An Online Computation of Critical Path Profiling

Jeffrey K. Hollingsworth†
Computer Science Department
University of Maryland
College Park, MD 20742
hollings@cs.umd.edu

Abstract

In this paper we introduce a runtime, non-trace based algorithm to compute the critical path profile of the execution of a message passing parallel program. 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.

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 proved useful is Critical Path Profiling[16]. 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[7]. In this paper we introduce a runtime, non-trace 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[12], we described an off-line (post-mortem) approach to computing the critical path profile that required recording all inter-process, procedure entry, and procedure exit events during execution. Once the program had finished, a graph containing all recorded events is constructed. The space required to compute Critical Path Profiling in this off-line manner is O(e) where e is the number of events. The time required is also O(e). Unfortunately for large, long running programs, it is not always feasible to log the events necessary to compute the critical path profile nor to explicitly build this graph.

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 a subset of e consisting of inter-process 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[11]. 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. Finally, we present results from running an initial implementation of our algorithm using several PVM[6], based parallel programs. Initial results indicate that our online critical path algorithm can profile up to eight procedures with a 3-10% slow down of the application program.

2. 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 review how to compute critical path profiling in an off-line environment and then introduce a new, efficient online algorithm.

2.1 Off-line Algorithm and its Limitations

Before we describe the off-line algorithm, we define a few relevant terms:

† This work was supported in part by NIST CRA award 70-NANB-5H0055 and a University of Maryland GRB award.

To Appear: SPDT96: 1st ACM SIGMETRICS Symposium on Parallel and Distributed Tools, May 22-23, 1996, Philadelphia, PA, pp. 11-20.

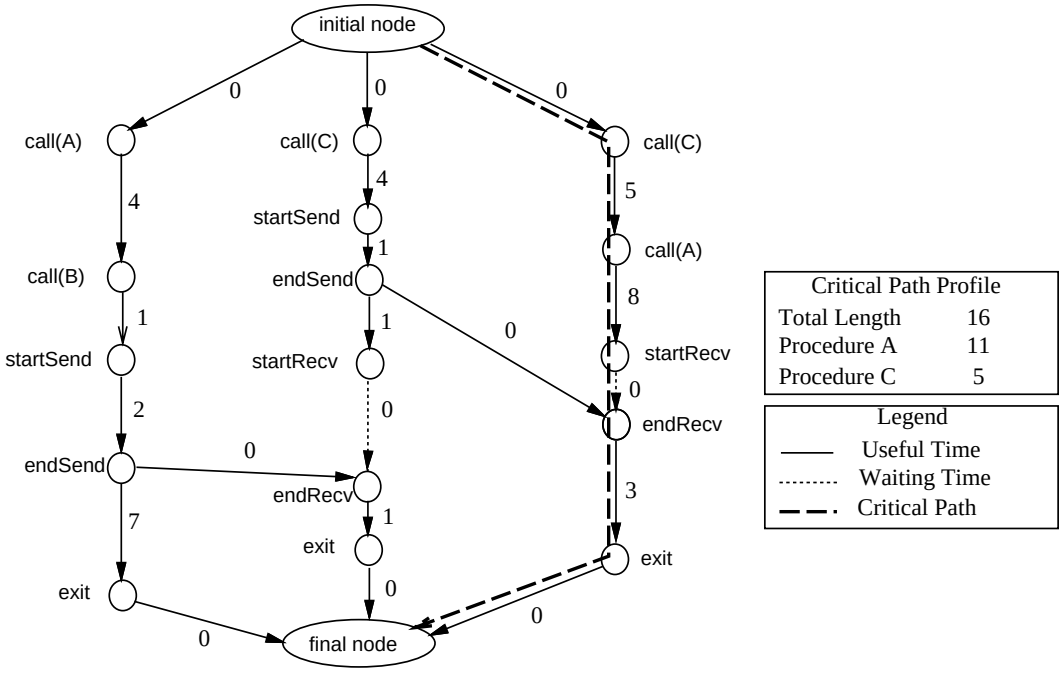

Figure 1: Calculating Critical Path

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 ore 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. Figure 1 shows a simple PAG for a parallel program with three processes.

