Performance Instrumentation and Measurement for Terascale Systems

Jack Dongarra[1], Allen D. Malony[2], Shirley Moore[1],
Philip Mucci[2], and Sameer Shende[2]

[1] Innovative Computing Laboratory, University of Tennessee
Knoxville, TN 37996-3450 USA
{dongarra,shirley,mucci}@cs.utk.edu
[2] Computer Science Department, University of Oregon
Eugene, OR 97403-1202 USA
{malony,sameer}@cs.uoregon.edu

Abstract. As computer systems grow in size and complexity, tool support is needed to facilitate the efficient mapping of large-scale applications onto these systems. To help achieve this mapping, performance analysis tools must provide robust performance observation capabilities at all levels of the system, as well as map low-level behavior to high-level program constructs. Instrumentation and measurement strategies, developed over the last several years, must evolve together with performance analysis infrastructure to address the challenges of new scalable parallel systems.

1 Introduction

Performance observation requirements for terascale systems are determined by the performance problem being addressed and the performance evaluation methodology being applied. Instrumentation of an application is necessary to capture performance data. Instrumentation may be inserted at various stages, from source code modifications to compile-time to link-time to modification of executable code either statically or dynamically during program execution. These instrumentation points have different mechanisms which vary in their ease of use, flexibility, level of detail, user control of what data can be collected, and intrusiveness.

To provide insight into program behavior on large-scale systems and point the way toward program transformations that will improve performance, various performance data must be collected. Profiling data show the distribution of a metric across source-level constructs, such as routines, loops, and basic blocks. Most modern microprocessors provide a rich set of hardware counters that capture cycle count, functional unit, memory, and operating system events. Profiling can be based one either time or various hardware-based metrics, such as cache misses, for example. Correlations between profiles based on different events, as well as event-based ratios, provide derived information that can help to quickly identify and diagnose performance problems. In addition to profiling data, capturing event traces of program events, such as message communication events, helps portray the temporal dynamics of application performance.

For terascale systems, a wide range of performance problems, performance evaluation methods, and programming environments need to be supported. A flexible and extensible performance observation framework can best provide the necessary flexibility in experiment design. Research problems to be addressed by the framework include the following: the appropriate level and location in the framework for implementing different instrumentation and measurement strategies, how to make the framework modular and extensible, and the appropriate compromise between the level of detail and accuracy of the performance data collected and the instrumentation cost.

The remainder of the paper is organized as follows. Section 2 describes the instrumentation mechanisms desirable to support in such a framework and Section 3 describes types of measurements. Section 4 explains how the instrumentation and measurement strategies are supported in the PAPI cross-platform hardware counter interface and in the TAU performance observation framework. Section 5 presents our conclusions.

2 Instrumentation

To observe application performance, additional instructions or probes are typically inserted into a program. This process is called *instrumentation*. Instrumentation can be inserted at various stages, as described below.

2.1 Source Code Instrumentation

Instrumentation at the source code level allows the programmer to communicate higher-level domain-specific abstractions to the performance tool. A programmer can communicate such events by annotating the source code at appropriate locations with instrumentation calls. Once the program undergoes a series of transformations to generate the executable code, specifying arbitrary points in the code for instrumentation and understanding program semantics at those points may not be possible. Another advantage of source code instrumentation is that once an instrumentation library targets one language, it can provide portability across multiple compilers for that language, as well as across multiple platforms. Drawbacks of source code instrumentation include possible changes in instruction and data cache behavior, interactions with optimizing compilers, and runtime overhead of instrumentation library calls.

Source code annotations can be inserted manually or automatically. Adding instrumentation calls in the source code manually can be a tedious task that introduces the possibility of instrumentation errors producing erroneous performance data. Some of these difficulties with manual source code instrumentation can be overcome by using a source-to-source preprocessor to build an automatic instrumentation tool. Tools such as Program Database Toolkit (PDT) [10] for C++, C and Fortran 90, can be used to automatically instrument subroutines, code regions, and statements.

2.2 Library Level Instrumentation

Wrapper interposition libraries provide a convenient mechanism for adding instrumentation calls to libraries. For instance, the MPI Profiling Interface [1] allows a tool developer to interface with MPI calls in a portable manner without modifying the application source code or having access to the proprietary source code of the library implementation. The advantage of library instrumentation is that it is relatively easy to enable and the events generated are closely associated with the semantics of the library routines.

