EXPERIENCES WITH DISTRIBUTED LONGEST PATH ALGORITHMS

by

Barton P. Miller
Cui-Qing, Yang

Computer Sciences Technical Report #681

January 1987

Experiences with Distributed Longest Path Algorithms

Barton P. Miller
Cui-Qing, Yang

Computer Sciences Department
University of Wisconsin-Madison
1210 W. Dayton Street
Madison, Wisconsin 53706

Abstract

As part of our research in performance tools for parallel and distributed programs we have developed longest path algorithms to calculate critical paths in program activity graphs. These longest path algorithms include both parallel distributed algorithms and centralized algorithms. Program activity graphs were derived from measurements of application programs and used to test these algorithms. This paper presents our experiences in developing, implementing and testing longest path algorithms for the critical path analysis. Both our centralized and distributed algorithms are based on the technique of diffusing computation. We have tested variations of these algorithms with graphs built from traces of different application programs. We compare the performance behavior of the distributed and centralized algorithms and discuss various aspects that affect the speed-up of distributed programs.

1. INTRODUCTION

In our current research of performance measurement tools for parallel and distributed programs[1], we are developing techniques for automatically guiding the programmer to performance problems in their application programs. One example of such techniques finds the critical path through a graph of a program's execution history. We create a precedence graph of a program's activities (computation and communication) with the data collected during the program's execution. The critical path, the longest path in the program activity graph, represents the sequence of the program activities that take the longest time to execute. The knowledge of this path helps to guide the programmer to performance problems.

Finding the longest path in an arbitrary directed graph is computationally expensive. However, if the given graph is acyclic, as is the case for program activity graphs, the algorithm becomes simpler and shortest path algorithms can be used to find the longest paths with only trivial changes. The shortest path algorithms are known to be linear to the number of vertices and edges in a graph[2].

We have developed several different algorithms for calculating the critical path through an activity graph in our measurement system. These algorithms include both distributed parallel algorithms and centralized serial algorithms. In this paper we present our experience in developing, implementing and testing longest path algorithms for the critical path analysis in program activity graphs. We compare the performance of distributed algorithms with the centralized version and discuss various performance aspects that affect the speed-up of distributed programs.

The use of the longest path algorithm is discussed elsewhere[1]; this paper focuses on the development of the parallel algorithms. We start with Section 2 to define some basic graph terms used in the paper and briefly describe our testing environment. Section 3 gives a brief description of the general structure of diffusing computations on graphs, on which we base our algorithms. Section 4 describes the implementation of our longest path algorithms and their corresponding performance measurement results. The final conclusions are in Section 5.

2. The Longest Path Problem

We present some definitions relevant to longest path algorithms used in this paper. We also give a description of the hardware and software environment in which we tested the longest path algorithms.

2.1. Definitions

A *directed graph (digraph)*, $G=(V,E)$, consists of a set of vertices $V=\{v_1,v_2,...\}$, a set of edges $E=\{e_1,e_2,...\}$, and a mapping Ψ that maps every edge onto some ordered pair of vertices (v_i,v_j). Edge (v_i,v_j) has an associated length w_{ij}. A *path*, P, is a finite sequence of edges $P = (v_1, v_2, \ldots, v_k)$, such that $(v_i,v_{i+1})\in E$ for all $i < k$. A path that starts and ends at the same vertex is called a *cycle*. The length $d(P)$ of a path P is defined to be $d(P) = w_{01} + w_{12} + \ldots + w_{k-1k}$. P is a *longest path* from v_i to v_j if $d(P)$ has the maximum length over all paths possible from v_i to v_j. The *one-to-all longest path problem* finds the longest path from a given vertex, called the *source*, to all the other nodes, the *destinations*.

2.2. Assumptions and Our Testing Environment

The graphs used in our longest path algorithms are program activity graphs in which vertices represent events in a program and edges represent the precedence relationships between events. The length of an edge is the time between the events of its two end vertices. Since all edges in a graph represent a forward progression of time, no cycles can exist in the graph. The acyclic property makes the difference between longest path problem and shortest path problem trivial. Therefore, in the following discussion, we consider that all shortest path algorithms are applicable to the longest path problem.

