Practical Experiences with Modern Parallel Performance Analysis Tools: An Evaluation

Adam Leko
leko@hcs.ufl.edu

Hans Sherburne
sherburne@hcs.ufl.edu

Hung-Hsun Su
su@hcs.ufl.edu

Bryan Golden
golden@hcs.ufl.edu

Alan D. George
george@hcs.ufl.edu

High-Performance Computing and Simulation (HCS) Laboratory
Department of Electrical and Computer Engineering
University of Florida, Gainesville, FL, USA

ABSTRACT

Achieving a significant fraction of peak performance on a modern high-performance computer is a challenging task. Fortunately, many performance analysis tools exist that can be used to improve the efficiency of parallel programs. However, while these tools can be very effective at troubleshooting performance problems, finding the right performance tool for each situation can be a time-consuming task. Since performance optimization is generally considered near the end of software development cycles, most developers cannot afford to spend time examining each available performance tool. Thus, it is common practice for developers to rely on ad-hoc performance analysis techniques.

We have recently concluded an extensive study of several existing performance analysis tools. This paper summarizes our findings and is meant to serve as a guide to the latest software in parallel performance analysis tools. We evaluate each tool using a standard methodology and highlight each tool's key features and relative strengths. Finally, we give general recommendations on how to best use each performance analysis tool.

Keywords

Parallel performance analysis tool, Tool evaluation, MPI performance tool

1. INTRODUCTION

While hardware advancements have led to ever-increasing maximum peak performance for modern high-performance computing platforms, most software has not been able to exploit these advances to attain similar performance increases. The gap between theoretical peak performance and real-world performance has been driven further apart by increasing hardware and software complexity. As a result, parallel performance tools are playing an increasingly important role in the software development process.

The existence of several performance tools gives software developers many choices for analyzing performance bottlenecks in their code. While the plethora of performance tools affords developers much flexibility, the sheer number and variety of tools can be daunting. A comprehensive overview of available parallel performance tools aids developers since it serves as a road map for what tools and analysis methods are currently available. Additionally, an overview also aids researchers developing new tools by enabling them to see what functionality is currently available and what open research issues still exist.

We are currently designing a next-generation performance analysis tool designed specifically for global address languages. As preparation for this task, we have performed an extensive review of performance tools. We have chosen a variety of tools that employ different analysis techniques and have evaluated them against an application suite exhibiting known performance problems. This paper presents the findings from that tool study, giving an indication of the relative strengths and weaknesses of each tool and giving recommendations on appropriate uses for each tool.

Since MPI [25] is a very popular standard for current high-performance computing software, most of the tools we examined are geared towards performance analysis of MPI programs. Also, while techniques such as modeling and simulation may be used to reason about a program's performance, experimental performance analysis has historically been the most effective method for troubleshooting performance problems in real code. Therefore, this paper focuses on experimental performance analysis tools as these types of tools will be the most useful for the majority of software developers.

The rest of this paper is organized as follows. We first give an introduction to the basic terminology and techniques used by performance tools in Section 2. We present our review of performance tools in Section 3 and finally present our conclusions in Section 4.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
Copyright 200X ACM X-XXXXX-XX-X/XX/XX ...$5.00.

2. BACKGROUND

In experimental performance analysis, there are two major techniques that influence the overall design and work flow of performance tools [31]. The first technique, profiling, keeps track of basic statistical information about a program's performance at runtime. This compact representation of a program's execution is usually presented to the developer immediately after the program has finished executing, and gives the developer a high-level view of where time is being spent in their application code. The second technique, tracing, keeps a complete log of all activities performed by a developer's program inside a trace file. Tracing usually results in large trace files, especially for long-running programs. However, tracing can be used to reconstruct the exact behavior of an application at runtime. Tracing can also be used to calculate the same information available from profiling and so can be thought of as a more general performance analysis technique.

Performance analysis in performance tools supporting either profiling or tracing is usually carried out in five distinct steps: instrumentation, measurement, analysis, presentation, and optimization. Developers take their original application, instrument it to record performance information, and run the instrumented program. The instrumented program produces raw data (usually in the form of a file written to disk), which the developer gives to the performance tool to analyze. The performance tool then presents the analyzed data to the developer, indicating where any performance problems exist in their code. Finally, developers change their code by applying optimizations and repeat the process until they achieve acceptable performance. This collective process is often referred to as the "measure-modify" approach, and each step will be discussed in the remainder of this section. For a more comprehensive background on performance analysis techniques, we refer the reader to [18].

2.1 Instrumentation

During the instrumentation step, an instrumentation entity (either software or a developer) inserts code into a developer's application to record when interesting events happen, such as when communication or synchronization occurs. Instrumentation may be accomplished in one of three ways: through source instrumentation, through the use of wrapper libraries, or through binary instrumentation. While most tools may use only one of these instrumentation techniques, it is possible to use a combination of techniques to instrument a developer's application.

Source instrumentation places measurement code directly inside a developer's source code files. While this enables tools to easily relate performance information back to the developer's original lines of source code, modifying the original source code may interfere with compiler optimizations. Source instrumentation is also limited because it can only profile parts of an application that have source code available, which can be a problem when users wish to profile applications that use external libraries distributed only in compiled form. Additionally, source instrumentation generally requires recompiling an entire application over again, which is inconvenient for large applications.

Wrapper libraries use interposition to record performance data during a program's execution and can only be used to record information about calls made to libraries such as MPI. Instead of linking against the original library, a developer first links against a library provided by a performance tool and then links against the original library. Library calls are intercepted by the performance tool library, which passes on the call to the original library after recording information about each call. In practice, this interposition is usually accomplished during the linking stage by including weak symbols for all library calls. Wrapper libraries can be convenient because developers only need to re-link an application against a new library, which means that there is less interference with compiler optimizations. However, wrapper libraries are limited to capturing information about each library call. Additionally, many tools that use wrapper libraries cannot relate performance data back to the developer's source code (e.g., locations of call sites to the library). Wrapper libraries are used to implement the MPI profiling interface (PMPI), which is used by most performance tools to record information about MPI communication.

Binary instrumentation is the most convenient instrumentation technique for developers, but places a high technical burden on performance tool writers. This technique places instrumentation code directly into an executable, requiring no recompilation or relinking. The instrumentation may be performed before runtime, or may happen dynamically at runtime. Additionally, since no recompiling or relinking is required, any optimizations performed by the compiler are not lost. The major problem with binary instrumentation is that it requires substantial changes to support new platforms, since each platform generally has completely different binary file formats and instruction sets. As with wrapper libraries, mapping information back to the developer's original source code can be difficult or impossible, especially when no debugging symbols exist in the executable.

