# Design and Testing of Parallel/Distributed Precedence Tree Scheduling Algorithms

**Dian R. Lopez**
**Department of Computer Science**
**University of Minnesota-Morris**
**lopezdr@umn.edu**

**Daniel D. O'Brien**
**Department of Computer Science**
**University of Minnesota-Morris**
**obriend@mrs.umn.edu**

**Jake Krohn**
**Department of Computer Science**
**University of Minnesota-Morris**
**krohnk@mrs.umn.edu**

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

**Rhett Wilfahrt**
**Department of Computer Science**
**University of Minnesota-Morris**
**wilf0015@mrs.umn.edu**

## Abstract

Our research focuses on the NP-hard problem of scheduling tasks on a pool of identical workstations in a network where message passing is used for data transfer and communication between processors. The work is based on a priority task graph representation of jobs that contain sequential segments with varying dependencies. We have created a testbed in which to store benchmarks for a distributed processor allocation model. An approximation algorithm for the scheduling problem was then developed. Software was written that runs the approximation algorithm on thousands of random precedence graphs to obtain completion times in order to develop an average running time. Finally, we used PVM (Parallel Virtual Machine) and a private distributed system of five workstations to measure the actual and practical results of our algorithms.

## Introduction

Distributed computing is the utilization of multiple workstations to simultaneously execute segments of an application that would be done sequentially on a single workstation. The result can often be a shorter total execution time of the application, allowing more computation to be done in any given amount of time. Distributed computing can be implemented on a pool of workstations connected over a network where message passing is used for data transfer and communication between processors.

As demand for computational speed rises and the price of processors fall, distributed computing is becoming a feasible and attractive option for many companies. This, in turn, increases the demand for applications and algorithms that use these systems.

When an application or program is to be run on a distributed system of workstations, it must first be separated into independent sequential sections that can be executed simultaneously. In this paper we will refer to the entire application as the *job* and the independent segments as *tasks*. Once this division has been done, the question then becomes: what is the most efficient way to assign, or *schedule*, these tasks among the available workstations so as to minimize the total computational time of the entire job (See Figure 1). Our research has focused on the design and development of algorithms that deal with this NP-Hard optimal scheduling problem.

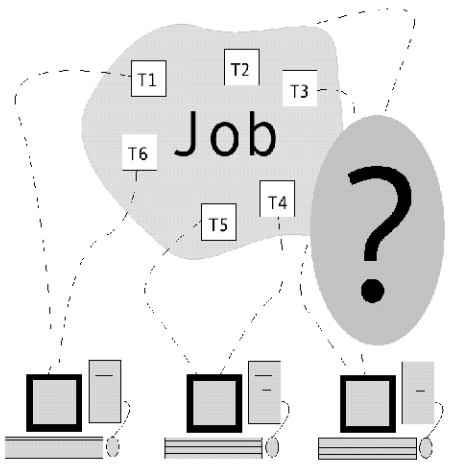



**Figure 1. What is the best way to schedule tasks (T1 – T6)?**

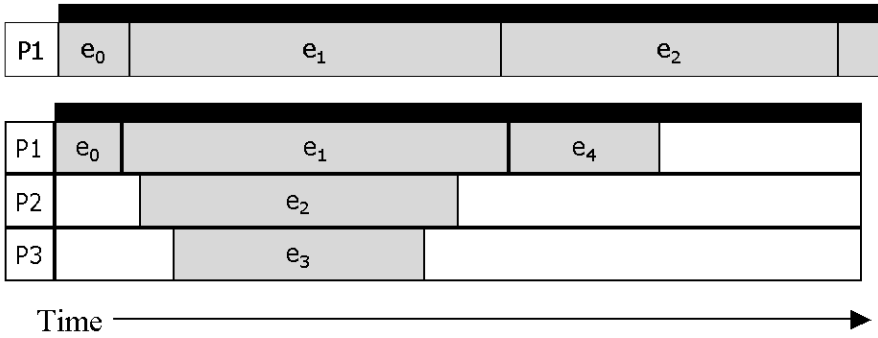
Our research differs from most other work being done in this area in that we take into account practical limitations such as network latency and communication time between processors. Much prior research has been on a purely theoretical basis and ignores these practical limitations. The reasoning is that these practical limitation times are small enough so as to be insignificant to the overall theoretical running time of the problem. On a real-life system, however, these time limitations can play a major factor in altering the optimal allocation, and they are likely to be different for every system.