A program activity graph consists of several sub-graphs that are stored in different host machines. The data to build these sub-graphs is collected during the execution of the application programs. We can copy the sub-graphs between host machines. The time to do the copying can be traded against the amount of parallel computation. All sub-graphs were sent to one machine to test the centralized algorithm. In testing distributed algorithm, sub-graphs were either locally processed or sent to some collection of machines to re-group into bigger sub-graphs.

We used two application programs to generate program activity graphs in testing the longest path algorithms. Application 1 is master-slave structure in which a master process repeatedly sends work to slave processes and collects results from the slaves. Application 2 is pipeline structure in which each process in a chain of processes gets the job from its predecessor and sends the partial result to its successor. There are adjustable parameters in each application program. By varying these parameters we vary the size of the problem solved and generate program activity graphs with different number of vertices. In

graphs generated from Application 1, more than 50% of the total vertices was in one sub-graph and other half of the vertices were evenly distributed among the other sub-graphs. The vertices in the graphs from Application 2 were evenly distributed among each sub-graph.

All our tests were run on VAX-11/750 machines. The centralized algorithms ran under 4.3BSD UNIX, and the distributed algorithms ran on Charlotte distributed operating system[3,4]. Charlotte is a message-based distributed operating system designed for the Crystal multicomputer[5], which connects 20 homogeneous node computers (VAX-11/750s) and several host computers using an 80MB/sec Pronet token ring[6].

The Charlotte kernel supports basic interprocess communication mechanisms and process management services. The interprocess communication is through full-duplex connections, called links, between processes. The basic communication primitives are **Send, Receive**, and **Wait**. The **Send** and **Receive** are non-blocking operations which initiate a transfer of message between two processes on the specified link. A process can do **Wait** on either **Send** or **Receive** operations (or both) and the process will be blocked until the operation completes.

3. Diffusing Computation on Graphs

Diffusing computation on a graph, proposed by Dijkstra and Scholten[7], is a general method for solving many graph problems. All of our algorithms for the longest path problem are variations of this method. Before we start the discussion on details of our algorithms, we first give a brief description of the general structure of diffusing computation.

We define that a *root* vertex of a directed graph is a vertex in the graph that has only out-going edges, a *leaf* vertex of a directed graph is a vertex in the graph that has only in-coming edges. A diffusing computation on a graph can be described as:

> From all root vertices in the graph, a computation (e.g., a labeling message) diffuses to all of its descendant vertices and continues diffusing until it reaches all leaf vertices in the graph.

We distinguish two variations of the diffusing computation: *synchronous execution* and *asynchronous execution*. In the synchronous execution, a non-root, non-leaf vertex will diffuse the computation to its descendant vertices only after it gets all computations diffused from all in-coming edges. In the asynchronous execution, a non-root, non-leaf vertex will diffuse the computation to its descendant vertices as

soon as it gets an new computation from any one in-coming edge. The synchronous execution can deadlock in a graph with cycles. However the computational complexity of synchronous execution is linear to the number of edges and vertices.

4. Tests of Longest Path Algorithms

We have implemented several algorithms for finding the one-to-all longest paths from a designated source vertex to all other vertices. The centralized algorithm was based on the PDM shortest path algorithm[2] and tested as a standard for comparison with distributed algorithms. A distributed algorithm, based on Chandy and Misra's distributed shortest path algorithm[8], was developed on Charlotte distributed system. Both centralized and distributed algorithms were tested with graphs derived from program activity graphs from measurements of Application 1 and 2.

This section discusses the details of our implementation of various algorithms and summarizes the test results. We compare the centralized and distributed algorithms and give some remarks based on our experiences.

