Critical Path Profiling of Message Passing and Shared-Memory Programs

Jeffrey K. Hollingsworth, *Member*, *IEEE Computer Society*

Abstract—In this paper, we introduce a runtime, nontrace-based algorithm to compute the critical path profile of the execution of message passing and shared-memory parallel programs. Our algorithm permits starting or stopping the critical path computation during program execution and reporting intermediate values. We also present an online algorithm to compute a variant of critical path, called critical path zeroing, that measures the reduction in application execution time that improving a selected procedure will have. Finally, we present a brief case study to quantify the runtime overhead of our algorithm and to show that online critical path profiling can be used to find program bottlenecks.

Index Terms—Parallel and distributed processing, measurement, tools, program tuning, on-line evaluation.

1 Introduction

In performance tuning parallel programs, simple sums of sequential metrics, such as CPU utilization, do not provide the complete picture. Due to the interactions between threads of execution, improving the performance of a single procedure may not reduce the overall execution time of the program. One metric, explicitly developed for parallel programs, that has proven useful is Critical Path Profiling [27]. Critical path profiling is a way to identify the component in a parallel program that limits its performance. Based on our experience with commercial and scientific users, Critical Path Profiling is an effective metric for tuning parallel programs. It is especially useful during the early stages of tuning a parallel program when load imbalance is a significant bottleneck [11]. In this paper, we introduce a runtime, nontrace-based algorithm to compute the critical path profile. Our algorithm also permits starting or stopping the critical path computation during program execution and reporting intermediate values.

Previous algorithms to compute the critical path profile are expensive. In an earlier paper [20], we described an offline (post mortem) approach to computing the critical path profile that required space proportional to the number of procedure calls and communication operations performed. For large, long running programs, this space requirement limited the usefulness of Critical Path Analysis.

To make critical path profiling practical for long running programs, we developed an online (during program execution) algorithm that incrementally computes the critical path profile for a selected procedure(s). It requires O(p) space, where p is the number of processes in the program. The time required is O(|e′|), where e′ is the set of interprocess events and call and return events for the selected procedure(s). Our online approach makes it possible to integrate critical path profiling into online performance monitoring systems, such as Paradyn [18]. By using Paradyn's dynamic instrumentation system, we only need to insert instrumentation code for the procedures whose share of the critical path we are currently computing.

We also present an online algorithm to compute a variant of critical path, called critical path zeroing. Critical path zeroing measures the reduction in application execution time that improving a selected procedure will have. We then show how the online critical path algorithm can be adapted to work with shared-memory programs. Finally, we present results from running an initial implementation of our algorithm using several PVM [9] based parallel programs. Initial results indicate that our online critical path algorithm can profile up to eight procedures with a 3-10 percent slow down of the application program.

2 Critical Path

A simple definition of the critical path of a program is the longest, time-weighted sequence of events from the start of the program to its termination. For an execution of a sequential program, there is only one path whose events are the procedure invocations and returns, and the length of the path is the CPU time consumed by the process. In a parallel program, communication and synchronization events result in multiple paths through a program's execution. Fig. 1 shows the sequences of events in a three process parallel program (indicated by the three vertical lines). The diagonal arcs show the communication events between processes and the dashed line shows the critical path through the program's execution. In this example, the length of the critical path is 16 and it is entirely contained in the rightmost process. Nodes represent events during the execution of a parallel program and arcs indicate the ordering of events. The weight assigned to each arch is the amount of time that elapses between events. The table at the right summarizes the amount of time each procedure contributes to the critical path.

• *The author is with the Computer Science Department, University of Maryland, College Park, MD 20742.*
E-mail: hollings@cs.umd.edu.

Manuscript received 25 June 1997; revised 21 Jan. 1998.
For information on obtaining reprints of this article, please send e-mail to: tpds@computer.org, and reference IEEECS Log Number 105302.

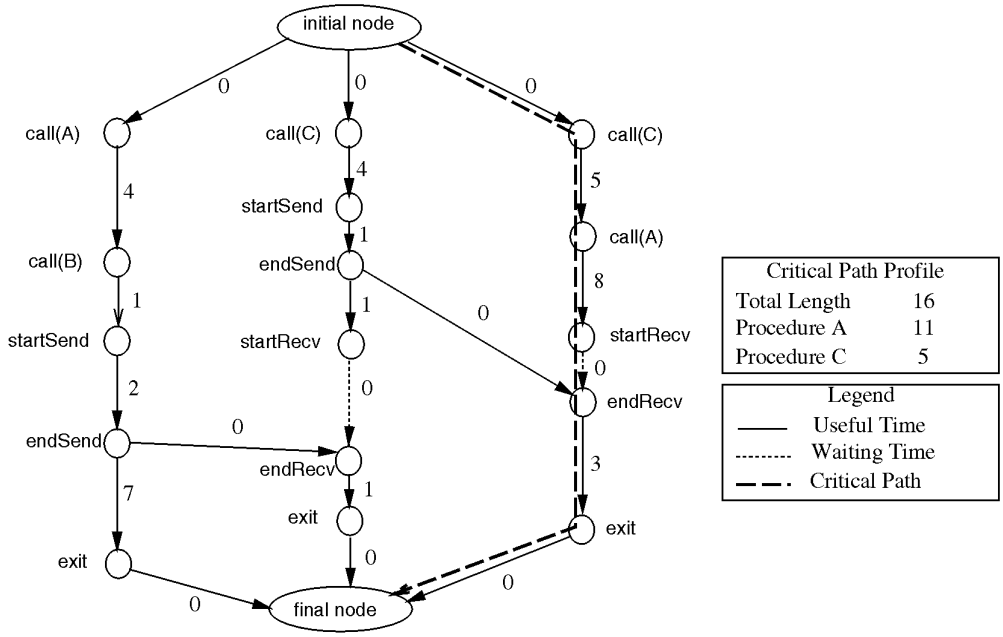

Fig. 1. A program activity graph and calculating its critical path.

The advantage of critical path profiling compared to metrics that simply add values for individual processes is that it provides a "global view" of the performance of a parallel computation that captures the performance implications of the interactions between processes. However, this advantage of providing a global view is exactly what makes it difficult to efficiently compute the metric. In a distributed system, extracting a global view during the computation requires exchanging information between processes. This exchange of information will require resources (e.g., processor cycles and communication bandwidth) and could potentially slow down the computation being measured. In this section, we introduce a new, efficient online algorithm to compute the critical path profile.

2.1 Formal Definition of Critical Path

Before we describe our algorithm, we define a few relevant terms:

Process: A thread of execution with its own address space.

Event: Observable operations performed by a process. A process communicates with other processes via messages. Message passing consists of start-send, end-send, start-receive, and end-receive operations, each of which is an event. Message events can be "matched" between processes. For example, an end-send event in one process matches exactly one end-receive event in another process. Processes also have local events for procedure calls and returns.[1]

Program: One or more processes that communicate during execution. This definition of a program captures SPMD, MIMD, and client-sever systems.

Program Execution: A single execution of a program on one or more processors with one input. A program execution consists of one or more processes. A program execution is defined by the **total** ordering of all events in all its processes. We denote a program execution P.

Program (Execution) Trace: A set of logs, one per process, that records the events that happened during the execution of that process. For a program execution P, let PT[p, i] denote the *i*th event in process p.

CPU Time: A per-process clock that runs when the process is executing on a processor and is not waiting for a message. Each event is labeled with the current CPU time at the time of the event.

