Critical Path Analysis for the Execution of Parallel and Distributed Programs

Cui-Qing Yang

Department of Computer Sciences
University of North Texas
P.O. Box 13886
Denton, Texas 76203

Barton P. Miller†

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

ABSTRACT

In our research of performance measurement tools for parallel and distributed programs, we have developed 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.

This paper presents the design, implementation and testing of the critical path analysis technique on the IPS performance measurement tool for parallel and distributed programs. We create a precedence graph of a program's activities (Program Activity Graph) with the data collected during the execution of a program. 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. Various algorithms are developed to track the critical path from this graph. The events in this path are associated with the entities in the source program and the statistical results are displayed based on the hierarchical structure of the IPS. The test results from the measurement of sample programs show that the knowledge of the critical path in a program's execution helps users identify performance problems and better understand the behavior of a program.

1. Introduction

The execution of a parallel or distributed program can be quite complex. Often individual performance metrics do not reveal the cause of poor performance, because a sequence of activities, spanning several machines or processes, may be responsible for slow execution. Consider an example from traditional procedure profiling. We might discover a procedure in our program that is responsible for 90% of the execution time. We could hide this problem by splitting the procedure into 10 subprocedures, each responsible for 9% of execution time. For this reason, it is necessary to detect a situation in which cost is dispersed among several procedures, and across process and machine boundaries.

There are other problems that are difficult to detect using simple performance metrics. It may be important to determine the effect of contention for resources. For example, the scheduling or planning of activities on different machines can have a great effect on the performance of the entire program[12].

Our strategy in designing a measurement system is to integrate automatic guidance techniques into such a system. Therefore, information from these techniques, such as that which concerns critical resource utilization, interaction and scheduling effects, and program time-phase behavior should be available to help users analyze a program's execution. In our research, we have developed one of the techniques — critical path analysis for the execution of distributed programs. This paper presents the design, implementation, and testing of this technique in IPS measurement tool[16]. Section 2 addresses the basic concepts for critical path analysis of the execution of distributed programs. Section 3 provides a definition of the critical path in the program activity graph and describes the construction of these graphs based on data collected from measurement. Different algorithms for calculating the critical path and the testing of these algorithms are presented in Section 4. Finally, Section 5 shows the application of critical path information to the performance analysis of the execution of distributed programs, and Section 6 presents conclusions.

2. Critical Path Analysis

Turnaround or completion time is an important performance measure for parallel programs. When turnaround time is used as the measure, speed is the major concern. One way to determine the cause of a program's turnaround time is to find the event path in the execution history of the program that has the longest duration. This ***critical path*** [13] identifies where in the program we should focus our attention. As an example, Figure 1 gives the execution history of a distributed program with three processes. This figure displays the program history at the process level, and the critical path (identified by the bold line) readily shows us the parts of the program with the greatest effect on performance.

We can view a distributed program as having the following characteristics:

(a) It can be broken down into a number of separate activities.

(b) The time required for each activity can be measured.

(c) Some activities must be executed serially, while others may be carried out in parallel.

(d) Each activity requires a combination of resources, e.g., CPU's, memory spaces, and I/O devices. There may be more than one feasible combination of resources for different activities, and each combination is likely to result in a different duration of execution.

Based on these properties of a distributed program, we can use the critical path method (CPM)[9, 13] to analyze a program's execution. The CPM method is commonly used in

† Research supported in part by the National Science Foundation grant CCR-8703373and an Office of Naval Research Contract.

CH2541-1/88/0000/0366$01.00 © 1988 IEEE

operational research for planning and scheduling, and has also been used to evaluate concurrency in distributed simulations[2].

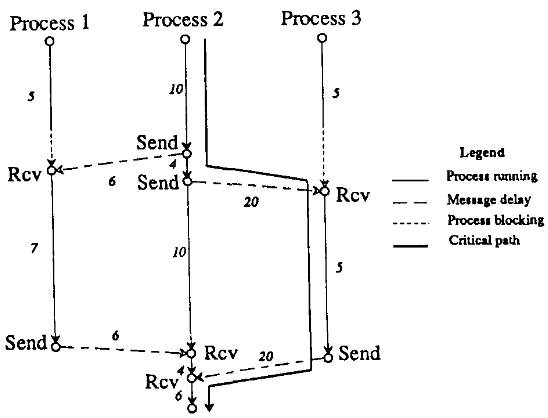


Figure 1: Example of Critical Path

In contrast to CPM, the technique used in our critical path analysis (CPA) is based on the execution history of a program. We can find the path in the program's execution *history* that took the longest time to execute. Along this path, we can identify the place(s) where the execution of the program took the longest time. The knowledge of this path and of the bottleneck(s) along it will help us focus on the performance problem.

