Slack: A New Performance Metric for Parallel Programs

Jeffrey K. Hollingsworth
hollings@cs.umd.edu

Computer Science Department
University of Maryland
College Park, MD 20902

Barton P. Miller
bart@cs.wisc.edu

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

Abstract

Critical Path Profiling is a technique that provides guidance to help programmers try to improve the running time of their program. However, Critical Path Profiling provides only an upper bound estimate of the improvement possible in a parallel program execution. In this paper, we present a new metric, called *Slack*, to complement Critical Path and provide additional information to parallel programmers about the potential impact of making improvements along the critical path.

1. Introduction

Performance debugging parallel programs is much harder than for sequential programs. For sequential programs, tools such as gprof[2] provide sufficient information to guide the programmer to the bottlenecks in their program. Unlike sequential programs, in parallel programs improving the procedure that consumes the largest amount of CPU time will not necessarily improve the program. More sophisticated metrics are required. Metrics provided by techniques such as Critical Path Profiling[5] help to guide parallel programmers to find performance bottlenecks. Critical Path Profiling computes a value for each procedure in the program indicating how much time is spent on the critical path for that procedure. However the values provided by Critical Path Profiling are an upper bound on the improvement because when a procedure along the critical path is improved, it can cause the critical path to shift from that procedure to another slightly sub-critical path. We have developed a new metric called *Slack* to get a better idea about how much improving a single procedure will improve the overall execution time of a program.

The rest of this paper is divided into three sections: an overview of Critical Path, a description of the Slack algorithm, and a short case study comparing the results of Critical Path and Slack for two applications. A complete case study comparing Critical Path, Slack, and several other parallel performance metrics can be found in [3].

2. Critical Path

Critical Path Profiling is based on identifying the path through the program's execution that consumed the most time. To calculate a program's Critical Path, we build a graph of the program's execution, called a Program Activity Graph (PAG). This graph consists of significant events in the execution of a parallel program (e.g., inter-process communication, and procedure calls). Figure 1 shows a sample PAG for a three process program. The solid arcs represent useful CPU time intervals. The dashed lines indicate non-useful CPU time from activities such as spinning on a lock. The numbers next to each arc represent the time between the two events. The Critical Path of a parallel program is the longest CPU time weighted path through the PAG. Non-productive CPU time, such as spinning on a lock, is assigned a weight of zero.