Program Activity Graph (PAG): A graph of the events in a single program trace. Nodes in the graph represent events in the program's execution. Arcs represent the ordering of events within a process or the communication dependencies between processes. Each arc is labeled with the amount of CPU time between events. Fig. 1 shows a simple PAG for a parallel program with three processes.

The critical path of a parallel program is the longest path through the PAG. We can record the time spent on the critical path and attribute it to the procedures that were executing. The Critical Path Profile is a list of procedures and the time each procedure contributed to the length of the critical path. The time spent in these procedures is the reason that the program ran as long as it did. Unless one of these procedures is improved, the application will not run any faster.

Since the number of nodes in the PAG is equal to the number of events during the program's execution, explicitly building the graph is not practical for long running programs. One way to overcome this limitation is to develop an algorithm that does not require storing events logs or building the graph. However, we want to compute the critical path profile for distributed memory computations and, therefore, any online approach will require instrumentation

1. This definition is extended to include locks and barriers in Section 4.

```
1   struct {        // one per process
2    longest;       // longest path in this process.
3    funcShare;     // selected  func's share of  the
                    // critical path across all processes
4    funcActive;    // (bool) the  selected function is
                    //  currently active in this process
5    lastTime;      // time of the last instrumented event
6    funcLastTime;// time of last instrumented func event.
7   };

8   Send(toHost):
9    now = CPUTime();
10   process.longest += now - process.lastTime;
11   process.lastTime = now;
12   if (process.funcActive) {
13       // add active time of this func.
14       process.funcShare += now -
          process.funcLastTime;
15       process.funcLastTime = now;
16    }
17    send(toHost,process.longest,
       process.funcShare);

18  Recv(fromHost):
19    now = CPUTime();
20  process.longest += now - process.lastTime;
21    process.lastTime = now;
22    recv(fromHost, rmtLongest, rmtFuncShare);
23    if (rmtLongest > process.longest) {
24       process.longest = rmtLongest;
25       process.funcShare = rmtFuncShare;
26    } else {
27       if (process.funcActive) {
28          process.funcShare += now -
             process.funcLastTime;
29       }
30    }
31    if (process.funcActive) {
32      // start the func clock here.
33      process.funcLastTime = now;
34    }

35  Entry: // to selected procedure or function
36    // mark active and start the func clock.
37    increment process.funcActive;
38    if process.funcActive is one
        process.funcLastTime = CPUTime();

39  Exit: // from selected procedure or function
40    decrement process.funcActive;
41    if process.funcActive is zero
        process.funcShare += CPUTime() -
                process.funcLastTime;
```

Fig. 2. Algorithm to compute the Critical Path of a procedure on-the-fly.

messages to co-exist with (and compete for resources with) application messages. Therefore, we need to keep the volume and frequency of instrumentation messages small. Since programs can have hundreds or thousands of procedures, an approach that requires sending messages whose lengths are proportional to the number of procedures can cause a significant interference with the application program due to message passing.

2.2 Online Computation of Critical Path

With these challenges in mind, we have developed an online algorithm to compute the Critical Path Profile. We describe our approach to computing the Critical Path Profile in three steps. First, we show an algorithm to compute the share (fraction) of the critical path for a specified procedure. Second, we describe how to calculate the fraction of the critical path for a single procedure for a specific subinterval of the program's execution starting at the beginning of the computation. Third, we discuss how to start collecting critical path data during program execution. In this section, we present the algorithm assuming a message passing program; the algorithm is extended to support shared memory programs in Section 4.

Rather than computing the Critical Path Profile for all procedures in the application, we compute the Critical Path Profile for a selected set of procedures. Currently, selecting the desired set of procedures to compute the Critical Path Profile for is left to the programmer. A good heuristic is to identify those procedures that consume a large percentage of the total CPU time of the application. Selecting high CPU time procedures works well since, although critical path profiling may assign a different ordering and importance to the top 10 procedures, the procedures generally remain the same [11]. If top procedures are not the same, this can be detected since their cumulative share of the critical path length will be small. In this case, the programmer selects a different set of procedures and computes the critical path share for them.

We could also automate the identification of the top items in the critical path profile. To do this, we take advantage of the fact that we are interested in the top m items, where m is a user supplied value on the critical path profile, and that the most expensive operation is to send messages between processes. Based on these two assumptions, it is possible to employ a variation on binary search to identify the top m items from a set of n items in $O(m \log_2 n)$ time. The details of this algorithm are given in Appendix B.

To compute the length of the critical path (but not the share due to any procedure), we harness the normal flow of messages in the application to traverse the PAG implicitly. For each message sent, we attach an extra value to indicate the length of the longest path to the point of the send operation. For each receive event, we compare the received value to the local length of the critical path. If the received length is longer than the local one, we set the value of the local copy to the received copy. At the end of the computation, the critical path length is the longest path through any of the processes. To compute the share of the longest path due to a selected procedure, we also keep track of (and pass) the amount of time the selected procedure is on the longest path.

Each process keeps a structure of five variables (shown in lines 2-6 of Fig. 2). These variables record the longest path ending in the process, the share of that path due to the selected procedure, a flag to indicate if the selected procedure is currently active, the time of the last recorded event in the process, and the time of the last recorded event when the selected procedure was active. To compute the Critical Path for multiple procedures, we would replicate the variables **`funcShare`**, **`funcActive`**, and **`funcLastTime`** for each procedure.

Four events in the application program require instrumentation: message send and receive, and calls to and returns from the desired procedure. Pseudocode for the algorithm is shown in Fig. 2. It consists of four short code segments, one for each of the four application events that require instrumentation. Since all of the code segments require constant time to execute, the complexity of the computation is linear in the number of events. The only data structures are one copy of the per process structure (shown in Fig. 2, lines 1-7) and the piggybacked data for messages that have been sent but not received (stored in message buffers).

For clarity, we described the inclusive critical path for a single selected procedure (i.e., cumulative time spent in the procedure and procedures it calls). To compute the noninclusive critical path for a procedure, we need to insert instrumentation before and after each subroutine called by the selected subroutine. Before the called subroutine, we could use the same instrumentation shown in lines 39-41 of Fig. 2. After the subroutine returns, we use the instrumentation shown in lines 35-38 of Fig. 2.

2.3 Critical Path of Partial Program Execution

The algorithm presented in the previous section works as long as we want to compute the critical path for the entire program's execution and report the result at application termination. However, we also would like to be able to compute the critical path for a fraction of the program's execution. Selecting a fraction of the program's execution is equivalent to dividing the PAG into three disjoint pieces (sets of vertices). The first piece contains those events before we want to compute the critical path, the second the events we wish to compute the critical path for, and the third piece the events after the selected interval. To have a single critical path through the selected subset of the PAG, we must insert a single start node that has the first selected event from each process as its successor. Likewise, we require a final node that is the successor to the selected node in each process (i.e., we assume that a selected region is terminated by a barrier).

Since our algorithm doesn't explicitly build the PAG, we must identify the desired division of the PAG into parts by sending data attached to application messages. First, we will describe how to stop critical path analysis and, then, we will return the question of starting it during execution.

Closely related to stopping critical path profiling during program execution is sampling intermediate values for the critical path profile. In this case, we compute the critical path profile up until a well-defined point and report its value. Sampling the critical path is the same as stopping the critical path at some point during program execution. We now describe how to calculate intermediate values of the critical path starting from the beginning of the program's execution.

