Review of Performance Analysis Tools for MPI Parallel Programs

http://www.cs.utk.edu/~browne/perftools-review/review.html

Shirley Browne
Jack Dongarra
Kevin London

This work was partially supported by the DoD High Performance Computing Modernization Program, CEWES Major Shared Resource Center, through Programming Environment and Training (PET), and by the National Aeronautics and Space Administration under Grant NAG 5-2736, as part of the National High-performance Software Exchange (NHSE) project.

1.0 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.

For this review, we have investigated 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 written in Fortran or C. MPI, which stands for Message Passing Interface, is a standard interface for the message passing model of parallel programming [1,4]. Some of these tools also support F90, C++, and/or HPF, or are language-independent, but we only evaluated their support for Fortran and C with MPI.

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 investigated the following tools:

- AIMS - instrumentors, monitoring library, and analysis tools
- MPE logging library and Nupshot performance visualization tool
- Pablo - monitoring library and analysis tools
- Paradyn - dynamic instrumentation and run-time analysis tool
- SvPablo - integrated instrumentor, monitoring library, and analysis tool
- VAMPIRtrace monitoring library and VAMPIR performance visualization tool
- VT - monitoring library and performance analysis and visualization tool for the IBM SP

We restricted our review to tools that are either publically or commercially available and that are being maintained and supported by the developer or vendor. To give our review continuity and focus, we followed a similar procedure for testing each tool and used a common set of evaluation criteria.

For tools not already installed by the vendor, we built and installed the software using the instructions provided. If we ran into problems with the installation, we contacted the authors for help. If a binary distribution was provided, we tried that first. Otherwise, we obtained the source and compiled it. After the software was installed successfully, we worked through any tutorial or examples that were provided so that could become familiar with the tool. Finally, we attempted to use the tool to analyze the following test programs:

- `CMPI-rbsor`, a parallel red black SOR code, written in C with MPI, distributed with SvPablo as an example
- `NPB 2.2 SP`, a 3D multi-partition algorithm for the solution of the uncoupled systems of linear equations resulting from Beam-Warming approximate factorization, written in Fortran 77 with MPI, distributed as part of the ParkBench suite

Our set of evaluation criteria consisted of the following:

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

Our evaluation of these criteria was admittedly subjective and qualitative. Because the tools are somewhat different in their approaches, it did not seem possible or useful to carry out a quantitative comparison. Thus we used these criteria to give our reviews a common focus, rather than to do a side-by-side objective comparison.

For robustness, we expected 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 looked 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 being able to say where to put the trace file, an API for turning tracing on/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 displays, 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 looked for whether the tool was multi-hosted or was easy to build for a new host, and what MPI implementations and languages it could handle.

For versatility, we looked 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.

2.0 Detailed Reviews

2.1 Automated Instrumentation and Monitoring System (AIMS)

URL	http://science.nas.nasa.gov/Software/AIMS/
Version	3.7
Languages	C, Fortran 77
Platforms	IBM SP2 with IBM MPI or MPICH Sun, SGI, and HP workstations with MPICH SGI Power Challenge with SGI MPI

Overview

AIMS is a software toolkit for measurement and analysis of Fortran 77 and C message-passing programs written using the NX, PVM, or MPI communication libraries [4]. In addition to the platforms and languages listed above, a version of AIMS supporting HPF is being developed jointly with the Portland Group. The developers of AIMS are working on a port for the SGI Origin 2000, and ports to other platforms are being considered. We tested AIMS with our two test programs CMPI-rbsor and NPB 2.2 SP on an IBM SP2 running AIX 4, Sun Sparcs running Solaris 5.5.1, an SGI workstation running IRIX 6.3, and an SGI Power Challenge Array running IRIX 6.2. For each test program, we instrumented the source code; compiled, linked, and ran the instrumented code; and analyzed the resulting trace files.

AIMS documentation includes a Users' Guide in HTML format that is accessible from the AIMS Web page, as well as README files and man pages included with the software distribution. The Users' Guide gives step-by-step instructions on how to instrument, run, and analyze your application program, as well as a clear explanation of the various features and options for each of the AIMS components. The documentation is somewhat incomplete, however, because for example the batch instrumentor and the User Marker and User Block instrumentation constructs are not mentioned in the Users' Guide or anywhere else that we could find. An example program is provided, with C and Fortran versions for both PVM and MPI, with a README file that can serve as a quick start guide to using AIMS. There is also on-line help available for the `xinstrument` and `VK` GUIs that explains the various menu options and views. This on-line help is good as far as it goes, but is incomplete for `xinstrument` because it does not explain all of the available features such as User Marker and User Block.

An Installation Guide is provided as an appendix to the Users' Guide, and installation instructions are also included in the software distribution. The major installation task consists of editing the top-level Makefile with system-specific definitions. Clear instructions are given on how to do this, and sample Makefiles are provided for different platforms. One minor problem was that choosing the SP2MPI architecture in the Makefile turns out to imply Portable Batch System (PBS). For an SP2 without PBS, we had to use MPI instead for the architecture. On the Sun Solaris platform, we did have to edit one source code file and change the Makefile in the monitor directory, but other than that installation proceeded smoothly on all three of our test platforms. For the IBM SP2, we had to replace the `collecttrace` and `collectstat` scripts that collect performance data files from remote nodes with scripts that worked on our system which does not use PBS. Initially we could not get AIMS to work with vendor MPI on our SGI Power Challenge Array test platform, which is at CEWES pca1.hpc.army.mil, because the only versions of SGI MPI installed on that machine are n32-bit and 64-bit, but AIMS would only compile and link with 32-bit. However, an email exchange with the authors produced a solution, which was to compile all of AIMS except for the monitor with the -32 flag, and to compile the monitor with -64.

AIMS consists of three major components:

1. A source code instrumentor, called `xinstrument`, that inserts calls to AIMS performance monitoring routines into the user's application. Although it is not documented in the Users' Guide, there is also a command-line instrumentor, named `batch_inst`, the usage for which gets printed when one types just the command name.
2. A runtime performance monitoring library that consists of a set of monitoring routines that measure and record various aspects of program performance.
3. A set of tools that process and display the performance data.

The reason AIMS developers chose to parse the application source code and insert calls to the AIMS monitoring routines, rather than using the MPI profiling interface, are so that they can provide source code click back capability in the analysis GUI and perform more detailed analysis than would be possible using the MPI profiling interface. In the future, the AIMS developers anticipate tighter integration with compilers that will generate the instrumentation automatically, and work is underway with the Portland Group to integrate AIMS with the Portland Group HPF compiler.

Source Code Instrumentation

Source code can be instrumented using either the `xinstrument` graphical user interface or the `batch_inst` command-line instrumentor. The AIMS instrumentation is rather fragile with respect to the source language, which must be standard Fortran 77 or ANSI C. Free-format Fortran is poorly supported, and most F90 constructs are not parsed correctly. The instrumentors also have trouble with include files.

The `batch_inst` command-line instrumentor is not mentioned in the AIMS Users' Guide. However, it's usage is printed if one types just the command name without any arguments.

The `xinstrument` GUI allows the user to select specific source code constructs to be instrumented. `xinstrument` is quite flexible in allowing the user to specify what constructs should be instrumented. The default is to instrument all communication constructs. The user may also select All Subroutines, All I/O, or Enable by Type. Enable by Type allows the user to select particular constructs that will be instrumented in all loaded files. Even more selective instrumentation may be done by pointing and clicking on particular constructs in Contruct Tree diagrams to select particular constructs in particular files. Because the AIMS instrumentation API is not meant to be called by the user, there does not appear to be a way provided to turn tracing on and off under program control during program execution.

Either Monitor Mode, which generates a trace file, or Statistics Mode, which generates a much smaller file of summary statistics, may be selected from `xinstrument`. The `batch_inst` command-line instrumentor does not provide a Statistics Mode option. However, the user can edit the `AIMS.monrc` file directly to turn Monitor and Statistics modes on or off, regardless of whether `AIMS.monrc` is generated by `xinstrument` or `batch_inst`. Because `AIMS.monrc` is read by the AIMS monitor at run time, the modes can be changed without reinstrumenting the application source code.

Another file created by the instrumentors is the `APPL_DB`, or application database, file that stores information about the static structure of the application. The analysis tools use this information to relate traced events to source code constructs.

Another file produced by the instrumentors is a profile file consisting of a set of flags, once for each construct in the application database. The user can edit this file directly to change the instrumentation on a source file. Again, because this file is read by the monitor at run time, changes can be made without reinstrumenting the application source code. The profile file can also be changed from the `xinstrument` GUI.

The formats of all the files generated by the instrumentors and of the trace and statistics files produced when an instrumented application is run are documented in an appendix to the Users' Guide.

Where the final merged trace file is put can be set from `xinstrument` or by editing the `AIMS.monrc` file.

Where the intermediate per-process trace files get put can be set by setting the `AIMS_TMP_DIR` environment variable, although this does not appear to be documented in the Users' Guide and we found out only by asking the authors.

Performance Data Generation

After the instrumented application source files are compiled and linked with the AIMS monitor library, they can be run in the usual manner to produce trace and/or statistics files. AIMS generates separate files for each process and then automatically collects and merges the per-process files to create a single trace or statistics file.

We used the default instrumentation (communication constructs only) to instrument the CMPI-rbsor test program using `xinstrument` and the NPB 2.2 SP test program using `batch_inst`. On the SP2 and SGI PCA platforms, we then compiled and linked with the vendor MPI and AIMS monitor library. On Solaris, we used MPICH 1.0.13 (AIMS would not compile with MPICH 1.1). On the SP2, we at first had a problem when monitoring NPB2.2 SP with running out of tmp space for sorting the trace file, but were successful after setting AIMS_TMP_DIR to a directory with enough space. The size of the trace files produced was 78K for CMPI-rbsor and 25M for NPB 2.2 SP.

A problem we ran into with using the AIMS monitor on our SP2 was with the temporary TRACE and STAT files. These have the same name for different users, so when we are running at the same time, or if a program fails and leaves files lying around, name conflicts can cause problems. A solution would be to append the user's uid number to the filename. The authors have promised to fix this problem in the next release.

Trace file Analysis

Trace files can be analyzed using the View Kernel (VK) and the `tally` statistics generator. VK has VCR-like controls for controlling tracefile playback. The user can also set breakpoints by time or on specific source code constructs. VK has two views that are relevant to MPI programs -- a timeline view called OverVIEW and a view called Spokes that animates messages passed between tasks while showing the state of each task.

