CS-TR-3703 UMIACS-TR-96-73

Grindstone: A Test Suite for Parallel Performance Tools*

Jeffrey K. Hollingsworth
Institute for Advanced Computer Studies
and Computer Science Department

Michael Steele
Computer Science Department

University of Maryland
College Park, MD 20742
{hollings,mike}@cs.umd.edu

Abstract

We describe Grindstone, a suite of programs for testing and calibrating parallel performance measurement tools. The suite consists of nine simple SPMD style PVM programs that demonstrate common communication and computational bottlenecks that occur in parallel programs. In addition, we provide a short case study that demonstrates the use of the test suite on three performance tools for PVM. The results of the case study showed that we were able to uncover bugs or other anomalies in all three tools. The paper also describes how to acquire, compile, and use the test suite.

* This work was supported in part by NIST CRA award 70-NANB-5H0055, DOE Grant DE-FG02-93ER25176, and the State of Maryland.

1. Introduction

Creating and debugging performance measurement tools for parallel systems is a difficult task. First, parallel tools are often parallel programs in their own right, and as such have all of the difficulties of writing parallel programs. Second, the input data for parallel tools is also a parallel program. Full parallel programs (and even "small" kernels) are often large programs that exhibit complex performance behavior. Performance tools exist to measure and analyze these complex performance characteristics. However, when debugging and validating performance measurement tools, it is useful to have simpler programs with well understood performance characteristics. For testing implementations of programming languages, validation suites exist that contain many small programs each testing one or a few features of the language. We feel that a similar suite of parallel programs would be useful to help develop parallel performance tools.

We have created a validation suite of programs, called Grindstone[1], that demonstrate simple but common problems which parallel programs commonly suffer from. Test Program suites such as SPLASH[5] and PERFECT Club[1] have proved useful to evaluate parallel compiles. We intend Grindstone as the starting point for a standardized validation suite which authors of performance analysis tools will be able to use to verify that their software gives correct prognoses for common problems. In this paper, we present nine simple PVM programs that demonstrate common performance problems. In addition, we present a brief case study that shows the resulting of running these programs with three performance measurement tools: XPVM, Paradyn, and PVaniM. The tests programs were able to uncover bugs in all three tools. The appendix at the end of the paper describes how to obtain, compile, and run the test programs.

2. Description of the Validation Suite

In this section, we describe each of the programs in the validation suite. Each program is a single PVM application that consists of a single executable image. The programs are all written in a SPMD style and spawn additional copies of themselves as needed. When it completes execution, each program prints the wall time, CPU time, messages sent, and messages received by each process. This information can be used to validate the metrics produced by each tool. The timing information can also be used to compute the overhead of a tool by comparing the execution time of each program run without the tool to the time to run with the tool. We have divided the test programs into two categories: communication bottleneck programs and computation bottlenecks. An obvious third category is programs that are limited by I/O performance. How-

[1] Like a physical grindstone, this test suite is intended to shape and hone other tools.

ever, since there is currently no standard way of performing I/O in PVM, we have omitted I/O test programs.

2.1 Communication Bottleneck Programs

The test suite contains six programs that demonstrate various types of communication problems. All of the programs use the blocking versions of either point-to-point (`pvm_send/pvm_recv`) or group (`pvm_barrier`) communication. Four of the programs demonstrate pure communication performance since they are limited by the latency of bandwidth of the message passing system. The other two illustrate the interaction between load imbalance and communication waiting time.

Passing Large Messages (big-message)

The first test program passes very large messages (an array of 100,000 integers). The intended bottleneck is that the overhead associated with setting up and sending a large message should slow the program down. This program is limited by speed at which messages can move through the communication network. Since the messages are large, communication bandwidth, not latency is the important problem here. Two processes are spawned: the parent starts by sending the large message to the child. When the child process receives the message, it sends it back. This continues for a preset number of iterations. The program can be configured to send its messages indirectly via either the PVM daemon process, or directly using the "direct route" option of PVM.

Passing Too Many Small Messages (small-messages)

The second program in the test suite demonstrates the problem of passing too many small messages. The program small-messages was designed to show how the overhead associated with message passing could become a bottleneck. In this example, the parent process starts up three children. Then, all of the children begin sending small messages (the size of two integers) to the parent in an attempt to swamp the parent with too many incoming messages. Then, all of the children begin sending small messages (the size of two integers) to the parent in an attempt to swamp the parent with too many incoming messages. Since none of the messages generates a reply, it is possible to combine these messages into larger units of work to amortize the per message overhead.

Latency Critical Messaging (ping-pong)

The third program in the test suite illustrates an application that is being slowed down by the round-trip latencies between processes. This program spawns one worker process. Like small-messages, it sends a short message from one process to the other. However, before the sender continues, it waits for a response from the other process. Message packing and unpacking time, PVM daemon latency, and message time-of-flight all contribute to the critical path of this program.