To compute the intermediate critical path, we periodically sample each process in the application and record both its current critical path length and the share of the critical path due to the selected procedure. This information is forwarded to a single monitoring process. During the computation, the current global critical path length is the maximum value of the individual sample values from each process. The value of the critical path for the selected procedure is the procedure component of the longest critical path sample. The sampling step is shown in Fig. 3.

```
1  Sample:     //  called by an alarm expiring in the
2              //    application process
3       send(monitorProcess, process.longest,
4            process.funcShare);

5  processSample:     //  in a central monitoring process
6       //  keep looping reading reports of the CP.
7       global.longest = 0;
8       global.funcShare = 0;
9       do until computation done {
10           recv(fromProc, sample.longest, sample.funcShare);
11           if (sample.longest > global.longest) {
12               // update length and func's share of length.
13               global.longest = sample.longest;
14               global.funcShare = sample.funcShare;
15               // report CP length and percentage
16               //  in the selected procedure.
17      }
```

Fig. 3. Sampling the Critical Path during program execution.

Since sampling of intermediate values of the critical path in each process possibly occurs at different times, an important question is can we combine the samples into a metric value that represents a consistent snapshot of the critical path during program execution? Our goal is to show that the sequence of intermediate values of the Critical Path at the central monitoring process corresponds to a consistent view. Conceptually, a consistent snapshot is one that is achieved by stopping all processes at once and recording the last event in each process. However, it is sufficient to show that a sample corresponds to a set of events (one per process) that could have been the last events if we had stopped all of the processes. To explain this property, we introduce two additional definitions:

Happen Before: Denotes the transitive partial ordering of events implied by communication operations and the sequence of local events in a process. For local events, one event happened before another event if it occurred earlier in the program trace for that process. For remote events, send happens before the corresponding receive event. Formally, it is the set of precedence relationships between events implied by Lamport's happened before relationship [14]. If event x happens before event y, we denote this by $x \rightarrow y$.

State Slice: For any event e in a program trace PT and any process p, a state slice is the last event in a process that is required to happen before e based on the happen before relation. Formally, slice[p, e] = (PT[p, i] : $PT[p, i] \rightarrow e$ and $(\forall j > i \neg (PT[p, j] \rightarrow e))$), where p is a process in PT, e is an event in PT, and i and j are integers between one and the number of events in process p. slice[*, e] is the set of events, one per process, that are the last events required to precede e.

In addition to collecting intermediate values that correspond to consistent snapshots of a program's execution, we also want to report values in a timely manner. An intermediate value of the critical path should correspond to a point in the program's execution that is a bounded amount of time since the last sample from each process. If we simply

use the state slice associated with the current longest value to define such a point, we can't ensure that the point is timely. The reason for this is that if one process doesn't communicate with the other processes, the event for the noncommunicating process in the state slice might be arbitrarily early.

To ensure the timeliness of samples, we need to combine the state slices for the latest sample from each process. To do this, we compute G, the latest event from each process known at the monitoring station. For sample i, G[p, i] = max(G[p, i – 1], slice[p, i]). Hence, the events in the combined state slice G will be no earlier than the last sample from each process. However, we must show that G produces a consistent snapshot of the program. The proof of this property of our sampling algorithm appears in Appendix A.

We would also like to be able to start computing the critical path once the program has started execution. To ensure the computed metric is meaningful, we need compute the critical path starting from a point that corresponds to a consistent snapshot of the program's execution. However, to start computing our metric, a central process must send messages to each application process requesting it to start collecting critical path data. In general, it is impossible to ensure that all of the processes will receive this message at the same time. Even if we could, we need to account for messages that were being transmitted (in flight) at the time we start to compute the critical path.

We assume that every message that is sent between two processes either contains a critical path message if the sending process has started recording CP, or not if it the sender has not. We further assume that any receiver can detect whether or not a message has a critical path message attached to it. Without loss of generality, we can assume that there is only one type of critical path message (i.e., we are computing the critical path for a single procedure). There are four cases to consider:

1) A message without CP data arrives at a process that is not computing the critical path.
2) A message with CP data arrives at a process that is already computing the critical path.
3) A message with CP data arrives at a process that is not computing the critical path.
4) A message without CP data arrives at a process that is already computing the critical path.

Cases 1 and 2 require no special treatment since they occur either before or after the point where the critical path computation starts. We handle Case 3 by starting to collect critical path data at that point. We handle Case 4 by doing nothing; the sending event occurred before we started calculating the Critical Path.

To ensure that we can start calculating the critical path during program execution, we must establish that no matter when each process receives the message to start calculating the critical path (either directly from the monitoring station or from another process) that the resulting calculation will correspond to computing the critical path starting from a consistent state during the execution of P.

This is a special case of the consistent global snapshot problem described by Chandy and Lamport in [6]. Chandy and Lamport describe an algorithm to record the global state of a computation by sending a marker token along communication channels. In our scheme, the receipt of a critical path start message is equivalent to receipt of a marker token in their scheme. We won't repeat their proof here, but the key idea of the proof is that it is possible to order the events in the computation such that all events before starting to calculate the critical path occur before all events after starting to calculate the critical path and that the reordering of events is a feasible execution of the program.

```
1  Recv(fromHost):
2      now = CPUtime();
3      process.longest += now-process.lastTime;
4      process.lastTime = now;
5      recv(fromHost, rmtLongest, rmtFuncShare);
6      if (rmtLongest - rmtFuncShare >
7          process.longest - process.funcShare) {
8          process.longest = rmtLongest;
9          process.funcShare = rmtFuncShare;
10     } else {
11         if (process.funcActive) {
12             process.funcShare +=
13                 now - process.funcLastTime;
14         }
15     }
16     if (process.funcActive) {
17         // start the func clock here.
18         process.funcLastTime = now;
19     }
```

Fig. 4. Computing Logical Zeroing.

2.4 Online Critical Path Zeroing

Critical Path profiling provides an upper bound on the improvement possible by tuning a specific procedure. However, it might be the case that slightly improving a procedure on the critical path could cause that procedure to become subcritical and that most of the effort to tune that procedure would be wasted. To provide better guidance in this situation, we previously proposed a metric called logical zeroing [11] that computes the reduction in the length of the critical path length due to tuning specific procedures. However, computing logical zeroing also required collecting a large amount of data and building a post mortem graph. Fortunately, a variation of our online critical path algorithm can be used to compute logical zeroing.

The key idea of this algorithm is the same as critical path; we piggyback instrumentation data onto application messages. The only difference is at merge nodes, where we compare the "net" path lengths for both the remote and local sample. The "net" path length is the path length minus the share of the path due to the selected procedure. Fig. 4 shows the pseudocode for the computation of logical zeroing at a "merge" (receive) node. The changes are at lines 6-7; before comparing the two path lengths, we subtract the corresponding share of each path due to the selected procedure. The only other change required is when the critical path value is sampled; we report the "net" critical path length, not the share of the critical path due to the selected procedure.

Number of CP Items	~75% Computation Wall Time	Overhead	~60% Computation Wall Time	Overhead
Base	154.1		91.7	
0	157.0	1.9%	94.9	3.4%
1	157.4	2.1%	95.5	4.1%
4	157.5	2.2%	95.8	4.4%
8	158.4	2.8%	95.8	4.4%
16	158.5	2.9%	95.9	4.6%
32	159.6	3.6%	97.0	5.8%

Fig. 5. Overhead required to compute per procedure critical path.