4.1. Test of Centralized Algorithm

Graphs in our tests are directed and acyclic. To find longest (shortest) path in such graphs is simpler than the general longest (shortest) path problem. We chose the PDM shortest path algorithm as the basis for our implementation[2]. The experiments of Denardo and Fox[9], Dial et al[10], Pape[11], and Vliet[12], show that on the average PDM algorithm is faster than other shortest-path algorithms if the input graph has a low edges to vertices ratio (in our graphs, the ratio is about 2).

We give a brief outline of the PDM algorithm in Figures 1 and 2. Figure 1 shows the asynchronous algorithm and Figure 2 shows the synchronous algorithm. A detailed discussion and the proof of the correctness of these algorithms can be found in [2].

The diffusing computation in this algorithm is to label the vertex with the current longest length. Q, in Figures 1 and 2, is a job queue for the diffusing computation. The original PDM algorithm is asynchronous. Every vertex in the graph will generate successor vertices in the queue as soon as an updated path to that vertex is discovered. Therefore, the number of all generated successor vertices in the queue from a single vertex can be proportional to the number of all possible paths from the source to the vertex. In the

contrast, the synchronous algorithm will generate successor vertices only when it finds the longest path from the source — i.e., when computations from all of its in-coming vertices have completed. We implemented this by setting a counter at each vertex that equals the number of all in-coming edges to that vertex. The counter is decremented every time the vertex gets a path information from its predecessor, and the successor vertices will be put in the queue only when the counter becomes zero. Since our graph is acyclic and only one central job queue exists in the program, this method guarantees the termination of the algorithm (when job queue is empty) and no deadlock will result.

```
for all u in G do
  D[u] := 0;
end
initialize Q to contain SOURCE only;
while Q is not empty do
  delete Q's head vertex u;
  for each edge (u,v) that starts at u do
    if D[v] < D[u] + w_uv then
      P[v] := u;
      D[v] := D[u] + w_uv;
      if v was never in Q then
        insert v at the tail of Q;
      else
        insert v at the head of Q;
      end
    end
  end
end
```

Figure 1: Asynch. Version of PDM Alg.

```
for all u in G do
  Counter[u] := # of in-coming edges;
  D[u] := 0;
end
initialize Q to contain SOURCE only;
while Q is not empty do
  delete Q's head vertex u;
  for each edge (u,v) that starts at u do
    dec(Counter[v]);
    if D[v] < D[u] + w_uv then
      P[v] := u;
      D[v] := D[u] + w_uv;
    end
    if Counter[v] = 0 then
      insert v at the tail of Q;
    end
  end
end
```

Figure 2: Synch. Version of PDM Alg.

We have tested our algorithms with graphs derived from the measurement of Application 1 and 2. The total number of vertices in the graphs varies from a few thousands to more than 10,000. About 33% of vertices have an outdegree of three, and 67% of vertices have an outdegree of one. The average edges to vertices ratio is around two. Figure 3 shows the test results of the execution time versus number of vertices in the graph for centralized algorithm. Since the asynchronous execution was too slow to test with larger graphs, we only show one result for the asynchronous execution. The execution time shown in the figures includes the copying time of nine sub-graphs to one host machine. However, our measurements indicate that this copying time is much less than one percent of the total execution time. The result in Figure 3 shows a significant performance difference between asynchronous execution and synchronous execution (around two orders of magnitude). The longer execution time for the data of Application 1 is because of

unevenly divided sub-graphs.