2.3 Binary Instrumentation

Executable images can be instrumented using binary code-rewriting techniques, often referred to as binary editing tools or executable editing tools. Systems such as Pixie, ATOM [5], EEL [9], and PAT [6] include an object code instrumentor that parses an executable and rewrites it with added instrumentation code. The advantage of binary instrumentation is that there is no need to re-compile an application program and rewriting a binary file is mostly independent of the programming language. Also, it is possible to spawn the instrumented parallel program the same way as the original program, without any special modification as are required for runtime instrumentation [12]. Furthermore, since an executable program is instrumented, compiler optimizations do not change or invalidate the performance optimization.

2.4 Dynamic Instrumentation

Dynamic instrumentation is a mechanism for runtime code patching that modifies a program during execution. DyninstAPI [3] provides an efficient, low-overhead interface that is suitable for performance instrumentation. A tool that uses this API is called a *mutator* and can insert code snippets into a running program, which is called the *mutatee*, without re-compiling, re-linking, or event re-starting the program. The mutator can either spawn an executable and instrument it prior to its execution, or attach to a running program. Dynamic instrumentation overcomes some limitations of binary instrumentation by allowing instrumentation code to be added and removed at runtime. Also, the instrumentation can be done on a running program instead of requiring the user to re-execute the application. The disadvantage of dynamic instrumentation is that the interface needs to be aware of multiple object file formats, binary interfaces (32/64 bit), operating system idiosyncrasies, as well as compiler specific information (e.g., to support template name de-mangling in C++ from multiple C++ compilers). To maintain cross language, cross platform, cross file format, cross binary interface portability is a challenging task and requires a continuous porting effort as new computing platforms and multi-threaded programming environments evolve.

3 Types of Measurements

Decisions about instrumentation are concerned with the number and type of performance events one wants to observe during an application's execution. Measurement decisions address the types and amount of performance data needed for performance problem solving. Often these decisions involve tradeoffs of the need for performance data versus the cost of obtaining it (i.e., the measurement overhead). Post-mortem performance evaluation tools typically fall into two categories: profiling and tracing, although some provide both capabilities. More recently, some tools provide real-time, rather than post-mortem, performance monitoring.

3.1 Profiling

Profiling characterizes the behavior of an application in terms of aggregate performance metrics. Profiles are typically represented as a list of various metrics (such as inclusive/exclusive wall-clock time) that are associated with program-level semantics entities (such as routines, basic blocks, or statements in the program). Time is a common metric, but any monotonically increasing resource function can be used, such as counts from hardware performance counters. Profiling can be implemented by sampling or instrumentation based approaches.

3.2 Tracing

While profiling is used to get aggregate summaries of metrics in a compact form, it cannot highlight the time varying aspects of the execution. To study the post-mortem spatial and temporal aspects of performance data, event tracing, that is, the activity of capturing events or actions that take place during program execution, is more appropriate. Event tracing usually results in a log of the events that characterize the execution. Each event in the log is an ordered tuple typically containing a time stamp, a location (e.g., node, thread) an identifier that specifies the type of event (e.g., routine transition, user-defined event, message communication, etc.) and event-specific information. For a parallel execution, trace information generated on different processors may be merged to produce a single trace file. The merging is usually based on the timestamp which can reflect logical time or physical time.

3.3 Real-time Performance Monitoring

Post-mortem analysis of profiling data or trace files has the disadvantage that analysis cannot begin until after program execution has finished. Real-time performance monitoring allows users to evaluate program performance during execution. Real-time performance monitoring is sometimes coupled with application performance steering.

4 PAPI and TAU Instrumentation and Measurement Strategies

Understanding the performance of parallel systems is a complicated task because of the different performance levels involved and the need to associate performance information to the programming and problem abstractions used by application developers. Terascale systems do not make this task any easier. We need to develop performance analysis strategies and technique that are successful both in their accessibility to users and in their robust application. In this section, we describe our efforts in evolving the PAPI and TAU technologies to terscale use.