3 Initial Implementation

We have added an implementation of our online critical path algorithm to the Paradyn Parallel Performance Tools. We were interested in learning two things from our implementation. First, we wanted to quantify the overhead involved in piggybacking instrumentation messages onto application messages. Second, we wanted to demonstrate that the information supplied by critical path analysis provides additional guidance to programmers compared to CPU time profiling.

Our initial implementation works with PVM programs on any platform that the Paradyn tools support. There is no fundamental reason to use PVM, but for each message passing library we need to write a small amount of code to support piggybacking critical path messages onto data messages. Due to the semantics of PVM and our desire not to modify the PVM source code, we were forced to use an implementation of piggybacking that requires that a separate message be sent right after every message, even if we are not currently computing the critical path. Although this extra message does add a bit of overhead, we show below that it is not significant. It is possible to eliminate this extra message with a slight modification of PVM.

3.1 Overhead of Instrumentation

To quantify the overhead of piggybacking instrumentation messages onto application data, we constructed a simple two process test program. Each process "computes" for some interval of time and then sends a message to the other process. By varying the amount of data transferred and the amount of "computation" done we can simulate programs with different ratios of computation to communication. We can also vary message size and frequency. Since "piggybacking" messages incurs a per message overhead, the more frequently messages are sent, the higher the overhead. PVM runs on a range of different networks from Ethernet to custom MPP interconnects. To permit investigating the per procedure overhead of piggybacking messages, our test application contained 32 procedures that were linked into the executable but never called. We then varied the number of selected procedures from 0 to 32. To gauge the impact of our instrumentation on different platforms, we conducted tests on two systems. The first was a pair of Sun Sparcstation-5s connected by 10 Mb/s Ethernet. The second was two nodes of an IBM SP-2 connected by a 320 Mb/s high performance switch. For all reported results, the times shown were the minimum time of three runs. Variation between runs was less than 1 percent.

The table in Fig. 5 shows two versions of the program run on the SPARC/Ethernet configuration. The first column shows the number of items (procedures) whose Critical Path is being computed. The first version computed for 75 percent of its execution time and spent the remaining 25 percent of the time sending and receiving messages (shown in the second and third columns). The second version spent 60 percent of its time in computation and 40 percent in message passing (shown in the fourth and fifth columns). Each program sends the same size and number of messages, but we varied the "computation" component between the two programs. Message Passing used a 10Mb/s Ethernet.

The time required to send an empty piggyback message is shown in the third row. For the 75 percent computation case, sending empty messages added a 1.9 percent overhead to the application and for the 60 percent computation case it was 3.4 percent. This provides an indication of the overhead required to simply enable the piggyback mechanism and send an empty message. We were also interested in measuring the per procedure cost of computing the Critical Path. For each version, we varied the number of procedures for which we calculated the critical path for from one to 32 procedures. In addition, we report the time required to run the uninstrumented version of the program. None of the critical path items (procedures) were called by the application, so the reported overhead represents the incremental cost of computing the critical path for a procedure compared to the cost of computing a simple CPU profile.

The data shown in Fig. 6 is for the IBM SP-2 configuration. In this case, we held the computation to communication ratio fixed at 75 percent computation and varied the message passing frequency and message size. We used a message passing rate of approximately five, 50, and 150 messages per second per processor. For these three cases, the size of each message was 48,000, 4,800, and 480 bytes respectively. In all three cases (five, 50, and 150 messages per second per processor), the overhead of sending the empty critical path message was less than 1 percent. As expected, when we increased the number of critical path items being computed, the overhead went up. Notice that the overhead for the 32 procedure case for 150 messages/sec results in an overhead of almost 50 percent! Clearly, this would not be acceptable for most applications. However, for the four and eight procedure cases, the overhead is 6 and 11 percent, respectively. We feel this amount of overhead would be acceptable for most applications. If

Number of CP Items	5 msgs/sec Time	Percent	50 msgs/sec Time	Pct	150 msgs/sec Time	Pct
Base	50.3		51.9		194.0	
0	50.4	0.3	51.8	0.0	194.5	0.3
1	50.9	1.2	52.7	1.7	197.6	1.9
4	50.8	1.1	53.3	2.7	206.0	6.2
8	51.4	2.2	55.4	6.9	215.4	11.1
32	55.1	9.5	64.0	23.5	287.8	48.4

Fig. 6. Comparison of overhead for different message rates. The first column shows the number of items (procedures) whose Critical Path are being computed. The second and third columns show the execution time and overhead when the test application sends five messages/second. The fourth and fifth columns show the results when the message passing rate is increased to 50 messages/second. The sixth and seventh columns show the results for a sending 150 messages/sec. All messages were sent using a 320Mbps SP-2 high performance switch. The row denoted base reports the time with no instrumentation enabled, and the 0 item case is when the piggyback mechanism is enabled, but not procedures are selected.

Procedure	CP	% CP	CPU	% CPU
nas_is_ben	8.8	31.7	103.1	67.8
do_rank	0.5	1.8	29.9	19.7
create_seq	18.5	66.5	18.7	12.3

Fig. 7. NAS IS Benchmark Results. The first column shows the procedures that consume the largest amount of CPU time in the application. The second and third columns show the share and percent of the critical path length due to each procedure. The fourth and fifth columns show the CPU time and percent of total CPU time consumed by each procedure. The table is ordered by decreasing amount of CPU time consumed. The procedure `create_seq` is the most important procedure according to Critical Path metric, but only the third most important based on CPU Time.

an even lower overhead were necessary for an application, it could be run several times and the critical path profile information could be computed for a small number of procedures each time.

3.2 Case Studies

To evaluate the effectiveness of Critical Path computation on applications, we measured the Critical Path and CPU time profiles for three application programs. Each program used the PVM message passing primitives for communication.

First, we measured the performance of Integer Sort, one of the NAS benchmarks. The program was run on eight nodes of an IBM SP-2 using the High Performance Switch for message passing. The results are shown in Fig. 7. This table summarizes the Critical Path values and CPU time for the top three procedures. For each metric, we present the value of the metric and the percentage of the total metric value. Since the total value varies with different metrics, the percentage value is the important one for comparison. The percentage is the "score" for each procedure, indicating the important assigned to it by that metric.

This example shows the benefit of the additional information provided by Critical Path compared to CPU time profiling. Although `create_seq` is only 12 percent of the total execution time of the overall program, it was responsible for over 66 percent of the length of the critical path. Ranked by fraction of the CPU time consumed, `create_seq` is the third most important procedure, yet it is the most important when

Procedure	CP	% CP	CPU	% CPU
BeManager	31	48.0	47	19.3
MinCircuit	29	44.9	196	80.7

Fig. 8. TSP application.

Program Component	CP Zero	% CP Zero	CP	% CP	CPU	% CPU
s_recv	7.3	18.4	9.7	24.5	36.6	29.8
step	5.5	13.9	8.5	21.3	23.9	19.4
s_send	4.2	10.6	7.1	17.9	21.4	17.4
tracer	1.7	4.3	1.9	4.8	5.3	4.3
clinc	1.5	3.9	1.5	3.9	5.8	4.7

Fig. 9. Metric values for Ocean application. The first column shows the procedures that consume the largest amount of CPU time in the application. The second and third columns show the amount and percent that the critical path length is reduced if the weight assigned to that procedure is set to zero. The fourth and fifth columns show the CP time and for each procedure. The sixth and seventh columns show the CPU time and percent CPU time for each procedure. The table is ordered by decreasing amount of CPU time consumed.

ranked by Critical Path. The reason for this is that the routine is completely sequential. Other sequential metrics would not have easily identified the importance of `create_seq`.

