LOAD BALANCING FACTOR (LBF): A WORKLOAD MIGRATION METRIC [*]

Hyeonsang Eom Jeffrey K. Hollingsworth

Institute for Advanced Computer Studies
Computer Science Department
University of Maryland
College Park, MD 20742
{hseom, hollings}@cs.umd.edu

CS-TR-4375
UMIACS-TR-2002-55

February 2, 1998

Abstract

We introduce a new performance metric, called Load Balancing Factor (LBF), to evaluate different tuning alternatives of workload migration within a distributed/parallel program. The metric is unique because it shows the performance implications of a specific tuning alternative rather than quantifying where time is spent in the program. Previously we developed a variation of the metric for coarse-grained process placement, and demonstrated that it accurately predicts the placement impact. In this paper we focus on a variation designed for fine-grained function shipping in a client/server environment and present its online algorithm. We use a synthetic application to show that LBF provides accurate guidance about procedure-level migration.

1. Introduction

To improve the performance of a program, performance bottlenecks must be located, the causes identified, and the solutions proposed and implemented. So far, most performance debuggers have focused on the first two components leaving the rest to the programmers. Therefore, if there are several tuning options proposed, which is often the case, programmers need to exhaustively try each to choose the best. However, they would be required to make significant efforts to implement each of the options in most cases. In this paper, we focus on solution selection among given tuning alternatives by answering "what-if" style questions and present effective methods to evaluate each of the alternatives.

In distributed/parallel performance debugging, it is especially important to figure out the performance of a tuning alternative and compare it with others. Distributed/parallel computing is basically characterized by workload and data distributed among multiple processors combined with their storage. Obviously, the distribution needs to be balanced with data affinity for best performance. A consumer process that needs a large amount of data has data affinity if its performance is improved by co-location of the data. It is natural to have questions about the performance change due to a different workload or data distribution of a distributed/parallel program.

To effectively provide the potential benefit of tuning alternatives, online computation is used with well-defined levels of distributed computation. For a tuning option, the performance is computed online by combining the execution of the current version of the program and dynamic prediction of the impact of the option using online measurements of the execution. The idea is to execute the original program while simulating the proposed changes to the program. In general, dynamic prediction is better than static one with source-code analysis or instrumentation because the analysis doesn't reveal the dynamic behavior and the instrumentation might result in a large amount of trace data to be collected and processed. It also requires distributed computation granularity to be well defined for their virtual migration in order to be effectively supported; for example, workload can be divided to processes, queries, or procedures. Only with such well-supported computational units, their instrumentation and dynamic predic-

[*] This work was supported in part by NSF awards ASC-9703212 and ACI-9711364, and DOE Grant DE-FG02-93ER25176.

tion using them can be easily performed using currently-available dynamic instrumentation tools such as Paradyn[8].

In this paper, we present a metric called Load Balancing Factor, LBF, that provides programmers with feedback about the performance implications of moving computation between processors. Computation can be shifted either at a fine-grained basis by migrating procedures or at a coarse-grained level by moving entire processes. As a result, we have developed two variants of the LBF metric: procedure LBF and process LBF. Both variants can be effectively computed during the execution of the current version of the program, and do not require post-mortem processing.

We focus on procedure LBF in the rest of this paper. Section 2 briefly describes the previous work on process LBF. Section 3 introduces procedure LBF, explains its online algorithm, and evaluates it using a synthetic application. Section 4 describes related work. Finally, Section 5 summarizes our work and outlines future directions for this research.

2. Previous work

We previously developed process Load Balancing Factor (LBF) that effectively predicts the performance enhancement due to process-level workload migration. It addresses the problem of assessing the impact of process migration by predicting the effect of changing the assignment of processes to processors in a distributed or parallel execution environment. Our goal was 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.

We implemented process LBF, and tested it by running a collection of application programs. We measured the execution times of the programs and compared them with the predicted times of LBF. The results show that in all cases, the predicted values are within 6% of the actual execution times. In most cases, the overhead to compute the LBF metric is under 5%. The details are explained in the preceding paper[3].

3. Procedure LBF

Procedure Load Balancing Factor (LBF) addresses the problem of accessing the impact of fine- grained computation migration by predicating the impact of moving a procedure between one client and multiple servers or one server and multiple clients in a distributed or parallel execution environment. Unlike process LBF, procedure LBF is restricted to client-server style computations with all communication via message passing. Our goal is to compute the potential improvement in execution time if we move a selected procedure, F, from the client to the server or visa-versa.

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.

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.

Critical Path (CP): the longest process time weighted path through a PAG. For an entire program's execution, the CP represents the execution time of the program as if there were one process per processor.

