LBF: A Performance Metric for Program Reorganization*

Hyeonsang Eom Jeffrey K. Hollingsworth
Computer Science Department
University of Maryland
College Park, MD 20742 USA
{hseom,hollings}@cs.umd.edu

Abstract

We introduce a new performance metric, called Load Balancing Factor (LBF), to assist programmers with evaluating different tuning alternatives. The LBF metric differs from traditional performance metrics since it is intended to measure the performance implications of a specific tuning alternative rather than quantifying where time is spent in the current version of the program. A second unique aspect of the metric is that it provides guidance about moving work within a distributed or parallel program rather than reducing it. A variation of the LBF metric can also be used to predict the performance impact of changing the underlying network. The LBF metric can be computed incrementally and online during the execution of the program to be tuned. We also present a case study that shows that our metric can predict the actual performance gains accurately for a test suite of six programs.

1. Introduction

To successfully tune a distributed or parallel program, the cause of a performance bottleneck must be identified, a solution proposed and implemented. Finally, the tuned program must be re-measured to verify the problem was corrected. Each step in the process is a difficult and time consuming task. Performance debugging tools exist to help the programmer with these tasks. However, the majority of the work on performance tools has concentrated on bottleneck identification. While this is an important problem, it is just the first step. In this paper, we concentrate on providing guidance with the next step: choosing between alternative tuning strategies.

Once the source of a problem has been located, a proposed change must be identified. Frequently, there are several different strategies to try such as changing data decomposition, changing the assignment of processes to processors, or even changing the computation or communication resources. However, each of these options might require significant effort to change the program, debug it, and then re-execute it. Performance tools need to help the programmer to evaluate the potential impact of different tuning options before changing a single line of code.

There are several ways for a tool to provide information about the potential benefit of tuning options. First, the tool could use a static prediction of the performance of the changed program based on analysis of the source code. However, such an approach suffers from the problem that the prediction ignores dynamic (execution) data that can provide important information about a program's actual behavior. A second approach is to instrument a program to measure its dynamic behavior, and then use this data to make off-line predictions about tuning alternatives. This approach could require a significant amount of data to be collected. Instead, we use a third approach that combines the execution of the current version of the program, online measurements of its execution, and algorithms to predict the impact of different tuning options. The idea is to combine the execution of the original program with a simulation of the proposed changes to the program. This technique has been successfully used to simulate changes in computer architectures[20]. Combining direct execution of the majority of the system with a simulation of the changed parts, permits faster execution than simulating the entire program's execution.

There is a tradeoff between efficiency and accuracy when predicting the change in execution time due to tuning. Consider, for example, trying to assess the impact of tuning a single procedure's performance. At one extreme, we could generate very accurate results by performing a detailed execution-driven simulation of the proposed modifications to the original program. Each instruction could be simulated and an appropriate time for that instruction recorded. To simulate the impact of tuning,

* This work was supported in part by NSF award ASC-9703212, DOE Grant DE-FG02-93ER25176, and NIST CRA award 70-NANB-5H0055.

whenever the tuned procedure is executed, simulation time would advance only by the "tuned" time of the procedure. This would produce a very accurate prediction of the improvement possible by tuning the target procedure. However, the speed of this simulation would likely be too slow to provide timely feedback to the programmer. At the other extreme, we could simply profile the target procedure and predict that any time removed from that procedure would directly reduce the execution time of the program. This produces a simple value to compute, but the accuracy suffers due to the fact that the improvement in execution time of a program does not necessarily result in a corresponding improvement in the program's execution time due to communication and work done on other processors. Our goal is to combine reasonable performance and accuracy to provide useful feedback to programmers.

Unlike sequential programs, in a distributed or parallel program, it is possible to tune where a computation is performed, in addition to how it is performed. For example, a process in a producer/consumer pipeline may exhibit *data affinity*. A consumer process has data affinity if it consumes a large amount of data and its performance is improved by co-location with its data source. Data can be either static (a disk file), or dynamic (a producer process). Due to either load balancing or data affinity, it might be more productive to move part of the computation from one processor to another rather than reducing its execution time. In this paper, we concentrate on providing answers to "what-if" questions involving changing where computation is performed rather than changing how the result is computed. We present a new metric called Load Balancing Factor, LBF, that provides programmers with feedback about the performance implications of moving computation between processors. In addition, we present a variant of LBF called Networking Factor (NF) that predicts the performance gains due to changing the underlying communication network.