The critical path of a parallel program is the longest CPU time weighted 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 PAG is a directed acyclic graph and none of the arcs are negative, a variation on the distributed shortest path algorithm described by Chandy and Misra in [4] can be used for this calculation. This algorithm passes messages along the arcs of the graph. Each message contains the value of the longest path to the current node. At split nodes (nodes with one inbound arc and two outbound arcs), the message is duplicated and sent on each of the outbound arcs. At merge nodes, those with two inbound arcs and one outbound arc, only the longest path is propagated. The first phase of the algorithm terminates when the last node in the graph has received messages on each of its inbound arcs. Once the path is found, a second (backwards) pass is made though the graph. This pass traverses the critical path and

[1] This definition could easily be extended to include other synchronization or communication events such as locks and barriers.

accumulates the time spent by each procedure on the path. The pseudo code for the algorithm is shown in Figure 2.

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 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.

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 assigns a different ordering and importance to the top ten procedures, the procedures generally remain the same[7]. If top procedures are not the same, this fact can be detected since their cumulative share of the critical path length will be small. In this case, the programmer can select a different set of procedures and compute the critical path share for them.

```
1  struct node {    // one record per PAG node (program event)
2       longest     // longest path to this node
3       cpPred      // predecessor node along longest path
4       inArcs      // count of inbound arcs processed
5       totalInArcs // total number of inbound arcs (set when graph is built)
6       outArc[2]   // pointer to successor nodes (set when graph it built)
7       activeFunc  // function that is active when event occurs
8  };

9  // First Pass: forward along all arcs of the PAG
10 while (not empty queue) {
11     dequeue(node, fromNode, length);
12     if (length > node.longest) {
13           node.longest = length;
14           node.cpPred = fromNode;
15     }
16     node.inArcs++;
17     if (node.inArcs == node.totalInArcs) {
18        for each out arc {
19           enqueue(node.outArc[i], node,
20             node.length + node.ourArc[i].length);
21     }
22 }

23 // second pass (from end of PAG to start along the critical path)
24 pred = final_node;
25 node = final_node.cpPred;
26 while (node != NULL) {
27    cpProfile[pred.activeFunction] += node.longest - pred.longest;
28    node = pred;
29    pred = pred.cpPred;
30 }
```

Figure 2: Algorithm to compute the Critical Path profile from a PAG.

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 can use a variation of the distributed algorithm used in pass one of the off-line version of the algorithm. Rather than recording the significant events and then building the full program activity graph, we use the normal flow of messages in the application to traverse the graph 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 Figure 3). 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

```
1   struct {                // one per process
2       longest;            // longest path in this process.
3       funcShare;          // function's share of the longest path.
4       funcActive;         // is selected function active in 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.funcLonest += now - process.funcLastTime;
15          process.funcLastTime = now;
16      }
17      send(toHost, process.longest, process.funcLongest);

18  Recv(fromHost):
19       now = CPUTime();
20       process.longest += now - process.lastTime.
21      recv(fromHost, rmtLongest, rmtFuncShare);
22      if (rmtLongest > process.longest) {
23          process.longest = rmtLongest;
24           process.funcShare = rmtFuncShare;
25           process.lastTime = now;
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      process.funcActive = 1;
38      process.funcLastTime = CPUTime();

39  Exit:     // from selected procedure or function
40      process.funcActive = 0;
41      process.funcShare +=
42            currentProcessTime () - process.funcLastTime;
```

Figure 3: Algorithm to compute the Critical Path of a function on-the-fly.

in the process, and the time of the last recorded event when the selected procedure was active.