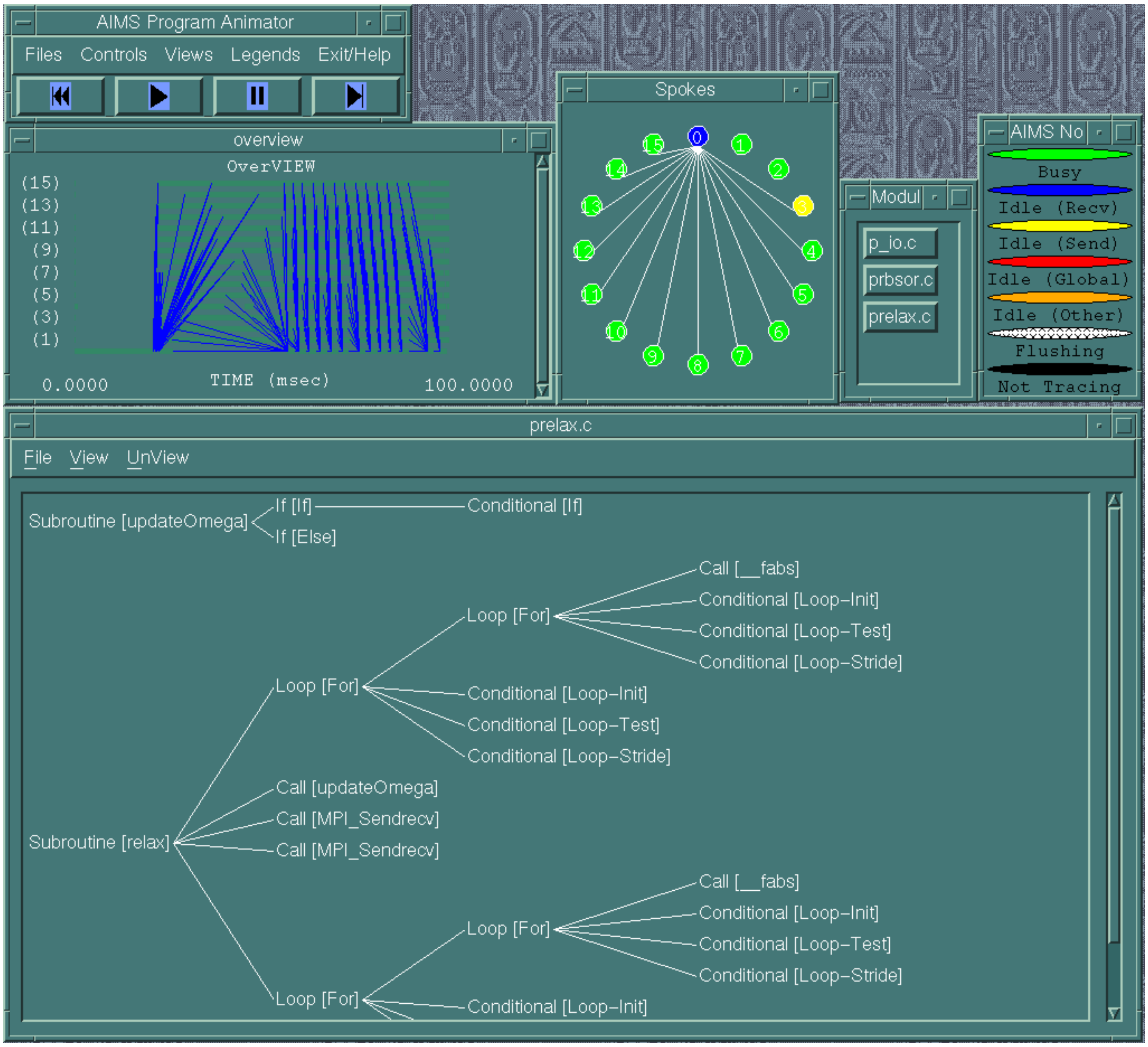
In the OverVIEW, each process is represented by a horizontal bar, with different colors for different instrumented subroutines and white space to indicate blocking due to a send or receive. Messages between processes are represented by lines between bars. Both bars and message lines can be clicked on for additional information, including source code click back to the line that generated the event.

The OverVIEW display can be toggled to show two additional views: I/OverVIEW for I/O activity and MsgVIEW for message activity. In these views, heights of the bars are used to represent size of I/O or messages, respectively.

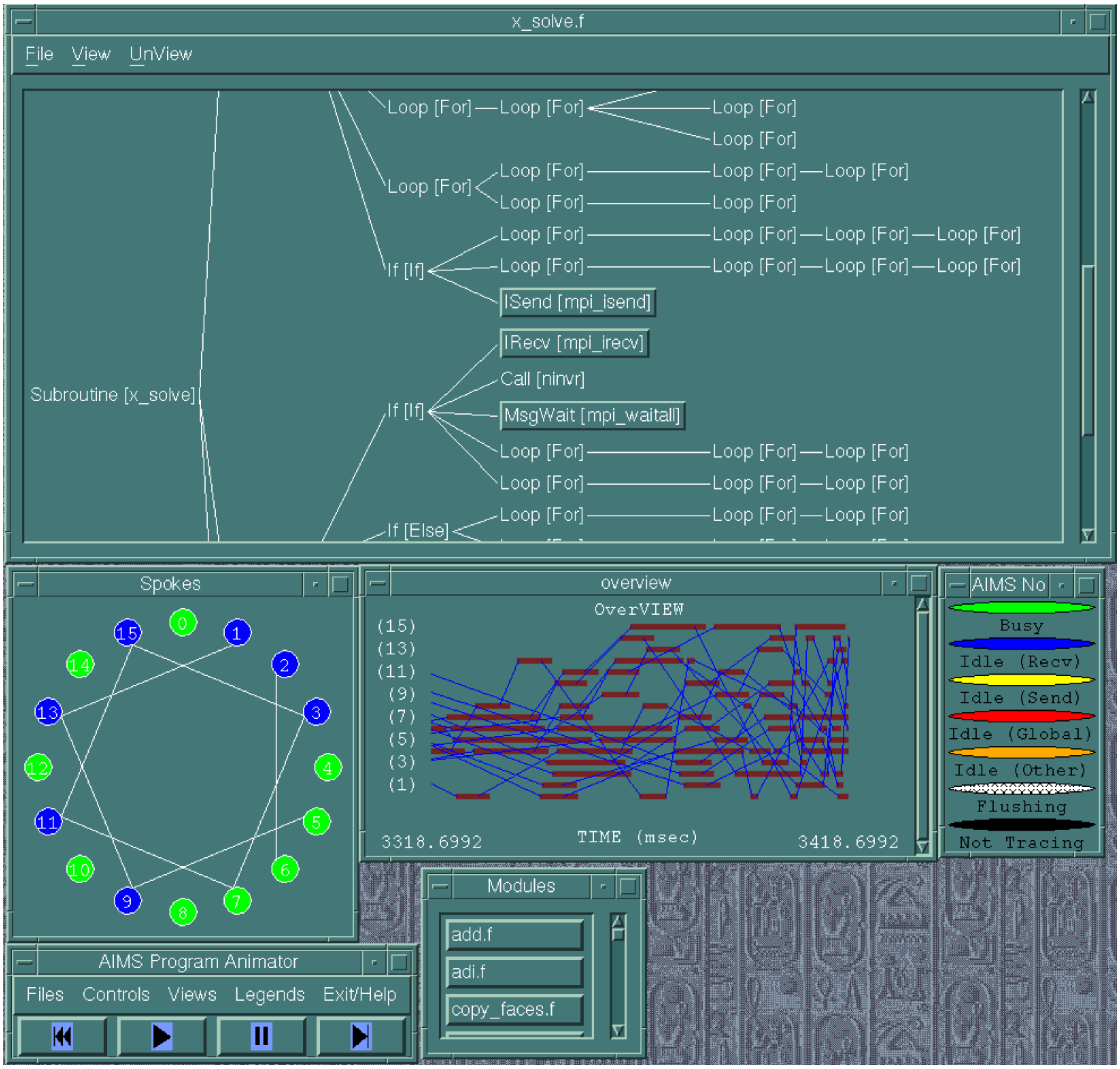
We were able to get the Spokes view to come up on our SGI IRIX 6.3 workstation, but not on any of our other test platforms for any of our test programs.

Although the playback of the trace file can be controlled with the VCR controls, and the time range of the timeline can be adjusted, there does not appear to be any way to scroll backward or to zoom in this view.

See the snapshots of VK displays of tracefiles produced for our test programs (These trace files were produced on the SP2 but analyzed on an SGI workstation):



CMPI-rbsor on the IBM SP2



NPB 2.2 SP on the IBM SP2

`tally` reads a trace file and generates resource-utilization statistics on a node-by-node and routine-by-routine basis. `tally` also reports the percentage of execution time spent in communication. The output from `tally` can be used as input to statistical analysis packages such as Excel. See Appendix A for the `tally` output for the NPB 2.2 SP test program (CLASS=A, NPROCS=16).

An interesting and unique feature of AIMS is that the time during which buffers are being written out to disk appears in the trace display or in the tally output.

Comparing the `tally` output NPA 2.2 SP code on the IBM SP2 with output from other tools, such as gprof, revealed a bug in how tally processes trace records. Tally showed that all nodes were spending a considerable amount of time in Send Blocking and Recv Blocking states, with five to six times more Send Blocking than Recv Blocking. Correspondence with the authors followed by further investigation on their part revealed that `tally` currently always reports `mpi_waitall` as Send Blocking, even though this function generates both Send Blocking and Recv Blocking. This problem will be fixed in the next release.

Evaluation Summary

We now evaluate AIMS according to the evaluation criteria given in the Introduction.

The AIMS software seems fairly robust and catches and handles most user errors with appropriate error messages. We tried most of the features of `xinstrument` and VK and found them to work correctly, with the exception of the Spokes view in VK which we could only get to come up on an SGI workstation but not on our other test platforms.

Although AIMS does not offer a way to turn tracing on and off, because AIMS by design has no trace library API, features are provided for limiting the number of contructs and parts of the program that are instrumented, so as to reduce the size of the generated trace files and focus on interesting events. For display scalability, it would be helpful to be able to zoom on the timeline display and to display subsets of processes in this view.

Although AIMS currently runs on only a few platforms, efforts are underway to port it to additional platforms. A CEWES-funded effort at the University of Tennessee is porting AIMS to DoD MSRC platforms. An ASCI effort at LANL is porting AIMS to ASCI platforms. Also, although it is necessary to port the monitor library, the analysis programs can be run on a different platform because the performance data files are in ASCII and thus portable.