In this paper, we introduce the LBF and NF metrics and evaluate them for several parallel programs. Section 2 introduces the LBF metric, describes an implementation of the metric, and quantifies its accuracy at predicting changes. Section 3 describes using NF, a network variant of LBF to predict the change in application execution time due to changing the performance of the networking infrastructure. Section 4 describes related work. Finally, Section 5 summaries our work and outlines future directions for this research.

2. Load Balancing Factor (LBF)

Load Balancing Factor (LBF) addresses the problem of assessing the impact of process migration by predicting the impact of changing the assignment of processes to processors in a distributed or parallel execution environment. Our goal is to compute the potential improvement in execution time if we change the placement. Our technique can also be used to predict the performance of a distributed or parallel program when it is executed on a larger number of nodes.

To assess the potential improvement, we predict the execution time of a program with a virtual placement, during an execution on a current one. Our approach is to instrument application processes to forward data about each message passing event to a central monitoring station that simulates the execution of these events under the target configuration.

Since there could be multiple processes contending for a CPU on a node in a target placement, we must select a realistic policy to schedule processes for an accurate prediction. We assume a fair round-robin scheduling policy, where the OS schedules each non-waiting process onto a processor for a fixed quantum of time, and then switches to the next non-waiting process. To speed the computation of the LBF metric, we do not simulate individual quanta. For each interval of time, every non-blocked process gets an equal share of the processor effectively making the quantum infinitely small.

Before describing our prediction algorithm, we define a few terms used to describe LBF:

Event: an observable operation performed by a process. A process communicates with other processes via messages. Message passing results in send, startRecv, and endRecv events being generated. Message events can be "matched" between processes. For example, a send event in one process matches exactly one endRecv event in another process.[1]

Process Time: a per-process clock that runs when the process is executing on a processor and is not waiting for a message.

Program Activity Graph (PAG): a graph of the events in a single program execution. 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 process time between events or communication time for inter-process arcs. The left half of Figure 1 shows a simple PAG for a parallel program with three processes.

Happen-Before: 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, a send event happens before the corresponding endRecv event. Formally, happen-before is the set of precedence relationships between events implied by Lamport's happened before relationship[12].

Critical Path (CP): the longest process time weighted path through a PAG. For an entire program's execution, the CP

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

represents the execution time of the program as if there were one process per processor.

Process Group: a set of processes that run on a single processor in a predicted (target) configuration.

Group Time: a per-group clock that runs when any process of the group is executing on a processor.

Group Activity Graph (GAG): a graph of the events in a single program execution. Nodes in the graph represent events in the program's execution. Arcs represent the ordering of events within a group or the communication dependencies between groups. Local arcs are labeled with the amount of group time between events. The right side of Figure 1 shows a simple GAG for a parallel program with three processes and two target groups. A GAG is effectively a PAG with all events from a group collected into a single "virtual" process.

Earliest Possible Time (EPT): the earliest time, measured in group time, an event can occur within a target group. EPT is equivalent to process time when there is only one process in a group.

To compute the execution time of a target configuration, we can construct a Group Activity Graph (GAG) and then compute the length of its longest path. For clarity of presentation, we first introduce the process of converting a PAG to a GAG in a postmortem fashion. We then describe the details of our algorithm that builds the GAG online during application execution.

Given a target process grouping for the execution of a program, the GAG is constructed from the corresponding PAG by combining the PAG components for the processes in a group, into a single "process" in the GAG. Each event from the PAG is placed into the GAG in the group time order. The arc between two adjacent events in the same group is labeled with the elapsed group time between them.