In each process, we keep track of the original CP and the new CP due to moving the selected procedure. We compute procedure LBF at each message exchange. At a send event, we subtract the accumulated time of the selected procedure from the CP of the sending process, and send the accumulated procedure time along with the application message. At a receive event, we add the passed procedure time to the CP value of the receiving process **before** the receive event. The value of the procedure LBF metric is the total effective CP value at the end of the program's execution. Procedure LBF only approximates the execution time with migration since we ignore many subtle issues such as global data references by the "moved" procedure. Our intent with this metric is to supply initial feedback to the programmer about the potential of a tuning alternative. A more refined prediction that incorporates shared data analysis could be run after our metric but before proceeding to a full implementation.

3.1 Algorithm

We describe our algorithm in terms of operations on a PAG. This is done to simplify our description. The actual computation of the metric does not require us to build the graph. There are two basic elements used for the computation of procedure LBF, the difference between the lengths of the CP in the sending and receiving process, and the length of execution of the procedure F to be moved from the sending process to the receiving process. The computation of procedure LBF for a single message send is shown in Figure 1.

When we consider moving a procedure F from one process to another, LBF is the new CP value after moving the procedure. We "move" the execution of the portion of F from between the send operation and the previous inter-process event of the sending process, to just before the receive operation of the receiving process. For each message sent, we allocate time for the selected procedure to the waiting time for the message receive (if any). Figure 1 illustrates how the critical path can change after moving a procedure F. Originally Cs is greater than Cr as seen in the left side of the figure. However, after moving F from S to R we obtain the shorter CP length shown in the right side of the figure, because Cr + F is less than Cs. The difference between the lengths of the two critical paths of both sides at *endRecv*, is the performance benefit due to the movement

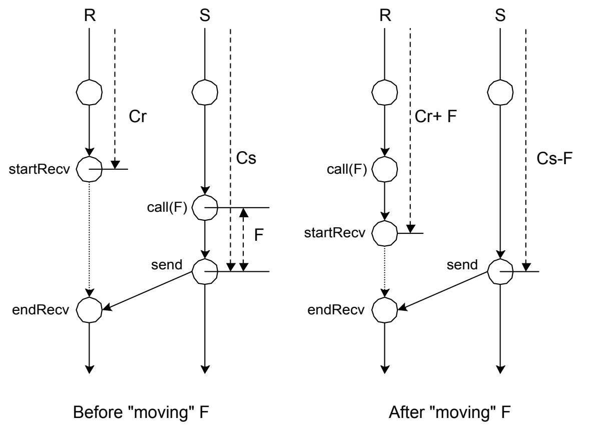

Figure 1: Computing procedure LBF

The PAG before and after moving the procedure F. The time for the procedure F is moved from the sending process (which is on the application's critical path) to the receiving one (which is not).

of F. During application execution, we add the benefit of procedure movement for each message exchange. The overall value of the procedure LBF metric is the predicted execution time due to the accumulation of these benefits.

To compute the length of the new CP due to the movement of a procedure F, we use the normal flow of messages in the application to traverse the PAG. The pseudo code for the algorithm is shown in Figure 2. On each message send, we piggyback the length of the procedure execution between the previous inter-process event and the send operation as well as the length of the CP up to the send operation. After sending the data, we subtract the length of the procedure execution from the length of CP and use the result as the new CP in sending process. For each message receive event, we compute the length of the new CP due to the movement of the procedure. Lines 16-26 of Figure 2 show this calculation. For each message passing operation, we reset the execution time of the selected procedure to zero to ensure that we don't double count the benefit of moving F (shown in lines 10 and 16).

We could remove the restriction that procedure LBF be used only with pure client-server style communication by incorporating Waiting Time Analysis[7]. This limitation arises since we currently use a conservative computation that resets the available procedure time for LBF to zero after each inter-process event to prevent double counting the benefit of migration. However, by incorporating waiting time analysis, we could allow server-to-server or client-to-client communication.

```
Send:
   now <- CPUTime
   longest += now - lastUpdate
   lastUpdate <- now

   IF (curFunc.active)
       F += now - curFunc.lastTime
       curFunc.lastTime <- now
   send(toHost, longest, F)
   longest -= F
   F <- 0;

Recv(fromHost, Cs, rmtF):
  now <- CPUTime()
  longest += now - lastUpdate;
  lastUpdate <- now
  Cr <- longest

  F <- 0
  IF (Cs - Cr > 0)
      IF (curFunc.active)
          curFunc.lastTime <- now;
      IF (rmtF)
          IF (Cs - rmtF > Cr + rmtF)
              longest <- Cs - rmtF
          ELSE
              longest <- Cr + rmtF
  ELSE
      longest += rmtFCP;

 selected procedure entry
     curFunc.active <- 1
     curFunc.lastTime <- CPUTime()

 selected procedure exit
     curFunc.active <- 0
     F += CPUTime() - curFunc.lastTime
```