2.2 Measurement

In the measurement stage, data is collected from a developer's program at runtime. The instrumentation and measurement stages are closely related; performance information can only be directly collected for parts of the program that have been instrumented.

During measurement, the most common metric collected by performance tools is the wall clock time taken for each portion of a program. This timing information can be further separated into time spent on communication, synchronization, and computation. In addition to wall clock time, a performance tool can also record the number of times a certain event happens, the amount of bytes transferred during communication, and other metrics. Many tools also use hardware counter libraries such as PAPI [1] to record hardware-specific information such as cache miss counts.

There is an obvious tradeoff between the amount of data that can be collected and the overhead imposed by collecting this data. In general, the more information collected during runtime, the more overhead experienced and thus the less accurate this data becomes. While early work has shown that it is possible to compensate for much of this overhead [17], overhead compensation has not become available for the majority of performance tools.

Profiling tools may also use an indirect method known as sampling to gather performance information. Instead of using instrumentation to directly measure each event as it occurs during runtime, metrics such as a program's call stack are sampled. This sampling can be perfomed at fixed intervals, or can be triggered by hardware counter overflows.

Using sampling instead of a more direct measuring technique drastically reduces the amount of data that a performance tool must analyze. However, sampled data tends to be much less accurate than performance data collected by direct measurement, especially when the sampling interval is large enough to miss short-lived events that happen frequently.

2.3 Analysis

During the analysis stage, data collected during runtime is analyzed in some manner. In some profiling or sampling tools, this analysis is carried out as the program executes. This technique is generally referred to as "online analysis." More commonly, analysis is deferred until after an application has finished execution so that runtime overhead is minimized. Performance tools using this technique are often referred to as "post-mortem analysis" tools.

The types of analysis capabilities offered varies significantly from tool to tool. Some performance tools offer no analysis capabilities at all, while others can compute only basic statistical information to summarize a program's execution characteristics. A few performance tools offer sophisticated analysis techniques that can identify performance bottlenecks. Generally, tools that provide minimal analysis capabilities rely on the developer to interpret data shown during the presentation stage.

We present a more complete discussion of the analysis capabilities of each tool in Section 3.

2.4 Presentation

After data has been analyzed by the performance tool, the tool must present the data to the developer for interpretation. Like the analysis stage, the types of presentations offered by performance tools vary significantly and will be discussed in detail in Section 3.

For tracing tools, the performance tool generally presents the data contained in the trace file in the form of a space-time diagram, also known as a timeline diagram. In timeline diagrams, each node in the system is represented by a line. States for each node are represented through color coding, and communication between nodes is represented by arrows. Timeline diagrams give a precise recreation of program state and communication at runtime. Figure 1 shows an example timeline view from Jumpshot-4 (discussed in Section 3.1.1).

For profiling tools, the performance tool generally displays the profile information in the form of a chart or table. Bar charts or histograms graphically display the statistics collected during execution. Text-based tools use formatted text tables to display the same type of information. A few profiling tools also display performance information alongside the original source code, as profiled data such as the percentage of time an instruction contributes to overall execution time lends itself well to this kind of display.

2.5 Optimization

In most performance tools, the optimization stage is left up to the developer. The majority of performance tools do not have any facility for applying optimizations to a developer's code. At best, the performance tool may indicate where a particular bottleneck occurs in the developer's source code and expects the developer to come up with an optimization to apply to their code.

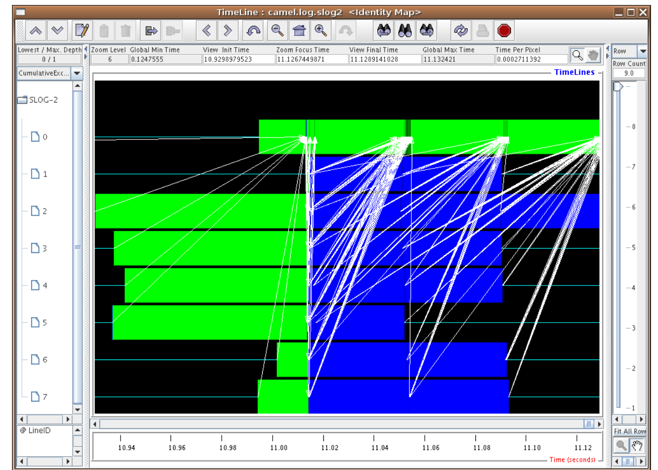

Figure 1: Example Timeline View from Jumpshot-4

3. PERFORMANCE TOOLS

In this section, we present an overview of each performance tool we examined. We categorize each tool as a tracing, profiling, or online analysis tool. For each tool, we describe what hardware platforms and programming languages/models are supported. We also describe how each tool can be used and point out any interesting analysis or display capabilities.

To compare each tool under similar circumstances, we devised a standardized evaluation methodology. After installing each tool, we used it on a suite of applications to search for performance problems. This application suite contained microbenchmarks from the PPerfMark suite [23] (based off of GrindStone [10]), an in-house cryptanalysis application developed by the HCS lab named CAMEL, and the LU benchmark from the NAS NPB-3.1 MPI benchmark suite [33]. We used the PPerfMark microbenchmarks to evaluate each tool's ability to identify different types of performance bottlenecks. The LU benchmark was used to determine how well the tool worked with a more realistic application code with no obvious bottleneck. Finally, we used the cryptanalysis application as a control to test for false positives for bottlenecks (e.g., when the tool identifies a bottleneck that does not exist). We also measured the overhead imposed by each tool by comparing the wall clock times for instrumented and non-instrumented runs of each application. Overhead times reported below refer to runs on a Linux cluster with eight nodes (each with two Pentium-3 1.0 GHz processors) using MPICH 1.2.6 and gcc 3.3.2 on RedHat 9, unless otherwise noted.

In addition to the application suite, we also devised a comprehensive set of over 20 characteristics (such as documentation quality and software stability) and rated each tool numerically on each scale. Space limitations prevent us from including all scores for each tool; these are available on our web site [15].