Four events in the application program require instrumentation: message send and receive, and calls to and returns from the desired procedure. Pseudo-code for the algorithm is shown in Figure 3. 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 Figure 3, lines 1-7) and the piggy-backed data for messages that have been sent but not received (stored in message buffers).

For clarity, we describe the inclusive critical path for a single selected procedure (i.e., cumulative time spent in the procedure and procedures it calls). To compute the non-inclusive 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 38-41 of Figure 3. After the subroutine returns, we use the instrumentation shown in lines 34-37 of Figure 3.

2.3 Critical Path of Partial Program Execution.

The algorithm we described 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. For the off-line version of critical path, 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.

In the on-line case, we don't explicitly build the PAG, and so 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 Figure 4.

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:

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

5  Sample:    // 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     }
```

Figure 4: Sampling the Critical Path during program execution.

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[9]. 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 a set of events that contains the last event in each 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 & j are integers between 1 and the number of events in process p.

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 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 currently 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 non-communicating 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 the 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 critical path data arrives at a process that is not computing the critical path.

(2) A message with critical path data arrives at a process that is already computing the critical path.

(3) A message with critical path data arrives at a process that is not computing the critical path.

(4) A message without critical path 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

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

Figure 5: Computing Logical Zeroing.

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 [3]. 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 re-ordering of events is a feasible execution of the program.

3. 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 sub-critical 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[7] 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 piggy-back 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. Figure 5 shows the pseudo code for the computation of logical zeroing at a "merge" (receive) node. The changes are at lines 5-6; before of 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.

4. 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 piggy-backing 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 piggy-backing 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 piggy-backing 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.

To quantify the overhead of piggy-backing 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 "piggy-backing" 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 gauge the impact of our instrumentation on these 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. Varia-

Number of CP Items	~75% Computation		~60% Computation	
	Wall Time	Overhead	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%

Figure 6: Overhead Required to Compute Per Procedure Critical Path.

Number of CP Items	5 msgs/sec		50 msgs/sec		150 msgs/sec	
	Time	Percent	Time	Percent	Time	Percent
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

Figure 7: Comparison of overhead for different message rates.

tion between runs was less than 1 percent.

The table in Figure 6 shows two versions of the program run on the SPARC/Ethernet configuration. The first version computed for 75% of its execution time and spent the remaining 25% of the time sending and receiving message. The second version spent 60% of its time in computation and 40% in message passing. Each program sends the same size and number of messages, but we varied the "computation" component between the two programs.

The time required to send an empty piggy-back message is shown in the third row. For the 75% computation case, sending empty messages added a 1.9% overhead to the application and for the 60% computation case it was 3.4%. This provides an indication of the overhead required to simply enable the piggy-back 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 we calculated the critical path for from 1 to 32 procedures. In addition, we report the time required to run the un-instrumented 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 Figure 7 is for the IBM SP-2 configuration. In this case, we held the computation to communication ratio fixed at 75% computation, and varied the message passing frequency and message size. We used a message passing rate of approximately 5, 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 (5, 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 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.