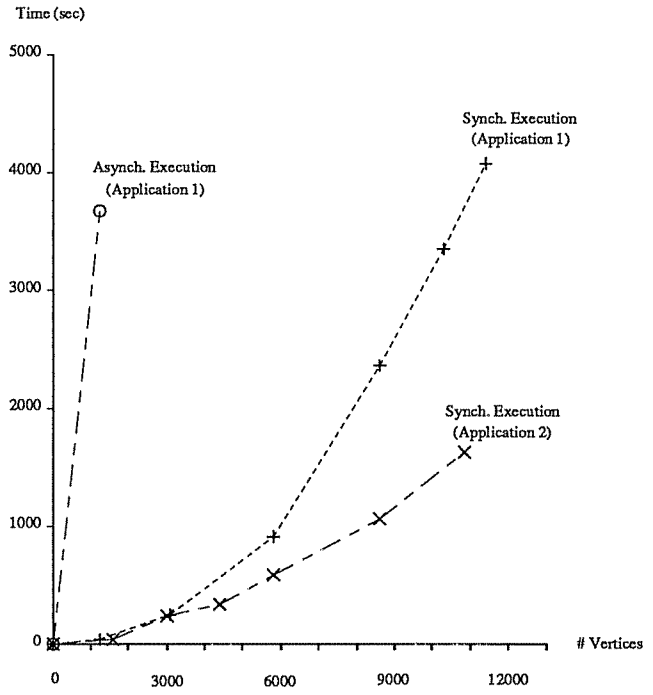

Figure 3: Execution Times for the Centralized Algorithm

4.2. Test of Distributed Algorithm

Our implementation of distributed longest path algorithm is based on Chandy and Misra's distributed shortest path algorithm[8]. Every process represents a vertex in the graph in their algorithm. However, we chose to represent a sub-graph instead of a single vertex in each process since the number of total processes in Charlotte system is limited by the size of physical memory and we were testing with graphs having thousands of vertices. Charlotte does not provide shared memory (as it is a network-based system), therefore we can not implement the algorithm using a shared job queue as in the modified PDM algorithm. We implemented the algorithm in such a way that there is a process for each sub-graph and each process has a job queue for the diffusing computation (labeling the current longest length of the vertex). Messages are send between processes for diffusing computations across sub-graphs (processes). Each process keeps individual message queues to its neighbor processes.

Unlike the original algorithm that has two phases and runs under asynchronous execution for detecting cycles in the graph, our algorithm only need a single phase to find the longest path from source to all vertices. In our asynchronous version, acknowledgements are used as in the Chandy and Misra's algorithm to determine the termination condition. However, in our synchronous version, a local termination condition is first checked in each process (by detecting the diffusion of computation to the leaf vertices), then the global termination condition is reached through the exchanges of messages among processes. Details of the two versions of the distributed algorithm appear in the Appendix.

We ran the distributed algorithm on the same test data as was used for the centralized algorithm. Figure 4 shows the test results. For the asynchronous execution, we only show one time result in the figure to compare with the time of synchronous execution. As same as in centralized algorithm, the execution times for asynchronous execution and synchronous execution differ dramatically. It is also interesting to note that the performance behavior of the distributed algorithm varies with different number of machines. When the input graph size is small, concurrency is low even for many machines. Communication overhead dominates the overall execution time of the algorithm. On the other hand, when graph size is big, there is more local processing to be done. That results in a higher computation to communication ratio and higher concurrency.