3.1 Tracing Tools

In this section, we present detailed overviews of several popular tracing tools that give a representative sample of the variety of available tracing tools. We also give brief descriptions of other tracing tools in Section 3.1.4.

3.1.1 MPE 1.26, Jumpshot-4 1.0.1.0

The Multi Processing Environment (MPE) [3] is a freely-available trace library distributed with MPICH meant to be used with Jumpshot, a trace file visualizer. MPE is available for all platforms that MPICH supports (including Linux, AIX, IRIX, and Tru64), and comes bundled with the MPICH installation source. Since MPE uses PMPI to record information about MPI applications, it can also be used with a wide variety of vendor-supplied MPI libraries, including Cray MPI.

MPE supports three modes of operation: echoing all MPI calls to `stdout`, animating communication for all MPI calls in an X-Windows session, and logging all MPI calls in a trace file. Of these three modes, the trace file creation library is by far the most useful, as it allows developers to visualize the behavior of their MPI code using the Jumpshot visualization tool. To enable trace file creation with MPICH, one uses `mpicc -mpilog` instead of `mpicc` when linking an MPI executable and runs the MPI program as normal. In addition, MPE also provides a simple set of functions that developers can use to manually instrument basic blocks or phases of their code.

The Jumpshot visualization tool is a Java program that gives a graphical timeline representation of trace files collected by the MPE logging library. By default, the MPE logging library generates trace files in the CLOG format, which must be converted to Jumpshot's native SLOG-2 format before Jumpshot can display them. In our tests, the conversion to SLOG-2 format was relatively quick – it took only a few minutes for a 700MB CLOG file to be converted to an SLOG-2 file, which shrunk the size of the file by approximately one-third. The SLOG-2 file format is a graphics-based format; it is more computationally expensive to create than CLOG, but allows Jumpshot to more efficiently display trace data.

Jumpshot provides an intuitive (though slightly cluttered) user interface that follows most modern user interface conventions and has excellent zooming and scrolling features that make navigating large trace files relatively painless. It tends to consume a lot of RAM during usage, but the interface remains responsive even in the face of large trace files. Since Jumpshot is written in Java, it can also run on a user's personal Windows or Linux workstation.

MPE and Jumpshot performed well on our application bottleneck suite, even though they only record and display information about MPI calls. For the PPerfMark benchmarks, Jumpshot's timeline display allowed us to clearly see MPI communication, which allowed us to pick out the communication bottlenecks in the PPerfMark benchmarks. Jumpshot did not show any anomalies on the CAMEL benchmark, and the timeline view illustrated a few places where message aggregation and collective functions could have been used to increase scalability. Finally, on the LU benchmark, Jumpshot clearly illustrated several places in the application where an excessive number of small messages were being sent and hinted at the data dependencies inherent in the benchmark code. In general, the overhead imposed by recording traces of application code was less than 10%, and the LU benchmark had only 5% overhead.

While we appreciate Jumpshot's intuitive interface and its ability to deal with large trace files, there are a few areas that could use improvement. The first notable limitation of Jumpshot is its inability to perform correlation with displayed events to lines in a user's source code. While users can manually insert instructions to indicate which function or region of code is currently being executed, this limited form of source correlation is not ideal and may take too long to be useful for applications with hundreds of thousands of lines of code. Second, while the Jumpshot viewer is able to handle large trace files, trace files containing data from many nodes can overwhelm users with too much data. Trying to use Jumpshot to display more than 32 separate processes becomes cumbersome, although Jumpshot does support histograms which can help cut down on the information being displayed and allow users to zero in on problematic sections of code. Third, MPE and Jumpshot only record information about the MPI activities of a program. This limitation of only collecting performance data directly related to MPI communication means that MPE and Jumpshot cannot be used with non-MPI programs unless a lot of manual instrumentation is done. Finally, Jumpshot provides no facility to show hardware counter data (e.g., information recorded by PAPI), which can be extremely valuable in troubleshooting performance bottleneck issues not related to communication.

In general, Jumpshot and MPE are valuable as a "first line of defense" for performance analysis of MPI programs, especially since they are freely available and distributed with most versions of MPICH. With a small number of processes, MPE and Jumpshot are able to display all MPI-specific events of a program without inducing a large amount of overhead. However, since they are tools specific to MPI programs, they are best suited to troubleshooting MPI-only problems.

3.1.2 Intel Trace Collector and Trace Analyzer

Trace Collector (v5.0.1.0) and Trace Analyzer (v4.0.3.1) are Intel's updated versions of Pallas' VampirTrace and Vampir tools [27]. Being Intel's tools, they are only supported on Intel architectures such as Xeon- and Itanium-based systems. Trace Collector, the trace collection component of Trace Analyzer, supports C/C++, Fortran, and Java programs. MPI programs are monitored through the PMPI interface, with direct support for Intel MPI, LAM, and MPICH. In the versions of Trace Analyzer and Trace Collector we tested, not much had been changed from the Pallas versions of Vampir, except the drop of support for non-Intel platforms.

Trace Collector supports static binary instrumentation of user function entry/exit points. It also supports manual instrumentation that enables users to record application states, communication events, and arbitrary counters values. Trace Collector uses the Structured Trace File (STF) file format to store trace information, which incorporates some indexing techniques to speed data loading.

Trace Analyzer is an X-Windows visualization tool for STF trace file, and supports several different visualizations of data, including timeline views (with user-supplied or hardware counter information), histograms, call trees, and communication statistics. We found the main interface to be very stable and responsive, although large trace files tended to make the interface sluggish. Trace Analyzer has excellent support for source code correlation; in the timeline view, for instance, you may right-click on any event displayed to find out what line of source code is attributed to that event.

On the bottleneck test suite, we were able to identify and troubleshoot nearly every test application. For the CAMEL benchmark, the timeline view clearly displayed the communication taking place and hinted that message aggregation and asynchronous communication could be used in a few places to increase scalability. The LU trace file clearly showed a large number of small messages and the complex communication patterns used by the benchmark. We were able to successfully identify each application in the PPerfMark suite, except for the `system-time` benchmark because Trace Analyzer has no way of showing time spent by the operating system vs. user process time. The overhead introduced by the Trace Collector library averaged less than 10% for all of our applications, with the LU benchmark experiencing an overhead of only 2%.