AIMS is very versatile in the type of instrumentation it allows, with the instrumentation done automatically without the user having to manually change the source code. The ability for the user to edit the AIMS database files for an application to change run time monitoring behavior also adds to versatility.

As for usability, AIMS has both a reasonable set of defaults for beginners and more advanced options for more expert users. The documentation is well-written and just needs to have a few things added that were left out. The developers have been very responsive to our questions, and they incoporated most of our comments and suggestions on an earlier release into the current release. AIMS is installed and in use at several sites, including Cornell and LLNL, and more are likely to follow soon.

2.2 nupshot

URL	http://http://www.mcs.anl.gov/mpi/mpich/
Version	1.1, April 1997
Languages	Language-independent
Supported platforms	Any that have the correct Tcl and Tk libraries
Tested platforms	SGI PCA, IBM SP, Sun Solaris

Overview

Nupshot is a performance visualization tool that displays logfiles in the alog format or the PICL version 1 format. The version of nupshot currently distributed with MPICH is the third version and is written in a combination of Tcl and C. The first version, called upshot, was written in entirely in C, and the second version, called nupshot, was all Tcl. The alog format was developed along with upshot and is documented in the alog.h file in the nupshot distribution. The PICL format was developed for use with the Portable Instrumented Communication Library (PICL). The PICL version 1 format is documented in the picl.h file in the nupshot distribution. We only tested the alog format in this review.

The user can create customized logfiles for viewing with nupshot by inserting calls the various MPE logging routines in his/her source code. A standard MPI profiling library is also provided that automatically logs all calls to MPI functions. MPE logging and nupshot are part of the MPICH distribution and come with their own configure scripts to set them up. MPE logging and nupshot are also distributed separately from MPICH from Parallel Tools Library (PTLIB). Nupshot needs to have the TCL 7.3 and TK 3.6 libraries to compile it succesfully. We tested nupshot with two test programs CMPI-rbsor and NPB 2.2 SP on Sun Sparcs running Solaris, a SGI Power Challenge Array running IRIX 6.2, and on an IBM SP. For both the SGI PCA and the IBM SP, we were able to get the MPE logging library working with vendor versions of MPI. For each test program, we linked to the MPE logging library and ran the instrumented code to produce a trace file. Then we analyzed the resulting tracefile with nupshot.

Nupshot documentation is minimal. It includes a short Readme file on how to compile the code, and a short todo list of things that the author would like to implement in future releases of nupshot. It does say that more documentation is on the way, but it had not been released by the date we did this review. However the controls are self-explanatory and it was easy to learn even with the lack of documentation. The MPE logging library is described briefly in the MPICH User’s Guide. Further documentation is provided in the MPE man pages.

The Unified Trace Environment (UTE) for generating tracefiles for MPI and MPL applications on the IBM SP, and a version of nupshot that analyzes tracefiles produced by UTE, have been developed by Eric Wu of IBM. However, there are no plans to officially support or do further development on these products. The IBM version of nupshot includes a help button with a brief help text which is not present in the MPICH version. The IBM version also has source code clickback capability which is not present in the MPICH version. UTE and IBM nupshot are installed on our SP2, along with a UTE User’s Guide, so we tried them. The IBM version of nupshot can also analyze alog files produced by the MPE logging library, although the source code clickback in this case only brings up the file, but not the current line location. We were able to link our test programs with the UTE trace library and to merge the resulting per-process trace files using utemerge. However, we were unable to get the ute2ups utility to convert this trace file to the ups format that can be analyzed by IBM nupshot. Our request for help with this problem was answered by Eric Wu, however, and he is attempting to solve the problem.

Installation

Installation of the MPICH version of nupshot was very easy. Nupshot comes with a configure script that will set up the makefile for you, and for us this compiled out of the box. The only problem is you must have TCL 7.3 and TK 3.6 to get it to compile. If you have a newer or older version of TCL or TK, it will not compile correctly. Because the configure script assumes MPICH, to get the MPE logging library working with IBM and SGI MPI required a few rounds of questions and answers to the MPICH support address, but ended in success.

Trace File Generation

To link to the MPE MPI profiling library, one gives the argument `-llmpi` to the linker, ahead of `-lpmpi` (if present) and `-lmpi`. The resulting profiled program then generates a logfile of timestamped events in the alog format. During execution, calls to `MPI_Log_event` are made to store events of certain types in memory, and these memory buffers are collected and merged during `MPI_Finalize`. during execution, `MPI_Pcontrol` can be used to suspend and restart logging operations.

Trace File Analysis

Nupshot includes three visualization displays: Timelines, Mountain Ranges, and state duration histograms. The Timelines and Mountain Ranges views can be toggled on and off using the pulldown Display menu. A state duration historgram display for each state can be invoked by clicking on the button for that state on the main display.

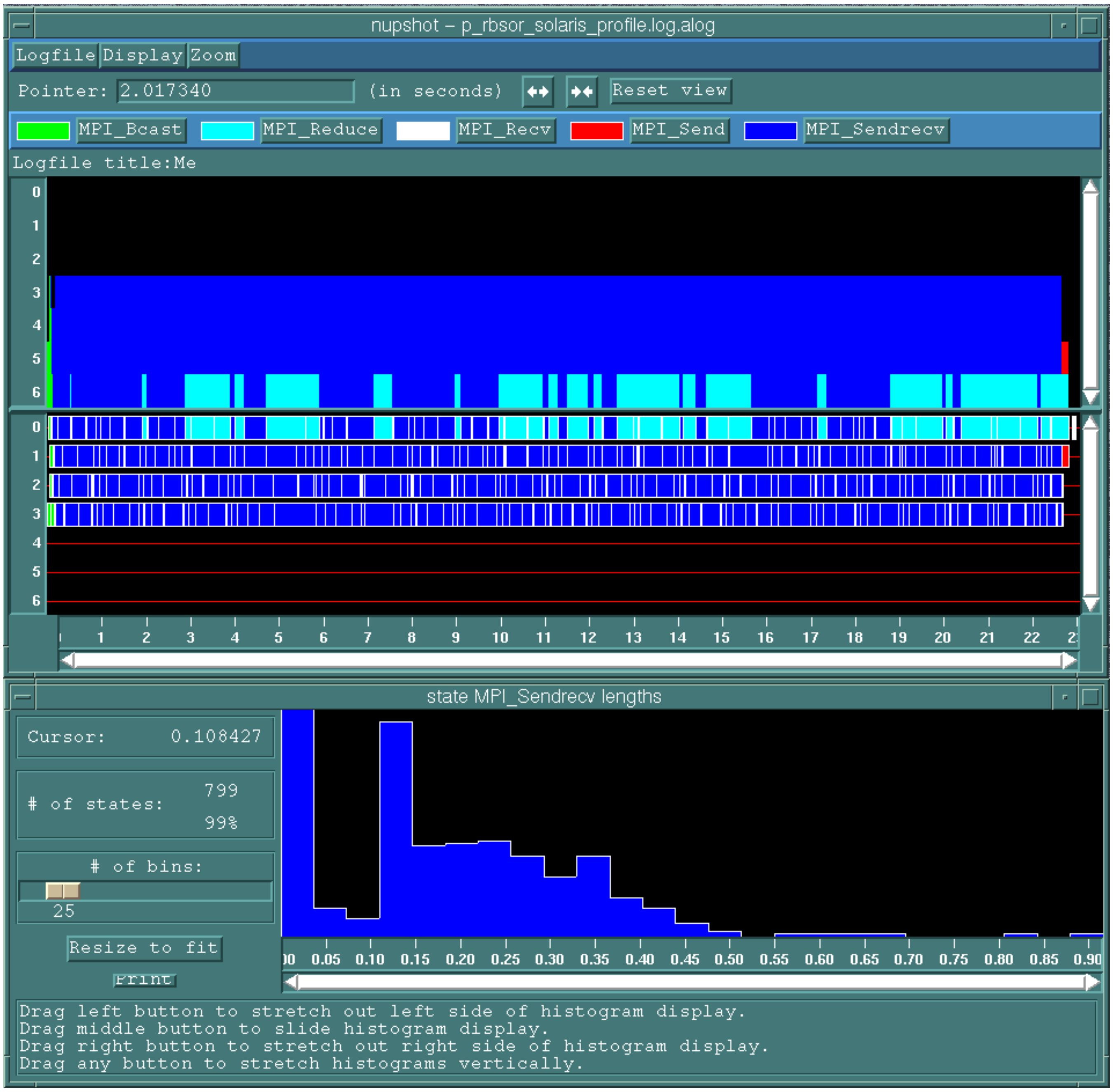
The Timelines view is present initially be default. Each line represents states of the process whose identifier appears along the left edge. Clicking on a bar with the left mouse button brings up an info box that gives the state name and duration. You must hold the left mouse button down and when you release the info box disappears, so you can't have several of these boxes open at once. Messages between processes are represented by arrows. Clicking on a message line does not have any effect. The Mountain Ranges view gives a color-coded historgram of the states that are present at a given time.

Time elapsed since the first event is shown along the bottom edge of the display, with the scale based on the assumption that the units in the timestamp field in the logfile are microseconds. As you move your pointer across the display, the Pointer box shows your position in the tracefile. One can zoom in or out to stretch the Timeline and Mountain Ranges views along the horizontal axis. The initial scale can be restored by the Reset button. When zoomed in, the entire width of the display can be scrolled.