In this paper, we begin with a short explanation of the benefits of parallel computing. We move on to introduce our model for this problem. A test-bed is then described which leads into the development of an approximation algorithm. Analysis of results has shown the approximate solution to be, on average, within 4% of the optimal solution. The paper will conclude with the implementation on an actual system of the makespans produced by our theoretical algorithms. Results of this implementation show that our theoretical work has relevant implications to real world distributed computing systems.

## Why run in parallel?

Parallel processing allows multiple parts of a single job to be simultaneously computed, thus decreasing total computation time. The two diagrams shown in Figure 2 are *makespans* of a job run sequentially (the top diagram) and in parallel (the bottom diagram). A makespan is a measure of the total execution time of a given schedule. In the diagram, each column represents a different processor on the network, and time passes along the horizontal axis from left to right.

The job being computed in Figure 2 consists of 5 tasks, labeled $e_0$ - $e_4$, and the width of each rectangle represents the execution time of the individual task. When running tasks in parallel, time is saved through concurrent execution. In the bottom makespan of Figure 2, tasks $e_2$ and $e_3$ have been sent to separate processors and can therefore execute simultaneously with each other and task 1. Given the situation depicted in Figure 2, it is easy to see that the job will complete earlier when run in parallel than when the tasks are executed sequentially.




**Above:** Multiple tasks of one job, run in sequential order on one processor
**Left:** The same job, run in parallel on multiple processors

**Figure 2. The Makespans of a job run sequentially and in parallel.**

## Model Representation

Each job is composed of several tasks that can be executed independently, but have varying dependencies among each other. Therefore, the job as a whole can be represented by a precedence graph, where each task is a node of the graph. (See Figure 3). In order to create a precedence task graph for a job, the job must first be divided into its sequential sections. The interdependencies of these tasks must be known along with

their individual execution times. Our work is based on a precedence task graph model that is an expanded version of the send-receive model originally discussed in [1].

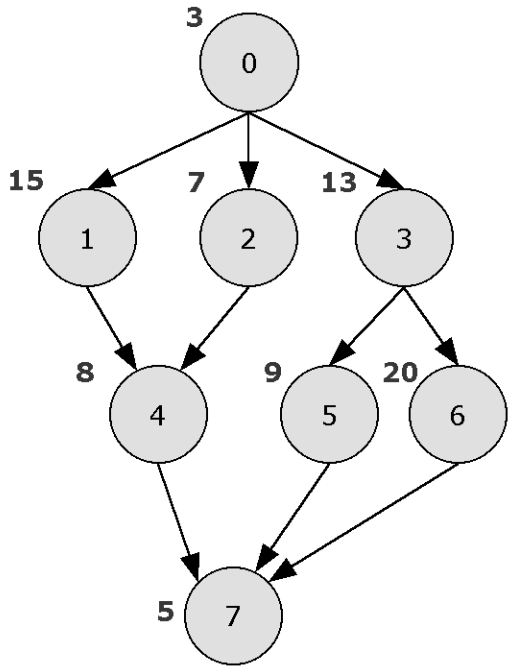



**Figure 3. The graph representation of a job consisting of eight tasks labeled 0 – 7. (Execution times of each task shown beside the node)**

Some of the salient features of this model will now be discussed. All jobs will start at a single root node (node 0 in Figure 3). The execution of this task usually consists of some overhead and setup that needs to take place before the other tasks can be executed. Every graph will also end at a single node at the bottom of the graph (node 7 in Figure 3). This is necessary to collect all the data produced by the preceding tasks and perhaps perform some culminating operations. The levels between the first and last (nodes 1 – 6 in Figure 3) can take the form of any valid directed graph as long as:

1. For every node, there exists a path to the bottom node and
2. There are no cycles.

Each edge from one task to another denotes a necessary transfer of data. For example, in Figure 3 a directed edge exists between nodes 1 and 4. This means that before task 4 can execute, it needs some data from task 1. Another way of saying this is that task 4 is *dependant* on task 1. Therefore, if tasks 1 and 4 were to be executed on different processors, lets say the former on processor 1 and the latter on processor 2, some necessary data must be sent from processor 1 to processor 2 over the network before task 4 would be able to execute. This implies the rule that no task can be executed until all of its predecessors have been executed. So, for example, since task 4 can't execute until tasks 1 and 2 have been executed, we can infer that task 0 has already been executed as well.

Since each task can be computed independently of the others, they can all be executed on different processors, with one exception. Our model forces the first and last task to be

executed on the same processor, for example, the Master workstation of the network. Thus all information is sent out from one processor and then collected at that same processor upon completion. Therefore, the maximum number of processors that can be used to run a job with n tasks is n-1 processors.

**Communication Time and Latency**

One might then ask the question: "Given a job with a known number of independent tasks, wouldn't the quickest way to complete the entire job always be to put each task on its own processor?" The answer is no, and the reason is network *latency* and *communication time* and the role they play in the data transfer indicated by the interdependencies of the precedence task-graph.

As stated earlier, when one task is dependent upon another and they are to be executed on separate processors, a segment of data must be sent across the network. In our model, this incurs delays called latency and communication time. *Latency* in a network is the amount of time it takes for the first bit of data sent from one processor to arrive at the destination processor. This is essentially "dead time" for the receiving processor, as it may be idle while it waits. In our algorithms, we assume that latency is the same between all processors in the system. This is done for simplicity, though it is likely in the case of an actual distributed system for latency between processors to be so close to equal that the difference is unimportant.

*Communication time* is the total amount of time it takes for every bit of the segment of data to be sent by a processor or received by a processor. Thus *both* the sending and receiving processors incur communication time. In a real-world problem, the communication time would be different for every pair of tasks, but in our algorithms, for simplicity, we assume a uniform communication time between all tasks.

Figure 4 is a visual depiction of the concepts of communication time and latency. A section of graph is shown (on the left) that indicates that task 1 is dependant on task 0. They are going to be executed on different processors, and the result is shown in the makespan (on the right). Remember that the horizontal axis indicates the passage of time, so the width of each box indicates duration. Task 0 is first executed (rectangle $e_0$) on Processor 0, the column labeled P0. Task 1 is then to be executed on processor 1 (P1), so data must be sent from processor 0 to processor 1 before task 1 can execute. The fist bit of this necessary data is sent from processor 0 at the point marked by the letter x, and that bit arrives at processor 1 at the point marked by the letter y. The amount of time that transpires between x and y, highlighted by the rectangle labeled L, is latency, or *dead time* for processor 1 in which it is doing nothing but waiting for data. The rectangle labeled $c_1$ denotes the amount of time it takes to send all the necessary data from processor 0 to processor 1. Once processor 0 has sent all the data, it is then free to work on other things. Processor 1, after waiting for the first bit of data, must then collect all the data as it comes in from processor 0 ($c_1$ on column P1). Once all the data has been collected, task 1 can then execute (rectangle $e_1$).