Turnaround time is not the only critical measure of the performance of parallel programs. Often the throughput is more important, e.g., in high-speed transaction systems[14]. While, our discussion concentrates on the issue of critical path analysis for turnaround time, the techniques we use to present CPA result (see Section 5.1) are directly applicable to throughput.

3. Program Activity Graph

To calculate critical paths for the execution of distributed programs, we first need to build graphs that represent program activities during the program's execution. We call these graphs *program activity graphs* (PAGs). The longest path in a program activity graph represents the critical path in the program's execution. In this section, we define the program activity graph and related ideas. We then describe how various communication primitives of distributed programs are represented in program activity graphs and how these graphs are built on the basis of information obtained from program measurement.

3.1. Definitions

The definition of program activity graph is similar to that of an activity network in project planning[6]. The execution of distributed programs can be divided into many nonoverlapping individual jobs, called *activities*. Each activity requires some amount of time, called the *duration*. A precedence relationship exists among the activities, such that some activities must be finished before others can start. Therefore, a PAG is defined as a weighted, and directed multigraph, that represents program activities and their precedence relationship during a program's execution. Each edge represents an activity, and its weight represents the duration of the activity. The vertices represent beginnings and endings of activities and are the *events* in the program (e.g., send/receive and process creation/termination events). A *dummy activity* in a PAG is an edge with zero weight that represents only a precedence relationship and not any real work in the program. More than one edge can exist between the same two vertices in a PAG.

The critical path for the execution of a distributed program is defined as the longest weighted path in the program activity graph. The length of a critical path is the sum of the weights of edges on that path.

3.2. Construction of Program Activity Graphs

A program activity graph is constructed from the data collected during a program's execution. There are two requirements for the construction of program activity graphs: first, the activities and events represented in a PAG should be measurable in the execution of programs; second, the activities in a PAG should obey the same precedence relationship as do program activities during the execution.

Two classes of activity considered in our model of distributed computation are computation and communication. For computation activity, the two basic PAG events are starting and terminating events. The communication events are based on the semantics of our target system, the Charlotte distributed operating system[1]. We can build similar graphs for systems with different communication semantics. In Charlotte, the basic communication primitives are message Send, Rcv and Wait system calls. A Send or Rcv call issues a user communication request to the system and returns to the user process without blocking. A Wait call blocks the user process and waits for the completion of previously issued Send or Rcv activity. Corresponding to these system calls are four communication events defined in the PAG: send_call, rcv_call, wait_send, and wait_rcv. These primitive events allow us to model communication activities in a program. We show, in Figure 2, a simple PAG for message send and receive activities in a program. Two extra events of transmit_start and transmit_end are included in Figure 2a to depict the underlying relationship among various communication events. They represent the actual data transmission inside the operating system. Since these extra events occur below the application program level, we do not consider them in our PAG. Therefore, we transform the graph in Figure 2a into that in Figure 2b and still preserve the precedence relationship among the basic communication events.

The weights of message communication edges in Figure 2b (t_{send}, t_{rcv}, t_{wait_send}, and t_{wait_rcv}), represent the message delivery time for different activities. Message delivery time is different for local and remote messages, and is also affected by message length. A general formula for calculating message delivery times is: $t=T_1+T_2\times L$, where L is the message length, and T_1 and T_2 are parameters of the operating system and the network. We have conducted a series of tests to measure values of these parameters for Charlotte. We calculated average T_1 and T_2 for different message activities (intra- and inter-machine sends and receives) by measuring the round trip times of intra- and inter-machine messages for 10000 messages, with message lengths from 0 to the Charlotte maximum packet size. These parameters are used to calculate the weight of edges when we construct PAGs for application programs.