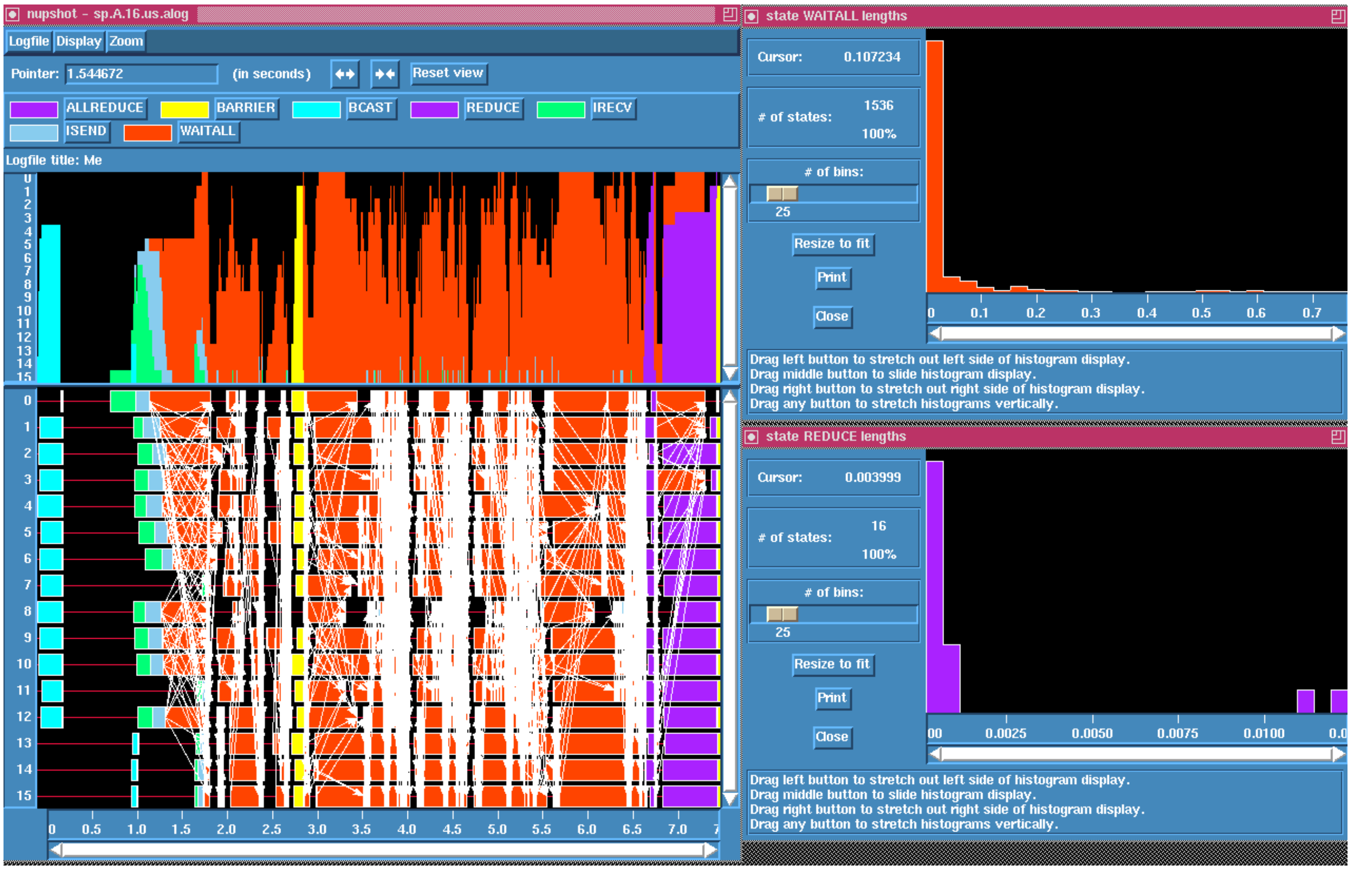
The state durations historgram views are accessed by menu button bars, and they pop up according to which routines were found to be traced. In our NPB2.2 SP example we had the following buttons: MPI_ALLreduce, MPI_Barrier, MPI_Bcast, MPI_Reduce, MPI_Irecv, MPI_Isend, and MPI_Waitall. In our CMPI-rbsor example we had the following buttons: MPI_Bcast, MPI_Reduce, MPI_Recv, MPI_Send, and MPI_Sendrecv. When you click one of these buttons it brings up a window that will show you a histogram view of the chosen call. You can zoom in and out as well as move along the timeline.

A nice feature is the print button on each of the windows that lets you print the image to a postscript file or directly to a printer. However, the images don't show up very well on a black and white printer, because the background color is black. However, if one knows Tcl, one can change this in the nupshot source code.

See the following snapshots of nupshot displays of tracefiles produced for our test programs:



CMPI-rbsor on a Sun Ultrasparc running Solaris



NPB 2.2 SP (4 iterations) on an IBM SP2

Evaluation Summary

nupshot crashes occasionally -- for example, with a segmentation fault when we were attempting to configure the views, although we did not figure out exactly what sequence of actions caused this. We found that the algorithim for zooming in has a slight glitch. If you have a large length right next to calls with very small lengths, when you zoom in to see those calls, the large call will disappear at times. Overall nupshot seems fairly robust, however. We did not experience any problems in producing trace files with the MPE logging library.

nupshot is easy to install and its usage is fairly self-explanatory. Its ability to use alog or PICL format makes it interoperable with any trace library that produces one of those formats. nupshot is a good tool for quickly getting an overview of what your program is doing. To do an in depth analysis of your program, however, you might want a tool with more views and features than this tool offers.

MPE logging and nupshot appear to be fairly easy to port to existing MPI platforms, except for the Tcl/Tk version problem with nupshot, which specifically requires Tcl 7.3 and Tk 3.6.

As with other MPE profiling libraries, once can turn off tracing by calling `MPI_Pcontrol(0)` and re-enable tracing by calling `MPI_Pcontrol(1)`. In this way, once can control the size of the generated trace file. However, MPE logging does not appear to provide any way to control where the log file is placed -- it ends up in the same directory as the executable on process zero with a .alog suffix. The nupshot display does not provide any way to aggregate or filter events. However the zooming and scrolling capabilities do provide some scalability for data display.

The MPE logging library and nupshot are somewhat versatile in that user-defined events and states can be generated and displayed by inserting calls to MPE logging library in one's source code.

2.3 Pablo Performance Analysis Environment

URL	http://www-pablo.cs.uiuc.edu/Projects/Pablo/
Version	5.0

svPablo

Languages	ANSI C, HPF
Platforms	SGI running IRIX 6 Sun running Solaris

TraceLibrary

Languages	Language-independent
Platforms	MPICH 1.0.13 on Convex Exemplar, Intel Paragon, Unix workstations, IBM SP

IO Analysis

Languages	C, Fortran 77
Platforms	Paragon with OSF/1 SGI running IRIX 6 Sun Solaris

Analysis GUI

Platforms	Built and tested on Sun Solaris

Overview

Pablo 5.0 consists of several components for instrumenting and tracing parallel MPI programs and for analyzing the trace files produced by the instrumented executables. Interoperation between the various Pablo components is based on the Pablo Self-Defining Data Format (SDDF) used for the performance trace files. The Pablo tracing components output trace files in SDDF, and the Pablo analysis components take SDDF files as input. Converters are provided with the Pablo distribution for converting other formats, such as PICL and AIMS, to SDDF, so that trace files produced by other trace libraries can be analyzed using the Pablo analysis tools. There is also a converter available from Cornell Theory Center for converting IBM VT trace files to SDDF.

Instrumentation

The instrumentation library consists of a basic trace library with extensions for procedure tracing, loop tracing, NX message passing tracing, I/O tracing, and MPI tracing. The basic trace library provides the functions `traceEvent`, `countEvent`, and the pair `startTimeEvent` and `endTimeEvent`. An event ID passed as an argument to these functions specifies the type of event that is being traced. The various extensions to the Pablo instrumentation library provide wrapper functions for management of event ID's for the various event types.

The Pablo trace library consists of portable and system-dependent portions. System-dependent source code is provided for Convex Exemplar, Intel Paragon, Unix workstations, and IBM RS6000 and SP. Procedure and loop tracing instrumentation must be done manually by inserting calls to TraceLibrary routines into the application source code.

I/O instrumentation requires changes to the application source code. With C programs, the user may either replace the standard I/O calls with their tracing counterparts or may define IOTRACE so that the pre-processor replaces standard I/O calls with their tracing counterparts. For I/O requests that are not implemented as library calls, for example the `getc` macro in C and Fortran I/O statements that are part of the language, I/O trace bracketing routines are provided that must be inserted manually in pairs around the actual I/O request. In addition to instrumenting the individual I/O calls or statements, the I/O trace initialization and termination routines must be called before and after calling any other I/O trace routines, respectively.

For MPI message tracing, the Pablo tracy library provides a profiled version of the MPI library. To trace MPI message events, the user need only link to this library and need not make any source code changes.

Trace File Generation

After compiling and linking with the TraceLibrary routines, the application executable can be run in the usual manner. In the case of a parallel MPI program, each MPI process outputs a trace file labeled with the process number. The user can insert a call to the Pablo TraceLibrary routine `Set TraceFileName()` immediately after the call to `MPI_Init()` to control where the trace file gets written. As with any MPI profiling library, tracing can be disabled by calling `MPI_Pcontrol(0)` and re-enabled by calling `MPI_Pcontrol(1).`

The per-process trace files can then be merged using the SDDF utility `MergePabloTraces`. The trace files are produced in binary format. The `SDDFconverter` utility can be used to convert a trace file to human-readable ASCII format.

The default mode of the Pablo instrumentation library is to dump trace buffer contents to a trace file, but it is possible to instead have trace data output sent to a socket -- e.g., for real-time analysis.

Trace File Analysis

SDDF trace files can be analyzed using the Pablo Analysis GUI. Trace files produced by the I/O instrumentation routines can also be analyzed by the Pablo I/O analysis command-line routines that produce plain-text tabular output. The command-line FileStats program is also provided that scans an SDDF file and reports the minimum and maximum values for each field and the total count of each record type.

The Pablo Analysis GUI is a toolkit of data transformation modules capable of processing SDDF records. The Analysis GUI supports the graphical interconnection of performance data transformation modules in the style of AVS to form a directed, acyclic data anlysis graph. By graphically connecting analysis and data display modules and then interactively selecting which trace data records should be processed by each data analysis module, the user specifies the desired data transformations and presentations. Expert users can develop and add new data analysis modules to the Pablo analysis environment. Although the Pablo documentation states that novice users will most likely load preconfigured data analysis graphs with a fixed set of data transformations and presentations, such preconfigured graphs are not included with the Pablo distribution of the Analysis GUI.

The Analysis GUI provides four primary module types out of which graphs can be built: data analysis, data presentation, trace file input, and trace file output. SDDF records flow through pipes connecting the modules when the graph is executed. The basic module set includes simple mathematical transforms (e.g, counts, sums, ratios, max, min, averages, trigonometric functions, etc.) as well as a set of synthesis modules that allow one to construct vectors and arrays from scalar input data. Data presentation modules include bar graphs, bubble charts, strip charts, contour plots, interval plots, kiviat diagrams, 2-D and 3-D scatter plots, matrix displays, pie charts, and polar plots.

We were able to do the tutorials with step-by-step instructions and get those graphs to work, but we were unable to devise a graph of our own to analyze the tracefile from our NPB2.2 SP test program, and our requests for help to the Pablo support address went unanswered. There is no assistance provided to the user by the Analysis GUI for debugging a graph that does not work.