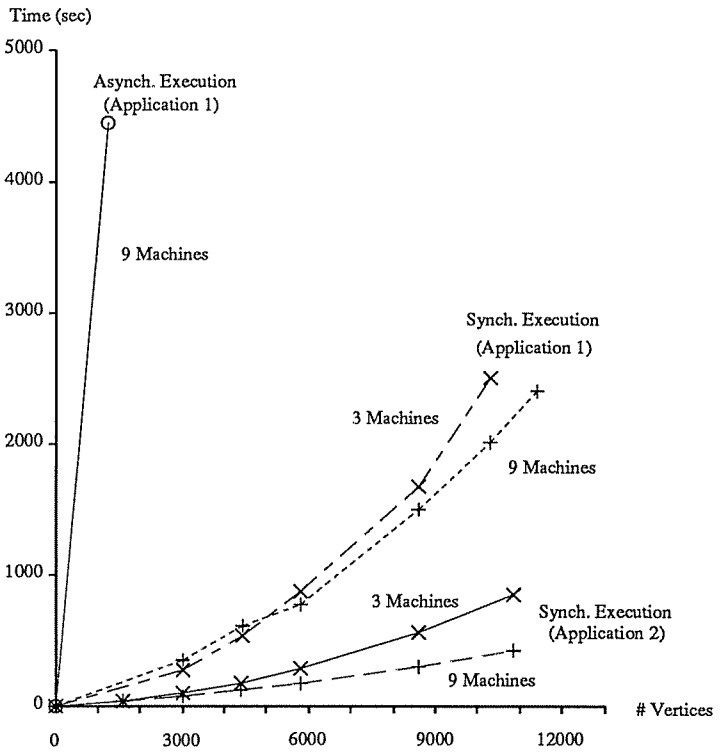

Figure 4: Execution Times for the Distributed Algorithm

Speed-up (S) and efficiency (E) are important measures for evaluating parallel programs. We use these measures to compare the performance of the distributed and centralized algorithms. The speed-up is defined as the ratio between the execution time of the centralized algorithm (T_c) to the execution time of the distributed algorithm (T_d): $S = T_c / T_d$. The efficiency is defined as the ratio of the speed-up to the number of machines used in the algorithm: $E = S/N$.

We used input graphs with different sizes and ran the centralized algorithm and distributed algorithm on up to 9 machines. The speed-up and efficiency are plotted against the number of machines. The results are shown in the Figures 6, 7, 8, and 9. We can see from these measurements that the distributed algorithm with bigger input graphs and more machines resulted a higher speed-up, but less efficiency. We will give detailed discussion on these test results in the next section.

