Review of Performance Analysis Tools for MPI Parallel Programs

Shirley Moore, David Cronk, Kevin London, and Jack Dongarra
Computer Science Department, University of Tennessee
Knoxville, TN 37996-3450, USA
{shirley,cronk,london,dongarra}@cs.utk.edu

Abstract. In order to produce MPI applications that perform well on today's parallel architectures, programmers need effective tools for collecting and analyzing performance data. Because programmers typically work on more than one platform, cross-platform tools are highly desirable. A variety of such tools, both commercial and research, are becoming available. Because the tools use different approaches and provide different features, the user may be confused as to what tool will best suit his needs. This paper reviews the available cross-platform MPI performance analysis tools and evaluates them according to a set of criteria that includes robustness, usability, scalability, portability, and versatility. Issues pertaining to hybrid and heterogeneous, distributed programming environments and to MPI I/O performance analysis are also discussed.

1 Introduction

The reasons for poor performance of parallel message-passing codes can be varied and complex, and users need to be able to understand and correct performance problems. Performance tools can help by monitoring a program's execution and producing performance data that can be analyzed to locate and understand areas of poor performance. We are investigating a number of performance tools, both research and commercial, that are available for monitoring and/or analyzing the performance of MPI message-passing parallel programs. This paper describes an on-going review of those tools that we are conducting.

A number of commercial and research tools are available for performance analysis of MPI programs. The most prevalent approach taken by these tools is to collect performance data during program execution and then provide post-mortem analysis and display of performance information. Some tools do both steps in an integrated manner, while other tools or tool components provide just one of these functions. A few tools also have the capability for run-time analysis, either in addition to or instead of post-mortem analysis.

We are restricting our investigation to tools that are either publicly or commercially available and that are being maintained and supported by the developer or vendor. We are also focusing on tools that work across different platforms, rather than on vendor tools that work only on one platform. To give our review continuity and focus, we are following a similar procedure for testing each tool and using a common set of evaluation criteria.

We first build and install the software using the instructions provided. After the software has been installed successfully, we work through any tutorial or examples that are provided so that we can become familiar with the tool. Finally, we attempt to use the tool to analyze a number of test programs.

Our set of evaluation criteria consists of the following:

1. robustness
2. usability
3. scalability
4. portability
5. versatility

In addition to these general criteria, we include the following more specific criteria:

1. support for hybrid environments
2. support for distributed heterogeneous environments
3. support for analysis of MPI-2 I/O

For robustness, we expect the tool to crash infrequently and features to work correctly. Errors should be handled by displaying appropriate diagnostic messages. Also, the tool should not cause the user to get stuck when he/she takes a wrong action by mistake. Research tools are not expected to be as robust as commercial tools, but if the tool has been released for public use, considerable effort should still have been invested in debugging it and on error handling.

To be useful, a tool should have adequate documentation and support, and should have an intuitive easy-to-use interface. On-line help and man pages are also helpful for usability. Although research tool developers cannot provide extensive support for free, we consider an email address for questions and bug reporting to be a minimum requirement for a tool that has been released for public use. Adequate functionality should be provided to accomplish the intended task without putting undue burden on the user to carry out low-level tasks, such as manual insertion of calls to trace routines, or sorting and merging of per-process trace files without assistance.

For scalability, we look for the ability to handle large numbers of processes and large or long-running programs. Scalability is important both for data collection and for data analysis and display. For data collection, desirable features are scalable trace file formats, a mechanism for turning tracing on and off, and filtering of which constructs should be instrumented. For data analysis and display, important scalability features include ability to zoom in and out, aggregation of performance information, and filtering.

Because of the short lifespan of high performance computing platforms and because many applications are developed and run in a distributed heterogeneous environment, most parallel programmers will work on a number of platforms simultaneously or over time. Programmers are understandably reluctant to learn a new performance tool every time they move to a new platform. Thus, we consider portability to be an important feature. For portability, we look for whether the tool works on most major platforms and for what MPI implementations and languages it can handle.

For versatility, we look for the ability to analyze performance data in different ways and to display performance information using different views. Another feature of versatility is the ability to interoperate with other trace formats and tools.