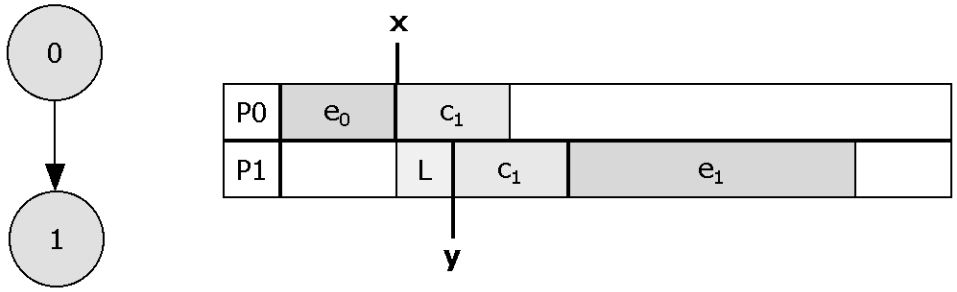



**Figure 4. Depiction of Communication Time and Latency of a single pass of data.**

It is easy to see how these two network limitations can effect the running time of an application executed on a distributed system. In an extreme example, lets say that, for the job in Figure 3, it takes 10 seconds to execute each task, communication time between all tasks is 50 seconds and latency between all processors is 30 seconds. In this case, total execution time of the entire job would be 400 seconds if each task were executed on its own processor, where it would only take 80 seconds to execute the tasks sequentially on a single processor. It will actually take longer to run the program in parallel than to do it sequentially!

Since the execution times of the individual tasks remain unchanged, in order to find the minimum total execution time of a job, we need to find the order of execution of tasks on processors that maximizes concurrent execution of tasks while minimizing total incurred communication time and latency. This is called task-scheduling.

## Problem Test-bed

One thing that makes the task scheduling problem so interesting and difficult is that it is *NP-Hard*. The only way to find the optimal solution is to look at every possible solution and choose the best. Therefore, the only way to find the *optimal* scheduling solution for a given job is through recursive brute force computation. Because of this, we created a testbed of solutions in order to compute and store the optimal results of the problem as benchmarks in a form that is versatile enough to be utilized by us and other researchers in the field.

Our brute force algorithm finds all possible processor/task combinations (schedules) for a given directed task graph using a depth-first implementation. An unlimited number of processors are assumed to be available when finding all possible schedules. Another algorithm then takes each of these schedules and, using randomly defined latency, execution, and communication times, computes its *makespan*, which is the total time required for the complete job to execute under the given schedule. The schedule with the shortest makespan is then our *optimal* schedule for those particular random values. The worst-case schedule, or the schedule that gives us the longest makespan, is also recorded.

This process is then repeated for thousands of variations of latency, execution and communication times on varying precedence graphs.

Since the problem is NP-hard, the test-bed can only find solutions for very small job graphs since the number of possible solutions grows exponentially as additional tasks are added to the task graph. The total number of possible combinations for a fully connected graph of n nodes is n!. But even though the directed task-graphs are not fully connected, there is an extra dimension of growth that greatly expands the realm of possible solutions. Given any order of task execution, the tasks can then be distributed in all possible ways among up to n-1 processors.

This rapidly expanding possibility set was extremely apparent when running precedence task graphs of different sizes. For a time, we stored all possible schedules as task/processor text strings in files, and the size of the files grew exponentially with the size of the graphs as shown in Figure 5. The largest graphs we could test, depending on the amount of possible parallelisation of the tasks, was between 9 and 11. After eliminating the write to the file, we attempted to run a specific 12-node task-graph and were forced to halt execution after three continuous months of execution, over a million possibilities computed, and with lower end estimates having it 1/10 completed.

| Input Size (nodes) | File Size |
|---|---|
| 5 | 6.7 kB |
| 6 | 38.9 kB |
| 7 | 2.9 mB |
| 8 | 80 mB |
| 9 | 266 mB |
| 10 | 1.43 GB |
| 12 | >> 2 GB |

**Figure 5. Size of graph and the file containing string representation of all possible schedules.**

## Approximation Algorithm

Given the running times of our test bed, it is apparent that attempting to find the optimal schedule for medium to large size task-graphs containing tens, hundreds, or even thousands of tasks through brute-force computation would take an impractical amount of

time. Therefore, in order to achieve solutions for NP-hard and NP-complete problems such as the task-scheduling problem, *approximation algorithms* are used.

An approximation algorithm is an algorithm that sacrifices some degree of accuracy in exchange for a solution that is produced in a reasonable amount of time. In other words, instead of spending an extensive amount of time computing the optimal solution, a good approximation algorithm will run for only a fraction of that time and give us a solution that is *close* to the optimal. In theoretical terms, finding the optimal solution will have an exponential time complexity, and by implementing an approximation algorithm we can reduce the complexity to polynomial time.