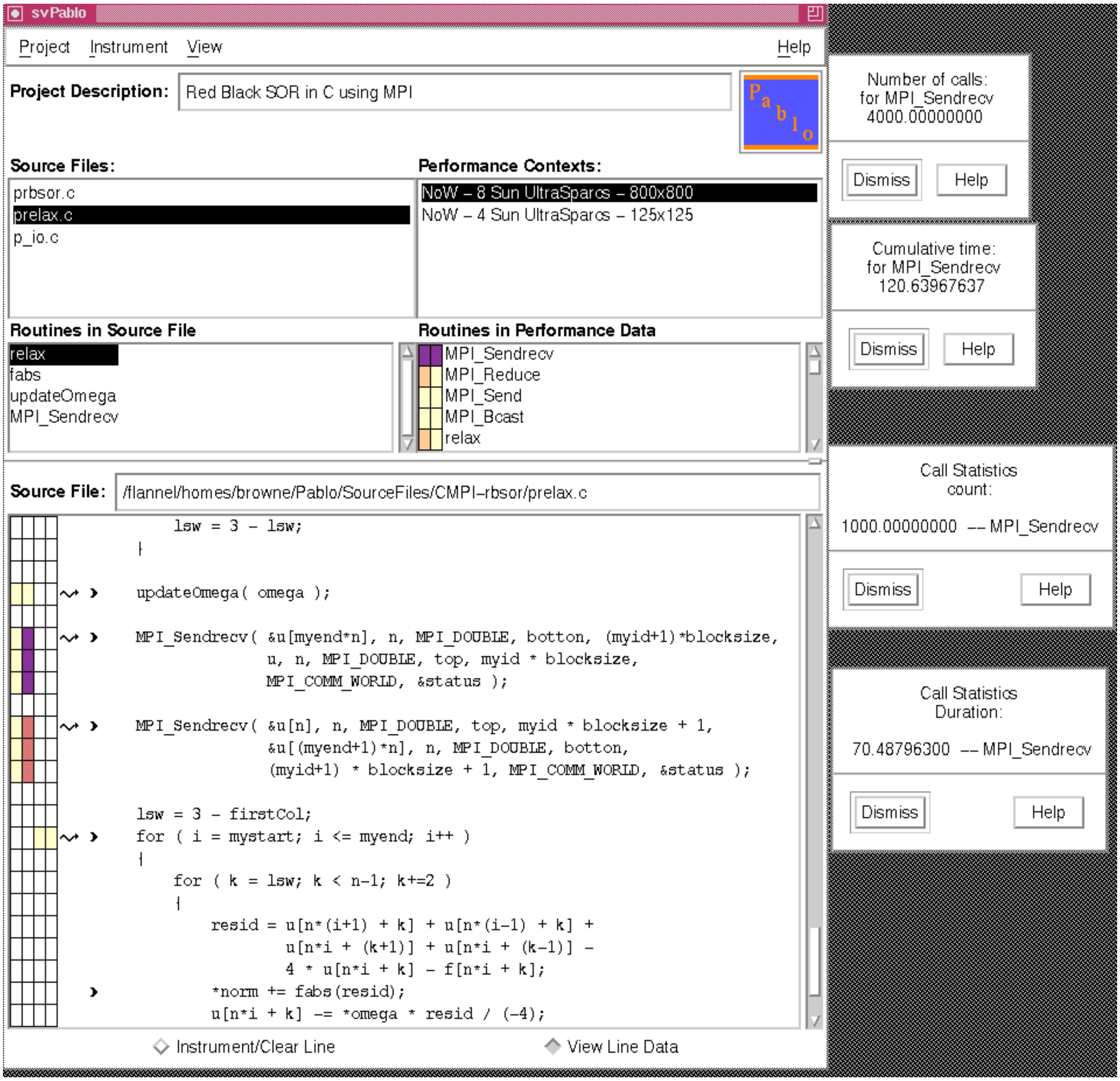
SvPablo

SvPablo, which stands for Source view Pablo, is a different approach from the other Pablo components in that the instrumentation, trace library, and analysis are combined in the single SvPablo component. The SvPablo "project" concept provides a way to organize your source code as well as to collect and organize traces done in the past. The current release supports ANSI C programs and HPF programs compiled with the Portland Group HPF compiler `pghpf`. HPF programs are automatically instrumented by the `pghpf` compiler when the appropriate flags are used.

C programs can be instrumented interactively using the SvPablo GUI which allows selective instrumentation of outer loops and function calls. SvPablo generates an instrumented version of the source code which is then linked with the SvPablo trace library and executed in the usual manner. For a C MPI program, the per-process SDDF performance files must then be collected and combined using the SvPablo CCombine utility before the merged performance file can be analyzed using the SvPablo GUI.

After the performance file has been loaded, the SvPablo GUI allows the user to view performance summary statistics for each instrumented routine and loop. To the left of the name for each instrumented routine, SvPablo shows two color-coded columns summarizing the number of calls made to the routine and the cumulative time for the routine. Detailed statistical information about a routine can be seen by clicking the mouse buttons. Selecting a function name for which SvPablo has access to the source code displays the source code along with color-coded performance information for instrumented constructs. Again, clicking on the colored boxes displays more detailed information.

See the snapshot of SvPablo analysis for our CMPI-rbsor test program.



SvPablo analysis of CMPI-rbsor

Evaluation Summary

The Pablo documentation is thorough for each component, and examples are provided for all components. The Analysis GUI distribution includes a tutorial that leads the user through the steps needed to build an analysis graph with a series of examples of increasing difficulty. It would be helpful, however, to have an up-to-date overview document that describes how the various components relate to one another and how they can be used together. Some example analysis graphs for the Analysis GUI -- e.g., one each for analyzing programs instrumented with the I/O trace library and the MPI trace library, respectively, would also be very helpful. Some minor usability improvements could also be made -- for example, a reset feature that allowed the user to rerun a graph would be convenient, rather than having to delete and reload a graph to run it again.

SvPablo is fairly easy to use, once one gets acquainted with svPablo Projects and how they are structured, which is well-documented in the SvPablo Users' Guide. The Pablo Analysis GUI, on the other hand, has a steep learning curve for learning how to construct the analysis graphs. For this tool to be embraced by the typical application scientist, it probably needs to be supplemented with example graphs for the particular application area or programming model, at least to get them started. On the other hand, the Analysis GUI is unquestionably quite versatile in the kind of analysis it allows the user to do, since theoretically any type of view could be constructed from the toolkit provided and the ability to define new modules.

Concerning support, some of our questions to the Pablo email support address received prompt answers, while others went unanswered.

According to the Pablo documentation, all the Pablo components are intended to be usable for a wide range of parallel platforms and programming languages. This is not currently the case. SvPablo currently runs only on Sun and SGI workstations and SGI Power Challenge. We are attempting to port SvPablo to the IBM SP, but are finding that this will take more than trivial effort. SvPablo currently supports only the HPF and ANSI C languages, and HPF only with the Portland Group compiler, but support for Fortran 77 and Fortran 90 is planned. The Pablo Trace Library MPI component has been tested by the developers only with MPICH 1.0.13, and we have been unsuccessful so far in getting it to work with IBM MPIF on the SP. The Pablo Analysis GUI has only been tested on Sun Solaris, and we have not yet attempted to build it on other platforms. So although a major goal of the Pablo project is portability, this goal has not yet been achieved.

As for scalability, the documentation states (An Overview of the Pablo Performance Analysis Environment, Nov 92, p. 9; Pablo Instrumentation Environment User's Guide, Appendex A.1) that the Pablo trace library monitors and dynamically alters the volume, frequency, and types of event data recorded by associating a user-specificied maximum trace level with each event and substituting less invasive data recording (e.g., event counts rather than complete event traces) if the maximum user-specified rate is exceeded. However, it is unclear if these measures are taken automatically by the high-level trace library routines or if they must be explicityly called by the user at a low level. Tracing with the Pablo MPI TraceLibrary can be turned off and on by using calls to `MPI_Pcontrol`. Scalable data presentation is certainly possible with the Analysis GUI, although it is up to the user to implement this.

2.4 Paradyn

URL	http://www.cs.wisc.edu/paradyn/
Version	Release 2.0, September 1997
Languages	Fortran, Fortran 90, HPF, C, C++
Platforms	Sun SPARC (PVM version only) Windows NT on x86 IBM RS6000 and SP with AIX 4.1 or greater

Overview

The purpose of the Paradyn project is to provide a performance measurement tool that scales to long-running programs on large parallel and distributed systems and that automates much of the search for performance bottlenecks [2]. The Paradyn designers wished to avoid the space and time overhead typically associated with trace-based tools. Their approach is to dynamically instrument the application and automatically control the instrumentation in search of performance problems. Paradyn starts by looking for high-level problems (such as too much total synchronization blocking, I/O blocking, or memory delays), using only a small amount of instrumentation. Once a general problem has been found, more instrumentation is selectively inserted to find specific causes of the problem. The Performance Consultant module, which automatically directs the placement of instrumentation, has a knowledge base of performance bottlenecks and program structure so that it can associate bottlenecks with specific causes and with specific parts of a program.

The Paradyn tool has two parts: 1) the Paradyn front-end and user interface, and 2) the Paradyn daemons. The user interface allows the user to display performance visualizations, use the Performance Consultant to find bottlenecks, start and stop the application, and monitor the status of the application. The Paradyn daemons monitor and instrument the application processes.

The Paradyn documentation includes a User's Guide, Installation Guide, and Tutorial. The tutorial is oriented towards PVM, however, with the example being a PVM program. Also, in the User's Guide, one is not told what daemon to select for an MPI program, only that the default daemon "defd" is appropriate for most uses, and that "pvmd" is the choice for PVM programs. It turns out that "mpid" is the correct choice for IBM MPI programs run under POE, but we had to find this out through an email exchange with the developers. For developers who wish to expand or port Paradyn, developer's guides are provided that describe the Paradyn source code as well as the performance visualization and thread package APIs. The dyninst library that provides a machine-independent interface for runtime program instrumentation is also available separately. Dyninst is used in Paradyn but can also be used to build other performance tools, debuggers, simulators, and steering systems.

Paradyn requires several files from the TCL 7.5 and TK 4.1 packages. The necessary files are available from the Paradyn ftp site.

Program Preparation

The documentation says that future releases of Paradyn will be able to instrument unmodified binary files. The current release requires linking an application with the Paradyn instrumentation libraries. On an IBM AIX system, static linking is required. The link command given in the Paradyn User's Guide is several lines long but worked fine for our C and Fortran test programs. The application must be compiled with the -g flag because Paradyn needs the debugging information.