4.1 PAPI

Most modern microprocessors provide hardware support for collecting hardware performance counter data [2]. Performance monitoring hardware usually consists of a set

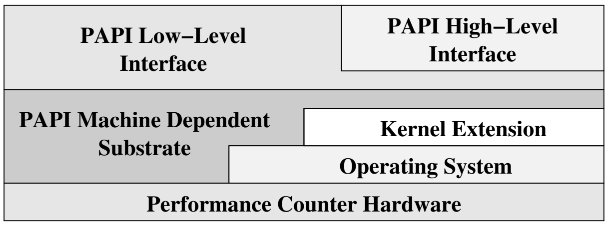

Fig. 1. Layered architecture of the PAPI implementation

of registers that record data about the processor's function. These registers range from simple event counters to more sophisticated hardware for recording data such as data and instruction addresses for an event, and pipeline or memory latencies for an instruction. Monitoring hardware events facilitates correlation between the structure of an application's source/object code and the efficiency of the mapping of that code to the underlying architecture.

Because of the wide range of performance monitoring hardware available on different processors and the different platform-dependent interfaces for accessing this hardware, the PAPI project was started with the goal of providing a standard cross-platform interface for accessing hardware performance counters [2]. PAPI proposes a standard set of library routines for accessing the counters as well as a standard set of events to be measured. The library interface consists of a high-level and a low-level interface. The high-level interface provides a simple set of routines for starting, reading, and stopping the counters for a specified list of events. The fully programmable low-level interface provides additional features and options and is intended for tool or application developers with more sophisticated needs.

The architecture of PAPI is shown in Figure 1. The goal of the PAPI project is to provide a firm foundation that supports the instrumentation and measurement strategies described in the preceding sections and that supports development of end-user performance analysis tools for the full range of high-performance architectures and parallel programming models. For manual and preprocessor source code instrumentation, PAPI provides the high-level and low-level routines described above. The `PAPI_flops` call is an easy-to-use routine that provides timing data and the floating point operation count for the bracketed code. The low-level routines target the more detailed information and full range of options needed by tool developers. For example the `PAPI_profil` call implements SVR4-compatible code profiling based on any hardware counter metric. Again, the code to be profiled need only be bracketed by calls to the `PAPI_profil` routine. This routine can be used by end-user tools such as VProf [3] to collect profiling data which can then be correlated with application source code.

Reference implementations of PAPI are available for a number of platforms (e.g., Cray T3E, SGI IRIX, IBM AIX Power, Sun Ultrasparc Solaris, Linux/x86, Linux/IA-64, HP/Compaq Alpha Tru64 Unix). The implementation for a given platform attempts to map as many of the standard PAPI events as possible to the available platform-

[3] `http:/aros.ca.sandia.gov/čljanss/perf/vprof/`

specific events. The implementation also attempts to use available hardware and operating system support – e.g., for counter multiplexing, interrupt on counter overflow, and statistical profiling.

Using PAPI on large-scale application codes, such as the EVH1 hydrodynamics code, has raised issues of scalability of the instrumentation. PAPI initially focused on obtaining aggregate counts of hardware events. However, the overhead of library calls to read the hardware counters can be excessive if the routines are called frequently – for example, on entry and exit of a small subroutine or basic block within a tight loop. Unacceptable overhead has caused some tool developers to reduce the number of calls through statistical sampling techniques. On most platforms, the current PAPI code implements statistical profiling over aggregate counting by generating an interrupt on counter overflow of a threshold and sampling the program counter. On out-of-order processors the program counter may yield an address that is several instructions or even basic blocks removed from the true address of the instruction that caused the overflow event. The PAPI project is investigating hardware support for sampling, so that tool developers can be relieved of this burden and maximum accuracy can be achieved with minimal overhead. With hardware sampling, an in-flight instruction is selected at random and information about its state is recorded – for example, the type of instruction, its address, whether it has incurred a cache or TLB miss, various pipeline and/or memory latencies incurred. The sampling results provide a histogram of the profiling data which correlates event frequencies with program locations.

In addition, aggregate event counts can be estimated from sampling data with lower overhead than direct counting. For example, the new PAPI substrate for the HP/Compaq Alpha Tru64 UNIX platform is built on top of a programming interface to DCPI called DADD (Dynamic Access to DCPI Data). DCPI identifies the exact address of an instruction, thus resulting in accurate text addresses for profiling data [4]. Test runs of the PAPI `calibrate` utility on the substrate have shown that event counts converge to the expected value, given a long enough run time to obtain sufficient samples, while incurring only one to two percent overhead, as compared to up to 30 percent on other substrates that use direct counting. A similar capability exists on the Itanium and Itanium 2 platforms, where Event Address Registers (EARs) accurately identify the instruction and data addresses for some events. Future versions of PAPI will make use of such hardware assisted profiling and will provide an option for estimating aggregate counts from sampling data.