Second, we measured a implementation of the Traveling Salesperson Problem (TSP) using Critical Path Analysis and CPU profiling. Although this is a small problem (taken from a program written as part of an introductory parallel programming class), it is illustrative of the types of performance problems that can happen in client-server applications. The program consists of one manager process and eight worker processes. Fig. 8 shows the CPU and Critical Path values (including time spent in called functions) for the two most important functions in the application. The first column shows the procedures (including called procedures) that consume the largest amount of CPU time in the application. The second and third columns show the share and percent of the critical path length due to each procedure. The fourth and fifth columns show the CPU time and percent of total CPU time consumed by each procedure. `BeManager` is the main procedure of the master process, and `MinCircuit` is the main procedure of the worker processes. This table shows that, based on CPU time, the `MinCircuit` procedure is the most important procedure to tune, consuming 80 percent of the time. However, the Critical Path value shows that the `BeManager` is responsible for 48 percent of the Critical Path time. The value is much larger for the `BeManager` procedure due to the master process becoming a bottleneck when eight clients are exchanging partial solutions with it (due to the program's dynamic load balancing).

We also measured an implementation of the GFDL Modular Ocean Model [5] developed by Webb [25]. The results of computing the CPU time profile, Critical Path Profile, and Critical Path Zeroing are shown in Fig. 9. The results show the importance of using Critical Path Zeroing to identify the potential decrease in the Critical Path length by improving selected procedures compared to CPU time profiling or even Critical Path Profiling. Although all three metrics indicate that `s_recv` is the most

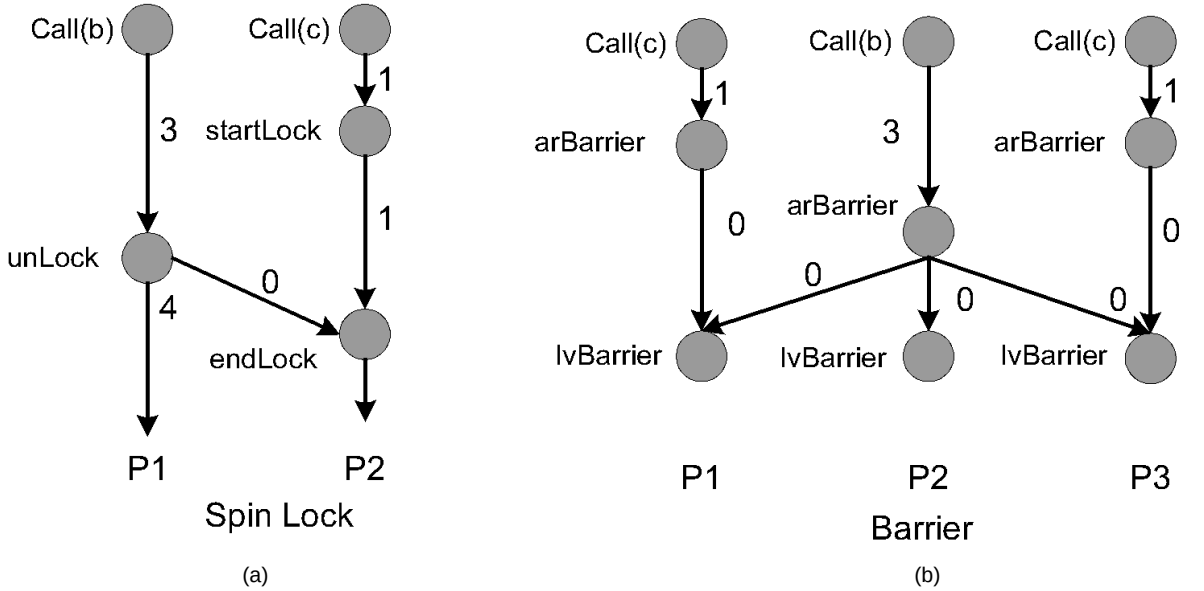

Fig. 10. PAG for Spin Locks and Barriers. (a) The left graph shows a typical PAG representation for a Spin Lock. The unLock of the previous use of the lock must proceed the endLock of the next use. The time spent waiting to acquire a lock is the time between the startLock and endLock events. (b) The right graph shows the PAG for a barrier. The lvBarrier events of all processes must wait for the final arBarrier event as shown by the diagonal lines.

important program component to fix, the weight assigned to it varied from 18 to 30 percent. CP Zeroing provides a lower value for this procedure because other subcritical paths in the program limit the improvement possible by just tuning this one routine.

4 Support for Shared-Memory Programs

Critical Path and Critical Path Zeroing are useful techniques for shared memory programs, too. In this section, we describe how the basic ideas of critical path can be applied to a shared memory program and present a few short case studies that demonstrate the utility of the information gathered. We also explain how the idea of Critical Path Analysis can be extended to apply not just to individual synchronization events, but to higher level events such as items entering and leaving a work queue.

4.1 Computing Critical Path for Shared-Memory Programs

In shared memory programs, the basic abstractions for synchronization between threads of execution are locks and barriers. For each of these operations, we need to have the correct representation in the program activity graph. Fig. 10 shows the PAG representation for spin locks and barriers. For spin locks, the *n*th call to endLock (lock aquire) depends on the n – 1st call to unLock having been completed. For Barriers, the last thread of control to arrive at the *n*th use of a barrier must happen before any thread may leave the barrier (shown as the lvBarrier event). In the case of Critical Path Zeroing, the last event to arrive at a barrier is determined by its **net** time rather than its current time.

The other significant difference between the message passing and shared-memory versions of the online critical path algorithm is how the information is shared between processes. In the shared-memory version, a data structure for each synchronization object (e.g., lock or barrier) is maintained in shared-memory and updated by the instrumentation code in each thread of control. However, in the message passing case, Critical Path data is exchanged via messages.

Tracking individual Spin Locks produces a useful metric for Critical Path. However, the nondeterminism present in many parallel programs can create problems for Critical Path Zeroing to accurately predict the correct execution time if a procedure is tuned. Since Critical Path Zeroing is based on using the PAG for a single program execution, changes due to tuning could alter the structure of the graph. For programs that are general race free [21] (i.e., lack nondeterminism), changing the weights assigned to arcs in a PAG will not result in structural changes to the graph. However, for programs that rely on nondeterminism, such as task queues, changing the value of one arc in the graph can result in changes in graph structure.

To correctly predict the performance for these cases, the PAG structure should track the items entering and leaving work queues rather than the mutual exclusion that is used to implement the work queue. To track activity in a work queue, we match the en-queue of an item with its subsequent removal from the queue. An example of how objects in an event queue can be tracked is shown in Fig. 11. For the shared memory programs described below, we have implemented item-based tracking of work queues. It is possible to track objects in an event queue for message passing programs, too. However, due to the substantially coarser grained communication in most message passing programs, tracking of event queues has not been required to provide a useful information to programmers.