Run-time Analysis

Unlike the other tools we reviewed that do post-mortem analysis of trace files, Paradyn does interactive run-time analysis. In order for this type of analysis to be effective, the program must be fairly long running. We set our NPB 2.2 SP test program input data file to 400 iterations to try to achieve this effect.

Paradyn is designed to measure an application either by starting up the application processes and killing them upon exit, or by attaching to and detaching from running (or stopped) processes. However, attaching to a running process is currently implemented only on Solaris platforms, and Paradyn currently does not detach from an application but only kills it upon exit.

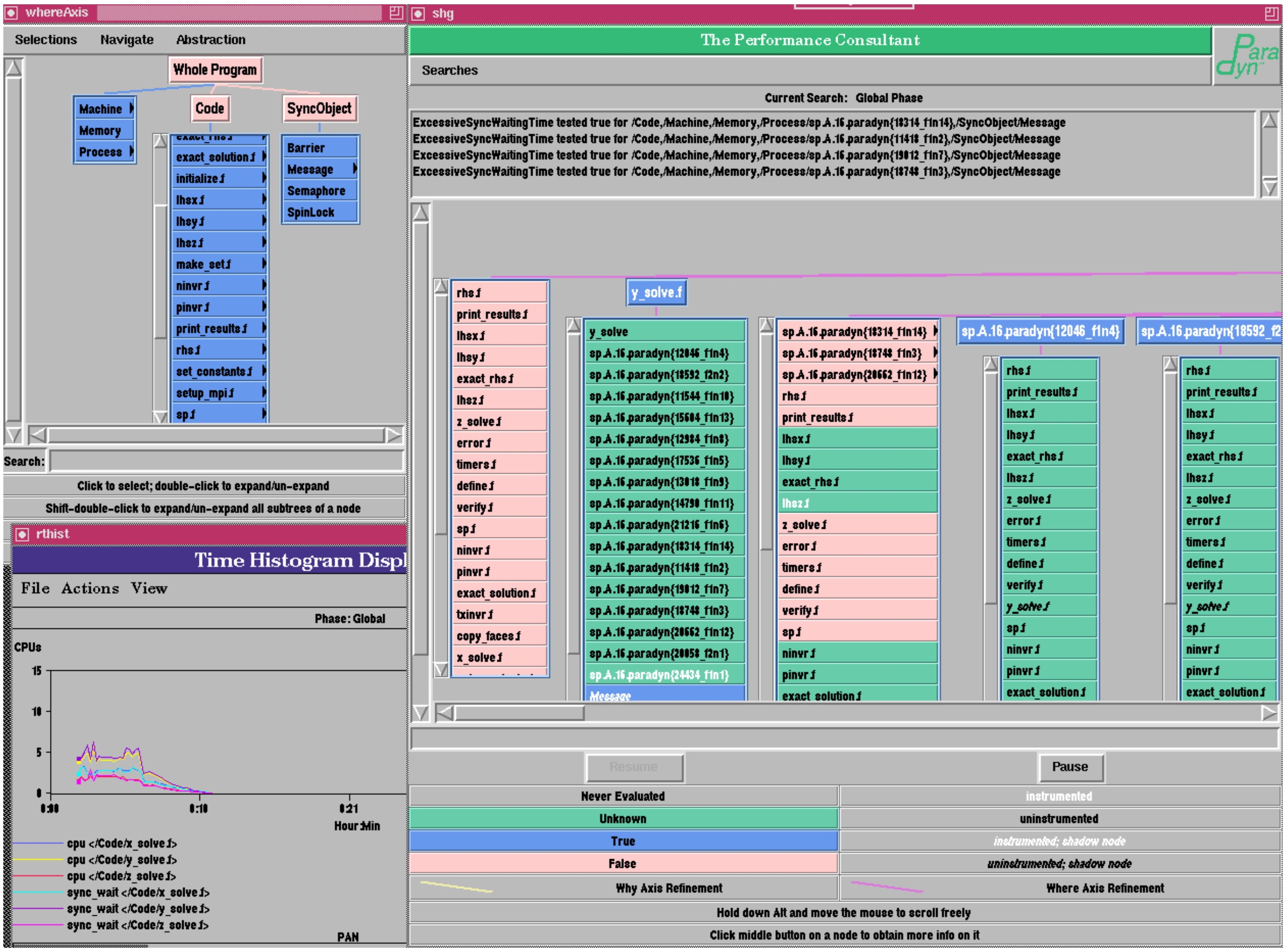
Paradyn uses the concepts of *metric-focus grid* and *time-histogram* for selecting, analyzing, and presenting performance data. A metric-focus grid is based on two vectors of information. The first vector is a list of performance metrics, such as CPU time, blocking time, message rates, I/O rates, or number of active processes. The second vector is a list of program components, such as procedures, processors, disks, message channels, or barrier instances. The cross product of these two vectors produces a matrix with each metric listed for each program component, and the user essentially selects metric-focus pairs from this matrix. The elements of the matrix can be single-valued (e.g., current value, average, min, or max) or time-histograms. A time-histogram is a fixed-size data structure that records the behavior of a metric as it varies over time.

After paradyn loads a program, it adds entries to the "Where Axis" window for resources such as files and procedures in the Code hierarchy, and processes and machines. Before running the program, the user can first define performance visualizations. These are defined in terms of metric-focus pairs. For example, the user might specify a histogram visualization with a metric of CPU time and a focus of a particular subset of processes executing a particular procedure. More than one metric-focus pair can be displayed in the same visualization. Alternatively or in addition, the user can start up the Performance Consultant. The Performance Consultant automatically enables and disables instrumentation for specific metric-focus pairs as it searches for performance bottlenecks. It displays a "Search History Graph", a graphical representation of the state of the search that shows how the Performance Consultant iteratively refines its search. By clicking the middle mouse button on any node in the search history graph, one can see a text string representation of the hypothesis associated with that node. Nodes that test true are colored blue, and refinements are made only on those nodes. The results of the search can be obtained by following the blue nodes from the root to a leaf node. One can then start a visualization to display performance data corresponding to a bottleneck found by the Performance Consultant.

Because Paradyn uses fixed-size data structures to store performance data, the granularity of performance data becomes coarser the longer the application runs. To obtain performance data at a finer granularity after a program has been running for some time, one can define a new *phase*. Phases are contiguous time intervals within an application’s execution. There are two types of phases: a *global phase* and one or more *local phases*. The global phase is the entire period of execution. A local phase can be started at any time and ends when a new local phase is started. At any time, the user can select performance data from the global phase or from the current local phase. The Performance Consultant can also be restricted to search just the current local phase.

In testing Paradyn, we tried both our CMPI-rbsor and NPB 2.2 SP test programs for four to sixteen processors. For up to twelve processors, we found that Paradyn worked fine. For more than twelve processors, however, we got failures about 80 percent of the time at various stages, with segmentation faults and core dumps. After corresponding with the developers, we found that the MPI version of Paradyn 2.0 for the IBM SP had not been tested on more than twelve processors. The developers said this scalability problem would be fixed in the next release.

See the following snapshot of using Paradyn to analyze the NPB 2.2 SP code for 400 iterations with 16 processors on an IBM SP2.



Paradyn analysis of NPB 2.2 SP

Evaluation Summary

For twelve or fewer processes, Paradyn seems fairly robust. For more than twelve processes, however, it experiences catastropic failure in about 80 percent of the runs we tried. However, there seems to have been fairly careful attention given to generating appropriate error messages in an error window. These messages are sometimes off the mark. For example, when we tried to run our NPB 2.2 SP example with the wrong number of processes, the error window told us "Could not find version number in instrumentation".

The MPI version of Paradyn currently works on the the IBM SP2 with IBM's POE MPI interface. The dyninst run-time instrumentation library currently works on SUN SPARC, HP RA-RISC, DEC Alpha, IBM Power2, and Intel x86 architectures. The developers report that they are planning to port Paradyn to the SGI Origin 2000 in the near future.

Paradyn is designed to be scalable to long running programs (hours or days) on large systems (thousands of nodes). As we found in our testing, however, the SP2 version of Paradyn currently scales only to twelve nodes. The developers have promised that this limitation will be removed in the next release. The Paradyn front end also using a large amount of memory, with the front end taking up over 27 megabytes when analyzing an execution of NPB 2.2 SP on nine processes. We would be interested to see what the memory usage is for larger numbers of processes.

Paradyn has an intuitive interface that is easy to use. The User's Guide clearly explains the various features available. We received prompt answers to our questions to the Paradyn developers. One disadvantage of the interface, however, is that the Performance Consultant graph can grow very large to the point where only a small portion of it can be viewed without scrolling.

Paradyn is very versatile in that it allows the user to select different focuses and views. It also provides an open interface for visualization, so that new visualizations can be added. However, Paradyn only works in interactive mode and cannot be used on batch processing systems.

2.5 VAMPIR -- Visualization and Analysis of MPI Resources

URL	http://www.pallas.de/pages/vampir.htm
Version	VAMPIR 1.0, VAMPIRtrace 1.5
Languages	Language-independent
Platforms	Cray T3D/T3E, Unicos DEC Alpha, OSF/1 Fujitsu VP 300/700, UXP/V Hitachi SR2201, HI-UX/MPP HP PA, HP-UX 9,10 IBM RS/6000, AIX 3, AIX 4 IBM SP-2, AIX 4 Intel Paragon, Paragon OS 1.5 NEC SX-4, S-UX 7 NEC EWS, EWS-UX4 NEC Cenju-3, Paralib/cj SGI, IRIX 5, IRIX 6 SGI Origin Series, IRIX 6 SGI PowerChallenge, IRIX 6 SPARC, Solaris 2.4 I86, Solaris 2.5 NEC EWS, EWS-UX 4 NEC SX-4, S-UX 6

Overview

VAMPIR is a commercial trace visualization tool from PALLAS GmbH. VAMPIRtrace, also available from PALLAS, is an instrumented MPI library. In addition to C and Fortran, VAMPIRtrace works with HPF compilers that emit MPI. We downloaded versions of VAMPIR and VAMPIRtrace with evaluation license keys from the VAMPMIR home page for our various test platforms. We tried VAMPIRtrace on T3E, IBM SP2, SGI PowerChallenge, and Sun Solaris, and VAMPIR on IBM SP2, SGI PowerChallenge, and Sun Solaris, and found them to work out-of-the box without any problems.