The `dynaprof` tool that is part of the most recent PAPI release uses dynamic instrumentation to allow the user to either load an executable or attach to a running executable and then dynamically insert instrumentation probes [11]. Dynaprof uses Dyninst API [3] on Linux/IA-32, SGI IRIX, and Sun Solaris platforms, and DPCL [4] on IBM AIX. The user can list the internal structure of the application in order to select instrumentation points. Dynaprof inserts instrumentation in the form of *probes*. Dynaprof provides a PAPI probe for collecting hardware counter data and a wallclock probe for measuring elapsed time, both on a per-thread basis. Users may optionally write their own probes. A probe may use whatever output format is appropriate, for example a real-time data feed to a visualization tool or a static data file dumped to

[4] `http://oss.software.ibm.com/developerworks/opensource/dpcl/`

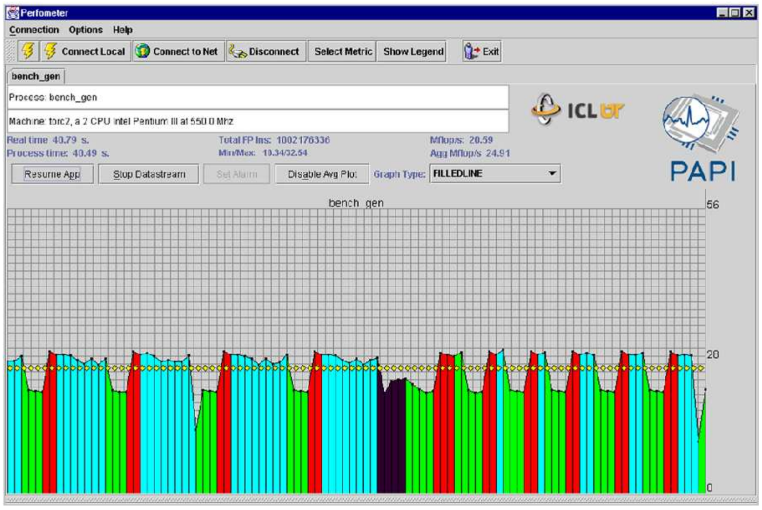

Fig. 2. Real-time performance analysis using Perfometer

disk at the end of the run. Future plans are to develop additional probes, for example for VProf and TAU, and to improve support for instrumentation and control of parallel message-passing programs.

PAPI has been incorporated into a number of profiling tools, including SvPablo [5], TAU and VProf. In support of tracing, PAPI is also being incorporated into version 3 of the Vampir MPI analysis tool [6]. Collecting PAPI data for various events over intervals of time and displaying this data alongside the Vampir timeline view enables correlation of event frequencies with message passing behavior.

Real-time performance monitoring is supported by the *perfometer* tool that is distributed with PAPI. By connecting the graphical display to the backend process (or processes) running an application code that has been linked with the perfometer and PAPI libraries, the tool provides a runtime trace of a user-selected PAPI metric, as shown in Figure 2 for floating point operations per second (FLOPS). The user may change the performance event being measured by clicking on the Select Metric button. The intent of perfometer is to provide a fast coarse-grained easy way for a developer to find out where a bottleneck exists in a program. In addition to real-time analysis, the perfometer library can save a trace file for later off-line analysis. The dynaprof tool described above includes a perfometer probe that can automatically insert calls to the perfometer setup and color selection routines so that a running application can be attached to and monitored in real-time without requiring any source code changes or recompilation or even restarting the application.