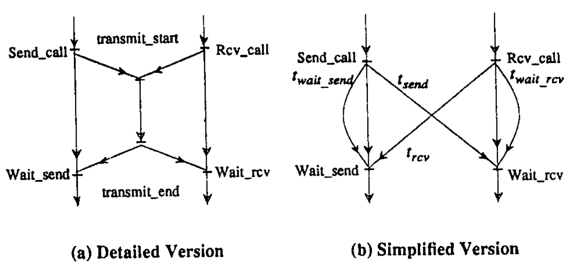


Figure 2: Construction of Simple Program Activity Graph

4. Algorithms of Critical Path Analysis

An important side issue is how to compute the critical path information efficiently. After a PAG is created, the critical path is the longest path in the graph. Algorithms for finding such paths are well studied in graph theory. We have implemented a distributed algorithm for finding the one-to-all longest paths from a designated source vertex to all other vertices. A centralized algorithm was also tested as a standard for comparison with distributed algorithms. In this section, we describe some details of the implementation and testing of these algorithms, and provide comparisons between them.

4.1. Assumptions and Our Testing Environment

Since all edges in a PAG represent a forward progression of time, no cycles can exist in the graph. To find the longest path in such graphs is a much simpler problem than in graphs with cycles. Most shortest path algorithms can be easily modified to find longest paths of the acyclic graphs. Therefore, in the following discussion, we consider those shortest-path algorithms to be applicable to our longest-path problem.

A program activity graph consists of several subgraphs that are stored in different host machines. The data to build these subgraphs are collected during the execution of application programs. We can copy subgraphs between node machines; the copying time is included in the execution time of algorithms. All subgraphs were sent to one machine to test the centralized algorithm. In testing the distributed algorithm, subgraphs were either locally processed or sent to some collection of machines to be regrouped into bigger subgraphs.

We used two application programs to generate PAGs for testing the longest path algorithms. Application 1 is a group of processes in a master-slave structure, and Application 2 is a pipeline structure. Both programs have adjustable parameters. By varying these parameters, we vary the size of the problem and the size of generated PAG's. In graphs generated from Application 1, more than 50% of the total vertices were in one subgraph, while the remaining ones were evenly distributed among the other subgraphs. The vertices in the graphs from Application 2 were evenly distributed among all subgraphs.

All of our tests were run on VAX-11/750 machines. The centralized algorithms ran under 4.3BSD UNIX, and the distributed algorithms ran on the Charlotte distributed operating system[1].

4.2. Test of Different Algorithms

We chose the PDM shortest-path algorithm as the basis for our implementation of centralized algorithm[7]. The experiments of Denardo and Fox[5], Dial et al[8], Pape[18], and Vliet[19] show that, on the average, the 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). An outline of the PDM algorithm and a brief proof for the correctness of the algorithm are given in [20]. More detailed discussion of the algorithm can be found in [7].

Our implementation of the distributed longest path algorithm is based on Chandy and Misra's distributed shortest path algorithm[3]. 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 because the number of total processes in the Charlotte system is limited and we were testing with graphs having thousands of vertices. The algorithm is implemented in such a way that there is a process for each sub-graph, and each process has a job queue for work at vertices in the sub-graph (labeling the current longest length to a vertex). Messages are sent between processes for passing information across sub-graphs (processes). Each process keeps individual message queues to its neighbor processes. An outline of the two versions of the distributed algorithm and a proof of the correctness of the algorithm appear in [20]. A detailed discussion of the algorithm is given by Chandy and Misra[3].

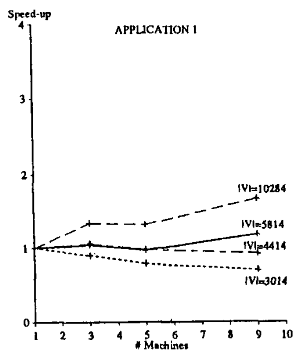
We tested our algorithms with graphs derived from the measurement of the execution of Applications 1 and 2. The total number of vertices in the graphs varies from a few thousand to more than 10,000. Speed-up (S) and efficiency (E) are used to compare the performance of the distributed and centralized algorithms. 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): $S = T_c \ / \ T_d$. 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 and distributed algorithms on up to 9 machines. Speed-up and efficiency were plotted against the number of machines. The results are shown in Figures 3, 4, 5, and 6. We can see from these measurements that the distributed algorithm with larger input graphs and more machines resulted in greater speed-up but less efficiency.

We have observed a speed-up of almost 4 with 9 machines for the distributed algorithm. 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. The sequential nature of synchronous execution of diffusing computations determines that the computations in an individual machine have to wait for synchronization at each step of the algorithm. As a result, the overall concurrency in the algorithm is restricted, and the communication overhead with more machines offsets the gain of the speed-up.