Intel's Trace Collector and Analyzer show how effective a tool can be that provides both profile-style statistical summary data *and* trace-style event timelines to users. However, the sheer number of available visualizations offered by the program can be daunting. The tool does not offer any automatic analysis capabilities that can guide users to problematic areas of execution, which can make it difficult to use with complex applications. Additionally, while the STF file format deals relatively well with larger files, loading trace files larger than 200MB has a negative effect on Trace Analyzer's responsiveness. While it is understandable for Intel to drop support for other non-Intel platforms, it severely limits the usefulness of the tool for high-performance computing users. Finally, we suggest that Intel should release the specifics of the STF file format so users of non-Intel platforms may benefit from Trace Analyzer's excellent display capabilities. If this is not possible, we suggest that Intel should continue to support the older VTF trace format, which is documented.

In general, Trace Analyzer is best used in conjunction with other performance tools such as TAU or KOJAK that can write to the VTF format, which allows Trace Analyzer to benefit from the portability and additional analysis capabilities of these tools. It is also worth mentioning that the University of Dresden is also working on a next-generation version of Vampir supporting advanced analysis capabilities, such as parallel analysis of trace files [2].

3.1.3 KOJAK 2.0

The Kit for Objective Judgement And Knowledge-based detection of performance bottlenecks (KOJAK) is a software suite aimed towards automatic performance analysis and bottleneck detection of parallel programs [22]. KOJAK supports MPI and OpenMP programs on almost every modern computing platform, including Itanium/Opteron, Power, MIPS, SGI Altix, SPARC, Cray X1 and T3E, NEC SX, Hitachi SR-8000, and IBM BlueGene/L based systems.

For most platforms, KOJAK relies on the MPI profiling interface and manual instrumentation to record performance data. For OpenMP programs, KOJAK has a specific tool, OPARI, that is able to perform automatic source instrumentation. One can also use TAU to record trace data (and thus benefit from TAU's automatic source instrumentation), and convert the TAU trace files to KOJAK's own EPILOG format. Additionally, KOJAK is also able to automatically instrument executables on a few platforms by taking advantage on undocumented compiler features. On the IBM AIX platform, KOJAK is also able to leverage DPCL [4], a software toolkit that provides dynamic binary instrumentation for distributed environments.

KOJAK's main feature is its automatic bottleneck detection trace analysis program, EXPERT. EXPERT searches through trace files for patterns of performance bottlenecks, which can be described in either C or Python code. The bottleneck analysis can take a long time, but the time taken is generally proportional to the size of the trace file. In our tests, the LU benchmark's class W trace file was 23MB and took just under 9 minutes to analyze on a 2.2+ GHz AMD Athlon XP processor.

To visualize the analyzed data from EXPERT, KOJAK relies on the CUBE viewer. The CUBE viewer is a simple viewer consisting of three main panels: a metric pane, a source code location pane, and a node/thread location pane. The metric pane uses a simple hierarchy for describing different metrics, such as time or bottleneck type. Each metric has a color-coded severity displayed alongside it, which can be further separated out into source code and machine locations by expanding nodes in the middle and rightmost panels. The interface used by CUBE does take a few minutes to get used to, but is able to compactly represent profile-style timing and bottleneck information. For visualizing the complete trace data, KOJAK also supports exporting EPILOG trace files to Vampir.

KOJAK's pattern-matching bottleneck detection worked well for identifying bottlenecks in the test suite. For the CAMEL benchmark, KOJAK was able to point out a few places where asynchronous communication could have been used by identifying a "late sender" bottleneck. KOJAK also identified a few late sender and receiver communications in the LU benchmark, but the CUBE viewer did not illustrate that too many small messages were being sent. For the rest of the PPerfMark tests, KOJAK was able to correctly identify bottlenecks except for a few load imbalance problems. The overhead imposed by tracing EPILOG files was reasonable, averaging at less than 10% and adding less than 2% overhead for the CAMEL and LU benchmarks.

We appreciated KOJAK's ability to identify multiple bottlenecks in each application and think its "severity" attribute for each bottleneck provides a good method for quantifying the effect of the bottleneck on overall performance. While it is nice to see that users can record arbitrary PAPI metrics in each run, it would be better if KOJAK supported more PAPI metrics in its built-in library of performance problems used during EXPERT's bottleneck searches, which currently only uses floating-point instructions and L1 cache misses. While the CUBE viewer does support source code correlation, it would be nice to see MPI functions related to call sites in a user's code for platforms where automatic instrumentation is not available. Finally, we would also like to see KOJAK export trace files to Jumpshot in addition to Vampir.

In general, we suggest that KOJAK be used when users want a detailed analysis of where potential bottlenecks exist in their code without having to manually search through an entire trace file. While the bottleneck detection process can be time-consuming for large files, the analysis can be carried out while users work on other tasks.

3.1.4 Other Tracing Tools

MPICL [34] and Paragraph [8] are two older tools that support recording and visualizing MPI traces. MPICL is

a trace capture library that supports many vendor-specific message-passing libraries that were in existence before the PVM or MPI standards. With the introduction of the MPI standard, this library is not as useful on modern platforms. Paragraph is a visualization tool first developed between 1989 and 1991 that supports over 30 different visualizations. While most of the visualizations such as timelines have survived in modern tools, Paragraph contains a few esoteric visualizations that no modern tool has, such as the "streak" and "phase plot" visualizations. While the visualizations provided by Paragraph allowed us to pinpoint most of the bottlenecks in our performance bottleneck suite, it was very difficult to know beforehand which visualization provided us with the correct type of data needed to identify those bottlenecks.

Paraver [14] is another trace visualization tool that is geared towards displaying timeline views of trace data. It has a very complicated user interface and uses a tape metaphor instead of scroll bars for navigating through trace data, which tends to be counter-intuitive and less flexible. However, Paraver supports many different analysis modes and can display just about any statistic you can think of once you spend enough time to master it. Additionally, Paraver can be paired with Dimemas, a simulation tool that can be used to perform what-if analysis on real traces, such as "What would happen if I doubled my network bandwidth?" Dimemas also generates trace files that can be read by Paraver, which gives users a unified way of dealing with either simulated or real data.

Kappa-PI [24] is a unique research prototype that analyzes MPI traces, searches for bottlenecks, and makes suggestions to users on how to fix each bottleneck. Kappa-PI is targeted towards new users, but we hypothesize that a similar knowledge-based system might also work well for experienced users. While Kappa-PI has not been made publicly available at the time of this writing, a version of it is scheduled to be released in December 2005.