4.2 Impelementation and Evaluation of Shared-Memory CPA

To evaluate the effectiveness of the shared-memory version of the online critical path algorithm, we implemented a version

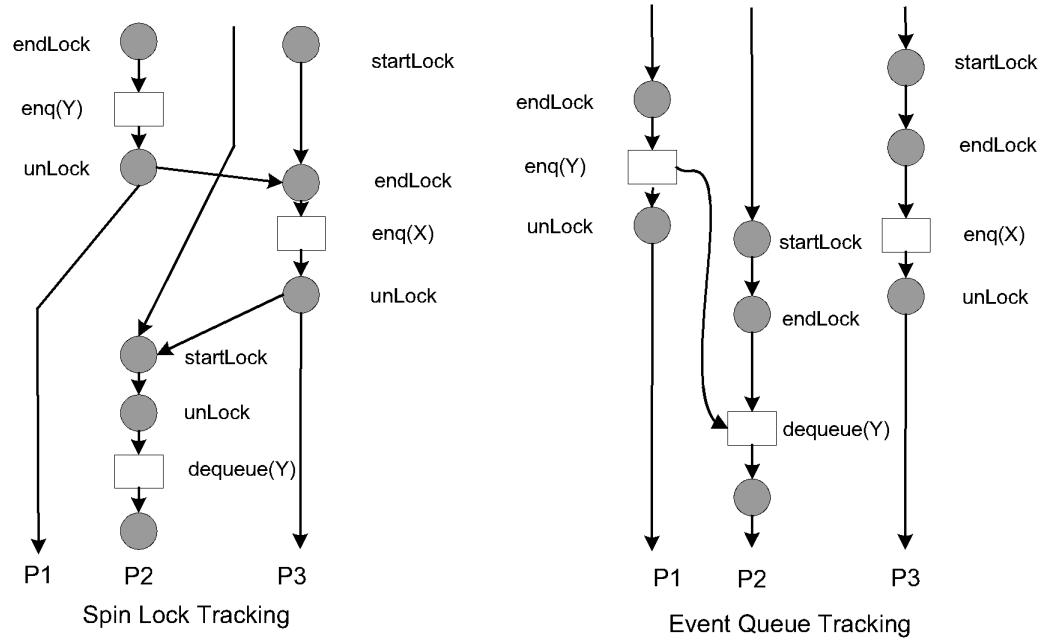

Fig. 11. Spin Lock vs. Event Queue Tracking. Two PAGs showing the different ways that the dependencies for an event queue can be represented. The left side shows a three process program tracking each spin-lock event. The right graph shows the same three process program when the contents of the event queue are tracked. In the right graph, the dequeue of object Y has an arc that forces it to follow the enqueue of the same object. On the left the graph, the dequeue of Y is also required to follow the enqueue of object X, even though these two events are not required to happen in that order.

of the algorithm that runs on DEC Alpha shared memory multi-processors and ran the Splash-II benchmarks applications [26]. For all programs, we ran Critical Path Zeroing, Critical Path, and UNIX prof/gprof profiling. For many of the applications, the results for critical path and traditional profiling produced nearly identical results. This is to be expected. For programs with good load balancing, the results for Critical Path and traditional profiling should be similar. However, for two of the applications, we noticed a significant difference between the two measurement techniques. The results for those two applications are shown below.

The values for the three metrics for the Raytrace application are shown in Fig. 12. For this application, the two critical path based metrics show that the sequential routine **`ReadGeoFile`** is about four times more important (3.5 vs. 13.5 percent) than the CPU profile metric indicated. This difference is due to the routine being entirely sequential.

The table shown in Fig. 13 shows the results of computing the three metrics for the Choleski Kernel. The table shows the 10 procedures that contribute the largest amount to the application's running time ordered by decreasing value of the CPU Profile metric.[2] For four of the procedures (marked inc), we show the inclusive time consumed by the target procedure and all procedures it calls. We used the inclusive metric for these routines since they contained frequent calls to small helper routines and, so, the reported routine and its called subroutines would likely be tuned as a unit. For the rest of the procedures, we present the value for the procedure without the time spent in any of its called routines. Due to differences in absolute measures, the different metrics can be best compared in a head-to-head fashion by looking at the percent columns that show the relative weight assigned to each procedure.

The differences between the results for the Prof metric and the Critical Path metrics is pronounced. Although all three metrics indicate the same two procedures as the top target for tuning, the weight assigned to each procedure is substantially different. For the **`ModifyTwoBy`** routine, the CPU Profile metrics predicts that tuning this procedure will reduce the running time of the application by over 1/3; however, the Critical Path and Critical Path Zeroing metrics each predict the improvement will be less that 1/4. For the procedure **`OneMatmat`**, the CPU profile metric predicts that tuning the procedure will reduce the running time of the application by 24 percent, but the CP Zero metric shows that the improvement will likely be less than half that amount, about 10.4 percent. For the third and fourth procedures, the difference between the three metrics is more pronounced. The CPU profile indicates that procedures **`GetBlock`** and **`OneDiv`** are the third and fourth most important. However, the Critical Path based metrics indicate that **`FillInNZ`** and **`CreateBlockedMatrix2`** are the third and fourth most important procedures. Of equal importance to rank is the weight assigned to each of these procedures. According the CPU Profile metric, the third and fourth ranked procedures are less than 5 percent each; however, the **`FillInNZ`** and **`CreateBlockedMatrix2`** routines are each responsible for almost 9 percent of the application's running time. We investigated the discrepancy between the two metrics and discovered that the procedures that received

2. The values reported for the CPU Time profile were obtained using UNIX Gprof. We also computed the metric using a direct timer based approach (as opposed to Gprof's sampling.) The results were almost the same, so we chose to show only the Gprof results here.

Procedure	CPU		CP		CP Zero	
	Time	Percent	Time	Percent	Time	Percent
IntersectHuniformPrimlist	44.5	54.1	12.0	46.7	11.9	46.6
Shade	28.9	35.1	9.1	35.3	8.4	32.9
ReadGeoFile	4.5	3.5	3.5	13.5	3.2	12.6

Fig. 12. CP vs. traditional profiling for the raytrace application. The results are for computing Prof, CP, and CP Zero for each three procedures. Both CP and CP Zero assign a significantly higher importance to `ReadGeoFile` than CPU profiling does.

Procedure	CPU Profile		CP		CP Zero	
	Time	Percent	Time	Percent	Time	Percent
Total	34.7		13.9		13.9	
ModifyTwoBy..	12.8	36.8	3.4	24.7	3.3	23.9
OneMatmat	8.2	23.5	2.1	14.9	1.4	10.4
GetBlock	1.6	4.6	0.3	2.1	0.1	0.8
OneDiv	1.3	3.7	0.3	2.2	0.1	0.8
FillInNZ (inc)	1.2	3.3	1.2	8.6	1.2	8.6
ReadSparse (inc)	1.0	3.0	1.1	7.6	1.0	7.5
CreateBlockedMatrix2 (inc)	1.1	3.0	1.2	8.5	1.2	8.5
FillInStructure (inc)	0.7	2.1	0.7	5.3	0.8	5.5
ComputeNZ	0.8	2.4	0.8	5.9	0.8	5.8
InsSort	0.8	2.2	0.8	5.4	0.8	5.5

Fig. 13. Critical path vs. traditional profiling for the Choleski kernel. The results are for computing Prof, CP, and CP Zero for each three procedures. Both CP and CP Zero assign significantly less importance to `GetBlock` and `OneDiv` than CPU profiling does. Likewise, the importance of the procedures `FillInNZ` and `CreateBlockedMatric2` is much higher with CP and CP Zero than CPU Profiling.

higher weight with the Critical Path based metric execute sequentially on one processor while the rest of the processors idle. Thus, the Critical Path metrics were able to rank the sequential routines as more important for tuning the application's execution time.