By hybrid environments, we mean a combination of shared and distributed memory, both with respect to hardware and parallel programming models. The evolving HPC architecture is that of clustered SMP systems where shared memory nodes with up to 32 or more CPUs are being interconnected by dedicated high performance networks. The different communication layers in such a hybrid system present problems for the performance of parallel applications. An emerging parallel programming paradigm is to use message passing between nodes and shared memory (e.g., OpenMP) within a node. Most performance anlaysis tools were originally developed to support only one of these models, but tools that support both models simultaneously are highly desirable.

Another emerging model of parallel computation is that of running large computations across a computational grid which consists of a distributed collection of possibly heterogenous machines [8, 7]. Thus there is a need for performance analysis tools that work in distributed heterogeneous environments.

As processing speeds increase, I/O is becoming a bottleneck for certain classes of applications. MPI I/O, which is part of the MPI-2 standard, simplifies parallel I/O programming and potentially improves performance o by allowing the programmer to make a single I/O call to access non-contiguous memory or file data, and by striping files and marshaling values to combine many small reads/writes into a few large reads/writes [9]. Tools are needed that not only trace MPI I/O calls but also assist the user specifically with analysis of MPI I/O performance.

2 Tools Reviewed

2.1 DEEP/MPI

URL	http://www.psrv.com/deep_mpi_top.html
Version	
Supported languages	Fortran 77/90/95, C, mixed Fortran and C
Supported platforms	Linux x86, SGI IRIX, Sun Solaris Sparc, IBM RS/6000 AIX, Windows NT x86

DEEP and DEEP/MPI are commercial parallel program analysis tools from Veridian/Pacific-Sierra Research. DEEP (Development Environment for Parallel Programs) provides an integrated graphical interface for performance analysis of shared and distributed memory parallel programs. DEEP/MPI is a special version of DEEP that supports MPI program analysis. To use DEEP/MPI, one must first compile the MPI program with the DEEP profiling driver **mpiprof**. This step collects compile-time information which is stored in a deep subdirectory and instruments the code. Then after executing the program in the usual manner the user can view performance information using the DEEP/MPI interface.

The DEEP/MPI interface includes a call tree viewer for program structure browsing and tools for examining profiling data at various levels. DEEP/MPI displays whole program data such as the wallclock time used by procedures. After identifying procedures of interest, the user can bring up additional information for those procedures, such as loop performance tables. The DEEP Performance Advisor suggests which procedures or loops the user should examine first. MPI performance data views allow users to identify MPI calls that may constitute a bottleneck. Clicking on a loop in a loop performance table or on an MPI call site takes the user to the relevant source code. CPU balance and message balance displays show the distribution of work and number of messages, respectively, among the processes. DEEP supports the PAPI interface to hardware counters and can do profiling based on any of the PAPI metrics [2, 3]. DEEP supports analysis of shared memory parallelism. Thus DEEP/MPI can be used to analyze performance of mixed MPI and shared-memory parallel programs.

2.2 MPE Logging and Jumpshot

URL	http://www-unix.mcs.anl.gov/mpi/mpich/
Version	MPICH 1.2.1, Jumpshot-3
Supported languages	Fortran, C, C++
Supported platforms	AIX, Compaq Tru64 UNIX, HP-UX, IRIX, LINUX, Solaris, WindowsNT/2000

The MPE (Multi-Processing Environment) library is distributed with the freely available MPICH implementation and provides a number of useful facilities, including debugging, logging, graphics, and some common utility routines. MPE was developed for use with MPICH but can be used with other MPI implementations. MPE provides several ways to generate logfiles which can then be viewed with graphical tools also distributed with MPE. The easiest way to generate logfiles is to link with an MPE library that uses the MPI profiling interface. The user can also insert calls to the MPE logging routines into his or her code. MPE provides two different logfile formats, CLOG and SLOG. CLOG files consist of a simple list of timestamped events and are understood by the Jumpshot-2 viewer. SLOG stands for Scalable LOGfile format and is based on doubly timestamped states. SLOG files are understood by the Jumpshot-3 viewer. The states in an SLOG file are partitioned into frames of data, each of which is small enough to be processed efficiently by the viewer. SLOG and Jumpshot-3 are designed to be capable of handling logfiles in the gigabyte range.