Figure 1 illustrates a PAG and the corresponding GAG. The weights of arcs in the GAG include the effect of the target grouping. The Earliest Process Time (EPT) of the startRecv in Group 1 is 2 because Processes 1 and 2 share a processor up to the startRecv. The EPT of the endRecv in Group 1 is 8 because Process 2 must run on the group processor so that the send event precedes the matching endRecv. The EPTs of the startRecv and the endRecv in Group 2 are all 1's, because there is only one process in the group. The endRecvs in the GAG show the two extreme cases of EPT calculation: the EPT of the endRecv in Group 1 is the same as that of the corresponding send event. The EPT of the endRecv in Group 2 is the same as that of the startRecv. The predicted execution time at each message receive is the maximum of the EPT of the endRecv event and the EPT of the send event plus the message time of flight. LBF is only an approximation of the execution time after migration since we ignore memory contention among processes in a target group. A complete description of the algorithm is presented in Section 2.1.

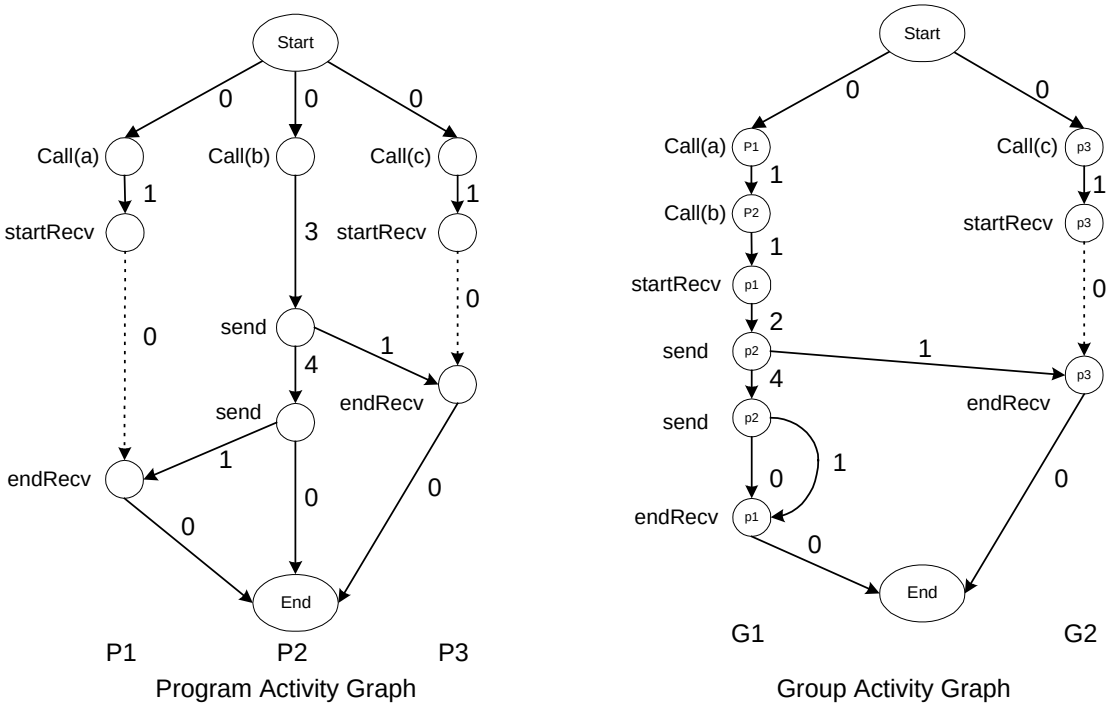

Figure 1: Transforming a PAG into a GAG.

An off-line algorithm to calculate LBF, would build a PAG, convert it to the corresponding GAG, and then compute CP along the longest EPT plus communication time path through the GAG. Since the number of nodes in the PAG is equal to the number of events during the program's execution, explicit graph construction, conversion, and computation would cause intolerable overhead for long-running programs. Instead, we have developed an online algorithm to compute LBF, building a PAG and converting the GAG incrementally. Our algorithm permits us to maintain only the part of the GAG that is currently being processed. To incrementally maintain the GAG, we adapt the on-the-fly topological sort algorithm developed by Kimelman and Zernak[11]. Our algorithm simulates the real execution on a target grouping of processes. To compute the predicted execution time of the target configuration during program execution, we use a variation of our online critical path algorithm[9].

