Performance and Memory Evaluation Using TAU

Sameer Shende, Allen D. Malony, Alan Morris
Department of Computer and Information Science
University of Oregon, Eugene, OR, USA
{sameer,malony,amorris}@cs.uoregon.edu

Peter H. Beckman
Mathematics and Computer Science Division
Argonne National Laboratory, Argonne, IL, USA
beckman@mcs.anl.gov

Abstract

The TAU performance system is an integrated performance instrumentation, measurement, and analysis toolkit offering support for profiling and tracing modes of measurement. This paper introduces memory introspection capabilities of TAU featured on the Cray XT3 Catamount compute node kernel. TAU supports examining the memory headroom, or the amount of heap memory available, at routine entry, and correlates it to the program's callstack as an atomic event.

Keywords: TAU, performance mapping, memory headroom, measurement, instrumentation, performance evaluation

1 Introduction

With the development of new generations of large-scale parallel machines comes the question of what performance data is important to observe and how it should be observed. While the increasing complexity of high-performance systems suggests that new machine features will require performance measurement facilities different from existing tools, there is a strong need to create empirical performance observation technologies that are cross-platform, reusable, and open source. Robust performance technology must address the dual goals of portability and specificity, providing for the latter means extending existing observation capabilities to support machine-specific features, without sacrificing usability and efficiency in how the technology is applied.

In this paper we discuss how the TAU parallel performance system contends with issues of system diversity while maintaining a consistent performance measurement model and observation flexibility. The parallel machine environment we study is the Cray XT3, in particular, the system interfaces for extracting performance and memory information. We describe how TAU incorporates these interfaces in the general TAU measurement facility. In Section §2 , we describe in detail TAU's instrumentation approach and capabilities. TAU's measurement options are explained in Section §3. Section §4 shows how we have enabled TAU to use the memory introspection capabilities in the XT3 Catamount compute node kernel. Results from experiments are shown. Conclusions are given in Seciton §5.

2 Instrumentation

Parallel performance evaluation tools typically employ either profiling or tracing modes of measurement. While profiling records aggregate performance metrics during program execution, tracing relies on timestamped event logs to re-create the temporal variation in performance, showing program events along a global timeline. Instrumentation calls are inserted in the program to activate events that characterize actions that occur in a program. Sampling and measured profiling are two common techniques to activate the instrumentation. Sampling incurs a fixed measurement overhead and relies on a periodic interrupt from the operating system based on an interval timer or hardware performance counters. When an interrupt takes place, the program state is recorded and at the end of execution, the tool estimates the contribution of different routines and statements based on the samples collected. Tools such as SGI's Open SpeedShop[27] and gprof [11, 10] fall into this category. Tools based on measured instrumentation, on the other hand, insert hooks at specific locations in the program, such as routines and loops. At the entry and exit points of these program entities instrumentation is activated and measurements are performed.

2.1 TAU

The TAU performance system relies on measured instrumentation to support both profiling and tracing performance models. The TAU framework architecture is organized into three layers - instrumentation, measurement and analysis - where within each layer multiple modules are available and can be configured to suit user needs. TAU supports a flexible instrumentation model that allows the user to insert performance instrumentation using the measurement API at different levels of program code representation, compilation, transformation, linking and execution. The key concept of the instrumentation layer is the definition of performance events. It supports several types of performance events to characterize program performance. Events describing a region of code such as routines, basic blocks or loops use a pair of markers to start and stop timers. Atomic user-defined events take place at a given point in the code and are triggered with some application level data (such as the size of message transmitted, the size of memory allocated etc.).

2.2 Program Database Toolkit

Techniques to insert code annotations range from manual instrumentation using the TAU API (available in C++, C, Fortran, Python and Java) to more automated modes of inserting these annotations. Using the Program Database Toolkit (PDT) [16], the *tau_instrumentor* tool can examine the source code locations and re-write the original source code with annotations for identifying regions of interest. PDT comprises of modified commercial-grade compiler front-ends that emit program information in the form of a program database (PDB) ASCII text files. These are subsequently parsed by the DUCTAPE library. Currently, PDT provides the EDG front-end for C++ and C and the GNU gfortran [25], Cleanscape Flint, and Mutek Solutions front-ends for Fortran. Using PDT, TAU supports automatic instrumentation of C++, C and Fortran programs using interval events. There are several ways to identify interval events and it is probably more recognizable to talk about interval events as static timers.