5 Discussion

Critical Path Profiling is a useful metric for many parallel programs. Its primary benefit is to help programmers understand where they can most productively spend time tuning their programs. This benefit comes from two factors:

1) Identifying procedures that may not be the largest consumers of CPU time but are, in fact, critical to performance since other activities in the program must wait for their completion.
2) Providing feedback about the limited impact a tuning option might have due to secondary and tertiary paths. This allows programmers to avoid spending time tuning procedures that will not result in a significant improvement in the application's execution time.

Critical Path provides the most value added compared to traditional profiling techniques for applications that either have a load imbalance or that contain processes executing different procedures at the same time. For example, client-server programs, control-parallel programs, and coupled data-parallel applications benefit from the additional information provided by Critical Path Analysis.

There are also some limitations to the technique. First, for many data parallel programs with good load balance, critical path provides an accurate metric, but no additional information over simple sequential metrics. Second, like any measurement tool, the results are based on a single execution of the program. The programmer is left with the responsibility of selecting a representative execution to measure.

Like any system that employs software-based instrumentation, our instrumentation can perturb the measurements and result in different event orderings and timings. Experience with both our new online algorithm, and previous offline algorithms has shown that most programs can tolerate a moderate level of instrumentation overhead (5-10 percent) without the instrumentation making any significant change in the length of or the procedures along the Critical Path. If perturbation were a major concern, we could employ the event-based perturbation compensation algorithms suggested by Malony [15]. However, instead we use a measurement based-technique to control the amount of instrumentation overhead [12].

6 Related Work

A graph representation of an execution of a parallel program similar to our PAG has been used extensively in the past [4], [7], [10], [19], [24]. Implicitly walking a PAG by attaching instrumentation messages onto applications messages has been used for online detection of race conditions in parallel programs [8], [13]. Similar instrumentation has also been used to reduce the number of events that must be logged for program replay [22].

Many metrics and tools have been developed to quantify the performance of parallel programs. The Paradyn [18] and Pablo [23] tools provide a wealth of performance metrics. One similar metric to Critical Path profiling is Normalized Processor Time [1]. Unlike NPT, Critical Path can be computed for either shared memory or message passing programs. Also, Critical Path provides information about

the relationships between sequences of events, while NPT relies on comparing the instantaneous behavior across all processors. Other metrics focus on specific sources of bottlenecks in parallel programs such as memory [16].

Many metrics and measurement techniques have been developed for sequential programs. Path Profiling [2] provides an efficient way to measure the frequently executed paths (sequences of basic blocks) through a sequential program and to associate both time and hardware events with each path. Digital's Continous profiling [3] combines hardware measurement with operating system support to measure time spent both in application programs, as well as executing operating system services. Hardware measurement facilities are included with many micro-processors today [1], [17], [28]. All of these sequential techniques complement Critical Path Profiling. In most cases, they can be used after Critical Path has isolated a performance problem to a specific procedure.

7 Conclusion

We have presented an online algorithm to compute the critical path profile of a parallel program and a variant of critical path called critical path zeroing. We showed that it is possible to start collecting the critical path during program execution and that sampling intermediate results for critical path profiling is possible and produces a meaningful metric. In addition, we showed that it is possible to compute this algorithm with minimal impact on the application program and presented a brief case study that demonstrated the usefulness of critical path for a PVM message passing program. Finally, we showed how the message passing online Critical Path algorithm can be extended to work with shared-memory programs.

Appendix A

In this appendix, we show that combing critical path samples corresponds to a feasible execution of P. In other words, we are not restricting ourselves to computing the critical path of the exact total ordering of the events in the computation, but instead to that of the family of feasible executions that satisfy the happened before relation.

Consider the sending of critical path samples to the monitoring station. Sending a sample message is an event in the application process. During program execution, a series of samples will arrive from the application processes. Let CP(i) represent the send sample event corresponding to the *i*th sample to arrive at the monitoring station. Therefore, slice[p, CP(i)] is the last event in process p that must have preceded the sample. For each sample i, let G[p, i] be the latest event for process p from all of the state slices for samples 0 to i (i.e., G[p, i] = max(G[p, i – 1], slice[p, i]). G[p, 0] = PT[p, 1]. Let G[*, i] denote the set of events for all processes p in G[p, i].

To show that the series of critical path samples correspond to a sampling of states during a feasible execution of P, we must show that all states G[*, 0], G[*, 1], …, G[*, n] correspond to a sequence of global states in feasible execution P′ of P.

THEOREM. *For all i,* G[*, i] *corresponds to a feasible global state of* P.

PROOF. The proof is by induction. G[*, 0] is trivially a feasible state since it represents that start of the program's execution. Now, assume G[*, i] is a feasible state. Let S denote the set of events that occur between G[*, i] and G[*, i + 1]. S consists of the events in each process that must occur after G[*, i] and at or before G[*, i + 1]. G[*, i + 1] is a feasible state of P if there exists a total ordering of S that satisfies the happen before constraint. To see that G[*, i + 1] is feasible, consider what it would mean if it were not. This would imply that there is no ordering of the events in S that satisfy happen before. For this to be the case, it requires that there exists events x, y, z in S such that $x \rightarrow y$, $y \rightarrow z$, and $z \rightarrow x$. However, this is not possible by the definition of HB; therefore, G[*, i + 1] is a feasible state of P. □

Finally, the sequence G[*, 0], G[*, 1], …, G[*, n] corresponds to a series of events in a single feasible execution of P. This can be shown by a construction, since G[*, 0] is the start of the computation, and we can construct a total ordering of the events in P such that G[*, 1] is a global state and, from there, such that the rest are global states. The constructed total ordering is then a feasible execution of P.

Appendix B

In this appendix, we present a simple algorithm to permit finding all items whose share (fraction) of the critical path is larger than 1/m, where m is an integer. Two key observations make this algorithm possible:

1) There are at most m items whose share of the critical path is greater than 1/m.
2) Since the major cost in computing the critical path corresponds to the sending of instrumentation messages, computing the aggregate critical path for a collection of procedures has about the same cost as computing the critical path for a single procedure.

We start the algorithm with n items to consider. We divide our n items into $2*m$ disjoint buckets each with $\lfloor n / (2*m) \rfloor$ or $\lfloor n/(2*m) \rfloor + 1$ items. We then compute the aggregate share of the critical path for each bucket. This step requires the same overhead as computing the CP share for $2*m$ procedures (i.e., sending $2*m$ values per message). At the end of the program, we compare the critical path share for each bucket. The critical path share of at most m buckets will be 1/m or larger. We discard the procedures in those buckets whose CP share is less than 1/m. This eliminates at least half of the procedures. We then repeat our algorithm with the remaining procedures put into $2*m$ buckets until the buckets contain only a single item (procedure). This pruning of procedures makes it possible to identify the up to m procedures responsible for at least 1/m of the overall execution of the program in $O(m \log_2 n)$ steps.

It is easy to remove the restriction that we must run the program n times to identify the program components that are responsible for more than 1/m of the total length of the critical path. To do this, we use the observation that, within

a single phase of a program's execution, its performance remains consistent. We can use the ability of our critical path algorithm to compute the critical path for part of a program's execution to compute the critical path for a fixed interval of the program, and then evaluate the next step in the search algorithm.