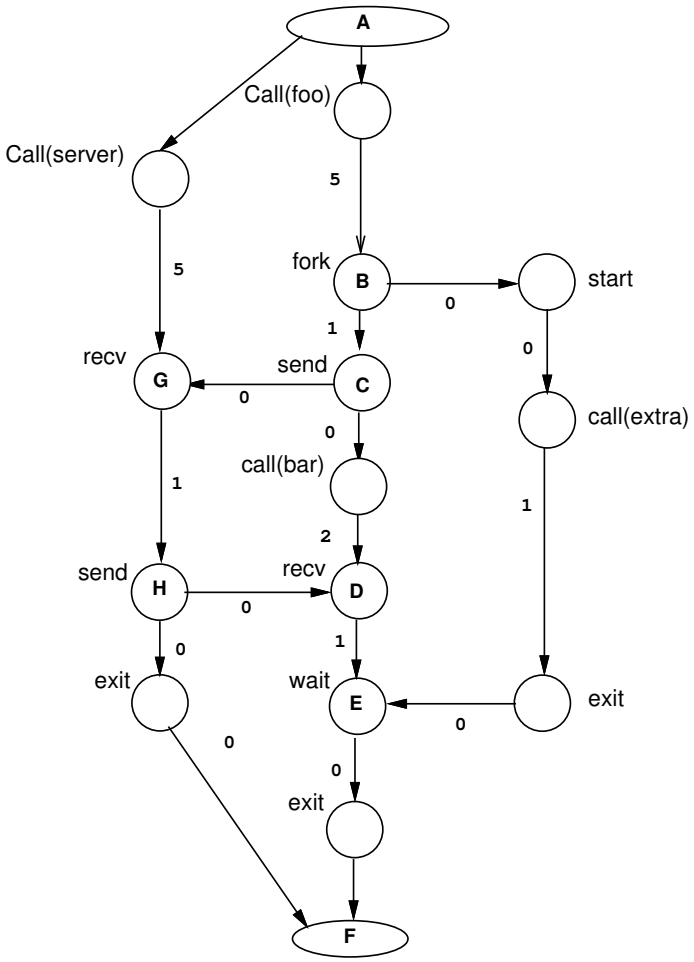

Figure 1. A Sample Program Activity Graph.

Nodes represent interesting events in the application (e.g., procedure calls and message passing). The arcs are labeled with the elapsed time between events.

To find the longest path through the graph, we use a variation on the parallel shortest path algorithm described by Chandy and Misra in [1]. Each process in the original program is assigned to a processor for the computation of the longest path. The 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 (nodes with two inbound arcs and one outbound one), only the longest path is propagated, and the arc responsible for this longest path is recorded at the node. The first phase of the algorithm terminates when the last node in the graph has received messages on each of its inbound arcs.

After the longest path has been found, a second pass is made along the Critical Path. To purpose of this path is to record the time spent in each procedure along the Critical Path. The path is discovered by walking backwards through the PAG. At each node along the path, we traverse the arc responsible for the longest path to that node. For each procedure along an arc, we add the amount of time that the procedure executed to that procedure's total critical path time. This process continues until we reach the root of the graph.

3. Computing Slack

To compute the Slack Profile, we use the same PAG that is used to calculate the Critical Path. The algorithm for computing the Slack Profile is divided into three phases. The first phase computes the Critical Path through the PAG. This path computation is done with the same algorithm used to compute the Critical Path Profile (if the Critical Path profile has already been computed, this step is omitted). In the second phase, potential slack along the Critical Path is identified. The third phase computes the slack profile for each procedure along the Critical Path.

The second phase consists of a forward pass similar to the Critical Path's forward pass. Figure 2 shows the messages sent by the forward pass of the slack algorithm for the DAG shown in Figure 1. Particular attention is paid to nodes along the Critical Path that represent divergences from the Critical Path (a *split* node) or the convergence of a non-critical path with a node on the Critical Path (a *merge* node). We define the set of arcs from a split node along the Critical Path to a merge node to be a *slack segment*. The difference between the length of a slack segment, and the Critical Path between these nodes is the potential slack for that segment. Messages sent from split nodes are tagged with the node at which they leave the Critical Path (for example the F(A,0) message sent from node A to node G in Figure 2). When a node on the Critical Path receives a message from an arc not on the Critical Path (for example node F in Figure 2), a message is sent from this node back to the split node. This message contains the name of the merge node and the potential slack between the split node and the merge node.

We now consider the case of split and merge nodes not on the Critical Path. At split nodes, a slack message is sent along each outbound arc after the appropriate arc weight has been added. At merge nodes, a slightly more