After a user loads a logfile into Jumpshot-3, a statistical preview is displayed to help the user select a frame for more detailed viewing. Selecting a frame and clicking on the Display button causes Jumpshot-3 to display a timeline view with rectangles which represent the various MPI and user defined states for each process and arrows which represent messages exchanged between those states. The user may select and deselect states so that only those states that are of interested are displayed. In the case of a multithreaded environment such as an SMP node, the logfile may contain thread information and Jumpshot-3 will illustrate how the threads are dispatched and used in the MPI program. The timeline view can be zoomed and scrolled. Clicking on a rectangle or an arrow brings up a box that displays more detailed information about the state or message, respectively.

2.3 Pablo Performance Analysis Tools

URL	http://www-pablo.cs.uiuc.edu/
Version	Trace Library 5.1.3, Pablo Performance Capture Facility (PCF) March 2001 release, SvPablo 4.1
Languages	Fortran 77/90, C, HPF (SvPablo)
Supported platforms	PCF has been tested on Sun Solaris, SGI IRIX, and Linux/x86. SvPablo runs on Sun Solaris, SGI IRIX, IBM AIX, and Linux/x86.

The Pablo Trace Library includes a base library for recording timestamped event records and extensions for recording performance information about MPI calls, MPI I/O calls, and I/O requests. All performance records generated by the Trace Library are in the Pablo SDDF (Self Defining Data Format) format. SDDF is a data description language that specifies both data record structures and data record instances. SDDF has both a compact binary version and a human-readable ASCII version.

The MPI extension is an MPI profiling library [11]. Trace records are written that capture timing and parameter information for MPI calls in SDDF format. A separate SDDF file is produced for each MPI process. Utility programs are provided for merging the per-process trace files, but they must be invoked manually. The MPI I/O extension provides additional wrapper routines for recording information about MPI I/O calls [12]. The user may choose between a detailed tracing mode that writes records for each MPI I/O event or a summary tracing mode that summarizes the performance data for each type of MPI I/O call. In addition to linking with the MPI I/O profiling library, the user must link with the base trace library and add calls to the application source code to initialize and end MPI I/O tracing. Utilities are provided that analyze the MPI I/O trace files and produce reports.

The Pablo Performance Capture Facility (PCF) supports MPI and provides several options for recording performance information for Unix I/O, Hierarchical Data Format (HPF), and MPI I/O operations. PCF can either write performance information to trace files in the SDDF format or allow the user to monitor performance in real-time using the Pablo Autopilot facility. Autopilot is an infrastructure for real-time adaptive control of parallel and distributed computing resources. Unlike the Pablo Trace Library, PCF is thread-safe and thus can be used with mixed MPI and threaded programs.

SvPablo is a graphical user interface for instrumenting source code and browsing runtime performance data. Applications can be instrumented either interactively or automatically. To interactively instrument an application code, SvPablo parses each source file and flags constructs (outer loops and function or subroutine calls) that can be instrumented. The user then selects the events to be instrumented and SvPablo generates a new version of the source code containing instrumentations calls. Alternatively, the SvPablo stand-alone parser can be used to instrument all subroutines and outer loops, including MPI calls. After the instrumented application has been run, SvPablo correlates the runtime performance data with application source code and performs statistical analyses. SvPablo displays a graphical representation that visually links performance information to the original source code. The SvPablo data capture library supports the use of the MIPS R10000 hardware performance counters to allow visualization of hardware events on SGI architectures. Plans are to use the PAPI portable hardware counter interface [2, 3] in future versions of SvPablo.