Given a target grouping, we must determine the order of events in the grouping to build the GAG incrementally. Like a topological sort, we must choose the next event to process by selecting events such that all events are processed in the order dictated by the happen-before relationship. Events not ordered by the happen-before relationship are ordered based on round-robin scheduling of a group's processes onto a processor.

In addition to selecting the next event to add to the GAG, we must also assign the correct weights to its arcs. For inter-group arcs, the communication time supplied in the PAG is used. Computing the weight of the arc between local events is more complicated; the weight is equal to the total amount of processing done by each non-blocked application process between the last event added to the GAG for the group, and the current event being processed.

2.1 Algorithm

We now present the details of our algorithm. We describe how to transform a stream of program events arriving from application processes (i.e., a PAG) into a GAG. By calculating the length of the longest path through the resulting GAG, we compute the execution time under the proposed grouping. Events arrive for processing from the application processes, and are maintained until they are inserted into the GAG. When events are no longer needed, they are deleted. While an event is being processed, it is in one of four states:

Queued: an event is queued if it has arrived at the monitoring station, but the event immediately before it in the same process has not yet been reported.

Current: a current event is a candidate for processing. There can be at most one current event per process.

Pending: a pending event is an endRecv that is waiting for the corresponding send event to be processed.

Reported: an event is reported when the processing of the event has been completed and is inserted into the Group Activity Graph (GAG). The DAG data structure for a reported event is freed once both its local and remote successors are reported.

Each event arrives from its application process and is processed by the function `EventArrival` (lines 19-44 of Figure 2). The `EventArrival` procedure inserts the new event into the PAG, the initial state of the event is determined based on the states of its predecessor events. The state of an event is updated in two places: when it arrives and when a predecessor event is reported. An event becomes current when all its predecessors are reported. Since only endRecv events have two predecessors, and events from individual processes arrive in FIFO order, only endRecv events can be marked as pending (waiting for non-local predecessors to be processed).

The event selected for processing is the earliest current event. To select among multiple current events, we use the function `EarliesEventTime` (lines 14-18 of Figure 2). The *Earliest Event Time* for an event is the time of an event if it were to be selected as the current event. If the event selected is a non-blocking event, its procTime is updated to simulate the amount of time it would have executed in the target configuration between the current and previously reported events in the group. For a blocked process, its waitTime is reduced by the total process time used by the runnable events in the group. Next, the waitTime and procTime of the other current and pending events in the group are updated, and the groupTime of the event's group is increased by the total process time consumed. The waitTime field represents the process time consumed by the group since its last event was added into the GAG.

For accurate prediction, it is necessary to integrate communication cost into the computation of the predicted time. Communication cost is due to protocol processing time at the end-points, and the time of flight of the message. Since protocol processing is local to a single process, it is easy to measure directly. However, due to problems with clock synchronization, it is generally impossible to accurately measure the time of flight of a message. As a result of this difficulty, we use a lookup table based on the number of message bytes transferred and whether the message is local (same processor) or remote. The values for this table are determined off-line (prior to application execution) by measuring one half of the round trip times for messages of varying lengths.

```
1.  UpdateState(Event):
2.    IF Event's type is endRecv AND its send event has not been reported
3.      Event.state <- pending
4.    ELSE Event.state <- current
5.    IF Event's type is endRecv AND its send event has been reported AND
6.       Event.remotePred.Cs > Event.localPred.Cr
7.      Event.waitTime += (Event.remotePred.Cs - Event.localPred.Cr)

8.  Report(Event):
9.    add Event into GAG
10.   Event.state <- reported
11.   IF (Event.remoteSuc && Event.remoteSuc.state == pending)
12.     UpdateState(Event.remoteSuc)
13.   IF (Event.localSuc) UpdateState(Event.localSuc)

14. EarliestEventTime(Event):
15.   IF Event's type is endRecv
16.     return Event.waitTime + group(Event).time
17.   ELSE
18.     return Event.procTime * |CNER[2] events| + group(Event).time

19. EventArrival(Event):
20.   insert Event into PAG
21.   IF (there is no unreported event for Event's Process) UpdateState(Event)
22.   ELSE Event.state <- queued
23.   WHILE (Each Process has a current or pending Event)
24.     neEvent <- CNER Event with the smallest EarliestEventTime(Event)
25.     eEvent <- current endRecv Event with smallest EarliestEventTime(Event)
26.     IF (neEvent AND
27.        (no eEvent OR EarliestEventTime(neEvent) < EarliestEventTime(eEvent)))
28.       FOR EACH (current or pending Event in neEvent's Group)
29.         IF (Event.state == pending)
30.           Event.waitTime -= |CNER Events in Event's Group| * neEvent.procTime
31.         ELSE Event.procTime -= neEvent.procTime
32.       group(neEvent).time += |CNER Events in neEvent's Group| * neEvent.procTime
33.       IF (neEvent is a send event)
34.         neEvent.Cs <- group(neEvent).time
35.       ELSE IF (neEvent is a startRecv event)
36.         neEvent.Cr <- group(neEvent).time
37.       Report(neEvent)
38.     ELSE
39.       FOR EACH (current or pending Event in eEvent's Group)
40.         IF (Event.state == pending)
41.           Event.waitTime -= eEvent.waitTime
42.         ELSE Event.procTime -= eEvent.waitTime/ |CNER Events in eEvent's group|
43.       group(eEvent).time += eEvent.waitTime
44.       Report(eEvent)
```