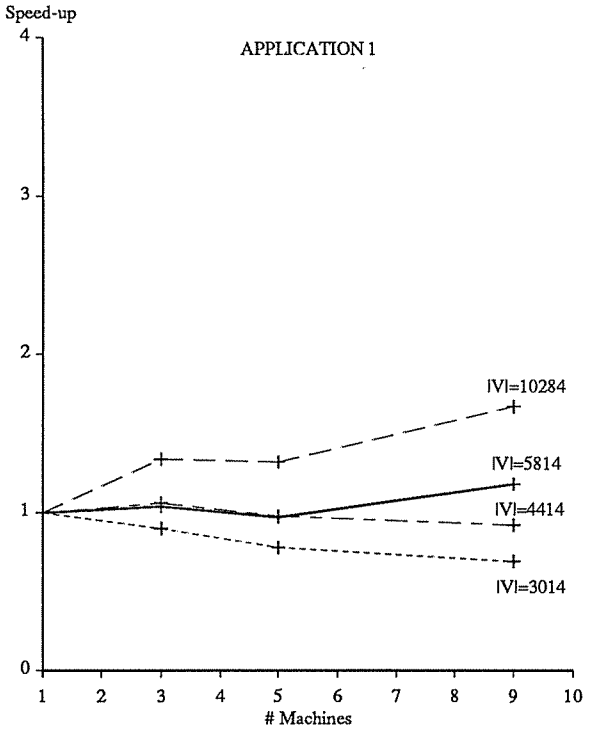

Figure 5: Speed-up of Distributed Algorithm

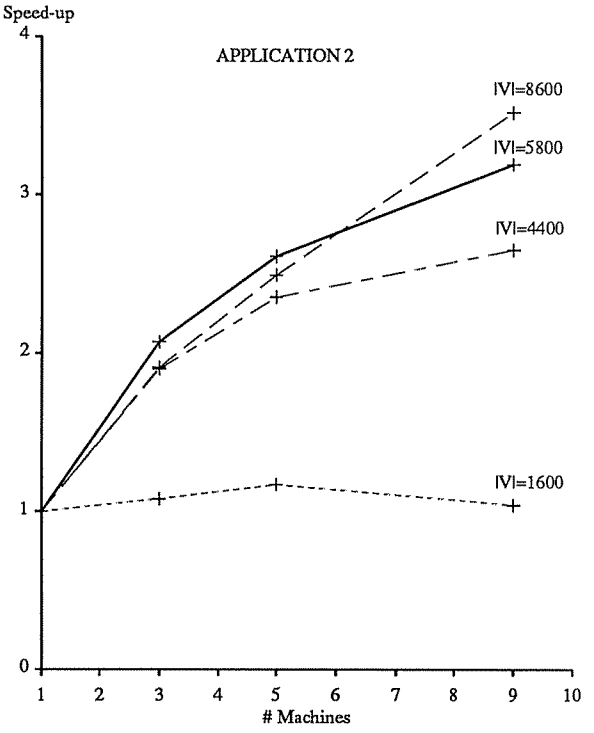

Figure 7: Speed-up of Distributed Algorithm

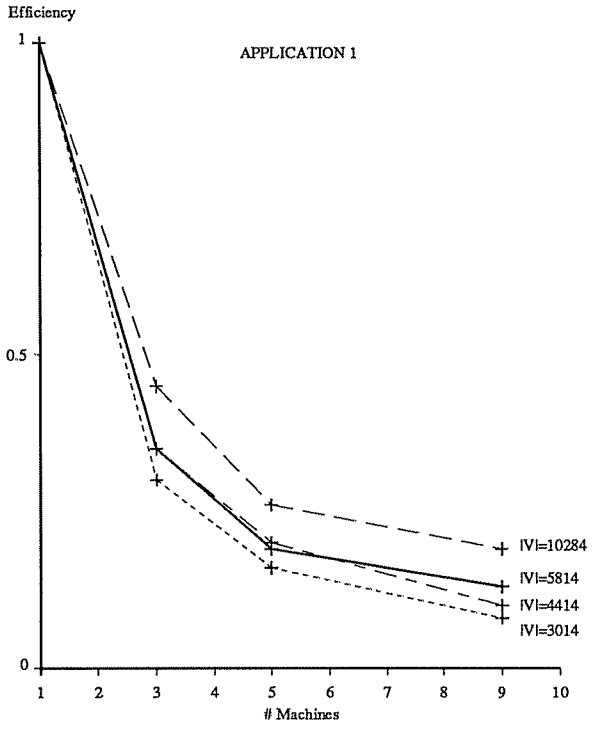

Figure 6: Efficiency of Distributed Algorithm

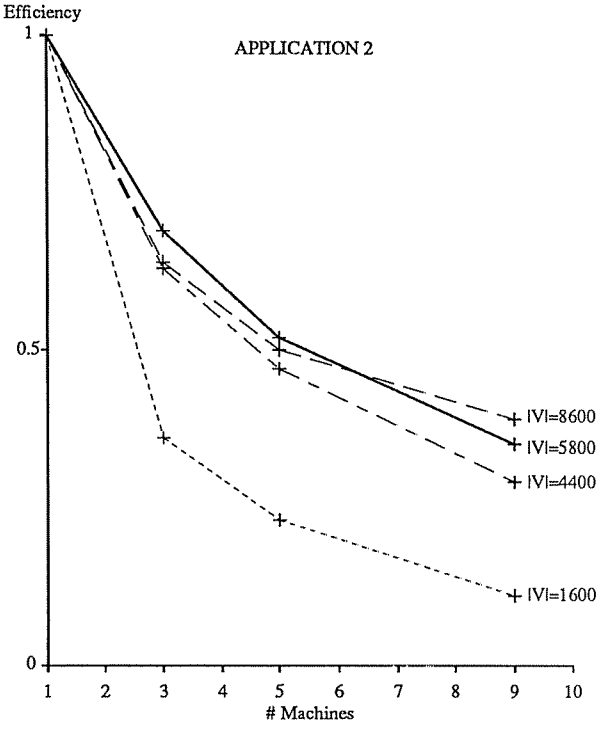

Figure 8: Efficiency of Distributed Algorithm

4.3. Discussions

We have implemented different algorithms for finding the longest path in program activity graphs and tested these algorithms with various input data. Both the centralized and distributed algorithms we tested are based on the method of diffusing computation. We first tested our centralized algorithm with asynchronous execution, as in the original PDM algorithm[2]. One major observation from the test was that the size of the job queue (Q) in the program grew rapidly with respect to the number of vertices in the graph. The length of the job queue was proportional to the number of all possible paths from the source vertex.