3.2 Profiling Tools

In this section, we present an illustrative sample of available profiling tools. We also quickly review some of the other available sampling tools in Section 3.2.4, relating their analysis and presentation techniques to the tools we have already discussed.

3.2.1 TAU 2.14.4

Tuning and Analysis Utilities (TAU) [21] is a portable performance tool written by researchers at the University of Oregon. One of TAU's greatest strengths is the number of hardware and software platforms it supports. TAU can be run on just about any modern high-performance computing platform and supports several languages, including C, C++, Java, Python, Fortran, SHMEM, OpenMP, MPI, and Charm.

Most of TAU's options are controlled at compile time, specified as options to a `configure` script. TAU can be configured to use profiling or tracing, and has many options controlling the type of profiling that can be done. While many of TAU's options are mutually exclusive, it is possible (though somewhat awkward) to install multiple versions of TAU and select between them by adjusting environment variables.

At the highest level, TAU relies on source instrumentation performed by the user (or a tool in conjunction with PDToolkit [16]) to record entry and exit events for user functions. When combined with "lightweight" functions (i.e., functions that do a small amount of work but are called many times), this can result in a lot of runtime overhead; to overcome this problem, TAU provides a `tau_reduce` tool that can be used with existing runtime data to automatically avoid instrumenting functions that are called many times. Additional instrumentation levels supported include dynamic binary instrumentation through DynInst [9], wrapper libraries for MPI (using the PMPI interface), virtual machine instrumentation for Java programs, and interpreter instrumentation for Python code.

TAU's default mode of operation is profiling. To this end, TAU is distributed with two profile visualization tools: `pprof` and `paraprof`. `pprof` is a text-based tool that produces output similar to `prof` or `gprof` in which functions are listed (per node) in order of increasing time consumption. `paraprof` shows the same data available as `pprof`, but displays it graphically using bar charts and three-dimensional visualizations. Additionally, in newer versions of TAU, `paraprof` can make use of the new PerfExplorer [11] interface to perform data mining and cross-experiment performance analysis. While TAU does not directly collect line-level source correlation, it can be configured to collect complete call paths to get a better idea of the context of each function call. In our testing, we found both `pprof` and `paraprof` to be exceptionally stable.

TAU does not contain any built-in trace viewers (although newer versions of TAU are being distributed with Jumpshot). TAU instead relies on exporting trace files from its native format to other formats, and currently supports Jumpshot's SLOG-2 format, KOJAK's EPILOG format, and Vampir's native format VTF.

For the bottleneck suite, we chose to use `paraprof` to troubleshoot performance problems since it was the most functional user interface distributed with TAU. On the CAMEL application, we were able to easily identify the most time-consuming functions, but we were unable to determine how communication could be restructured to gain efficiency. For the LU benchmark, `paraprof` showed that much of the application time was spent on communication, but we gained little insight on what communication patterns were causing performance degradation. Finally, on the PPerfMark suite, we found that `paraprof` was able to identify most computation bottlenecks easily, but did not provide us with enough information to identify bottlenecks that resulted from variable communication patterns or poor dynamic load balancing (such as the `wrong-way` and `random-barrier` benchmarks).

We should note that when TAU's profiles were combined with TAU's traces and another visualization tool such as Jumpshot or Vampir, we were able to identify almost every performance bottleneck in our suite. We will return to this observation later.

TAU's measurement libraries also did a good job recording profile data with minimal overhead. TAU's profile modes generally introduced negligible overhead (less than 1%) as long as lightweight functions were not profiled. TAU also includes a configuration option that instructs it to compensate for the overhead introduced by profiling. This option works fairly well, but often the simpler solution of excluding

lightweight functions from instrumentation can be more effective. TAU's measurement libraries also introduced a minimal overhead (less than 5%) for our tracing experiments.

While we liked TAU's portability and depth of features, we encountered a few issues worth mentioning. First, TAU relies on other software such as PDToolkit for some basic functionality; this makes the installation process a bit more involved, since extra software has to be downloaded and installed first. Second, even though TAU provides a very helpful user manual, getting acquainted with all of TAU's features can take some time. Additionally, sometimes it is easier to use other tracing tools by themselves instead of using them through TAU. For instance, it is generally easier to use `mpicc -mpilog` and Jumpshot than the equivalent sequence of operations to get the same functionality through TAU. Finally, while TAU's wrapper Makefiles make it relatively easy to use the correct compilation and linking arguments to instrument code and link against TAU's libraries, getting these working correctly with an existing application can take some trial and error. However, newer versions of TAU provide scripts that can be used in place of compilers invocations (e.g., `tau_cc.sh` instead of `cc`); this makes it much easier to instrument applications that have unique build processes (such as the NAS benchmark suite).

In general, TAU is a powerful but complex tool well suited to users that require a lot of functionality and are willing to invest time to learn how to use its advanced features.

3.2.2 HPCToolkit 1.1.0

HPCToolkit [19] is a suite of tools that helps programmers collect, organize, and display profile data. It runs on many Linux-based platforms, including IA32, Opteron, and Itanium systems with the PAPI library installed. It also runs on AlphaServer Tru64, IRIX64 MIPS machines, and Solaris SPARC machines. Since HPCToolkit relies on sampling metrics, it can work with any compiled executable, including threaded, MPI, and OpenMP code. HPCToolkit requires no instrumentation phase, but does require executables to be compiled with debugging information for source correlation to work.

HPCToolkit is composed of several smaller tools that provide most of the functionality of the tool set. The `hpcrun` tool is used to perform monitored runs of executables using PAPI, which produces several text profile files. These profiles are merged with executable format structure information from `bloop` by the `hpcprof` tool, which creates simple profile information. Additionally, the profiles can be used with the `hpcview` tool, which packs up the performance information and source code referenced in the files into a report directory that can be opened by the `hpcviewer` program.

HPCToolkit allows users to collect wallclock metrics or any PAPI metrics for each sampled run. When viewed in the `hpcviewer` Java-based viewer, these metrics are displayed alongside a user's code. Users may aggregate the metrics in any way they wish by constructing MathML expressions, which the `hpcviewer` tool uses when displaying the data. Additionally, since HPCToolkit makes no assumptions about the nature of performance data, it is very easy to include metrics collected from different runs in a single `hpcviewer` session.