Figure 2: Pseudo Code for LBF.

To report an event, we need to know that no other event that casually preceded it remains unreported. If a process is not generating events (i.e., it does not communicate with other processes) for a long period of time, we can't process any current events in other processes. To prevent this, we use periodic alarms in each application process to create additional keep-alive events. Keep-alive events are treated like normal events and advance the group time of their target group; the difference is that they are discarded rather than being added to the GAG.

2.2 Experimental Validation of LBF Metric

We implemented LBF as an extension to the Paradyn Parallel Performance Measurement Tools[16]. Using Paradyn provided an easy way to implement the algorithm since it already included support for instrumentation of a

[2] CNER (Current Non-End-Receive) events are all current events except endRecv.

running program and periodic sampling callbacks. We tested LBF by running a collection of application programs. The programs consisted of a Synthetic Parallel Application (SPA), a program to solve the Traveling-Salesman Problem (TSP), and a selection of the NAS benchmark programs. The NAS applications are an embarrassingly parallel program (EP), a parallel FFT computation (FT), an integer sort program (IS), and a multigrid solver (MG). The data size used for the NAS applications was "class A" which is intended for execution on a network of workstations. All programs were run on an IBM SP-2 and used PVM[4] for communication. We measured the execution times of the programs and compared them with the predicted times of LBF. We also report the overhead of computing LBF.

All measurements were conducted on dedicated SP-2 nodes, and so there was no interference with other applications. The metric computation is not influenced by the overhead of other applications running on the same processors as the target application because the prediction is based only on the process times of the processes in the application and table driven communication time. However, the load on the system influences the timing of the actual configurations.

The summary of the measured and predicted execution times is shown in Figure 3. We use N/M to describe a target or actual configuration where N is the number of processes and M is the number of nodes. For each target configuration, we ran the program in two actual configurations: one identical to the target configuration and the other with no more than half of the nodes of the target configuration. By predicting the performance of a target configuration that was identical to the running configuration, we were able to evaluate how well our communication prediction information worked. The results show that in all cases, the predicted values are within 6% of the actual execution times.

We also measured the overhead of computing the LBF metric. To do this, we ran the same six applications with and without computing LBF. The resulting overhead, shown in Figure 4, represents the extra time required to run the application when computing the LBF metric. For most applications and configurations, the overhead to compute the LBF metric is under 5%. However, for the IS application, the overhead is 7.4%. We investigated the source of this relatively high overhead, and determined that it was caused mainly by the overhead of running the application program with the Paradyn performance tool[3].

[3] We suspect this is due to an interaction between Paradyn and the ptrace facility in programs that make many blocking system calls, but are still investigating this point.