It was clear that the next phase of our research was to develop an approximation algorithm for the task-scheduling problem. As we can see through the use of greedy algorithms, simple algorithms are often the most effective. Therefore, with simplicity in mind, we developed a greedy approximation algorithm whose actions can be split into two distinct steps, or *decisions*. First, the algorithm decides which task to execute next; then it decides which processor to execute it on.

**The Task Decision**

In order to make the first decision, the algorithm uses a variable called *summed_execution_time*. The summed_execution_time of a given task is the sum of the execution times of all of its descendants. The task with the highest summed_execution_time is the task that is chosen. If there are two or more tasks with equal summed_execution_time, then the task with the largest execution time is chosen. The idea is to quickly execute tasks that have a lot of tasks dependent upon them. In this way, the computation-intensive parts of the graph can be sent out to processors early so as to maximize the amount of concurrent execution.

**The Processor Decision**

The second decision is based on a method used in [1]. To illustrate this method, lets assume that a task has just finished executing. All of its children, or dependent tasks, are now able to execute. According to this method, the parent task sends off all tasks but one to a different processor for concurrent execution. The final remaining task it keeps on the same processor as itself and, in that way, the last task doesn't incur any communication time or latency.

In our algorithm, once the next task to be executed is chosen, its siblings are then checked. If all of its siblings have already been placed on a processor (or if there are no siblings) then it is the only unassigned child-task of its parent, and should be executed on the same processor as its parent unless it is the final task of the entire job. If siblings exist that have not been assigned to a processor, then the task is assigned to an unused

processor. As in the test-bed, the approximation algorithm assumes the availability of an unlimited number of processors.

These two steps are repeated until all tasks have been assigned to a processor. Thus far, the results of the approximation algorithm have been very promising.

## Results of the Approximation Algorithm

In order to evaluate our approximation algorithm, we wanted to test it on thousands of random task-graphs in order to gauge its effectiveness and to obtain average completion times that will be used to develop an average lower bound running time. To aid us in this process, we developed software that creates hundreds of random precedence task-graphs and then runs our algorithms on them.

### The Random Graph Generator

The random graph generator creates a large number of graphs with n nodes, where n is a predetermined number that is fed to the code as an argument. Each graph generated has six different random characteristics.

1. The number of levels in the graph.
2. The number of nodes on each level (except for the first and last)
3. The percentage of completeness. For example, 100% would be a complete graph where 50% would have half of all possible edges.
4. Which nodes the directed edges connect. Edges that create cycles are not allowed. Also, every node will have a path to the final node.
5. Communication time between tasks. (One value)
6. Latency between processors. (One value)

The generator creates $x^6$ graphs where x is passed to the code as an argument. Each of the six characteristics is forced to be distributed fairly evenly across the range of possible values. For example, if you are creating graphs with 10 nodes, possible values for the "number of levels" characteristic are between 3 and 10 levels (with 3 levels there is one node on the top, one on the bottom, and 8 on the middle level, and with 10 levels there is one node on all levels). If you simply randomly generate 4 different numbers between 3 and 10 then there is a fair possibility that you will end up with numbers that are all 8, 9 or 10. This would create very degenerate graphs that don't adequately test the approximation algorithm. Instead, the generator is forced to randomly select values across the entire range. In this way we are guaranteeing a widely representative sample, and are improving the chances of acquiring best, worst, and average cases within every characteristic.

**Approximation Algorithm Results**

Results of the approximation algorithm thus far have been very good, as the worst it has performed on any given graph is 1.5 times (or 50% off of) the optimal makespan. The top table of Figure 6 displays data on fourteen different graphs of various sizes.