The complexity of the synchronous version of the longest path algorithms is linear to the number of vertices and edges in the graph, because the diffusing computations only go through each edge and vertex exactly once. In the asynchronous execution, the complexity is proportion to the total number of all possible paths from the source to each vertex in the graph, which is far worse than the linear case. The asynchronous execution can increase concurrency in a distributed computation by generating more diffused computations in the job queue and releasing the synchronization requirements among executions. But we sacrifice the economy of work because the asynchronous algorithm does less careful bookkeeping. The work in the asynchronous execution grows so fast that even a parallel algorithm is not viable.

We can only use asynchronous execution to find longest paths in an arbitrary graph with cycles (as in Chandy and Misra algorithm). But the asynchronous version of the distributed algorithm is not practical to implement. Our experiences suggest that it is more practical to use a more deterministic method for detecting cycles in graph as the first step, and then use synchronous execution in second step for finding longest paths.

We have observed a speed-up of almost 4 with 9 machines in the synchronous execution of the distributed algorithm. The speed-up increases with the size of the input graph and the number of machines participating in the algorithm. On the other hand, the efficiency of the algorithm decreases as more machines are involved in the algorithm. Because of the sequential nature of the synchronous execution of diffusing computations, the computations in an individual machine have to wait for synchronization at each step of the diffusion. As a result, the overall concurrency in the algorithm is restricted.

Our test results (Figures 5, 6, 7, 8) indicate that many factors affect the speed-up in a distributed algorithm. The structure of the input graph, the number of vertices in the graph, and the number of machines in the execution all influence the overall speed-up of the algorithm. Our experiences demonstrate that an important consideration in design of distributed algorithm is to keep balance between the CPU load and message costs when decomposing the original problem into concurrently executing tasks (processes). In the input data of Application 1, one of the task was overloaded by more than 50% of the total graph size. Therefore, the overall speed-up was worse than with input data of Application 2. Similarly, when the size of input graph was small and the number of machines in the execution was big, the unit of local work was small relative to the cost of message communication. The speed-up was dominated by the message communication overhead.

5. CONCLUSION

We developed centralized and distributed algorithms, based on diffusing computation, for finding the longest paths in program activity graphs. We observed that there is a dramatic performance difference between asynchronous execution and synchronous execution of algorithms. The combinatorial explosion of CPU and message loads in the asynchronous algorithm easily overwhelm the possible increase of the concurrency in the algorithm. The complexity of the synchronous algorithm is linear to the number of vertices and edges in the graph. However, the inherent sequential property of the diffusing computations restricts the concurrency of the distributed algorithm.

Our test results demonstrated that parallel, distributed computation can be used to speed up determination of longest paths in large graphs. We have observed a speed-up of almost 4 with 9 machines. The speed-up of the distributed algorithm depends on the decomposition of the original problem. An important consideration in design of a distributed algorithm is to keep a balance between the CPU load and message costs when decomposing the problem into concurrently executing tasks (processes). To get good speed-up in a distributed algorithm, we should decompose the problem in the way such that to keep tasks busy but do not overload or underload any individual tasks. The ratio between the work load in each task and the costs of message communication affects the overall speed-up of a distributed algorithm.

6. ACKNOWLEDGEMENT

The authors gratefully acknowledge the advice and useful ideas offered by Udi Manber.

7. REFERENCES

[1] Barton P. Miller and Cui-qing Yang, "IPS: An Interactive and Automatic Performance Measurement Tool for Parallel and Distributed Programs"," Tech. Report 613 Computer Sciences Dept. , Univ. of Wisconsin-Madison (Sept. 1985).

[2] Narsingh Deo, C. Y. Pang, and R. E. Lord, "Two Parallel Algorithms for Shortest Path Problems," *Proc. of the 1980 International Conference on Parallel Processing*, pp. 244-253 (Aug. 1980).