The VAMPIR documentation includes a User's Manual and an Installation Guide. A separate VAMPIRtrace Installation and User's Guide is provided for each platform (e.g., VAMPIRtrace for the AIX Parallel Environment). The VAMPIR User's Guide has a "Getting Started" chapter that gives an overview of the various views and features then follows with chapters that go into more detail. The documentation is lengthy but incomplete on some parts of the VAMPIR user interface.

Instrumentation and Trace File Generation

Instrumentation is done by linking your application with the VAMPIRtrace library which interposes itself between your code and the MPI library by means of the MPI profiling interface. Running the instrumented code produces a merged trace file that can be viewed using VAMPIR. Clear instructions are given in the VAMPIRtrace Users' Guide for each platform for linking C and Fortran programs with VAMPIRtrace on that platform. We were able to link and run instrumented C and Fortran programs on all our test platforms, using vendor MPI where available and MPICH 1.1 otherwise without any problems.

The VAMPIRtrace library also has an API for stopping and starting tracing and for inserting user-defined events into the trace file, but we did not try these features.

The trace files generated by the MPI processes when the program is run are automatically collected and merged into a single trace file.

Trace File Analysis

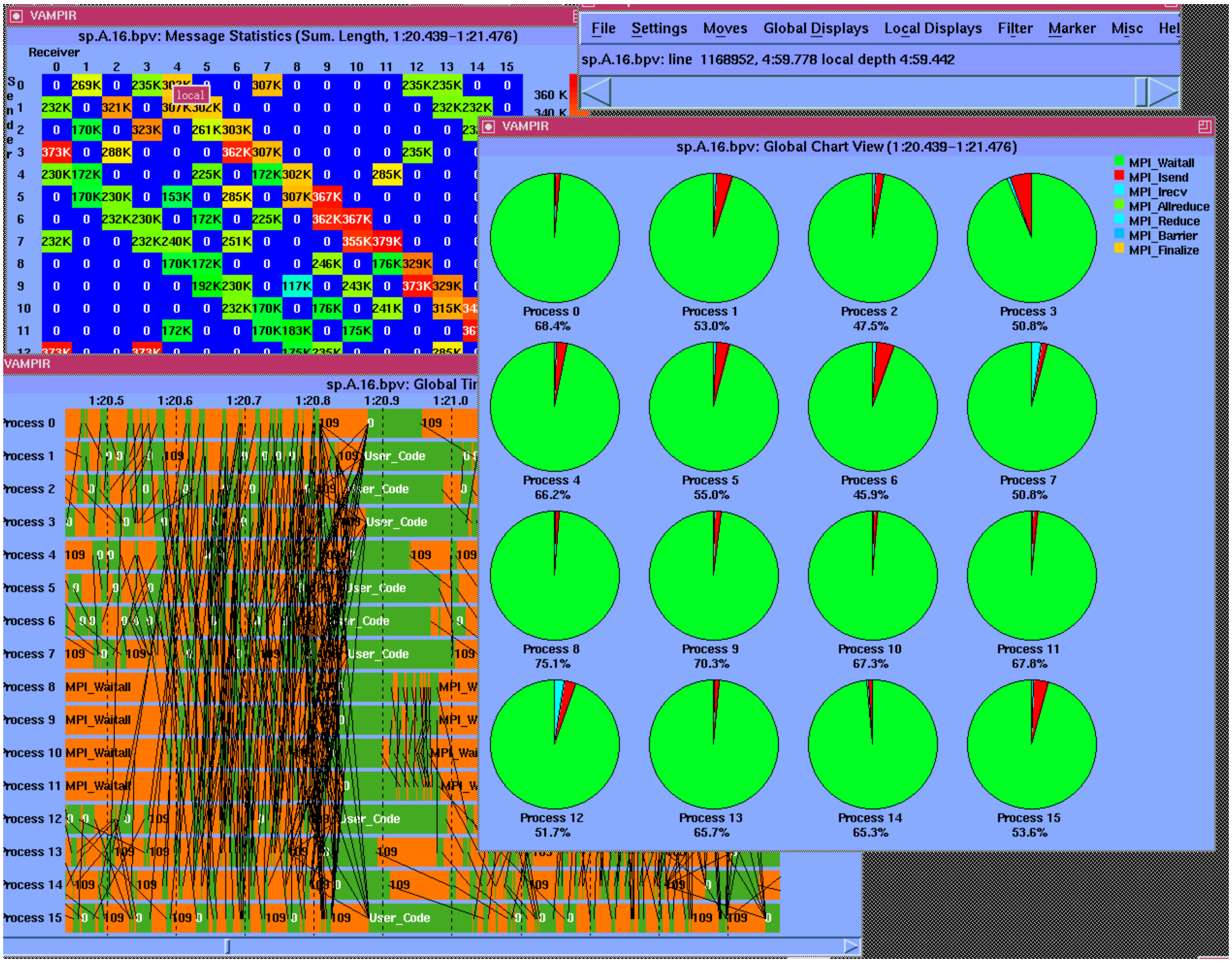
VAMPIR includes three main categories of visualization displays, which can be configured by the user by using pull-down menus to change options which are saved to a configuration file. The Process State Display displays every process as a box and displays the process state at one point in time. The Statistics display shows the cumulative statistics for the complete trace file in pie chart form for each process. The Timeline display shows process states over time and communication between processes by drawing lines to connect the sending and receiving process of a message.

The Process State and Statistics displays have both global and local forms, where local is for just one process. Initially, the only states shown in the Process State display were Idle, MPI, and Application. However, by selecting MPI from the "Display" submenu, one can obtain a display that subdivides the single MPI state into substates with different colors for the individual MPI calls.

The VAMPIR Timeline display shows the time during which a process has constant state as a colored bar along the horizontal axis. The Timeline display has both global and local forms. In the local version, the vertical dimension is used to display different states for a single process at different vertical positions, so that short time durations of one state are not obliterated by long durations of other states. In the global timeline display, timelines for all processes are displayed with a common time scale and with lines between bars to indicate messages, but without the vertical expansion of different states. Both timeline displays may be zoomed by dragging over the desired time range. One can obtain additional information about a state (or message) by selecting Identify State (or Identify Message) from the popup menu and then clicking on a colored bar (or message line). The Ruler from the popup menu can be used to obtain the length of an arbitrary time period selected by dragging the mouse on the display. Selecting Communication Statistics from the popup menu diplays a matrix showing the cumulative lengths of messages sent between pairs of processes. Although when using the default MPI tracing, only the Application and MPI states are shown in different colors, the different MPI activities are labelled as such with text on each bar.

By default, the Process State and Statistics displays summarize the complete history of the tracefile, but if the timeline display is open, the "Use Timeline Portion" option can be selected. In this mode, the displays show statistics about the time interval displayed in the timeline window and track changes in the timeline window.

See the following snapshot of VAMPIR analysis of our NPB 2.2 SP test program.



VAMPIR analysis of NPB 2.2 SP

Evaluation Summary

VAMPIR and VAMPIRtrace are easy to install and seem very robust. The number of architectures supported is impressive, and VAMPIR is the only commercial tool we are aware of that supports multiple platforms.

VAMPIR does not have source code clickback -- i.e., you cannot relate an event or state on the timeline to a specific program source line. However, by inserting calls to VAMPIRtrace routines in the source code, one can define events or states that will appear on the timeline display.

VAMPIR tracefiles can be stored in compressed binary format and uncompressed on the fly, and thus they require a minimal amount of disk space for amount of information stored. The VAMPIR tracefile format is documented in an appendix to the User’s Guide. Instructions are given for integrating a new converter into the VAMPIR user interface, should the user wish to supply one for converting from a different trace file format.

VAMPIRtrace is designed to use scalable data collection. Trace data collection can be turned on and off during runtime, and a filtering mechanism is provided to limit the amount of trace data and focus on relevant events. The filtering is done by means of a configuration file which is read by VAMPIRtrace at runtime and thus does not require application source code changes. VAMPIRtrace limits intrusion by keeping trace data locally in each processor's memory and post-processing it and saving it to disk when the application is about to finish. For systems without a globally consistent clock, VAMPIRtrace automatically corrects clock offset and skew so that timestamps in the merged trace file are consistent.

As for presentation scalability, VAMPIR allows simultaneous display of up to 512 processes, and the processes may be grouped into "jobs". VAMPIR can handle a time range of up to 2^52 clock cycles (e.g., three months for a machine with a clock cycle of 2 nanoseconds). VAMPIR has filtering capabilities that allow the user to extract a subset of the trace information for display. For example, filtering can be used to select a subset of processes or only certain message types to be displayed.

2.6 Visualization Tool (VT) for the IBM SP Parallel Environment

Version	May 28, 1997
Languages	C, C++, Fortran 77, Fortran 90, IBM HPF
Platforms	IBM SP2 with IBM MPI or MPL

Overview

VT can be used either for post-mortem trace visualization or for on-line performance monitoring. With post-mortem analysis, both communication events and AIX kernel statistics may be displayed. With on-line monitoring, only system statistics are available.

On-line documentation is available on the Web at AIX Online Publications and Books or locally on your SP using IBM InfoExplorer. The documentation explains in detail how to generate trace files and view them using VT, as well as how to use VT for on-line runtime monitoring. A VT Quick Operation table is provided that can get you started quickly using the basic features of VT. Sample trace files are also provided.

Trace File Generation

To generate a trace file, you must run your program with tracing turned on, either by setting the `MP_TRACELEVEL` environment variable or by using the `tracelevel` flag when invoking the program. The default trace level is zero, which means that tracing is turned off. The following four types of trace records may be generated:

- point-to-point message passing
- collective communication
- AIX kernel statistics
- application markers

We ran our test programs with the trace level set to 9, which generates all four types of trace records, although we did not use any application markers.

Generation of trace records for all enabled events for the duration of the program does not require any source code changes or special linking. However, you must have compiled your program with the **-g** option to be able to take advantage of VT's Source Code view. To gain greater control over trace file generation, for example to turn tracing on and off or to call particular VT trace generation routines, you can insert calls to VT routines into your application program.

VT uses its own routines to create trace records rather than utilizing the AIX trace facility. Communication trace records are written by instrumentation in the communication library. System statistics records are written by a spawned process that samples the kernel at a specified interval. The default sampling interval is 20 milliseconds, but it can be changed by setting the `MP_SAMPLE_FREQ` environment variable or the `samplefreq` command-line flag to a value in milliseconds. VT writes the trace file for each node to a buffer in memory local to that node. When the memory buffer is full, records are appended to a file in `MP_TMPDIR`, which should be local on each node. Separate buffers and temporary files are used for communications and statistics records. At the end of program execution, trace files from all nodes are automatically merged into a single file. The default name for the final trace file is the program name appended by `.trc`. The user can override the defaults by specifying the directory for the temporary trace files, the directory and name for the final trace file, the maximum buffer and temporary file sizes, and use of buffer wraparound rather than writing to a temporary file.