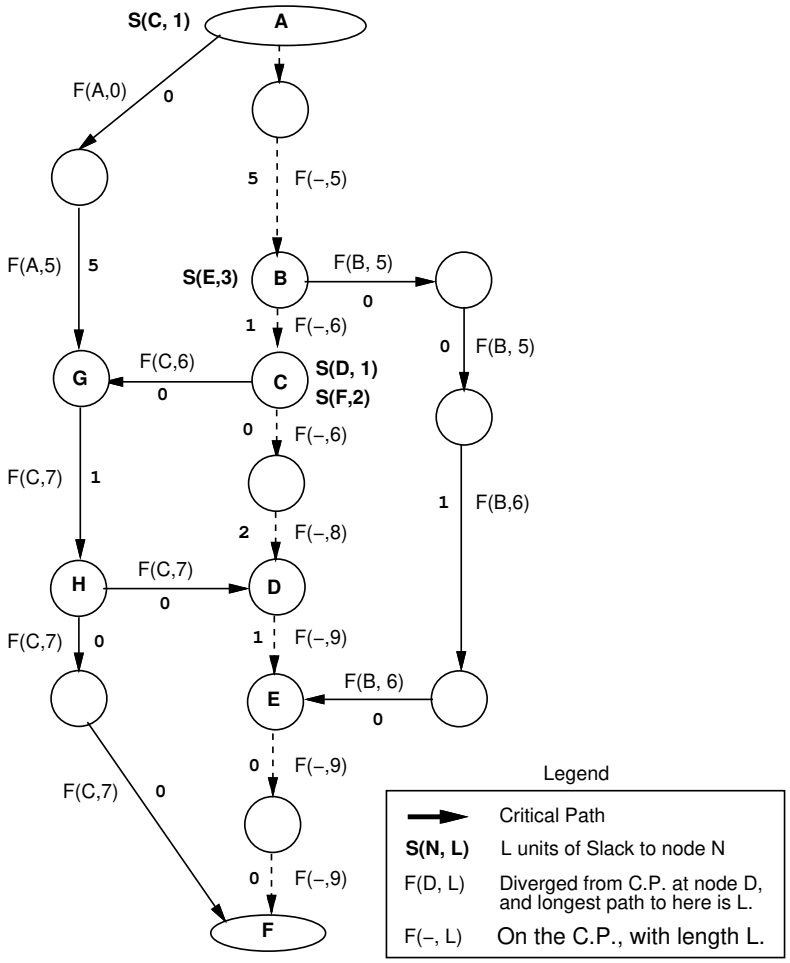

Figure 2. Slack Calculation

A sample slack computation. There are four slack segments in this graph. The actual slack value for each arc along the Critical path is the minimum of the slack values for each of the slack segments active for the arc.

complex operation must be performed. First, the value of the longer path is propagated out the outbound arc. Second, a slack message is sent to the first (earlier on the Critical Path) split node indicating the identity of the second split node and the difference between the lengths of the paths at the merge node. Node G in Figure 2 is an example of this kind of node. From this node two messages are sent, one message, F(C,2), is sent to node H to propagate the longest path. The second message, **S(C,1)**, is sent to node A, indicating that a slack segment exists from node A to node C and that it is 1 unit shorter than the Critical Path between these nodes. This phase terminates when there are no more messages to send.

After the second phase, the graph has been annotated with the potential slack opportunities (shown as **S(N, L)** in Figure 2). A third pass is made though the graph to identify the exact amount of slack that is available. Two facts complicate this pass. First, a single function might be called several times during a slack segment and the actual slack available for the function during the segment is limited to the total slack of the segment. Second, slack

segments can overlap (i.e., nodes can be part of two difference slack segments).

Similar to the second pass in Critical Path, the third phase of the slack algorithm is implemented as a traversal of the Critical Path. However, unlike Critical Path, the slack algorithm starts at the root node and works forward to the last node. This pass requires an table for each procedure along the Critical Path. The table holds the available slack for each active slack segment. When the start of a slack segment (i.e., a split node) is encountered, the potential slack for that segment is inserted into the list of active segments for each procedure. When the end of a segment (i.e., the merge node) is reached, the segment is removed from each procedure's list. Figure 3 shows an example of this data structure. When an a procedure call is encountered, its total slack is incremented by the minimum of the available slack for each of the active slack segments. In addition, each of the active slack segments for that procedure is decremented by this value. This pass terminates when the final node in the graph is reached. The value of the slack metric for a procedure is the total slack that has been accumulated for that procedure.

Procedure	Total Slack	Slack Segments
foo	1	(A-C,1)
bar	2	(C-D,1) (C-F,2) (B-E,0)

Figure 3. Computing the Slack for each procedure along the Critical Path.

4. Discussion and Examples

To get an idea how well the Slack metric performs in practice, we added an implementation of Slack Profiling to the IPS-2 parallel program measurement system[4]. We then used IPS-2 to measure the performance of two programs. The first was a client-server program running on a network of workstations. The second is a numerical application running on a multi-processor.