2.4 Paradyn

URL	http://www.cs.wisc.edu/paradyn/
Version	3.2
Languages	Fortran, C, C++, Java
Supported platforms	Solaris (SPARC and x86), IRIX (MIPS), Linux (x86), Windows NT (x86), AIX (RS6000), Tru64 Unix (Alpha), heterogeneous combinations

Paradyn is a tool for measuring the performance of parallel and distributed programs. Paradyn dynamically inserts instrumentation into a running application and analyzes and displays performance in real-time. Paradyn decides what performance data to collect while the program is running. The user does not have to modify application source code or use a special compiler because Paradyn directly instruments the binary image of the running program using the DyninstAPI dynamic instrumentation library[6]. MPI programs can only be run under the POE environment on the IBM SP2, under IRIX on the SGI Origin, and under MPICH 1.2 on Linux and Solaris platforms. Additional MPI support is under development. Although Paradyn can instrument unmodified executable files on most platforms, it is recommended to build the executable to include debug information.

Paradyn has two main components: the Paradyn front-end and user interface, and the Paradyn daemons. The daemons run on each remote host where an application is running. The user interface allows the user to display performance visualizations, use the Performance Consultant to find bottlenecks, start or stop the application, and monitor the status of the application. The daemons operate under control of the front-end to monitor and instrument the application processes. The Paradyn front-end requires Tcl/Tk.

The user specifies what performance data Paradyn is to collect in two parts: the type of performance data and the parts of the program for which to collect this data. The types of performance data that can be collected include metrics such as CPU times, wallclock times, and relevant quantities for I/O, communication, and synchronization operations. The performance data can be displayed using various visualizations which include barcharts, histograms, and tables. The data can be displayed for either the *global phase* which starts at the beginning of program execution and extends to the current time, or for a *local phase* which is a subinterval of the execution. As an alternative to manually specifying what performance data to collect and analyze, the user can invoke the Paradyn Performance Consultant. The Performance Consultant attempts to automatically identify and diagnose the types of performance problems a program is having by testing various hypotheses such as whether the application is CPU bound or is experiencing excessive I/O or synchronization waiting time. The user can change the behavior of the Performance Consultant by adjusting the thresholds used to evaluate the hypotheses.

2.5 TAU

URL	http://www.acl.lanl.gov/tau/
Version	2.9.11 (beta)
Supported languages	Fortran, C, C++, Java
Supported platforms	SGI IRIX 6.x, Linux/x86, Sun Solaris, IBM AIX, HP HP-UX, Compaq Alpha Tru64 UNIX, Compaq Alpha Linux, Cray T3E, Microsoft Windows

TAU (Tuning and Analysis Utilities) is a portable profiling and tracing toolkit for performance analysis of parallel programs written in Java, C++, C, and Fortran. TAU's profile visualization tool, Racy, provides graphical displays of performance analysis results. In addition, TAU can generate event traces that can be displayed with the Vampir trace visualization tool.

TAU requires Tcl 7.4/Tk 4.0 or higher (8.x is recommended) which is available as freeware. TAU is configured by running the configure script with appropriate options that select the profiling and tracing components to be used to build the TAU library. The default option specifies that summary profile files are to be generated at the end of execution. Profiling generates aggregate statistics which can then be analyzed using Racy. Alternatively, the tracing option can be specified to generate event trace logs that record when and where an event occurred in terms of source code location and the process that executed it. Traces can be merged and converted to Vampitrace format so that they can be visualized using Vampir. The profiling and tracing options can be used together.

In order to generate per-routine profiling data, TAU instrumentation must be added to the source code. This can be done automatically for C++ programs using the TAU Program Database Toolkit, manually using the TAU instrumentation API, or using the **tau_run** runtime instrumentor which is based on the DyninstAPI dynamic instrumentation package. An automatic instrumentor for Fortran 90 is under development. With C++, a single macro, TAU_PROFILE, is sufficient to profile a block of statements. In C and Fortran, the user must use statement level timers by inserting calls to TAU_PROFILE_TIMER (declares the profiler to be used to profile a block of code), TAU_PROFILE_START (starts the timer for profiling a set of statements), and TAU_PROFILE_STOP (stops the timer used to profile a set of statements). In addition to time-based profiling, TAU can use either PAPI [2, 3] or PCL [1] to do profiling based on hardware performance counters.