If the High Performance Communication Adaptor (HPCA) is configured, trace records are timestamped with the switch clock value, whether or not the adaptor is used for communication. If the adaptor is not present, the system clock is used.

The trace file we generated for our CMPI-rbsor test program was 445 Kbytes in size.

Trace File Analysis

The VT Control Panel has VCR-like controls that allow the user to control playback of a trace file. The control panel also has a magnification control that allows increasing or decreasing the amount of time displayed in the views, a replay speed control, and a trace file time control that shows and allows the user to change the current playback position.

VT has two types of views:

1. instantaneous - for information at a specific point in time
2. streaming - for showing a range of time

A total of thirty-one views are available from the VT View Selector panel, grouped into the following categories:

1. Computation - utilization of processor nodes
2. Communication/Program - message passing events between processor nodes; program source code
3. System - system activities and events such as page faults and context switches
4. Network - number of TCP/IP packets sent or received by processor nodes
5. Disk - number of disk reads, disk writes, and disk transfers

With all views, one can click with the left mouse button to get more information, and with the right mouse button to configure the view. For beginners, the documentation suggests the following initial selection of views:

- Message Status Matrix (a Communication/Program view)
- User Load Balance (a Computation view)
- Source Code (a Communication/Program view)

The Message Status Matrix uses a grid to visualize messages sent between processor nodes. Rectangular intersections on the grid represent message paths between the sending node (the row) and the receiving node (the column). The rectangles light up from when the message begins to be sent to when the receive completes, and the color of the rectangles indicates message size. The view can be toggled between instantaneous and cumulative presentation of the message information, with instantaneous being the default.

The User Load Balance view is an instantaneous view that uses three overlapping polygons to illustrate CPU utilization for each processor node, as well as the overall processor load balance. The outer polygon represents 100 percent utilization for all processor nodes. A second polygon is drawn inside the first to represent instantaneous CPU utilization for each of the processor nodes. The more regular this polygon, the better the processor load balance. A third polygon drawn inside the first represents average CPU utilization for each of the nodes.

The Source Code view shows the C, C++, Fortran, or HPF source code file associated with the most recent trace event. Colored bars across the top of the display represent different processes. During trace file playback, the bars move through the code to show each task's position in the source. The source code view only displays source code for one executable, by default the one running on task 0, but this can be initialized to a different task for MPMD programs.

The Interprocessor Communication view is a streaming view that corresponds to the timeline view that is provided by almost all trace visualization tools. In VT this view is limited to the range of events currently in the display's history buffer -- i.e., it cannot be scrolled backwards. There is also no way to zoom on this display. This view seems best suited for animation of a trace history, rather than for a detailed examination of the timeline.

Evaluation Summary

VT seems fairly robust, although it produced some spurious error messages on our CMPI-rbsor example. For this example, which uses the NULL option for the MPI SendRecv call, VT reported unmatched send/receive errors. Some of the Computation view output seemed questionable. For example, when animating the trace file for our CMPI-rbsor example, the Processor Utilization view showed no processor utilization for some time for process zero, either kernel or user, even though other views indicated that this process was executing code.

Although VT has an overwhelming number of views and features and thus a relatively steep learning curve, there is detailed on-line documentation available if one has the patience to work through it. We discovered a few features that didn't work quite as described in the documentation -- for example, in the Interprocessor Communication view, range search 0-3 didn't work, but 0 1 2 3 worked. Also, for the Source Code and the Processor Utilization views, clicking with the left mouse button doesn't display the time index, although the documentation says it does.

VT is definitely not portable, since it only works on the IBM RS6000 and SP platforms.

VT is versatile in that it offers a large number of views that can be configured in different ways. VT interoperates with other tools only to the extent that developers of other analysis tools or third parties are willing to write converters from VT trace format to their formats. For example, Donna Bergmark at the Cornell Theory Center has written a utility for converting from VT trace file format to Pablo SDDF format.

3.0 Comparison and Conclusions

We first give results for execution time and tracefile size for the various tracing libraries. Times are on an IBM SP2 with dedicated use of the switch and CPU. Times are averaged over three runs and had low variation (i.e., differences of a few hundredths of a second). Times are wallclock but not including merging and writing the final tracefile -- i.e., just while the original application was running.

From the results below, one can see that the AIMS monitoring library, the MPE logging library, and the VAMPIR tracing library add essentially no overhead to the application runtime. VT has about 10 percent overhead and produces larger tracefiles, but with the trace level set to 9, VT is also collecting more information, both kernel statistics and communication traces, than the other trace libraries. AIMS starts to show some overhead for 40 iterations because it is flushing the per-process trace files to disk, but this overhead could be eliminated by increasing the default tracefile buffer size in the AIMS.monrc file if the memory is available. The MPE logging library fails to generate a trace file for NPB2.2 SP with 40 iterations because it buffers the per-process tracefile entirely in memory, but the per user/per node memory limit on our SP2 in only 32 megabytes.

NPB2.2 SP, 4 iterations, 4 processors		
	time (wallclock-sec)	tracefile size (bytes)
untraced	3.14	
AIMS	3.14	107301
MPE logging	3.14	47573
VAMPIRtrace	3.17	77746
VT (tlevel 9)	3.36	238619

NPB2.2 SP, 4 iterations, 9 processors		
	time (wallclock-sec)	tracefile size (bytes)
untraced	1.48	
AIMS	1.46	209996
MPE logging	1.49	159056
VAMPIRtrace	1.47	249836
VT (tlevel 9)	1.54	524537

NPB2.2 SP, 4 iterations, 16 processors		
	time (wallclock-sec)	tracefile size (bytes)
untraced	0.92	
AIMS	0.93	422331
MPE logging	0.91	383981
VAMPIRtrace	0.90	584330
VT (tlevel 9)	1.01	1240157

NPB2.2 SP, 40 iterations, 16 processors		
	time (wallclock-sec)	tracefile size (bytes)
untraced	9.33	
AIMS	10.23	2578421
MPE logging	9.91	memory exhausted
VAMPIRtrace	9.45	4456138
VT (tlevel 9)	10.6	7408029

For the tracefile analysis tools, we noted memory usage on the IBM SP2 for AIMS VK, nupshot, VAMPIR for analyzing the tracefile for NPB2.2 SP with 40 iterations and 16 processes. For Paradyn, we noted memory usage for the front end during runtime analysis with a visualization open and the Performance Consultant running. Note that the tracefile analysis tools that have the capability for scrolling and/or zooming (e.g., nupshot and VAMPIR) have higher memory usage than AIMS VK which does not have this capability.

	Memory usage
AIMS VK	1800
IBM nupshot	5356K
MPICH nupshot	4836K
Paradyn	27652K
VAMPIR	4800K
VT	9732K

Based on the testing and observations described in the individual tool reviews, we provide the following summary of our evaluation of each tool on each of the evaluation criteria.

	Robustness/Accuracy	Usability	Portability	Scalability	Versatility
AIMS	Fair	Good	Fair	Good	Good
nupshot	Good	Good	Good	Good	Good
Pablo Analysis GUI	Good	Fair	Fair	Good	Excellent
Paradyn	Fair	Good	Fair	Fair	Good
SvPablo	Good	Good	Fair	Good	Good
VAMPIR	Excellent	Good	Excellent	Excellent	Good
VT	Good	Good	No, platform-specific	Good	Good

If one can afford to buy a commercial tool, then VAMPIR is clearly a good choice because of its availability on all important HPC platforms and because it works well and has excellent support. All the tools we reviewed should continue to be of interest, however, because each offers unique capabilities.

AIMS provides both tracefile visualization and statistical analysis, along with great flexibility in what parts of the program are instrumented. The source code clickback capability allows the user to quickly determine what part of the program is causing the visualized behavior. Some bugs remain to be fixed, however, such as the incorrect attribution of wait time in the statistical analysis.

MPE logging appears to be readily portable to platforms with conforming MPI implementations. The alog format used by MPE logging and nupshot provides flexibility in the states that can be defined in the tracefile. nupshot is very simple and does not have as many features as the other tools, but it is simple and easy to use and provides a quick overview of the states of different processes over time, along with the ability to zoom for a detailed view.

Pablo is an ambitious project but needs further development to move from being a research to a production tool. SvPablo support for Fortran and Fortran90 is needed. The MPI TraceLibrary needs to work with vendor MPI implementations in addition to MPICH. The Pablo Analysis GUI needs to provide greater assistance to the user in debugging a graph that does not work.

Paradyn is another ambitious research project. The Performance Consultant which automatically searches for performance bottlenecks is a unique feature. However, the MPI version needs to be made to work for more than twelve processes and to be ported to additional platforms.

VT is specific to the IBM RS6000 platform, but is a solid tool with many useful features.

Information about the above tools, as well as the actual tools themselves where possible, will continue to be available from the Parallel Tools Library (PTLIB), which is part of the National High-performance Software Exchange (NHSE).

4.0 References

1. Message Passing Interface Forum. MPI: A Message Passing Interface Standard. *International Journal of Supercomputer Applications* 8, 1994. Special issue on MPI.
2. Barton P. Miller, Mark D. Callaghan, Jonathan M. Cargille, Jeffrey K. Hollingsworth R. Bruce Irvin, Karen L. Karavanic, Krishna Kunchithapadam, and Tia Newhall. The Paradyn Parallel

Performance Measurement Tools. *IEEE Computer* 28:11 (November 1995). Special issue on performance evaluation tools for parallel and distributed systems.

3. Daniel A. Reed, Ruth A. Aydt, Roger J. Noe, Phillip C. Roth, Keith A. Shields, Bradley Schwartz, and Luis F. Tavera, "Scalable Performance Analysis: The Pablo Performance Analysis Environment," In Anthony Skjellum, editor, *Proceedings of the Scalable Parallel Libraries Conference*, IEEE Computer Society, 1993.
4. Marc Snir, Steve Otto, Steven Huss-Lederman, David Walker, and Jack Dongarra. MPI: The Complete Reference. MIT Press, 1996.
5. Jerry Yan, S. Sarukhai and P. Mehra. Performance measurement, visualization and modeling of parallel and distributed programs using the AIMS toolkit. *Software -- Practice and Experience* 25:4 (April 1995), pp. 429-461.