5. Measurement Tests with the Technique of Critical Path Analysis

We have conducted a set of tests in using the technique of critical path analysis for the performance measurement of distributed application programs. The goal of these tests is to show how automatic guidance techniques can be integrated in a performance measurement tool, and how these guidance information can help us to better understand the performance



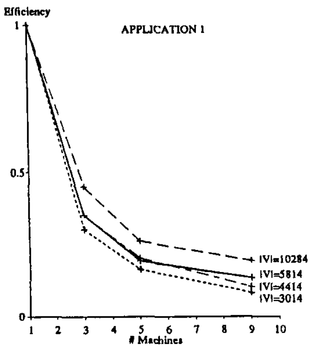
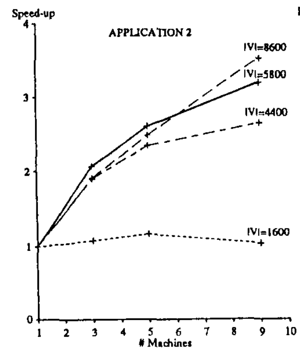


Fig. 3: Speed-up of Distributed Algorithm Fig. 4: Efficiency of Distributed Algorithm



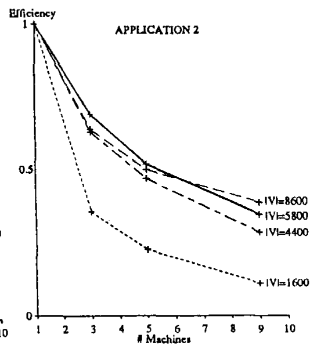


Fig. 5: Speed-up of Distributed Algorithm Fig. 6: Efficiency of Distributed Algorithm

behavior of a program. In this section, we first give a brief description of the system and the program with which we have conducted our tests. Then, we present some of the measurement results in relating to the information from the critical path analysis. The final discussion addresses the problem of the minimum length of the critical path in a program's execution.

5.1. Test System and Test Program

All of our tests with the technique of critical path analysis were run on the IPS performance measurement system for parallel and distributed programs[16, 20]. IPS uses a hierarchical model as the framework for performance measurement. The hierarchical model maps program's behavior to different levels of abstraction, and unifies performance data from the whole program level down to procedure and statement level. IPS provides a wide range of performance information about a program's execution. This information includes performance metrics at different levels of detail, histograms of the basic metrics, and guidance information from the critical path analysis. An interactive user interface with simple command menu is supported in the IPS. Programmers can interactively evaluate the performance history of a distributed program after the program terminates normally.

The presentation of results of critical path analysis offers some interesting problems. A PAG may contain more than 100,000 nodes; the critical path may contain a nontrivial percentage of these nodes. We use statistical presentation techniques to display the (time weighted) most commonly occurring nodes, and the most commonly occurring sequences in the path. We then use high-level language debugging techniques to relate these events directly to the source program. Observing the most commonly occurring sequences allows us to detect performance bottlenecks that span procedure, process, or machine boundaries. Performance problems that are divided among several procedures, or even among processes or machines, are readily apparent.

The program we have chosen for measurement tests on the IPS system is an implementation of the *Simplex method* for linear programming[4, 11]. The so-called ***columnwise distribution algorithm*** works in a master-slave model. With the given problem defined as a matrix, a controller (master) process coordinates the computation in multiple calculator (slave) processes to obtain an optimal solution. All slave processes use the same program code but work on different columnwise data of the matrix. The configuration for our test is set as follows: the input matrix size is 36×36, the program has a controller process and 8 calculator processes, and these processes run on 3 node machines.

5.2. Some Test Results

We start our measurement session by examining the information from various metrics and histograms in IPS. We can learn about many aspects of the program's behavior from this information, and have a general picture of the program's execution. For instance, the parallelism of the program is not high (around 1.15); there is considerable communication between the controller and calculator processes (about 700 messages); and each calculator process has light work load and spends most of the time in waiting for messages (11% of the time in computing, 85% of the time in waiting for messages, and 4% of the time in waiting for CPU). However, all this information is mainly applicable to individual items in the program. It tells us little about the interactions among different parts of a program, and about how these interactions affect the overall behavior of a program's execution. Therefore, it is still difficult to discover why the parallelism is low, how much communication costs affect the program's execution, and which process (controller or calculator) has a bigger impact on the program's behavior.