On the bottleneck test suite, HPCToolkit was able to identify computation-bound code sections. However, as there are no high-level communication metrics supported by PAPI, HPCToolkit was not able to show any communication characteristics of the application other than time spent inside each MPI function call. This inability to characterize communication patterns also affected the remainder of our benchmark suite: HPCToolkit makes it very easy to identify compute-bound sections of code, but it can be difficult or impossible to infer communication problems when only profiled data for metrics is available. HPCToolkit averaged less than 20% overhead for all applications in our bottleneck test suite.

HPCToolkit is exceptionally good at debugging performance problems in sequential code and other code with little communication, such as threaded code. HPCToolkit also excels in its ability to relate performance metrics to lines of source code, even in the face of aggressive loop transformations. However, the data presented by HPCToolkit can be very low-level, often showing where time is being spent inside a library (such as time spent in internal MPI function calls). It would be helpful to provide an option where timing information is related to callsites inside a user's code, rather than to internal routines used by libraries, since most users are unable or unwilling to change external library code for better performance.

In general, HPCToolkit is best used to troubleshoot performance problems in sequential parts of a parallel application. HPCToolkit's integration with PAPI and its excellent source code correlation make it very valuable for tuning code to minimize cache misses and maximize floating-point operations per second.

3.2.3 mpiP 2.8

mpiP [32] is a lightweight tool for profiling MPI applications. It supports Linux, Tru64, IBM AIX, Cray UNICOS, and IBM's BlueGene/L platforms. While many tools have problems running on systems with thousands of processors, mpiP has been successfully run on massively-parallel machines. mpiP also contains a comprehensive, portable address-to-source translation and stackwalking API that it uses to collect information about MPI callsites while only using the PMPI instrumentation interface.

mpiP records statistical information about the MPI communication for a program: time per MPI task, aggregate message size and time per MPI callsite, time and message stats per MPI callsite, and I/O stats per MPI callsite. It also provides a basic summary of the amount of time in a user's application that can be attributed to MPI calls. After running an MPI program, mpiP produces a simple text output file. The contents of this text file are human-readable, but the output file can also be used with the `Mpipview` program (distributed as part of Tool Gear [7]) which shows the information alongside the user's source code.

In the bottleneck suite, mpiP was able to identify problematic MPI callsites in code, but was not able to provide any information about non-MPI parts of each benchmark, because it only uses PMPI to gather performance data. For the CAMEL benchmark, mpiP showed that the application spends a small fraction of time within MPI functions and illustrated a few MPI send and receive pairs that had a time imbalance. For the LU benchmark, it showed that a significant fraction of execution time was spent in MPI calls, but since the communication calls are spread out in the application, mpiP was not able to narrow down the excessive

communication to a problematic set of callsites. mpiP also performed well on the PPerfMark tests, but had trouble with the benchmarks that had time-varying problems (such as `diffuse-procedure` and `hot-procedure`). The overhead imposed by mpiP was small, averaging less than 3% for all benchmarks.

While mpiP provides only a small amount of performance information when compared with other, more full-featured tools, it can still be quite valuable when trying to troubleshoot MPI communication problems that occur as the system size is scaled up to several thousand processors.

3.2.4 Other Profiling Tools

Profiling tools have long been used with sequential programs, and many vendors have shipped tools such as `prof`, `gprof`, or `pixie` as part of their standard software installations. Most of these vendor-supplied tools rely on callstack sampling and debug symbols to relate performance information back to lines of source code. Many parallel profiling tools try to provide a straightforward parallel extension to `prof`-like tools, such as PGI's pgprof [28], vprof [12], HPM Toolkit [29], and PerfSuite [13], in addition to the text outputs from HPCToolkit's `hpcprof` and TAU's `pprof` tools mentioned above. Two more ambitious commercial profiling tools that attempt to provide a user with suggestions on how to fix bottlenecks in their code are Crescent Bay Software's DEEP/MPI and Intel's VTune software (although VTune is currently available for sequential programs on Intel platforms only).

SvPablo [5] is a profiling tool that displays performance metrics color coded alongside a user's source code. Unlike the other profiling tools mentioned above, SvPablo has a graphical interface that lets users perform source instrumentation interactively. SvPablo also is able to do basic cross-experiment performance analysis by automatically producing parallel efficiency charts based on recorded performance data. SvPablo also uses a language-independent file format for recording performance data, which lets the interface visualize profile data from any language. In our testing with SvPablo, we had problems getting the built-in C parser to recognize modern C99 code. SvPablo performed well on our bottleneck suite, but suffered from similar problems experienced by other profile-only tools; namely, communication bottlenecks are sometimes impossible to pinpoint when only profiling information is available.

AIMS [35] is a tool written by the NASA AMES research center that is probably closest in functionality to TAU. AIMS supports overhead compensation, along with some interesting visualizations and statistical analysis capabilities. Unfortunately, AIMS has not been updated since 1999, and we have not been successful at running it on any modern platforms.

Cray's PAT [6] offers support for sampling, profiling, and tracing, and is available on most of Cray's platforms such as the X1/X1E and XD1. Cray PAT uses static binary instrumentation and can export data in XML format for use with other tools. The basic analysis capabilities provided by Cray PAT are similar to `prof`-like tools, and Cray PAT can also be coupled with Apprentice[2] for visualizing both MPI traces and sampled program data. Cray PAT supports a wide variety of languages, including UPC, SHMEM, Co-Array Fortran, MPI, and OpenMP programs.

3.3 Online Analysis Tools

In this section, we describe tools that mainly rely on online analysis techniques instead of post-mortem analysis techniques. We categorize these tools separately from the profiling and tracing tools above because their online analysis techniques result in vastly different usage. We also present brief descriptions of other online analysis tools in Section 3.3.2.

3.3.1 Paradyn 4.1.1

Paradyn [20] is an online performance analysis tool developed by the University of Wisconsin-Madison. Paradyn runs on most current platforms, including Solaris 8 and 9 (SPARC), Linux (x86), Windows 2000 and Windows XP (x86), AIX (PowerPC), IRIX 6.5 (MIPS) and Tru64 (AlphaServer). Paradyn supports both threaded C code, in addition to C and Fortran MPI programs, and has been in existence for several years.