[5] `http://www-pablo.cs.uiuc.edu/Project/SVPablo/SvPabloOverview.htm`
[6] `http://www.pallas.com/e/products/vampir/index.htm`

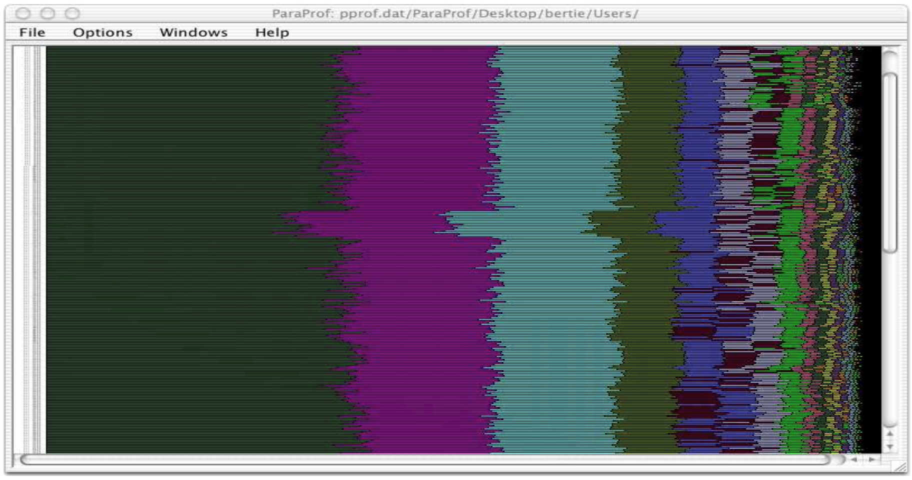

Fig. 3. Scalable SAMRAI Profile Display

4.2 TAU Performance System

The TAU (Tuning and Analysis Utilities) performance system is a portable profiling and tracing toolkit for parallel threaded and or message-passing programs written in Fortran, C, C++, or Java, or a combination of Fortran and C. The TAU architecture has three distinct parts: instrumentation, measurement, and analysis. The program can undergo a series of transformations that insert instrumentation before it executes. Instrumentation can be added at various stages, from compile-time to link-time to run-time, with each stage imposing different constraints and opportunities for extracting program information. Moving from source code to binary instrumentation techniques shifts the focus from a language specific to a more platform specific approach. TAU can be configured to do either profiling or tracing or to do both simultaneously.

Source code can be instrumented by manually inserting calls to the TAU instrumentation API, or by using PDT [10] to insert instrumentation automatically. PDT is a code analysis framework for developing source-based tools. It includes commercial grade front end parsers for Fortran 77/90, C, and C++, as well as a portable intermediate language analyzer, database format, and access API. The TAU project has used PDT to implement a source-to-source instrumentor (`tau_instrumentor`) that supports automatic instrumentation of C, C++, and Fortran 77/90 programs. TAU can also use DyninstAPI [3] to construct calls to the TAU measurement library and then insert these calls into the executable code. In both cases, a selective instrumentation list that specifies a list of routines to be included or excluded from instrumentation can be provided. TAU uses PAPI to generate performance profiles with hardware counter data. It also uses the MPI profiling interface to generate profile and/or trace data for MPI operations.

Recently, the TAU project has focussed on how to measure and analyze larg-scale application performance data. All of the instrumentation and measurement techniques discussed above apply. In the case of parallel profiles, TAU suffers no limitations in

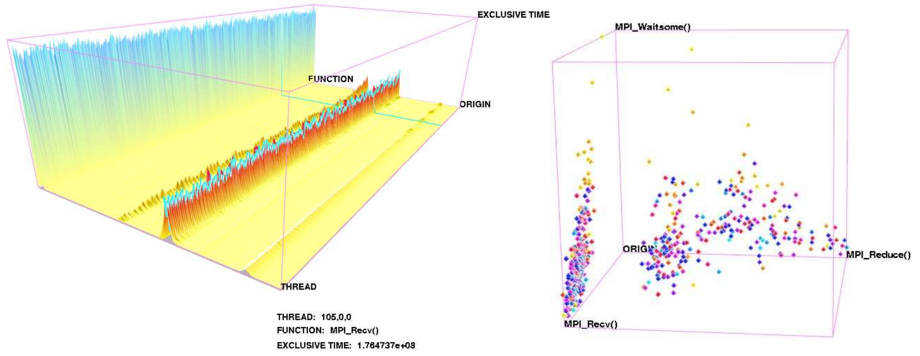

Fig. 4. Performance Profile Visualization of 500 Uintah Threads

the ability to make low-overehead performance measurement. However, a significant amount of performance data can be generated for large processor runs. The TAU ParaProf tool provide the user with means to navigate through the profile dataset. For example, we applied ParaProf to TAU data obtained during the profiling of a SAMRAI [8] application run on 512 processor nodes. Figure 3 shows a view of exclusive wall-clock time for all events. The display is fully interactive, and can be "zoomed" in or out to show local detail. Even so, some performance characteristics can still be difficult to comprehend when presented with so much visual data.