To evaluate the effectiveness of Critical Path computation on applications, we ran our algorithm on a PVM version of Integer Sort, one of the NAS benchmarks. The program was run on a network of two Sun Sparcstation 5's connected by an Ethernet. The results are shown in Figure 9. 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.

Procedure	CP	% CP	CPU	% CPU
nas_is_ben	12.4	56.4	54.8	74.1
create_seq	9.2	42.0	9.2	12.4
do_rank	0.4	1.6	9.2	12.5

Figure 9: NAS IS Benchmark Results.

This example shows the benefit of the additional information provided by Critical Path compared to CPU time profiling. Although create_seq is only 12% of the total execution time of the overall program, it was responsible for

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

Figure 8: Metric values for Ocean Application.

42% of the length of the critical path. The reason for this is that the routine is completely sequential. Other sequential metrics would not have easily identified the importance of creat_seq.

We also measured a PVM application running on an IBM SP-2. For this program, we measured an implementation of the GFDL Modular Ocean Model[2] developed by Webb[15]. The results of computing the CPU time profile, Critical path Profile, and Critical Path Zeroing are shown in Figure 8.

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 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 sub-critical paths in the program limit the improvement possible by just tuning this one routine.

5. Related Work

Implicitly walking a PAG by attaching instrumentation messages onto applications messages has been used for online detection of race conditions in parallel programs[5, 8]. Similar instrumentation has also been used to reduce the number of events that must be logged for program replay[13].

Many metrics and tools have been developed to quantify the performance of parallel programs. The Paradyn[11] and Pablo[14] tools provide a wealth of performance metrics. Other metrics focus on specific sources of bottlenecks in parallel programs[1, 10].

6. 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. Finally, we presented a brief case study that demonstrated the usefulness of critical path for a PVM message passing program.

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 proceeded 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 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. At then end of the program[2], 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, it 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

References

1. T. E. Anderson and E. D. Lazowska, "Quartz: A Tool for Tuning Parallel Program Performance", *1990 SIGMETRICS Conference on Measurement and Modeling of Computer Systems*. May 1990, Boston, pp. 115-125.
2. K. Bryan, "A numerical method for the circulation of the World Ocean", *Journal of Computational Physics*, 1969. **4**(1), pp. 347-.
3. K. M. Chandy and L. Lamport, "Distributed Snapshots: Determining Global States of Distributed Systems", *ACM TOCS*, Feb 1985. **3**(1), pp. 63-75.
4. K. M. Chandy and A. J. Misra, "Distributed computation on graphs: Shortest path algorithms", *CACM*, Nov 1982. **25**(11), pp. 833-837.
5. R. Cypher and E. Leu, "Eficient Race Detection for Message-Passing Programs with Nonblocking Sends and Receives", *IEEE Symposium on Parallel and Distributed Processing*, pp. 534-541.
6. J. Dongarra, A. Geist, R. Manchek, and V. S. Sunderam, "Integrated PVM framework supports heterogeneous network computing", *Computers in Physics*, March-April 1993. **7**(2), pp. 166-174.
7. J. K. Hollingsworth and B. P. Miller, "Parallel Program Performance Metrics: A Comparison and Validation", *Supercomputing 1992*. Nov. 1992, Minneapolis, MN, pp. 4-13.
8. R. Hood, K. Kennedy, and J. Mellor-Chrummey, "Parallel Program Debugging with On-the-fly Anomaly Detection", *Supercomputing 1990*. Nov 1990, New York, pp. 78-81.
9. L. Lamport, "Time, Clocks, and the Ordering of Events in a Distributed System", *CACM*, July 1978. **21**(7), pp. 558-564.
10. M. Martonosi, A. Gupta, and T. Anderson, "MemSpy: Analyzing Memory System Bottlenecks in Programs", *1992 SIGMETRICS Conference on Measurement and Modeling of Computer Systems*. June 1-5, 1992, Newport, Rhode Island, pp. 1-12.
11. 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", *IEEE Computer*, Nov. 1995. **28**(11), pp. 37-46.
12. 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 Transactions on Parallel and Distributed Systems*, April 1990. **1**(2), pp. 206-217.
13. R. H. B. Netzer and J. Xu, "Adaptive Message Logging for Incremental Replay of Message-Passing Programs", *Supercomputing 1993*. 1993, Portland, OR, pp. 840-849.
14. 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, Editor. 1993, IEEE Computer Society.
15. D. J. Webb, Personal Communication.
16. C.-Q. Yang and B. P. Miller, "Critical Path Analysis for the Execution of Parallel and Distributed Programs", *8th Int'l Conf. on Distributed Computing Systems*. June 1988, San Jose, Calif., pp. 366-375.

[2] We will show how to permit this computation do be done in a single execution of the program at the end of this Appendix.