Application Target	Meas. Time	Pred.	Error		Pred.	Error	
SPA		4/4			4/1		
4/4	158.7	159.0	-0.3	(-0.2%)	158.5	0.2	(0.1%)
4/1	240.2	235.5	4.7	(2.0%)	236.2	4.0	(1.7%)
TSP		4/4			4/1		
4/4	85.6	85.5	0.1	(0.1%)	85.9	-0.3	(-0.4%)
4/1	199.2	197.1	2.1	(1.1%)	198.9	0.3	(0.2%)
EP (class A)		16/16			16/8		
16/16	258.2	255.6	2.6	(1.0%)	260.7	-2.5	(-1.0%)
FT (class A)		16/16			16/8		
16/16	140.9	139.2	1.7	(1.2%)	140.0	0.9	(0.6%)
IS (class A)		16/16			16/8		
16/16	271.2	253.3	17.9	(6.6%)	254.7	16.5	(6.0%)
MG (class A)[4]		16/16			16/8		
16/16	172.8	166.0	6.8	(4.0%)	168.5	4.3	(2.5%)

Figure 3: Measured and Predicted Time for LBF.

For each application, we show one or two target configurations and the second column shows the measured time running on this target configuration. The rest of the table shows the execution times predicted by LBF when run under two different actual configuration.

Application Config.	**Msgs**	**Msg Bytes**	**Time** W/o Inst	With Inst	**Overhead** Sec.	%
SPA						
4/4	56	248	158.7	164.2	5.5	3.5%
4/1	56	248	240.2	247.0	6.8	2.8%
TSP						
4/4	6	2.3K	85.6	88.6	3.0	3.5%
4/1	6	2.3K	199.2	203.6	4.4	2.2%
EP (class A)						
16/16	45	1.8K	258.2	268.8	10.6	4.1%
FT (class A)						
16/16	3,480	1.8G	140.9	146.7	5.8	4.1%
IS (class A)						
16/16	7,725	670.5M	271.2	291.2	20.0	7.4%
MG (class A)						
16/16	3,396	400.2M	172.8	178.7	5.9	3.4%

Figure 4: Overhead of Computing LBF.

3. Networking Factor (NF)

Networking Factor addresses the problem of assessing the impact of a network upgrade by predicting the effect of changing a communication network in a distributed or parallel execution environment. Our goal is to compute the potential improvement in execution time if we change the network. The algorithm can also be used to simulate the performance characteristics of long haul networks when the application is run on a local network. Similarly to LBF, we predict the execution time of a program with a virtual network to assess the potential improvement of using the network rather than the currently available network. To validate the NF metric, we com-

[4] The PVM option direct route was used for this application.

pared the execution times of the programs with the predicted times of NF.

To compute NF, we use the same algorithm used for LBF, substituting the communication cost lookup table of a target (predicted) network for the one of the current network. Since we had access to both networks used in our study, we constructed the table by measuring each network. However, if we wished to evaluate a proposed network, we could simply create an appropriate table based on its expected performance. The overhead of computing NF is identical to that of computing LBF.

We implemented NF as a variation of LBF by using the communication cost lookup table for the target network rather than the one for the current network. We tested NF by running the same subset of the NAS benchmarks used to evaluate LBF. We again compared the execution times of the programs running on the real network with the predicted times when running on a different network. The summary of the measured and predicted execution times is shown in Figure 5. For each application, the measured performance is shown for two networks: High Performance Switch (HPS), and a traditional Ethernet. The high performance switch is a 320Mbps switched network, and the Ethernet is a bus based 10Mbps network. We also implemented and tested a combination of LBF and NF by using the target configuration and network communication cost lookup table at the same time. The validation is performed in the same manner as that of NF, and its summary is shown in Figure 6.

The results of running four of the NAS applications with the NF metric are shown in Figure 5. For each application, the second column shows the measured running time of the applications using the HPS, the third column the measured running time using Ethernet, and the fourth column the predicted running time using the HPS when we were running on Ethernet. The last two columns show the error in the prediction relative to the measured HPS execution time. For the MG application, we were able to predict the execution time on the HPS to within 1% even though the measured running time on Ethernet was over twice as long. Likewise, for IS we were able to predict the running time to within 8% when our target and actual configurations had running times that differed by almost a factor of 10. Finally, for FT our prediction was within 4% and the running time was 30 times slower than the target configuration.