2.3 Timers and Phases

Static timers store the name of a program entity and its performance data and are constructed only once. When we aggregate the cost of each routine invocation, we use static timers (e.g. the time spent in 100 invocations of MPI_Send is 100 seconds. TAU also supports dynamic timers that are constructed each time with a unique name. Dynamic timers are useful for capturing the cost of execution of program entities that have different cost for each invocation. By embedding an iteration count in a routine name, we can invoke a dynamic timer and provide a unique instance specific name to it (e.g. the time spent in the second invocation of routine foo was 12 seconds). Sometimes, it is useful to characterize the program performance based on higher user-level abstractions such as application phases. TAU supports both static and dynamic phases. Phases record the time spent in a given region as well as the time spent in all program entities called directly and indirectly in a given phase. Phases may be nested but may not overlap. Static phases, like static timers aggregate performance data for all invocations (e.g. the total time spent in MPI_Send when it was called by all instances of the IO phase was 10 seconds). Dynamic phases can associate an instance specific name with the phase performance data (e.g., the time spent in MPI_Send when it was called by the fourth iterate phase was 4 seconds).

2.4 Preprocessor-Based Memory Instrumentation

Besides source-level instrumentation, TAU supports pre-processor based replacement of routines. We have implemented a memory tracking package that includes a wrapper library for malloc and free calls in C and C++. By re-directing references to these calls and extracting the source line number and file name, TAU computes the sizes of memory allocated and deallocated at different locations in the source code. Performance data is stored in atomic user-defined events that track statistics such as the number of samples, maximum, minimum, mean and standard deviation of the memory size. This data helps locate potential memory leaks. Tools such as dmalloc[26] employ similar instrumentation techniques for automated memory leak detection.

2.5 OpenMP Instrumentation

TAU can also invoke an OpenMP directive rewriting tool such as Opari[12] to re-write the OpenMP directives and insert hooks in the program prior to instrumentation. Opari relies on a fuzzy parser and has some restrictions such as requiring OpenMP end directives to be placed at the appropriate locations in the source code. To help this instrumentation process, we have developed the *tau_ompcheck* tool that examines the locations of loops using PDT and inserts the missing OpenMP directives and helps the Opari tool in instrumenting the source.

2.6 MPI and SHMEM Library Instrumentation

MPI provides a name-shifted interface that allows a performance tool to intercept any MPI call in a portable manner without requiring a vendor to supply the proprietary source code of the library and without requiring any modifications to the source code. This is provided by providing hooks into the native MPI library with a name-shifted PMPI interface and employing weak bindings. We have developed a TAU MPI wrapper library that acts as an interposition library that internally invokes the name-shifted MPI interface. Wrapped around the call, before and after, is TAU performance instrumentation. TAU and several other tools such as Upshot[29], VampirTrace, and EPILOG [12] use this approach.

Cray Inc. provides similar profiling interfaces to its SHMEM interface. We have created a TAU wrapper library for the Cray XT3 that targets its PSHMEM interface for profiling all SHMEM calls. In both cases, an application needs to be re-linked with the TAU MPI or SHMEM wrapper library.

2.7 Library Pre-loading

Operating systems such as Linux provide an LD_PRELOAD environment variable that specifies the name of a dynamic shared object that is loaded in the context of an application prior to execution. Using this scheme, TAU can preload the MPI wrapper interposition library on systems such as the Cray XD1 using the *tau_load.sh* shell script that ships with TAU. Cray XT3 does not support shared objects on the quintissential compute node kernel - Catamount. Hence, we need to re-link with the TAU MPI wrapper library.

2.8 Binary Instrumentation

TAU uses DyninstAPI [20] for instrumenting the executable code of a program. DyninstAPI is a dynamic instrumentation package that allows a tool to insert code snippets into a running program or to rewrite a binary, using a portable C++ class library. TAU's mutator program *tau_run* loads a TAU dynamic shared object in the address space of the mutatee (the application program) and instruments it at the level of routines. The Cray Apprentice[2] [28] tool also uses binary rewriting mechanism to insert instrumentation in the application.

2.9 Runtime Instrumentation for Python, Java and Component Software

TAU also supports instrumentation at the Python interpreter level and Java Virtual machine-based runtime instrumentation. Also, TAU's component [13, 15] interface for the Common Component Architecture (CCA) permits us to create proxy components that are placed between caller and callee ports. The port proxies are created using PDT to parse the source code. All these approaches load the TAU shared object in the context of the executing application for instrumentation.

As the source code undergoes a series of transformations in the compilation process, it poses several constraints and unique opportunities for program instrumentation. Instead of restricting the choice of instrumentation to one layer, TAU permits multiple instrumentation interfaces to be deployed concurrently for better coverage. It taps into performance data from multiple levels and presents it in a consistent and a uniform manner.