Server Intensive Applications (intensive-server)

The fourth program in the validation suite demonstrates the pitfall of having a client-server model in which the server is overloaded. In the intensive-server program, the parent process (which will be the server) spawns three child processes (which become the clients). The clients initially wait for a message from the server telling them to start, and then go into a cycle of sending the server a message and waiting for a response. Meanwhile, the server is receiving the messages from clients, but then wastes time before sending out responses. This simulates the server having too much work to do in comparison with the clients.

Random Barriers (random-barrier)

The next bottleneck demonstration is similar to the intensive-server problem, only this time no single process is the bottleneck. The parent process starts three children, and then sits and wastes time. The other processes wait at a `pvm_barrier` call. When the parent finishes wasting time, it passes a token to another randomly selected process (hence, this program is called random-barrier), and that process wastes a predetermined amount of time. The process continues until each process has gone through a predetermined number of iterations. The effect of this program is to simulate a program with a load imbalance that is dependent on the data and moves around to different processors during different iterations of a major loop (such as a time-step loop in a simulation of a physical system).

Passing Messages Out Of Order (wrong-way)

The final message passing test program highlights the problem of passing messages "out-of-order." This problem could arise if one process is expecting messages in a certain order, but another process is sending messages which are not in the expected order. This could also arise if the two processes have to communicate over a "noisy" network, and message packets are dropped or have to be resent. In the wrong-way program, an extreme case of having messages out-of-order is used. Two processes are spawned. The first process sends messages with messages numbered 1 through n (where n set to 1,000 in the program). The second process waits to receive the messages, but expects to receive the messages in the order n through 1. This program is also known to stress test tools that record and match message send and receive operations.

2.2 Computation Bottleneck Programs

This section describes the three computation bottleneck programs we have created. Some of these program illustrate computation bottlenecks that occur only in parallel programs, and some that also can happen in serial programs.

One Bottleneck Procedure (hot-procedure)

This program demonstrates a bottleneck created by a single procedure that is consuming the majority of the time. It is a completely serial program, and not additional processes are created. The program contains 21 procedures that are called 100 times each. Almost all of the processor time is due to one procedure `bottleneckProcedure`.

Diffused Bottleneck Procedure (diffuse-procedure)

This program demonstrates a procedure bottleneck that is distributed among several processes in an application. The procedure `bottleneckProcedure` is responsible for 50% of the CPU time in the overall program. Each of the other procedures consumes approximately 2.5% of the program's CPU time. The program consists of four "rounds" where the `bottleneckProcedure` is active in each of the program's five processes. When a process is not executing `bottleneckProcedure`, is it runs one of the 20 `irrelevantProcedure*`. The only communication or synchronization in the program is 21 barrier operations per process.

Excessive System time (system-time)

This program spends most of its time executing code in the operating system kernel. The program is completely serial and does not create any additional processes. Depending on the operating system, it should spend about 75% of its time as "system time" and the remaining 25% is user time. System time is consumed by having the program repetitively call the UNIX system call kill to send itself a continue signal.

3. Experience Using the Validation Suite

We ran the suite with three performance tools indented to work with the PVM system. Performance tools can be evaluated by two different criteria, their quantitative accuracy, or the quality of the guidance supplied to the programmer. Quantitative accuracy refers to the ability of a tool to correctly time or count the events during a program's execution. Quality of guidance is an indication of the value the tool provides the programmer in locating the source of a performance bottleneck. Due to the difficulties in comparing the quality of guidance, we choose to focus on quantitative measures. We hope as we expand and refine this tool set to be able to use it to compare the quality of guidance too.

For all three tools we tested, we were able to find bugs or anomalous behavior for at least one of the test programs. Below we summarize the bugs we found in each tool. It is not our intent to single out these tools for specific criticism. We selected them not because they were buggy, but rather that they are freely available and are easy to install and start using. We had tried several other tools, but were never able to get them to work in our environment.

3.1 PVaniM

The Georgia Tech Graphics, Visualization, and Usability Center's PVaniM[6] visualization tool was the first program tested. Using PVaniM (version 2.0) required a few minor modifications to the test suite source code (using the PVaniM include file, and calling a function to register the process with the PVaniM monitor program). Two problems were discovered for this tool using the test suite. One problem manifested itself by causing the application processes die, and the other by reporting that all the processes were doing productive work when all but one of them was stopped at a barrier.

For the random barrier application, PVaniM reports that all of the processes are doing work, when in fact, they are waiting at a barrier. This problem is caused by PVaniM using macros to instrument PVM message passing routines. The barrier function in PVM is built using the normal send and receive procedures and was not explicitly instrumented by PVaniM. However, since PVaniM only requires recompiling the application, the calls made by the implementation of the barrier routine to `pvm_send` and `pvm_recv` from within the PVM library are not instrumented.