The first program is a simple client-server program running on two workstations. The program consists of a compute process which calculates data that is consumed by the other process. The Critical Path and Slack profiles of this application appears in Figure 4. For all of the procedures shown (the top 4 by length of critical path), the value of the Slack and Critical Path metrics is the same. Since Critical Path is a upper bound on the amount of improvement possible and Slack is a lower bound, this table indicates that any provement in the top four procedures would have a corresponding improvement in the running time of the program.

Procedure	Critical Path	Slack
calc2	13.92	13.92
calc1	8.27	8.27
Atan2	6.61	6.61
sockstuff	0.33	0.33

Figure 3. Critical Path and Slack for a client server application.

We now consider a second application written in the Single Program Multiple Data (SPMD) style. This application is an implementation of the Sparse Choleski factorization kernel. The program was run on a four processor Sequent Symmetry, which is a shared memory multi-processor. Figure 5 shows the values of both Critical Path Profiling and Slack Profiling for the 7 most important procedures in the program. In addition, the fourth column (Logical Zeroing) shows the reduction in the length of the critical path if the time spent in each procedure was set to zero and the new critical path computed. Since Logical Zeroing computes a new critical path profile based on updating the weights of the nodes in the PAG, it provides an indication of the improvement possible if the selected procedure is tuned. The value for Logical Zeroing will be between Critical Path's upper bound and Slack's lower bound estimates of the importance of improving a procedure. Therefore, we can compare the values for Logical Zeroing, Critical Path Profiling and Slack Profiling to gauge the effectiveness of the latter two metrics. For this application, Slack provided little useful information because of the combination of two factors. First, the program was written in data parallel style (i.e., the same procedures execute on each processor). Second, the work is well balanced among the processors. As a result, important procedures appear not only on the critical path, but also on each of the sub-critical paths. Since Slack does not consider the impact improving of a procedure on a sub-critical path, the values for this metric are too low for this particular program.

Procedure	Critical Path	Slack	Logical Zeroing
printCmat	14.00	1.87	14.00
Tcol	10.63	0.76	9.28
sElem	7.49	1.00	7.49
cmod	6.99	0.11	5.54
fillCmp	5.69	0.76	5.69
next	5.99	0.62	5.99
mycalloc	4.19	0.56	4.12

Figure 5. Summary of metric values for sparse Choleski.

5. Conclusion

We have described the slack metric that provides a new way to estimate the potential improvement in execution time of a parallel program that caused by improving procedures along the Critical Path. It is intended to complement Critical Path Profiling by proving a lower bound on the impact of improving each of the procedures along the critical path. However, its practical uses are limited because it often provides too conservative of an estimate of the improvement possible. This is especially true for SPMD style programs were the same procedure often appear on both the Critical Path and several slightly sub-critical paths.

References

1. K. M. Chandy and J. Misra, "Distributed computation on graphs: Shortest path algorithms", *CACM 25*(Nov. 1982), pp. 833-837.
2. S. L. Graham, P. B. Kessler and M. K. McKusick, "gprof: a Call Graph Execution Profiler", *SIGPLAN '82 Symposium on Compiler Construction*, Boston, June 1982, pp. 120-126.
3. J. K. Hollingsworth and B. P. Miller, "Parallel Program Performance Metrics: A Comparison and Validation", *Supercomputing 1992*, Minneapolis, MN, November 1992, pp. 4-13.
4. B. P. Miller, M. Clark, J. Hollingsworth, S. Kierstead, S. Lim and T. Torzewski, "IPS-2: The Second Generation of a Parallel Program Measurement System", *IEEE Transactions on Parallel and Distributed Systems 1*, 2 (April 1990), pp. 206-217.
5. C. Yang and B. P. Miller, "Critical Path Analysis for the Execution of Parallel and Distributed Programs", *8th Int'l Conf. on Distributed Computing Systems*, San Jose, Calif., June 1988, pp. 366-375.