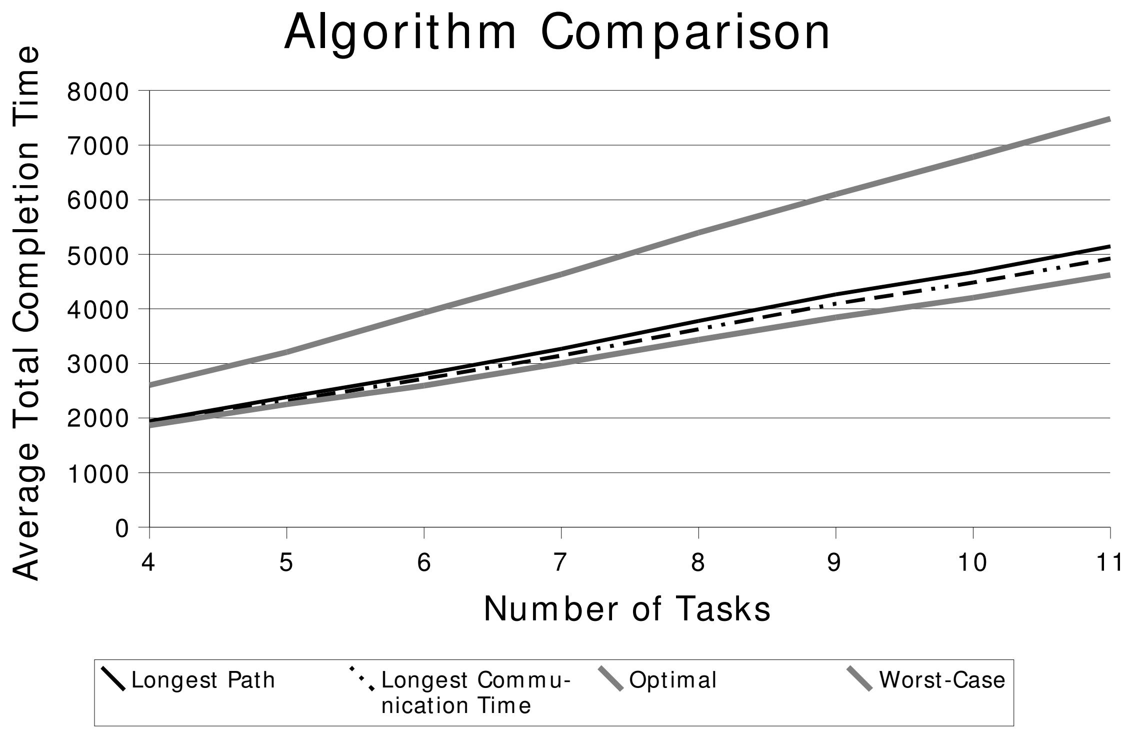

Figure 7
A comparison of algorithm results

As you can see on the graph, the Longest Communication Path algorithm gives better times than the Longest Path 2 algorithm. The difference may not seem important at a glance; however, algorithms tend to diverge from the optimal solutions at a fairly constant rate. Since the rate at which the Longest Communication time algorithm diverges from the optimal solution is smaller than that of the Longest Path algorithm, it will become an increasingly relevant difference as the number of nodes becomes large.

The fact that the Longest Path 2 algorithm does so well in this new environment with variable communication times is impressive, and speaks well of its original creators. The fact that the Longest Communication Time algorithm does better than the Longest Path algorithm is no real surprise, however; the designers of the Longest Path algorithm never had to deal with variable communication times. They could not have used a Longest Communication Time algorithm if they had wanted to - all the edges in their graphs had the same communication times.

The most significant result from the table is how close to optimal our approximate solutions are. Both of the algorithms are obtaining very close to optimal results. We added an entirely new variable to our graphs, and yet even our older algorithms can still hold their ground. This is a great victory for us - it tells us we've been pointed in the correct direction

Future Research

In the next year we plan to bring our theoretical research into a practical environment by implementing a parallel computing system using the Message Passing Interface (MPI) interface. Right now, our research is purely theoretical. In order to bring our research to the next level, we need to prove that our theoretical implementations work in a real-world environment. Parallel computing is starting to take off right now. The fastest supercomputer in the world, IBM's BlueGene, uses MPI to communicate between its 60,000+ processors. There are currently very few tasks suitable to be run on a platform such as BlueGene. If more efficient ways can be found to schedule jobs in a parallel environment, it could open up many new directions for parallel computing.

We plan on first doing research on the MPI protocol, and finding an implementation suitable to our needs. After that, we will find a problem that we can adapt for a parallel environment, and integrate it into an MPI environment. We hope to model actual problems and use our approximation algorithms to come up with appropriate schedules for running them in parallel on networked machines.

As always, we plan to improve our approximation algorithms even more. We should be able to do some simple optimization by analyzing our data. We already have some ideas on how we can do this. First, we can compare our results for each graph to the optimal solutions and figure out which cases we did worst on. We can then use this data to figure out what special cases we may have missed in our code. Right now our code is very general and getting down to more specific cases should help us optimize existing code or develop ideas for new algorithms.

We also have ideas for algorithms that incorporate more variables. Right now, both of our best algorithms are very narrowly focused. One only looks at execution times and the other only looks at communication times. They work very well for what they are, but the best algorithms should look at a combination of these two factors and judge the schedule accordingly. For example, looking at factors such as whether the communication time is greater than the execution time. In that case, a task should always be scheduled on the same processor as its parent node, because time would be lost to communication if it is put on a different processor.

Conclusions

The added variable of communication in our problem changes the applications of the task allocation problem greatly, and allows us to model many more situations than we could previously. The fact that our algorithms still return near-optimal results on this new instance of the problem is a great victory for our research team. Our goals were to add in the variable communication and latency, (which involved rewriting a lot of code from scratch to properly implement) and improve our scheduling algorithms. We succeeded on both accounts. We now have a more practical model for our problem, and a more efficient way of scheduling it.

References

[1] Adeola Adewola, Jon Quarfoth, Dan Wespetal, and Dian Lopez (Advisor), "Development of a Test Bed to Determine Performance of Scheduling Algorithms," Proceedings, Midwest Instructional Computing Conference (MICS), Morris, April 2004.

[2] Daniel Wespetal, Joel Nelson and Dian Lopez, "Approximating a Parallel Task Schedule using Longest Path," Proceedings, Midwest Instructional Computing Conference (MICS), Duluth, April 2003.

[3] Hsu, T.s., Lee, J., 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.