Using the random graph generator and some aggregation software, we were able to measure average optimal and approximate makespans over 500 graphs of size 5, 6, 7, and 8. Over these sizes of graphs, the approximate makespan, on average, was between 1.022 and 1.044 times (or 2.2 to 4.4 percent off) the optimal makespan. This is exceptionally close, which means our approximation algorithm is very effective.

| # Nodes | Optimal Makesp | Approx. Makesp | $^{Approx}/_{Opti}$ |
|---|---|---|---|
| 1 | 5 | 5 | 1.00 |
| 2 | 13 | 13 | 1.00 |
| 3 | 16 | 16 | 1.00 |
| 4 | 14 | 16 | 1.14 |
| 5 (I) | 38 | 46 | 1.21 |
| 5 (II) | 38 | 48 | 1.26 |
| 6 (I) | 45 | 52 | 1.15 |
| 6 (II) | 44 | 48 | 1.09 |
| 7 | 49 | 53 | 1.08 |
| 8 | 57 | 59 | 1.04 |
| 9 (I) | 69 | 100 | 1.45 |
| 9 (II) | 63 | 72 | 1.14 |
| 9 (III) | 47 | 49 | 1.04 |
| 10 | 60 | 73 | 1.21 |

| # Nodes (500 random graphs) | Avg. Optimal Makespan | Avg. Worst-case Makespan | Average Approximate Makespan | Approx/opti. |
|---|---|---|---|---|
| 5 | 224.0 | 299.0 | 229.0 | 1.02232 |
| 6 | 242.0 | 362.0 | 251.0 | 1.03719 |
| 7 | 275.0 | 432.0 | 284.0 | 1.03273 |
| 8 | 292.0 | 496.0 | 305.0 | 1.04452 |

**Figure 6. Table of individual graph makespans (top). Table of aggregate makespans of 500 random graphs of a specified size (bottom).**

## Makespans Produced for an Example Job

In order to illustrate more clearly how a task graph translates into a makespan, makespans for an optimal schedule and the schedule produced by our approximation algorithm are shown in Figure 7 for the job displayed in Figure 3. The top schedule is the one produced by our approximation algorithm; the bottom schedule is an optimal one produced by our brute force algorithm. The execution times for each of the eight tasks are shown in Figure 3 by the numbers outside the nodes. The units are theoretical, so they could be seconds, microseconds, milliseconds, or any other measurement of time. Communication between tasks is set to four units, and latency between processors is set to two units. It should be noted that all of these times were chosen randomly within realistic ranges. The task executions are shown in rectangles labeled with an e, communication times by rectangles labeled with c, and latency is not marked since it is idle time for the processor. The numbers behind the letters indicate the corresponding task number. Even though the schedules are quite a bit different, with the approximate schedule requiring one less processor, the total computation time for both schedules is very close, with the approximate total execution time of 59 units being only 3.5% off of the optimal total execution time of 57 units.