Generation of MPI trace data may be done in two ways. One way is to use the TAU instrumentation API and insert calls to TAU_TRACE_SENDMSG and TAU_TRACE_RECVMSG into the source code. The other way is to link to the TAU MPI wrapper library which uses the MPI profiling interface. Hybrid execution models can be traced in TAU by enabling support for both MPI and the thread package used (e.g., Pthreads or OpenMP).

2.6 Vampir

URL	http://www.pallas.de/pages/vampir.htm
Version	Vampitrace 2.0, Vampir 2.5
Supported languages	Fortran 77/90, C, C++
Supported platforms	All major workstation and parallel platforms

Vampir is a commercially available MPI analysis tool from Pallas GmbH. Vampirtrace, also from Pallas, is an MPI profiling library that produces trace files that can be analyzed with Vampir. The Vampirtrace library also has an API for stopping and starting tracing and for inserting user-defined events into the trace file. Instrumentation is done by linking your application with the Vampirtrace library after optionally adding Vampirtrace calls to your source code. Vampirtrace records all MPI calls, including MPI I/O calls. A runtime filtering mechanism can be used to limit the amount of trace data and focus on relevant events. For systems without a globally consistent clock, Vampirtrace automatically corrects clock offset and skew.

Vampir provides several graphical displays for visualizing application runtime behavior. The timeline and parallelism display shows per-process application activities and message passing along a time axis. Source-code click-back is available on platforms with the required compiler support. Other displays include statistical analysis of program execution, statistical analysis of communication operations, and a dynamic calling tree display. Most displays are available in global and per-process variants. The timeline view can be zoomed and scrolled. Statistics can be restricted to arbitrary parts of the timeline display.

Although the current version of Vampir can display up to 512 processes, that number of processes in a timeline display can be overwhelming to the user and simultaneous display of thousands of processes would clearly be impractical. A new version of Vampir that is under development uses a hierarchical display to address display scalability [5]. The hierarchical display allows the user to navigate through the trace data at different levels of abstraction. The display is targeted at clustered SMP nodes and has three layers (cluster, node, process) that can each hold a maximum of 200 objects, with the process layer displaying trace data for individual threads. To handle hybrid parallel programming models (e.g., MPI + OpenMP), a new tool is under development that combines Vampir with the Intel/KAI GuideView tool [10].

3 Evaluation Summary

We now address the evaluation criteria that were described in section 1 with respect to each of the tools reviewed in section 2.

Although we have encountered problems with some of the tools not working properly in certain situations, it is too early to report on robustness until we have tired to work out these problems with the tool developers.

With the exception of Jumpshot, all the tools have fairly complete user guides and other supporting material such as tutorials and examples. The documentation provided with Jumpshot is scant and poorly written. Jumpshot itself has a fairly intuitive interface as well as online help. However, some of the features remain unclear such as the difference between connected and disconnected states and between process and thread views. A nice usability feature of MPE logging and jumpshot is their use of autoconf so that they install easily and correctly on most systems.

Paradyn has the drawback that it is difficult or impossible to use in the batch queueing environments in use on most large production parallel machines. For example, we have been unable to get Paradyn to run an MPI program in such an environment. This is not surprising since Paradyn is an interactive tool, but with sufficient effort other tools such as interactive debuggers have been made to work in such environments.

Manual instrumentation is required to produce subroutine-level performance data with the MPE and Pablo trace libraries and with TAU for Fortran 90, and this can be tedious to the point of being impractical for large programs. The source-code click-back capability provided by DEEP/MPI, SvPablo, and Vampir is very helpful for relating performance data to program constructs.

Scalable log file formats such as MPE's SLOG format and the new Vampirtrace format [5] are essential for reducing the time required to load and display very large trace files. The summary option provided by the Pablo MPI I/O trace library allows generation of summary trace files whose size is independent of the runtime of the program. Although Paradyn achieves scalability in one sense by using dynamic instrumentation and by adjusting instrumentation granularity, some of the user interface displays such as the Where Axis and the Performance Consultant graph become unwieldy for large programs. The new hierarchical Vampir display is a promising approach for visualizing performance on large SMP clusters in a scalable manner.