Figure 2: Computing procedure LBF

3.2 Experiments

We implemented procedure LBF as an extension to Paradyn Parallel Performance Measurement Tools[8]. Using Paradyn provided an easy way to implement the algorithm since it already included support for instrumentation of a running program and periodic sampling callbacks. We tested procedure LBF by running a simple synthetic parallel application. It is more difficult to calibrate the accuracy of procedure LBF than process LBF. In order to evaluate it, we need to change the application to move the functionality from one place to another. Since this is a tedious task and requires detailed knowledge of the application, we only attempted this for a synthetic parallel application.

We created a Synthetic Parallel Application (SPA) that demonstrates a workload where a single server becomes the bottleneck responding to requests from three clients. In the server, two classes of requests are processed: *servBusy1* and *servBusy2*. *ServBusy1* is the service requested by the first client and *servBusy2* is the service requested by the other two clients.

The results of computing procedure LBF for the synthetic parallel application are shown in Figure 3. We then computed procedure LBF for each of these two procedures. To validate these results, we created two modified versions of the synthetic parallel application (one with each of *servBusy1* and *servBusy2* moved from the server the clients) and measured the resulting execution time. The results of the modified programs are shown in the third column of Figure 3. In both cases, the error is small indicating that our metric has provided good guidance to the application programmer.

Procedure	**Procedure LBF**	**Measured Time**	**Difference**	**Percent Error**
ServBusy1	25.3	25.4	0.1	0.4%
ServBusy2	23.0	23.1	0.1	0.6%

Figure 3: Validating procedure LBF accuracy for SPA program

For comparison to an alternative tuning option, we also show the value for the Critical Path Zeroing metric[5]. It is a metric that predicts the improvement possible due to optimally tuning the selected procedure (i.e., reducing its execution time to zero) by computing the length of the critical path resulting from setting the time of the selected procedure to zero. We compare LBF with Critical Path Zeroing because it is natural to consider improving the performance of a procedure itself as well as changing its execution place (processor) as tuning strategies.

The length of the new CP due to the movement of *servBusy1* is 25.4 and the length due to *servBusy2* is 16.1 while the length of the original CP is 30.7. With the Critical Path Zeroing metric, we achieve almost the same benefit as tuning the procedure *ServBusy1* by simply moving it from the server to the client. Likewise, we achieve over one-half the benefit of tuning the *ServBusy2* procedure by moving it to the client side. For any of the tuning alternatives, we report the performance potential if the program were so tuned, it is left to the user to decide which alternative is most feasible to attempt.

Procedure	procedure LBF	Improvement	Critical Path Zeroing	Improvement
ServBusy1	25.3	17.8%	25.4	17.4%
ServBusy2	23.1	25.1%	16.1	47.5%

Figure 4: Procedure LBF and Critical Path for SPA Program

4. Related Work

To track down performance change due to procedure migration on a PAG while incrementally maintaining it, we adapt the on-the-fly topological sort algorithm developed by Kimelman and Zernak[6]. Our algorithm simulates the execution on processes with migrated procedures. To compute the predicted execution time after procedure migration during program execution, we use a variation of our online critical path algorithm[5].

Performance prediction is closely related to our online "what-if" computation. Performance prediction uses a model or simulation to predict the execution time of an algorithm or 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[2] 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. However, this technique requires implementations of the different tuning options to be available for execution. Static prediction[1, 4] 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 online performance metric called Load Balancing Factor, LBF, that provides insights into how proposed tuning strategies will improve an application's execution time. We have developed procedure LBF that predicts performance improvement due to procedure migration in a client/server environment. We have shown for a synthetic application 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 to evaluate. In most cases, there are multiple tuning alternatives to consider. A future direction is to investigate automatic selection of candidate tuning alternatives. Eventually, 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.

References

1. 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.
2. 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.
3. H. Eom and J. K. Hollingsworth, "LBF: A Performance Metric for Program Reorganization," to appear, 18th Int'l Conf. Distributed Computing Systems. May 26-29 1998, Amsterdam, The Netherlands.
4. 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.
5. 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.
6. 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, 1996, San Diego, CA, vol.1, pp. 12-20.
7. W. Meira, T. J. LeBlanc, and A. Poulos, "Waiting Time Analysis and Performance Visualization in Carnival," SPDT'96: SIGMETRICS Symposium on Parallel and Distributed Tools. May 22-23, 1996, Philadelphia, PA, pp. 1-10.
8. 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.