Paradyn's authors noted the problems associated with storing and analyzing large amounts of trace data, and came up with a solution that tries to avoid these problems. Instead of recording a complete trace of all application behavior, Paradyn relies on online performance analysis. Paradyn performs instrumentation of binaries at runtime as needed, trying to keep instrumentation overhead minimal. Instrumentation code can be inserted or removed at will by users at runtime. Additionally, since users cannot hope to keep up with the flow of incoming performance information in large-scale systems, Paradyn also provides an automated performance bottleneck search routine named the Performance Consultant.

Paradyn is composed of several complementary pieces of software, all attached to a graphical user interface (GUI). When users launch their program using the main GUI, Paradyn also launches several monitoring daemon processes on each node. When users select a visualization or perform another action that requires performance data, the GUI communicates with each daemon and requests instrumentation code to be inserted in the running program. Each daemon performs the instrumentation, begins recording data, and periodically sends back data samples to the main GUI. In this manner, Paradyn can be thought of as a dynamic sampling tool. The sampled performance data is stored in a round-robin database, which is then presented to the user graphically through one of Paradyn's visualizations. Paradyn has several visualizations that are geared towards displaying sampled data, including two- and three-dimensional bar charts, histograms, and tables.

When the Performance Consultant is used, a similar sequence of actions occur, except they are controlled by the Performance Consultant's search routine instead of by the user. The bottleneck search uses the W^3 model to guide the bottleneck search over a set of candidate bottlenecks. The W^3 model attempts to answer *why*, *where*, and *when* the application is performing poorly by correlating performance bottlenecks to specific classes of bottlenecks, nodes of a machine and functions in source code, and phases of a user's program (which the user has to manually specify based on time intervals).

Dynamic binary instrumentation avoids many of the problems associated with other forms of instrumentation (such as interference with compiler optimizations and amount of work that must be done by a user), but is a technically chal-

lenging problem. Thankfully, the dynamic instrumentation performed by Paradyn is exposed to other tools through the DynInst library.

The Performance Consultant performed well on our bottleneck suite. The Performance Consultant was especially good at identifying computational and synchronization bottlenecks in the CAMEL and LU benchmarks. However, it did not perform as well with communication problems in the PPerfMark suite that could not be described using simple threshold values for metrics, such as the `wrong-way` benchmark. In many cases it was able to point us in the general direction of the problem, but could not provide enough information on what was causing the bottleneck (e.g., messages being sent in backwards order). And while the overhead experienced with Paradyn was almost unnoticeable in most cases, allowing the Performance Consultant to perform searches on code with lightweight functions generally resulted in several orders of magnitude of overhead.

We appreciate the technical challenges that Paradyn has set out to solve, but do have a few comments based on our experiences with the tool. While the Performance Consultant can be especially valuable for inexperienced users (and useful for experienced users as a "first pass" for identifying potential problem spots), it tends to be easily confused on applications with complex performance problems such as the LU benchmark. It would also be nice to have a notion of severity for each benchmark, instead of a simple cutoff scheme as currently used. Also, during normal use, the user interface tends to get cluttered with lots of different open windows. The interface could also benefit from the use of a more modern widget set. While binaries are provided for almost all supported platforms, it would be beneficial to have an easier process for users wishing to build from source, as building Paradyn and DynInst from source is very error-prone. And while we appreciated the ability to attach Paradyn to existing long-running processes, the version we tested had no way to detach from a monitored process. We also would like to see support for other popular high-performance computing platforms, such as the Cray X1E and Opteron-based platforms. Finally, we feel Paradyn would benefit from an "unsupervised" mode that could make it usable in batch systems, as it currently cannot be used in non-interactive environments.

In general, Paradyn is a useful research vehicle to test out online performance analysis ideas and is also useful for researchers wishing to interact with an application while it is running. Since no recompilation is required to use the tool, it seems well-suited for troubleshooting performance problems in "production" applications.

3.3.2 Other Online Analysis Tools

DynaProf [26] merges the DynInst library with PAPI, enabling users to dynamically instrument and run executables using a `gdb`-like interface. It works especially well with sequential and threaded programs, but can be difficult to use with MPI programs since the spawned MPI processes have to be overseen by DynPprof.

DynTG [30] is another performance tool that marries two separate performance tool frameworks: DPCL [4] and Tool Gear [7]. The Tool Gear component provides a source-level browser where users can instrument their program during runtime by clicking on source code lines in the browser. In effect, DynTG is similar to SvPablo, except that the instrumentation and measurement changes occur at runtime and do not require a recompilation. DynTG is currently not available, but according to the developers, a release is planned soon.

4. CONCLUSIONS

In this paper, we presented the findings of our tool evaluation study. We have reviewed many different types of performance tools, and have given recommendations on when it is appropriate to use each tool.

Based on our experiences with the tool evaluations, we have learned that the most effective performance tools can display data to the user at both high levels (profiled data) and low levels (traced data and hardware counters). Profile data can provide a top-down view of time spent in an application, which a user can interpret to focus their tuning efforts. On the other hand, trace data can better illustrate the temporal relationships and data dependencies of application behavior, albeit at a very low level. Additionally, we found that relating performance information back to *actual lines* in users' source code is critical for them to determine how to fix performance bottlenecks in their code.

Hardware counters are also useful for finding out how well an application is running on a given hardware platform, as they can be used to directly compare metrics such as FLOPS with the theoretical peak numbers for a given platform. Matching these metrics with other data such as data cache misses can also give provide more insight into where performance is being lost in an application.

One other observation we have made with performance tools is that most tended to be difficult to install and hard to learn. While performance tools have made several strides with regards to usability, most tools could benefit from clearer documentation and "quickstart" guides that allow users to reduce the time needed to become productive with a tool. Additionally, performance tools should follow established guides for user interface design whenever possible, and most tools would greatly benefit from usability studies. Tool developers should strive to minimize the amount of effort a user has to exert in order to use their tool. In general, the more work a user has to do in order to use the tool, the less likely they are to actually use it.

5. REFERENCES

[1] S. Browne, C. Deane, G. Ho, and P. Mucci. PAPI: A portable interface to hardware performance counters. In *Department of Defense HPCMP Users Group Conference*, Monterey, CA, United States, June 1999.

[2] H. Brunst, D. Kranzlmüller, and W. E. Nagel. Tools for scalable parallel program analysis - vampir vng and dewiz. In *Distributed and Parallel Systems: Cluster and Grid Computing (DAPSYS)*, pages 93–102, Budapest, Hungary, September 2004.

[3] A. Chan, W. Gropp, and E. Lusk. Scalable log files for parallel program trace data(draft), 2000.