Running wrong-way and small-messages with PVaniM caused the PVM daemons to die on all the hosts except for the host on which PVaniM was running. Reducing the number of iterations allowed the processes to finish running. The cause of this problem is that when the PVaniM process gets behind, it does not de-queue instrumentation messages from the PVM daemons. Eventually, the PVM daemons exhaust the available virtual memory and exit. It would appear that this is really two bugs: PVaniM can't keep up with a high volume of traffic, and PVM does not gracefully handle large backlogs of messages.

3.2 XPVM

XPVM[2] is a performance visualization and configuration management tool for PVM that was developed by Oak Ridge National Labs. For our study we used version 1.1. The tool is implemented in TCL/TK and provides several visualizations to display the communication behavior of PVM programs. We found three types of problems with XPVM. First, sometimes the tool is not able to keep up with a large volume of messages being exchanged. Second, the tool caused the PVM daemons to crash in two configurations. Third, the visualizations were not able to differentiate individual communication operations when the volume of message passing was high.

For the small-message program, the time required for XPVM to draw the animation adds considerable overhead, and XPVM is unable to keep up with the rate at which messages are generated. The server and one client process couldn't run to completion in over 40 minutes of runtime, and it appeared that while XPVM was still running, the processes had become hung.

Also in the too small message passing program, there are so many messages being rapidly passed that the message lines are close enough to each other to appear as a solid triangle. This appears to be a problem with not supporting a zoom feature that would let the programmer expand the time scale to differentiate the individual messages.

Finally, the documentation claims that the trace output files are in SDDF format, ready for Pablo to use for analysis, but we found that this wasn't completely the case. XPVM outputs data types and trace output interleaved, but Pablo[4] insists that all the data types be defined before any of the trace output lines. Also, XPVM's SDDF files don't begin with the appropriate magic string. A simple perl script could be used to overcome these problems, but it would be helpful if XPVM generated the traces in the correct format. We are not sure if this problem was due to a version miss-match between Pablo and XPVM or if they never worked together.

3.3 Paradyn

The third program tested is Paradyn[3], developed at the University of Wisconsin. To test the automated search features of Paradyn, when we increased the number of iterations to one million for the small-messages program. However, Paradyn couldn't handle it: some of the Paradyn daemon processes died before the small-messages processes were done running. As these processes died, error message dialog boxes popped up announcing which machine they had died on. Further investigation showed that the daemon processes were crashing as a result of running out of memory. It would have been helpful if the tool had printed a more descriptive error message for this case.

4. Conclusion

This paper presented a nine program PVM-based test suite for parallel measurement tools and then showed the results of running the test-suite with three such existing analysis tools: XPVM, Paradyn, and PVaniM. The six programs tested passing large messages, a client-server model where the server did all the work, programs taking turns being the barrier at random, passing lots of small messages, and passing messages out-of-order. The Grindstone suite highlights some of the basic problems parallel programs suffer; however, despite the simple nature of the tests, the suite of programs found flaws or exposed bugs in every performance analysis tool we tried. Clearly, there is room for improvement in the field of parallel performance analysis tools.

Each of these three tools provides useful information which complements the information from the other tools. Separately, each can give the programmer a view of what is going on, but any single tool will have its shortcomings. By using multiple tools together, one can get a more complete picture of how a parallel algorithm is working and where the performance bottlenecks are.

Our goal is to develop a more comprehensive set of test programs for parallel performance and correctness tools. We welcome and encourage people to submit suggestions for additional common problems.

5. Compiling and Testing

This appendix describes how to get, compile, and run the Grindstone package.

The source code for the test suite is available via anonymous ftp from `ftp.cs.umd.edu` in

`pub/faculty/hollings/grindstone.tar.gz.`

The package comes with a single Makefile that should build all of the test suite. To compile the software for your specific platform, you will need to create a directory for the target platform (using the PVM_ARCH environment). If the performance tool to be tested requires additional libraries to be linked into the application, edit the Makefile and insert the libraries to the appropriate location. For all of the tests conducted in this report, we used gcc 2.7.2 on a cluster of 4 Sun Microsystems SPARCstation5's running SunOS 4.1.3. In addition, we used PVM version 3.3.10.

6. Acknowledgments

Charles Lin and Alex Kaplunov developed an early prototype of this package. Pete Keleher suggested the name Grindstone. Brad Topol patiently tracked down a problem caused by our not following the directions supplied with PVaniM.

References

1. M. Berry, *et al.*, "The PERFECT Club benchmarks: Effective performance evaluation of supercomputers", *The International Journal of Supercomputing Applications*, 1989. **3**(3), pp. 5-40.
2. A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam, *PVM : Parallel Virtual Machine*. 1994, Cambridge, Mass: The MIT Press.
3. 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.
4. 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.
5. J. P. Singh, W.-D. Weber, and A. Gupta, "SPLASH: Stanford Parallel Applications for Shared-Memory", *Computer Architecture News*, March 1992. **20**(1), pp. 5-44.
6. B. Topol, J. T. Stasko, and V. S. Sunderam, *Monitoring and Visualization in Cluster Environments*,GIT-CC-96-10,Georgia Institute of Technology, March 1996.