We have also been experimenting with three-dimensional displays of large-scale performance data. For instance, Figure 4 [7] shows two visualizations of parallel profile samples from a Uintah [7] application. The left visualization is for a 500 processor run and shows the entire parallel profile measurement. The performance events (i.e., functions) are along the x-axis, the threads are along the y-axis, and the performance metric (in this case, the exclusive execution time) is along the z-axis. This full performance view enables the user to quickly identify major performance contributors. The `MPI_Recv()` function is highlighted. The right display is of the same dataset, but in this case each thread is shown as a sphere at a coordinate point determined by the relative exclusive execution time of three significant events. The visualization gives a way to see clustering relationships.

5 Conclusions

Terascale systems require a performance observation framework that supports a wide range of instrumentation and measurement strategies. The PAPI and TAU projects are addressing important research problems related to construction of such a framework. The widespread adoption of PAPI by third-party tool developers demonstrates the value of implementing low-level access to architecture-specific performance monitoring hardware underneath a portable interface. Now tool developers can program to a single in-

[7] Visualizations courtesy of Kai Li, University of Oregon

terface, allowing them to focus their efforts on high-level tool design. Similarly, the TAU framework provides portable mechanisms for instrumentation and measurement of parallel software and systems.

Terascale systems require scalable low-overhead means of collecting relevant performance data. Statistical sampling methods, such as used in the new PAPI substrates for the Alpha Tru64 UNIX and Linux/Itanium/Itanium2 platforms, yield sufficiently accurate results while incurring very little overhead. Filtering and feedback schemes such as those use by TAU lower overhead while focusing instrumentation where it is most needed. Both PAPI and TAU projects are developing online monitoring capabilities that can be used to control instrumentation, measurement, and runtime performance data analysis. This will be important for effective performance steering in highly parallel environments.

Together, the PAPI [8] and TAU [9] projects have begun the construction of a portable performance tool infrastructure for terascale systems designed for interoperability, flexibility, and extensibility.

References

1. MPI: A message passing interface standard. *International Journal of Supercomputing Applications*, 8(3/4), 1994.
2. S. Browne, J. Dongarra, N. Garner, G. Ho, and P. Mucci. A portable programming interface for performance evaluation on modern processors. *International Journal of High Performance Computing Applications*, 14(3):189–204, 2000.
3. B. Buck and J. Hollingsworth. An API for runtime code patching. *International Journal of High Performance Computing Applications*, 14(4):317–329, 2000.
4. J. Dean, C. Waldspurger, and W. Weihl. Transparent, low-overhead profiling on modern processors. In *Workshop on Profile and Feedback-directed Compilation*, October 1998.
5. A. Eustace and A. Srivastava. ATOM: A flexible interface for building high performance program analysis tools. In *Proc. USENIX Winter 1995*, pages 303–314, 1995.
6. J. Galarowics and B. Mohr. Analyzing message passing programs on the Cray T3E with PAT and VAMPIR. Technical report, ZAM Forschungszentrum: Juelich, Germany, 1998.
7. J. S. Germain, A. Morris, S. Parker, A. Malony, and S. Shende. Integrating performance analysis in the uintah software development cycle. In *High Performance Distributed Computing Conference*, pages 33–41, 2000.
8. R. Hornung and S. Kohn. Managing application complexity in the samrai object-oriented framework. *Concurrency and Computation: Practice and Experience*, special issue on Software Architectures for Scientific Applications, 2001.
9. J. Larus and T. Ball. Rewriting executable files to measure program behavior. *Software Practice and Experience*, 24(2):197–218, 1994.
10. K. Lindlan, J. Cuny, A. 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 *Proc. SC 2000*, 2000.
11. P. Mucci. Dynaprof 0.8 user's guide. Technical report, Nov. 2002.
12. S. Shende, A. Malony, and R. Bell. Instrumentation and measurement strategies for flexible and portable empirical performance evaluation. In *International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'2001)*, 2001.

[8] For further information: `http://icl.cs.utk.edu/papi/`

[9] For further information: `http://www.cs.uoregon.edu/research/paracomp/tau/`