The critical path analysis technique in IPS provides guidance for finding possible bottlenecks in a program's execution. The critical path information is represented by the percentages of communication and CPU time of the various parts of the program along the total length of the path. Figure 7 gives the critical path information at the program level. We can see that the communication cost (including inter-machine and intra-machine messages) is more than one third of the total length of the critical path. This reflects the fact that the communication overhead in Charlotte is relatively high compared to other systems[1].

Entry Name	Time(ms)	%
CPU	10347	62
Inter-machine Msg	4960	30
Intra-machine Msg	1360	8
Total	16667	100

Figure 7: Critical Path Information at Program Level

Entry Name	Time(ms)	%
P(1,3) CPU	9740	58
P(1,3)->P(3,5) Msg	840	5
P(3,5)->P(1,3) Msg	840	5
P(1,3)->P(2,5) Msg	480	3
P(2,5)->P(1,3) Msg	480	3
P(3,4)->P(1,3) Msg	480	3
P(1,3)->P(3,4) Msg	480	3
P(1,3)->P(2,4) Msg	440	3
P(2,4)->P(1,3) Msg	440	3
P(1,3)->P(1,5) Msg	408	2
P(1,5)->P(1,3) Msg	408	2
P(1,3)->P(1,4) Msg	272	2
P(1,4)->P(1,3) Msg	272	2
P(1,3)->P(3,3) Msg	240	1
P(3,3)->P(1,3) Msg	240	1
P(3,5) CPU	159	1
P(1,5) CPU	108	1
P(2,5) CPU	88	1
P(3,4) CPU	79	*
P(2,4) CPU	67	*
P(1,4) CPU	64	*
P(3,3) CPU	42	*
Total	16667	100

(P(i,j) denotes process j in machine i, * denotes less than 1%)

Figure 8: Critical Path Information at Process Level

The critical path information at the process level (see Figure 8) gives us more details about the program's execution. The execution of the controller process takes 58% of the whole length of the critical path, while the execution of all calculator processes take less that 5% of the whole length. The domination of the controller process in the critical path restricts the overall concurrency of the program. This explains why the parallelism for the current configuration is so low. From the length of the critical path, we can calculate the maximum parallelism of the program[10, 15], which equals the ratio between the total CPU time and the length of the critical path. This maximum parallelism depends upon the structure and the interactions among the different parts of the program. The maximum parallelism for the program under our tests (with 8 calculators running on 3 machines) is only 1.91. The communication costs and the CPU load effects in different machines lowered the real parallelism to 1.15.

Procedure Name	Mach. Process ID	Time(%)
MainLoop	(Mach 1, Proc 3)	21
SendChild	(Mach 1, Proc 3)	17
Init	(Mach 1, Proc 3)	11
readl	(Machl, Proc 3)	4
CheckWaiting	(Mach 1, Proc 3)	4
MainLoop	(Mach 3, Proc 4)	1
recv	(Mach 1, Proc 3)	1
MainLoop	(Mach 3, Proc 5)	1
MainLoop	(Mach 1, Proc 5)	1
MainLoop	(Mach 2, Proc 4)	1
MainLoop	(Mach 1, Proc 4)	*
MainLoop	(Mach 2, Proc 5)	*
Total CPU		62

(* denotes less than 1%)

Figure 9: Critical Path Information at Procedure Level

Finally, we display critical path information at the procedure level in Figure 9. This information is useful in locating performance problems across machine and process boundaries. The top three procedures, that take 49% of the entire length of the critical path, are in the controller process. Procedure *MainLoop* in the calculator processes, which is in charge of communications between calculators and the controller, takes 33% of the entire execution time of each calculator process. However, they are much less noticeable in the critical path because of the dominance of the controller process.

5.3. Discussion

We have seen that the execution of the controller process dominates the performance behavior. This is because, in our test configuration, the controller process serves too many (8) calculator processes, but each calculator process is lightly loaded. One way to cope with the problem is to reduce the number of calculator processes in the program. We have conducted a set of measurement tests with our test program having 2 to 8 calculator processes for the same 36×36 input matrix, running on 3 machines. The test results are shown in Figure 10.