[4] L. DeRose, T. Hoover, and J. K. Hollingsworth. The dynamic probe class library-an infrastructure for developing instrumentation for performance tools. In *International Parallel and Distributed Processing Symposium (IPDPS)*, San Francisco, CA, United States, April 2001.

[5] L. DeRose, Y. Zhang, and D. Reed. Svpablo: A multi-language performance analysis system. In *10th International Conference on Computer Performance Evaluation - Modeling Techniques and Tools - Performance Tools*, pages 352–355, Palma de Mallorca, Spain, September 1998.

[6] J. Galarowicz and B. Mohr. Analyzing message passing programs on the Cray T3E with PAT and VAMPIR. In *4th European CRAY-SGI MPP Workshop*, pages 29–49, Garching/Munich, Germany, October 1998.

[7] J. Gyllenhaal and J. May. Tool gear website. `http://www.llnl.gov/CASC/tool_gear/`.

[8] M. T. Heath and J. A. Etheridge. Visualizing the performance of parallel programs. *IEEE Softw.*, 8(5):29–39, 1991.

[9] J. K. Hollingsworth, B. P. Miller, and J. Cargille. Dynamic program instrumentation for scalable performance tools. In *Scalable High Performance Computing Conference (SHPCC)*, Knoxville, TN, United States, May 1994.

[10] J. K. Hollingsworth and M. Steele. Grindstone: A test suite for parallel performance tools. Technical Report CS-TR-3703, University of Maryland, 1996.

[11] K. A. Huck and A. D. Malony. Perfexplorer: A performance data mining framework for large-scale parallel computing. In *SC2005*, Seattle, WA, United States, November 2005.

[12] C. Janssen. Vprof web site. `http://aros.ca.sandia.gov/~cljanss/perf/vprof/`.

[13] R. Kufrin. Perfsuite: An accessible, open source performance analysis environment for Linux. In *6th International Conference on Linux Clusters: The HPC Revolution*, Chapel Hill, NC, United States, April 2005.

[14] J. Labarta, S. Girona, V. Pillet, T. Cortes, and L. Gregoris. Dip: A parallel program development environment. In *Euro-Par '96: Proceedings of the Second International Euro-Par Conference on Parallel Processing-Volume II*, pages 665–674, London, UK, 1996. Springer-Verlag.

[15] A. Leko, H. Sherburne, H.-H. Su, B. Golden, and A. George. Tool evaluation website. `http://www.hcs.ufl.edu/upc/toolevals/`.

[16] K. A. Lindlan, J. Cuny, A. D. Malony, S. Shende, B. Mohr, R. Rivenburgh, and C. Rasmussen. A tool framework for static and dynamic analysis of object-oriented software with templates. In *SC2000*, Dallas, TX, United States, November 2000.

[17] A. D. Malony. *Performance Observability*. PhD thesis, University of Illinois at Urbana-Champaign, October 1990.

[18] A. D. Malony. Tools for parallel computing: A performance evaluation perspective. In *Handbook on Parallel and Distributed Processing*, pages 342–363. Springer Verlag, 2000.

[19] J. Mellor-Crummey, R. J. Fowler, G. Marin, and N. Tallent. HPCVIEW: A tool for top-down analysis of node performance. *The Journal of Supercomputing*, 23(1):81–104, August 2002.

[20] B. P. Miller, M. D. Callaghan, J. M. Cargille, J. K. Hollingsworth, R. B. Irvin, K. L. Karavanic, K. Kunchithapadam, and T. Newhall. The paradyn parallel performance measurement tool. *IEEE Computer*, 28(11):37–46, November 1995.

[21] B. Mohr, D. Brown, and A. Malony. TAU: A portable parallel program analysis environment for pC++. In *CONPAR 94 - VAPP VI*, pages 29–40, University of Linz, Austria, LNCS 854, September 1994.

[22] B. Mohr and F. Wolf. KOJAK - a tool set for automatic performance analysis of parallel applications. In *European Conference on Parallel Computing (EuroPar)*, pages 1301–1304, Klagenfurt, Austria, LNCS 2790, August 26-29 2003. Springer-Verlag.

[23] K. Mohror and K. L. Karavanic. Performance tool support for MPI-2 on Linux. In *SC2004*, Pittsburgh, PA, United States, November 2004.

[24] A. E. Morales. *Automatic Performance Analysis of Parallel Programs*. PhD thesis, Department d'Informàtica, Universitat Autònoma de Barcelona, 2000.

[25] MPI Forum. MPI: a message passing interface. In *SC1993*, pages 878–883, Portland, OR, United States, 1993.

[26] P. Mucci. Dynaprof tutorial. In *SC2003*, Phoenix, AZ, United States, September 2003.

[27] W. E. Nagel, A. Arnold, M. Weber, H. C. Hoppe, and K. Solchenbach. VAMPIR: Visualization and analysis of MPI resources. *Supercomputer*, 12(1):69–80, 1996.

[28] Portland Group, Incorporated. pgprof web site. `http://www.pgroup.com/products/pgprof.htm`.

[29] L. D. Rose. The hardware performance monitor toolkit. In *7th International Euro-Par Conference on Parallel Processing*, pages 122–131, London, UK, 2001. Springer-Verlag.

[30] M. Schulz, J. May, and J. C. Gyllenhaal. Dyntg: A tool for interactive, dynamic instrumentation. In *5th International Conference on Computation Science, Part II*, pages 140–148, Atlanta, GA, United States, May 22-25 2005.

[31] S. Shende. *The Role of Instrumentation and Mapping in Performance Measurement*. PhD thesis, University of Oregon, August 2001.

[32] J. Vetter and M. McCracken. Statistical scalability analysis of communication operations in distributed applications. In *Principles and Practice of Parallel Programming (PPOPP)*, Snowbird, UT, United States, 2001.

[33] R. V. D. Wijngaart. NAS parallel benchmarks, version 2.4. Technical Report NAS Technical Report NAS-02-007, NASA Ames Research Center, Moffett Field, CA, 2002.

[34] P. H. Worley. A new PICL trace file format. Technical Report ORNL/TM-12125, Oak Ridge National Labs, 1992.

[35] J. Yan, S. Sarukkai, and P. Mehra. Performance measurement, visualization and modeling of parallel and distributed programs using the AIMS toolkit. *Software – Practice and Experience*, 25(4):429–461, 1995.