Application	**HPS**	**Ethernet**			
	Meas.	**Meas.**	**Pred.**	**Error**	
EP (class A)	258.2	257.4	262.3	-4.1	-1.6%
FT (class A)	140.9	4134.1	135.3	5.6	4.0%
IS (class A)	271.2	2686.7	251.1	20.1	7.4%
MG (class A)	172.8	495.0	174.0	-1.2	-0.7%

Figure 5: Measured and Predicted Time for NF.

Application Conf., Network	Measured Time	Pred.	Error	
EP (class A)		16/8, Ethernet		
16/16, HPS	258.2	259.9	-1.7	-0.7%
FT (class A)		16/8, Ethernet		
16/16, HPS	140.9	136.5	4.4	3.1%
IS (class A)		16/8, Ethernet		
16/16, HPS	271.2	254.4	16.8	6.2%
MG (class A)		16/8, Ethernet		
16/16, HPS	172.8	174.1	-1.3	-0.7%

Figure 6: Comparison of Measured and Predicted Time for a Combination of LBF and NF.

The results of running four of the NAS applications with a combination of the LBF and NF metrics are shown in Figure 6. It shows that in all cases, the predicted values are within 7% of the actual execution times.

4. Related Work

There are two areas that are closely related to our online "what-if" computation: performance measurement tools and performance prediction tools. Performance measurement tools quantify the behavior of an actual program execution and allocate time to specific operations or program components. Performance prediction uses a model or simulation to predict the execution time of an algorithm or program.

There are three major types of performance measurement tools: profilers, visualizations, and search tools. Profile metrics[1, 6, 15, 22] associate a value with each component of a distributed or parallel application (frequently procedures), and are presented as sorted tables. Visualizations[8, 13, 14, 18, 23] explain application performance using pictures. Search tools[10, 17, 21] help users to manage performance data information overload by treating the problem of finding a performance bottleneck as a search problem. However, all of these tools focus on the measurement and analysis of a specific program for a single execution. One type of tool that permits programmers to evaluate alternatives is application steering[7, 19]. Application steering permits programmers to change selected aspects of their program while it is in execution. This technique can be very effective in tuning program parameters, but is by necessity limited in the type of data decomposition and algorithmic changes that can be accommodated within the currently running executable image. Complex algorithmic changes require re-writing part of the program.

Performance predictions can be based either on extrapolations of executions of the program in a controlled environment, or on stochastic models derived from static program analysis. Lost Cycles Analysis[3] predicts performance at different operating points by running a controlled set of experiments that vary an orthogonal set of parameters and record the resulting execution time. How-

ever, this technique requires implementations of the different tuning options to be available for execution. Static prediction[2, 5] uses modeling languages or source code analysis to predict the execution time of a program. By necessity, this technique ignores many details about the interactions between the application, system software, and hardware.

5. Conclusions and Future Directions

We have presented a new performance metric that provides insights into how proposed tuning strategies will improve an application's execution time. We have shown for a collection of six programs that our metric is able to accurately predict the execution time of a modified configuration.

Although LBF is useful for programmers in its current form, there are many directions to expand this research. First, LBF doesn't provide any guidance about what tuning options of a program to evaluate. In most cases, there are multiple tuning alternatives to consider. A future direction is to investigate automatic selection of candidate tuning alternatives. Second, automated selection of candidate configurations combined with LBF provides a basis for dynamic program adaptation where we automatically change programs during execution based on observed behavior to enhance their performance. Third, to permit automatic adaptation, we will need to consider dynamic migration and incorporate migration cost into our metric. In addition, we have developed a finer-grained, function-shipping version of LBF, but haven't presented it in this paper because of space limitations.