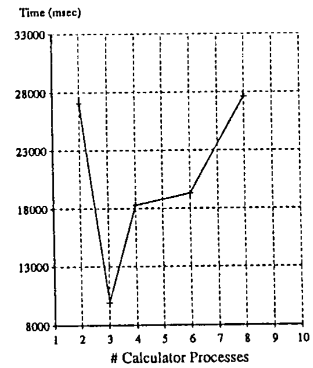
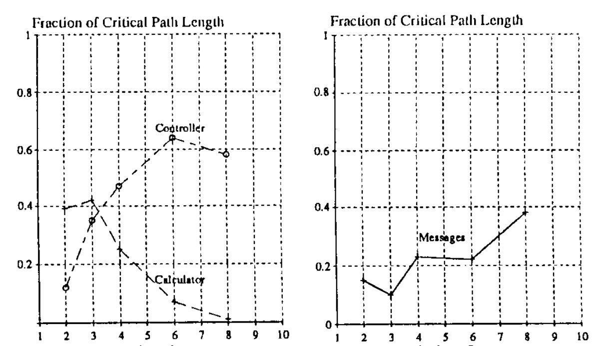


Figure 10: Program Elapsed Time in Different Configurations

We can observe that, to a certain extent, for this fixed initial problem, having fewer calculator processes gives a better result. The execution time (also the length of the critical path) has its minimum when the program runs with 3 calculators. However, if the number of calculator processes gets too small (2 in this case), each calculator has to do too much work and creates a bottleneck. Note that the test using 2 calculator processes is best with respect to the assignment of processes to machines (only one process per machine). While in the 3 calculator case the controller process is running on the same machine as a calculator process. Therefore, the contention for CPU time among processes is not the major factor that affects the overall execution time of the program.



(a) Process Components (b) Message Components

Figure 11: Components of the Critical Path

The critical path information for these tests (shown in

Figure 11) supports our observation. For the configuration of 3 calculator processes, the controller and calculator processes have the best balanced processing loads, and the lowest message overhead. This coincides with the shortest execution time in Figure 10. The Simplex program has a master-slave structure. The ratio between computation times for the controller and calculator processes ***on the critical path*** reflects the balancing of the processing loads between the master and slaves in the program. We have observed that when the master and slave processes have evenly distributed processing loads (dynamically, not statically), the program shows the best turnaround time. Otherwise, if the master process dominates the processing, the performance suffers due to the serial execution of the master process. On the other hand, if the slave processes dominated, it would be possible to add more slaves. The Appendix contains a proof that supports our claim that for programs with the master-slave structure, the length of the critical path in the program's execution is at its minimum when the path length is evenly distributed between master and slave processes.

The last observation from our tests is that the critical path information in our discussion ignores the delays caused by competition for the external resources as CPU's, so that it depends entirely on the structure of the program. Actually, we can calculate the critical path, using the real elapsed time with delays caused by processes waiting for the CPU on a machine. Therefore, the results of CPA will also reflect the interactions and scheduling of the concurrent events in a program.

6. Conclusion

The technique of critical path analysis (CPA) is one method that we developed to provide guidance for locating performance problems in the program. A PAG is created from the data collected during program's execution. The longest path in this graph represents the critical path in the execution of the program. We have implemented different algorithms to calculate the critical path in a PAG. Due to the acyclic nature of PAG's, these algorithms are simple and efficient.

The experimental measurements conducted on the IPS system with sample application programs show that the knowledge about the critical path in a program's execution helps programmers identify the possible bottlenecks in the program. In addition, this information also allows users to predict the program behavior under different configurations. It is possible to accommodate various guidance techniques in a performance measurement tool. Developing these guiding techniques for performance measurement of parallel and distributed programs exposes a new research area which requires a combined knowledge of disciplines such as performance measurement, program semantics, and algorithm design.

7. Appendix

In the following discussion, we give a simple proof to support our claim that for programs with the master-slave structure, the length of the critical path in the program's execution reaches the minimum when the whole path length is evenly distributed between master and slave processes. Our proof applies the related study in Mohan's thesis[17] to the aspect of critical path length.