[3] Raphael Finkel, Marvin Solomon, David DeWitt, and Lawrence Landweber, "The Charlotte Distributed Operating System -- Part IV of the First Report on the Crystal Project," Tech. Report 510 Computer Sciences Dept. , Univ. of Wisconsin-Madison (Sept. 1983).

[4] Yeshayahu Artsy, Hung-Yang Chang, and Raphael Finkel, "Interprocess Communication in Charlotte," *IEEE Software* **4**(1) pp. 22-28 (January 1987).

[5] D. DeWitt, R. Finkel, and M. Solomon, "The Crystal multicomputer: design and implementation experience," *To appear in IEEE Trans. on Software Engineering* , (1986).

[6] Proteon Associates, *Operation and Maintenance Manual for the ProNet Model p1000 Unibus.* 1982.

[7] E. W. Dijkstra and C. S. Scholten, "Termination detection for diffusing computations," *Inf. Process. Lett* **11**(1) pp. 1-4 (Aug. 1980).

[8] K. M. Chandy and J. Misra, "Distributed Computation on Graphs: Shortest Path Algorithms," *Communications of the ACM* **25**(11) pp. 833-837 (November 1982).

[9] E.V. Denardo and B.L. Fox, "Shortest-route methods: 1. reaching, pruning, and buckets," *Operations Research* **27**(1) pp. 161-186 (Jan.- Feb. 1979).

[10] R.B. Dial, F. Glover, D Karney, and D. Klingman, "A computational analysis of alternative algorithms and labeling techniques for finding shortest path trees," *Networks* **9**(3) pp. 215-248 (Fall 1979).

[11] U. Pape, "Implementation and efficiency of Moore-algorithms for the shortest route problems -- a review," *Math. Programming* **7**(2) pp. 212-222 (Oct. 1974).

[12] D. Van Vliet, "Improved shortest path algorithm for transporation networks," *Trasportation Research* **12**(1) pp. 7-20 (Feb. 1978).

APPENDIX

Following is a brief sketch of the two variations of our distributed longest path algorithm. Discussions of the distributed algorithm can be found in Section 4.2.

```
for all u in sub-graph G do
  Counter[u] := 0;
  D[u] := 0;
end
initialize Q;
while not termination do
  while Q is not empty and
        msg queues not full do
    delete Q 's head element;
    if element = (Length[u], Pred) then
      if D[u] < Length[u] then
        if Counter[u] > 0 then
          put Ack[P[u]] in Q or msg queue;
        end
        P[u] := Pred;
        D[u] := Length[u];
        for each edge (u,v) that starts at u do
          put length msg: (D[u]+w_uv,Pred=u)
          in Q or msg queue;
        end
        Counter[u] += # of out-going edges;
        if Counter[u] = 0 then
          put Ack[P[u]] in Q or msg queue;
        end
      else
        put Ack[Pred] in Q or msg queue;
      end
    else
      dec(Counter[u]);
      if Counter[u] = 0 then
        put Ack[P[u]] in Q or msg queue;
      end
    end
  end
  Send all msg queues that are not empty;
  Receive from all neighbor processes;
  if get any msg from other processes then
    processing msg and put length msg
    and acks in Q;
  end
end
```

I. Asynch. Version of Distributed Algorithm

```
for all u in sub-graph G do
  Counter[u] := # of in-coming edges;
  D[u] := 0;
end
initialize Q;
while not local_termination do
  while Q is not empty and
        msg queues not full do
    delete Q 's head: (Length[u],Pred);
    dec(Counter[u]);
    if D[u] < Length[u] then
      P[u] := Pred;
      D[u] := Length[u];
    end
    if Counter[u] = 0 then
      for each edge (u,v) that starts at u do
        put length msg: (D[u]+w_uv,Pred=u)
        in Q or msg queue;
      end
    end
  end
  Send all msg queues that are not empty;
  Receive from all neighbor processes;
  if get any msg from other processes then
    processing msg and put length msg in Q;
  end
end
exchange msg with neighbors to get consensus
of global_termination
```

II. Synch. Version of Distributed Algorithm