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. V. Balasundaram, G. Fox, K. Kennedy, and U. Kremer, "A Static Performance Estimator to Guide Data Partitioning Decisions," 1991 ACM SIGPLAN Symposium on Principals and Practice of Parallel Programming. April 21-24 1991, Williamsburg, VA, pp. 213-223.
3. M. E. Crovella and T. J. LeBlanc, "Parallel Performance Prediction Using Lost Cycles," Proceedings of Supercomputing '94. Nov. 14-18, 1994, Washington, DC, pp. 600-609.
4. A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam, PVM: Parallel Virtual Machine. 1994, Cambridge, Mass: The MIT Press.
5. A. J. C. v. Gemund, "Performance Prediction of Parallel Processing Systems: The PAMELA Methodology," International Conference on Supercomputing (ICS). July 1993, Tokyo, Japan, pp. 318-327.
6. A. J. Goldberg and J. L. Hennessy, "Performance Debugging Shared Memory Multiprocessor Programs with MTOOL," Supercomputing'91. Nov. 18-22, 1991, Albuquerque, NM, pp. 481-490.
7. W. Gu, G. Eisenhauer, E. Kraemer, K. Schwan, J. Stasko, J. Vetter, and N. Mallavurupu, "Falcon: On-line Monitoring and Steering of Large-Scale Parallel Programs," Frontiers '95. Feb 6-9, 1995, McLean, VA, pp. 422-429.
8. M. T. Heath and J. A. Etheridge, "Visualizing Performance of Parallel Programs," IEEE Software, 8(5), 1991, pp. 28-39.
9. J. K. Hollingsworth, "An Online Computation of Critical Path Profiling," SPDT'96: SIGMETRICS Symposium on Parallel and Distributed Tools. May 22-23, 1996, Philadelphia, PA, pp. 11-20.
10. J. K. Hollingsworth and B. P. Miller, "Dynamic Control of Performance Monitoring on Large Scale Parallel Systems," 7th ACM International Conf. on Supercomputing. July 1993, Tokyo, pp. 185-194.
11. D. Kimelman and D. Zernik, "On-the-Fly Topological Sort - A Basis for Interactive Debugging and Live Visualization of Parallel Programs," ACM/ONR Workshop on Parallel and Distributed Debugging. May 17-18, 1993, San Diego, CA, pp. 12-20.
12. L. Lamport, "Time, Clocks, and the Ordering of Events in a Distributed System," CACM, 21(7), 1978, pp. 558-564.
13. F. Lange, R. Kroger, and M. Gergeleit, "JEWEL: Design and Implementation of a Distributed Measurement System," IEEE Transactions on Parallel and Distributed Systems, 3(6), 1992, pp. 657-671.
14. T. Lehr, Z. Segall, D. F. Vrsalovic, E. Caplan, A. L. Chung, and C. E. Fineman, "Visualizing Performance Debugging," IEEE Computer, 21(10), 1989, pp. 38-51.
15. M. Martonosi, A. Gupta, and T. Anderson, "MemSpy: Analyzing Memory System Bottlenecks in Programs," 1992 SIGMETRICS. June 1-5, 1992, Newport, Rhode Island, pp. 1-12.
16. 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, 28(11), 1995, pp. 37-46.
17. S. E. Perl and W. E. Weihl, "Performance Assertion Checking," 14th SOSP. December 5-8, 1993, pp. 134-145.
18. 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. p. 104-113.
19. D. A. Reed, K. A. Shields, W. H. Scullin, L. F. Tavera, and C. L. Ellford, "Virtual Reality and Parallel Systems Performance Analysis," IEEE Computer, 28(11), 1995, pp. 57-68.
20. S. K. Reinhart, J. R. Larus, and D. A. Wood, "The Wisconsin Wind Tunnel: Virtual Prototyping of Parallel Computers," SIGMETIRCS. May 1993, pp. 46-60.
21. W. Williams, T. Hoel, and D. Pase, The MPP Apprentice Performance Tool: Delivering the Performance of the Cray T3D, in Programming Environments for Massively Parallel Distributed Systems. 1994, North-Holland.
22. 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.
23. D. Zernik and L. Rudolph, "Animating Work and Time for Debugging Parallel Programs Foundation and Experience," 1991 ACM/ONR Workshop on Parallel and Distributed Debugging. May 20-21, 1991, Santa Cruz, CA, pp. 46-56.