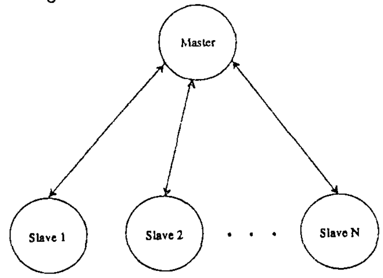


Figure 12: Master-Slave Structure

Assume that a general master-slave structure is represented as N slave processes working synchronously under the control of a master process (see Figure 12). Let a computation have a total computing time of C, consisting of the time for master, C_m, and the time for slaves, C_s (for simplicity, all times are deterministic). The computation time in the master process includes one part for a fixed processing time (e.g., initialization, result reporting time), F_m, and another part of per slave service time (e.g., job allocating, partial results collecting, and communication times with slaves in the program of the Simplex method), c_m. Therefore,

$$C_m = Nc_m + F_m.$$

Assume F_m is negligible compared to Nc_m, i.e., $F_m >> Nc_m$; we have:

$$C_m = Nc_m.$$

The nature of the synchronization pattern in the master-slave structure determines that the execution of the master process is serialized with the concurrent execution of N slave processes. Hence, the length of the critical path in the program's execution, $L_c(N)$, is:

$$L_c(N) = C_m + \frac{C_s}{N} = Nc_m + \frac{C_s}{N}.$$

To find the minimum of $L_c(N)$, we have:

$$\frac{d(L_c(N))}{dN} = c_m - \frac{C_s}{N^2} = 0,$$

and

$$N = \sqrt{\frac{C_s}{c_m}}.$$

Since $\frac{d^2(L_c(N))}{dN^2} > 0$ when $N = \sqrt{\frac{C_s}{c_m}}$, $L_c(N)$ has its minimum value at the point. Therefore, the minimum length of the critical path is:

$$min(L_c(N)) = Nc_m + \frac{C_s}{N} = \sqrt{c_m C_s} + \sqrt{c_m C_s}.$$

In this equation, both master and slaves have the same amount of share $(\sqrt{c_m C_s})$ in the length of the critical path. This result indicates that the length of the critical path reaches to the minimum when the entire length is evenly distributed in master and slave processes.

8. References

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

[2] O. Berry and D. Jefferson, "Critical Path Analysis of Distributed Simulation," *Proc. of Conf. on Distributed Simulation 1985*, (January 1985).

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

[4] G. B. Dantzig, *Linear Programming and Extensions,* Priceton University Press, Princeton, NJ (1963).

[5] 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).

[6] Narsingh Deo, *Graph Theory with Applications to Engineering and Computer Science,* Prentice-Hall, Inc., Englewood Cliffs, N. J. (1974).

[7] 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 (August 1980).

[8] 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).

[9] W. E. Duckworth, A. E. Gear, and A. G. Lockett, "A Guide to Operational Research," *John Wiley & Sons, New York,* (1977).

[10] Derek L. Eager, John Zahorjan, and Edward D. Lazowska, "Speedup Versus Efficiency in Parallel Systems," Tech. Report 86-08-01, Dept. of Computer Science, University of Washington (August 1986).

[11] Raphael Finkel, Bahman Barzideh, Chandreshekhar W. Bhide, Man-On Lam, Donald Nelson, Ramesh Polisetty, Sriram Rajaraman, Igor Steinbeig, and G. A. Venkatesh, "Experience with Crystal, Charlotte and Lynx (Second Report)," Tech. Report #649,, Computer Sciences Dept., University of Wisconsin-Madison (July 1986).

[12] Bernard Lint and Tilak Agerwala, "Communication Issues in the Design and Analysis of Parallel Algorithms," *IEEE Transactions on Software Engineering* **SE-7**(2) pp. 174-188 (March 1981).

[13] K. G. Lockyer, *An Introduction to Critical Path Analysis,* Pitman Publishing Company (1967).

[14] L.F. Mackert and G. M. Lohman, "R* Optimizer Validation and Performance Evaluation for Distributed Queries," Research Report, IBM Almaden Research Center (January 1986).

[15] B.P. Miller, "DPM: A Measurement System for Distributed Programs," *IEEE Transactions on Computers,* (to appear 1987).

[16] Barton P. Miller and Cui-qing Yang, "IPS: An Interactive and Automatic Performance Measurement Tool for Parallel and Distributed Programs," Proceedings of the 7th International Conference on Distributed Computing Systems, IEEE Computer Society, Berlin, FRG (September 21-25, 1987).

[17] J. Mohan, "Performance of parallel programs: model and analyses.," CMU-CS-84-141, Ph.D. Thesis, Carnegie Mellon U. Comp.Sci.Dept. (1984.).

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

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

[20] Cui-qing Yang, "A Structured and Automatic Approach to the Performance Measurement of Parallel and Distributed Programs," Tech. Report 713 Computer Sciences Dept. , Univ. of Wisconsin-Madison (August 1987).