Of all the tools reviewed, Vampir is the only one that has been tested extensively on all major platforms and with most MPI implementations. The MPE and Pablo trace libraries are designed to work with any MPI implementation, and explicit directions are given in the MPE documentation for use with specific vendor MPI implementations. However, on untested platforms there will undoubtedly be glitches. Because of platform dependencies in the dynamic instrumentation technology used by Paradyn, especially for threaded programs, implementation of some Paradyn features tends to lag behind on all except their main development platforms.

TAU provides a utility for converting TAU trace files to Vampir and SDDF formats. SDDF is intended to promote interoperability by providing a common performance data meta-format, but most other tools have not adopted SDDF. Except for TAU and Vampir, tool interoperability has not been achieved. However, several of the tools make use of the PAPI cross-platform interface to hardware performance counters and make use of hardware performance data in their performance displays.

DEEP/MPI, MPE/Jumpshot, and TAU all support mixed MPI+OpenMP programming, as will the next version of Vampir. The MPE and Pablo trace libraries, when used with MPICH, can generate trace files for heterogeneous MPI programs. Paradyn supports heterogeneous MPI programs with MPICH. Pablo PCF and Autopilot have been installed and tested with Globus. Although profiling of MPI I/O operations is possible with several of the tools, the only tools that explicitly address MPI I/O performance analysis are the Pablo I/O analysis tools.

4 Conclusions

Considerable effort has been expended on research and development of MPI performance analysis tools. A number of tools are now available that allow MPI programmers to easily collect and analyze performance information about their MPI programs. We hope that this review will encourage further development and refinement of these tools as well as increase user awareness and knowledge of their capabilities and limitations. Detailed reviews of these and other tools, along with testing results and screen shots, will be available at the National High-performance Software Exchange (NHSE) web site at http://www.nhse.org/. The current review is an update to a similar review conducted four years ago that has been widely referenced [4].

References

[1] R. Berrendorf and H. Zeigler, *PCL -- the Performance Counter Library*, Version 2.0, September 2000.
[2] S. Browne, J. J. Dongarra, N. Garner, G. Ho and P. Mucci, *A Portable Programming Interface for Performance Evaluation on Modern Processors*, International Journal of High Performance Computing Applications, 14:3 (Fall 2000), pp. 189-204.
[3] S. Browne, J. J. Dongarra, N. Garner, K. London and P. Mucci, *A Scalable Cross-Platform Infrastructure for Application Performance Optimization Using Hardware Counters*, *Proc. SC'2000*, Dallas, Texas, November 2000.
[4] S. Browne, J. J. Dongarra and K. London, *Review of Performance Analysis Tools for MPI Parallel Programs*, NHSE Review, 3:1 (1998), http://www.nhse.org/NHSEreview/.
[5] H. Brunst, M. Winkler, W. E. Nagel and H.-C. Hoppe, *Performance Optimization for Large Scale Computing: The Scalable VAMPIR Approach,, International Conference on Computational Science (ICCS2001) Workshop on Tools and Environments for Parallel and Distributed Programming*, San Francisco, CA, May 2001.
[6] B. Buck and J. K. Hollingsworth, *An API for Runtime Code Patching*, Journal of High Performance Computing Applications, 14:4 (2000), pp. 317-329.
[7] I. Foster and N. Karonis, *A Grid-Enabled MPI: Message Passing in Heterogeneous Distributed Computing Systems, Proc. SC '98*, November 1998.
[8] I. Foster and C. Kesselman, *The Grid: Blueprint for a Future Computing Infrastructure*, Morgan Kaufmann, 1998.
[9] W. Gropp, S. Huss-Lederman, A. Lumsdaine, E. Lusk, B. Netzberg, W. Saphir and M. Snir, *MPI: The Complete Reference, Volume 2 -- The MPI-2 Extensions*, MIT Press, 1998.
[10] Kuck and Associates, Inc., *The GuideView performance analysis tool*, http://www.kai.com/.
[11] H. Simitci, *Pablo MPI Instrumentation User's Guide*, Department of Computer Science, University of Illinois, June 1996.
[12] Y. Zhang, *A User's Guide to Pablo MPI I/O Instrumentation*, Department of Computer Science, University of Illinois, May 1999.