ACKNOWLEDGMENTS

This work was supported in part by U.S. National Institute of Science and Technology award 70-NANB-5H0055, DOE Grant DE-FG02-93ER25176, and U.S. National Science Foundation Grant ASC-9703212.

REFERENCES

[1] DEC, *DECchip 21064 and DECchip21064A Alpha AXP Microprocessors—Hardware Reference Manual*, EC-Q9ZUA-TE, June 1994.
[2] G. Ammons, T. Ball, and J.R. Larus, "Exploiting Hardware Performance Counters With Flow and Context Sensitive Profiling," *Programming Language Design and Implementation*, pp. 85-96, Las Vegas, Nev., June 1997.
[3] J.M. Anderson et al., "Continuous Profiling: Where Have All the Cycles Gone," *Symp. Operating System Principles*, pp. 1-14, Saint-Malo, France, Oct. 1997.
[4] D.F. Bacon and R.E. Strom, "Optimistic Parallelization of Communicating Sequential Processes," *Proc. SIGPLAN '91 Symp. Principals and Practice of Parallel Programming*, pp. 155-166, Williamsburg, Va., 21-24 Apr. 1991.
[5] K. Bryan, "A Numerical Method for the Circulation of the World Ocean," *J. Computational Physics*, vol. 4, no. 1, pp. 347-376, 1969.
[6] K.M. Chandy and L. Lamport, "Distributed Snapshots: Determining Global States of Distributed Systems," *ACM Trans. Computer Systems*, vol. 3, no. 1, pp. 63-75, 1985.
[7] J.-D. Choi and S.L. Min, "Race Frontier: Reproducing Data Races in Parallel-Program Debugging," *Proc. SIGPLAN '91 Symp. Principals and Practice of Parallel Programming*, pp. 145-154, Williamsburg, Va., 21-24 Apr. 1991.
[8] R. Cypher and E. Leu, "Eficient Race Detection for Message-Passing Programs With Nonblocking Sends and Receives," *Proc. IEEE Symp. Parallel and Distributed Processing* (*SPDP*), pp. 534-541.
[9] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam, *PVM: Parallel Virtual Machine*. Cambridge, Mass.: MIT Press, 1994.
[10] A.P. Goldberg, A. Gopal, A. Lowry, and R. Strom, "Restoring Consistent Global State of Distributed Computations," *Proc. ACM/ONR Workshop Parallel and Distributed Debugging*, pp. 144-154, Santa Cruz, Calif., 20-21 May 1991.
[11] J.K. Hollingsworth and B.P. Miller, "Parallel Program Performance Metrics: A Comparison and Validation," *Proc. Supercomputing 1992*, pp. 4-13, Minneapolis, Minn., Nov. 1992.
[12] J.K. Hollingsworth and B.P. Miller, "Using Cost to Control Instrumentation Overhead," *Theoretical Computer Science*, pp. 241-258, Apr. 1998.
[13] R. Hood, K. Kennedy, and J. Mellor-Chrummey, "Parallel Program Debugging With On-the-Fly Anomaly Detection," *Proc. Supercomputing 1990*, pp. 78-81, New York, Nov. 1990.
[14] L. Lamport, "Time, Clocks, and the Ordering of Events in a Distributed System," *Comm. ACM*, vol. 21, no. 7, pp. 558-564, 1978.
[15] A.D. Malony, "Event-Based Performance Perturbation: A Case Study," *Proc. 1991 ACM SIGPLAN Symp. Principals and Practice of Parallel Programming*, pp. 201-212, Williamsburg, Va., 21-24 Apr. 1991.
[16] M. Martonosi, A. Gupta, and T. Anderson, "MemSpy: Analyzing Memory System Bottlenecks in Programs," *Proc. 1992 SIGMETRICS Conf. Measurement and Modeling of Computer Systems*, pp. 1-12, Newport, R.I., 1-5 June 1992.
[17] T. Mathisen, "Pentium Secrets," *Byte*, vol. 19, no. 7, pp. 191-192, 1994.
[18] B.P. Miller, M.D. Callaghan, J.M. Cargille, J.K. Hollingsworth, R.B. Irvin, K.L. Karavanic, K. Kunchithapadam, and T. Newhall, "The Paradyn Parallel Performance Measurement Tools," *Computer*, vol. 28, no. 11, pp. 37-46, 1995.
[19] B.P. Miller and J.-D. Choi, "A Mechanism for Efficient De-bugging of Parallel Programs," *Proc. SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging*, pp. 141-150, Madison, Wis., 5-6 May 1988.
[20] B.P. Miller, M. Clark, J. Hollingsworth, S. Kierstead, S.-S. Lim, and T. Torzewski, "IPS-2: The Second Generation of a Parallel Program Measurement System," *IEEE Trans. Parallel and Distributed Systems*, vol. 1, no. 2, pp. 206-217, 1990.
[21] R.H.B. Netzer and B.P. Miller, "What Are Race Conditions? Some Issues of Formalizations," *ACM Letters on Programming Languages and Systems*, vol. 1, no. 1, pp. 74-88, 1991.
[22] R.H.B. Netzer and J. Xu, "Adaptive Message Logging for Incremental Replay of Message-Passing Programs," *Proc. Supercomputing 1993*, pp. 840-849, Portland, Ore., 1993.
[23] D.A. Reed, R.A. Aydt, R.J. Noe, P.C. Roth, K.A. Shields, B.W. Schwartz, and L.F. Tavera, *Scalable Performance Analysis: The Pablo Performance Analysis Environment, in Scalable Parallel Libraries Conference*, A. Skjellum, ed. Los Alamitos, Calif.: IEEE CS Press, 1993.
[24] R. Title, "Connection Machine Debugging and Performance Analysis: Present and Future," *ACM/ONR Workshop Parallel and Distributed Debugging*, pp. 272-275, Santa Cruz, Calif., 20-21 May 1991.
[25] D.J. Webb, personal communication, 1996.
[26] S.C. Woo, M. Ohara, E. Torrie, J.P. Singh, and A. Gupta, "The SPLASH-2 Programs: Characterization and Methodological Considerations," *Proc. 22nd Ann. Int'l Symp. Computer Architecture*, pp. 24-37, 1995.
[27] C.-Q. Yang and B.P. Miller, "Critical Path Analysis for the Execution of Parallel and Distributed Programs," *Proc. Eighth Int'l Conf. Distributed Computing Systems*, pp. 366-375, San Jose, Calif., June 1988.
[28] M. Zagha, B. Larson, S. Turner, and M. Itzkowitz, "Performance Analysis Using the MIPS R10000 Performance Counters," *Proc. Supercomputing '96*, Pittsburgh, Pa., Nov. 1996.

Jeffrey K. Hollingsworth received his PhD and MS degrees in computer science from the University of Wisconsin in 1994 and 1990, respectively. He earned his BS in electrical engineering from the University of California at Berkeley in 1988. Currently, he is an assistant professor in the Computer Science Department at the University of Maryland, College Park, and is affiliated with the Department of Electrical Engineering and the University of Maryland Institute for Advanced Computer Studies. His research interests include performance measurement tools for parallel computing, enabling infrastructure for high performance distributed computing, and computer networks. He received a U.S. National Science Foundation CAREER Award in 1997. Dr. Hollingsworth is a member of the IEEE Computer Society and the ACM.