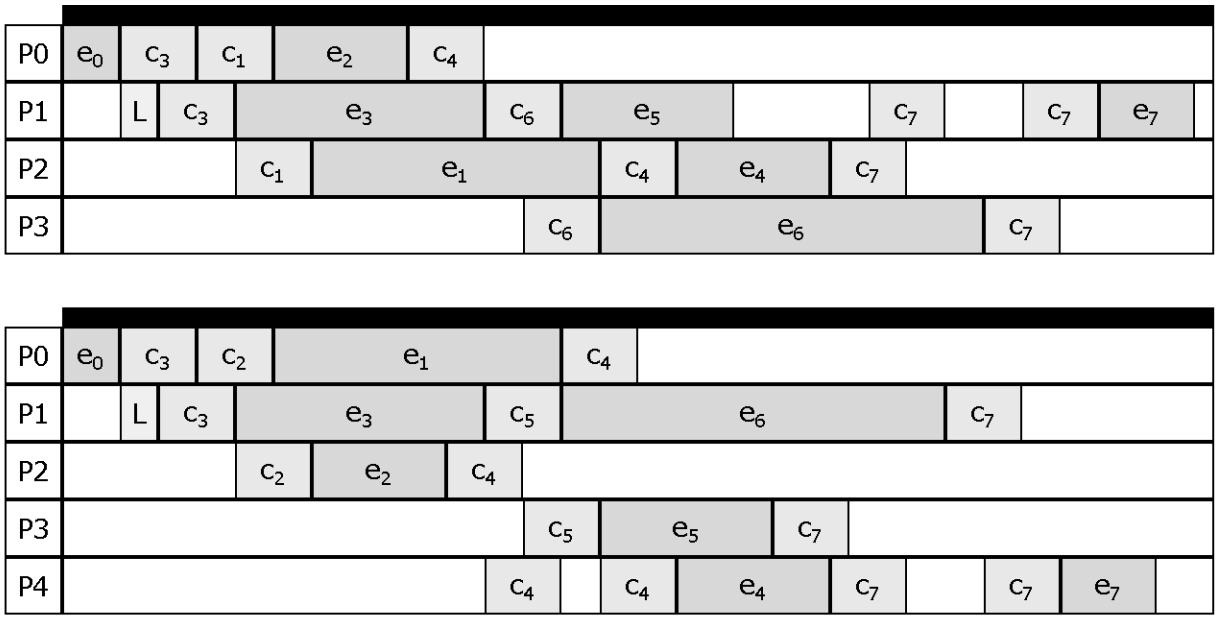



**Figure 7. Makespans of the job in Figure 3. Top is the schedule given by the approximation algorithm and total execution is 59 time units. Bottom is an optimal schedule and total execution is 57 time units.**

## PVM Implementation of the Problem

After collecting the results of our algorithms on a theoretical system, the next step was to see how they performed on an actual distributed system. In this way we could compare the actual running times of the schedules produced by our algorithms with the theoretical

running times and measure how much they differed. In order to run jobs on a distributed system, we used PVM (Parallel Virtual Machine), which is software that allows a computer program to be divided into independently executable modules and sent to different processors in a network in order to execute segments of the code in parallel.

We chose one precedence task graph that is representative of a practical parallel problem to test our algorithm and system timing results. The job we emulated was solving the factorial problem, with its directed graph containing 8 total tasks, 1 in the first level, 4 in the second, 2 in the third, and 1 in the fourth. The first node is connected to all the nodes in the second level, and from there it is an inverted binary tree (see Figure 8).

**The Theoretical Algorithms**

In order to test our algorithms, we needed execution times for each task, communication time between tasks, and latency between processors. Actual problems to be done in parallel will most likely have varying execution times per task, which could be measured by executing each task separately. For this theoretical problem, we randomly picked execution times for each task.

The communication and latency times, however, had to be measured from the distributed system we were running the schedules on. In order to find latency on our distributed system, we timed the sending and return of a single integer passed between two processors. Since the measured time was two-way travel, the time had to be divided by two, giving a one-way latency of 300 μs (microseconds). This time can be considered almost entirely latency since the time it takes to communicate a single integer is infinitesimal. In order to emulate communication, we decided that every task would pass on an array of 10,000 integers. Communication time was then found by sending and receiving the array, dividing the measured time by two, and subtracting out latency. It turned out to be 11,000 μs, and it will be the same between all tasks since the same amount of data is passed between each task.

After obtaining the time values for communication, latency, and simulated execution times, we could run the approximate and brute-force optimal scheduling algorithms on the precedence task-graph. These theoretical algorithms produced an optimal, approximate, and worst-case schedule and their associated makespans.

**PVM Implementation**

The next step was to implement these three schedules on the distributed system and to compare the actual running times with the theoretical ones. We used PVM to accomplish this implementation. A timing function was used to measure total computation time of the job, and sleep functions were used to emulate the individual task execution times randomly chosen.

We analyzed the PVM results by comparing the total computation time of the approximate schedule, the optimal schedule, and the worst-case schedule on the distributed system against the make-spans produced by the theoretical research algorithms.

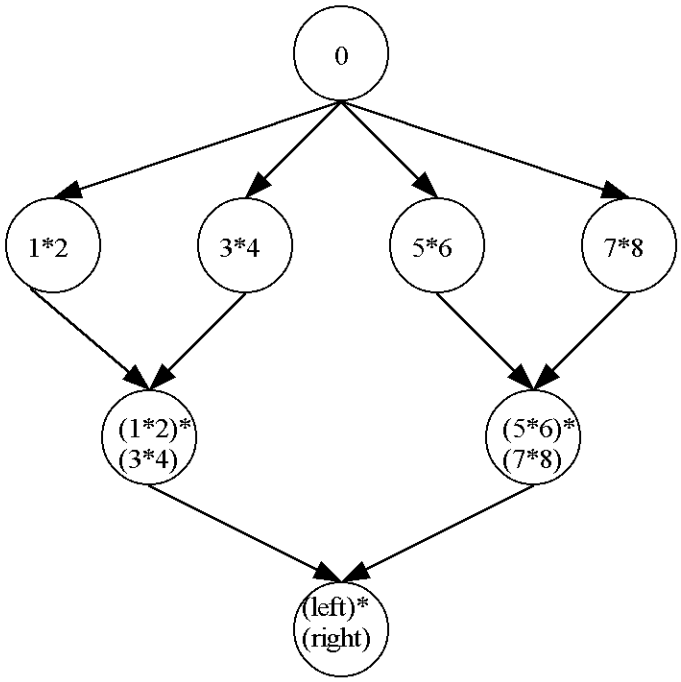



**Figure 8. The eight-node factorial directed task graph**

It should be noted here that there are usually many possible optimal schedules for any given directed task graph. In the case of this problem, our approximation algorithm produced one of the optimal schedules. Therefore, in order to have three schedules to test, we chose an optimal schedule different from the approximate schedule to represent the "optimal schedule" referred to in the remainder of the paper.

**PVM Results**

The PVM running time for the optimal schedule was 561,020 μs and the predicted time was 453,600 μs. The approximate schedule makespan prediction was 453,600 μs, but the PVM program ran in 575,493 μs. The worst-case schedule ran for 803,438 μs on PVM, and the predicted time was 868,900 μs. The exact reasons for the differences in predicted algorithm makespans and actual measured PVM times are not clear, though they are probably due to the unknown internal mechanics of PVM and unpredictable and unaccounted for network fluctuations. The shorter-than-predicted worst-case running time, for example, is probably due to some PVM optimizations. However, the predicted and actual total execution times are very close, especially considering the size of the measured units, with realized PVM running times varying from predicted theoretical

running times by a maximum of 23%, or 107,000 microseconds. These results indicate that our theoretical results can be effectively implemented on a real-world system.

## Conclusion

This paper has dealt with the NP-Hard problem of optimal task scheduling on a distributed system of workstations where message passing is used for communication between processors. The problem was discussed, and a precedence task-graph model for representing a job or application of independent sequential segments was introduced. After highlighting the model's salient features, with special attention given to communication time and latency, the testbed was described that uses brute-force computation to find optimal schedules for given graphs and stores them as benchmarks that can be used by ourselves and other researchers in the field.

A two-step approximation algorithm was then used to find possible solutions to the problem in polynomial time. The schedules produced by the approximation algorithm were compared to the brute-force optimal solutions, and the results were analyzed. The approximation algorithm performed well, as, on average, it was off from the optimal by only four percent.

The final pages of the paper discuss the implementation of the schedules produced by our theoretical algorithms on an actual distributed system through the use of PVM. A representative problem was chosen, and the theoretical running times of the schedules were within 23% of their measured actual running times. This indicates that our theoretical research can be duplicated on a physical system.

We have also done some work on imposing an option to limit the number of processors available for the task-scheduling problem. Formerly, we had assumed that an unlimited number of processors were available when computing the approximate and optimal solutions. This change will allow our algorithms to more accurately addresses limitations encountered on real-world systems. Testing and data collection for this implemented change were not sufficient at the time of the writing of this paper to be included, so it will be left for a future exposition. Future work also includes optional variation of communication times between tasks and latency between processors.

## References

[1] 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