2.10 Instrumentation Optimization

TAU differs from other tools that provide automatic instrumentation capabilities in the way it addresses the issue of instrumentation optimization. To improve accuracy of performance measurements, instrumentation of low-level lightweight routines that are invoked frequently must be disabled. When the user defines an environment variable `TAU_THROTTLE` to control instrumentation overhead, it throttles and disables instrumentation in routines that are called over 100000 times and take less than 10 microseconds per call. These default values may be modified by the user by setting other environment variables (`TAU_THROTTLE_NUMCALLS` and `TAU_THROTTLE_PERCALL` respectively). This feature can reduce profiling overhead significantly. When tracing is chosen, it reduces the volume of trace data generated. Other forms of instrumentation optimization include the use of the *tau_reduce* tool [14, 4] that generates a selective instrumentation file from a profile dataset. This file contains a list of routines that should be excluded from instrumentation. The source instrumentor uses it to re-instrument the program. Other approaches include removing instrumentation overhead from the performance data by measuring the timer overhead during the measurement phase [6, 7, 8].

3 Measurement

TAU provides a range of measurement options that extend from the default option of flat profiles at one end to event tracing at the other. Between these two extremes, the user can choose phase-based profiling, memory profiling, memory headroom profiling, callpath profiling, calldepth profiling, and use dynamic timers, phases and context events to extend the profiling information. Phase based profiles allow the user to choose a set of representative phases in an application. Timers and phases that are invoked directly or indirectly by this phase are recorded [3]. Callpath profiles highlight the parent-child relationship between events and allow the user to set a runtime parameter (`TAU_CALLPATH_DEPTH`) to truncate its depth. The profiles show the timings along a branch of the program callgraph based on the leaf node. Call depth profiling allows the user to disable the instrumentation in timers that are a certain distance away from the root node. This is specified using an environment variable (`TAU_DEPTH_LIMIT`) and it works with all flat profiles, callpath profiles and traces. When callpath and depth profiling are enabled simultaneously and set to the same depth (k), we can see the contribution of all edges of the callgraph that are k deep from the root (instead of the leaf). Both Kojak's CUBE [9] and TAU's ParaProf [5] profile browsers support expansion and contraction of calltrees.

3.1 High Resolution Timers

TAU employs low-overhead high-resolution timers for performance measurements on several systems. On Cray XD1, TAU uses the high resolution time stamp counter register. On Cray X1, it uses the *_rtc()* call to measure clock ticks. On Cray XT3, it uses the *dclock()* call. By default, the ubiquitous *gettimeofday()* call is used. Profiling does not need a globally synchronized real-time clock like tracing. Thus, it is better to use the free-running timers on individual cpus for profiling and to use the default synchronized clocks for tracing program executions. The user can choose these high resolution timers during configuration by specifying the `-CRAYTIMERS` configuration option.

3.2 Multiple Metric Measurement

Tracing and profiling use a single metric for performance measurements by default. During profiling, TAU measures inclusive and exclusive time, the number of calls, and the child calls for each routine. However, there are cases where the user may want to measure more than one metric. In addition to the wallclock time, counts of hardware performance metrics and message sizes may be useful. TAU supports access to hardware performance counters using the PAPI interface [19]. The user can configure TAU using the `-MULTIPLECOUNTERS` option and set environment variables `COUNTER`*`1-N`* to specify the nature of performance metrics that should be recorded. The user can set these counters by choosing from preset events provided by PAPI e.g., `PAPI_FP_INS` for floating point instructions as well as native events that are processor specific e.g., `PAPI_NATIVE_`*`event name`*.

3.3 Memory Profiling

At the entry of each routine, TAU can inspect the heap memory used and trigger this value using user-defined events that tie it to the routine

USER EVENTS Profile :NODE 0, CONTEXT 0, THREAD 0

NumSamples	MaxValue	MinValue	MeanValue	Std. Dev.	Event Name
1	655	655	655	0	MFIX - Memory Headroom Available (MB)
486	579	575	576.9	1.04	BLANK_LINE - Memory Headroom Available (MB)
1	577	577	577	0	CHECK_DATA_00 - Memory Headroom Available (MB)
1405	503	502	503	0.1398	ACCUMULATION - Memory Headroom Available (MB)
300	503	502	503	0.14	ADJUST_A_U_G - Memory Headroom Available (MB)
300	503	502	503	0.14	ADJUST_A_U_S - Memory Headroom Available (MB)
2400	503	502	503	0.14	BOUND_X - Memory Headroom Available (MB)
174	525	525	525	0	CALC_CELL - Memory Headroom Available (MB)
1.408E+05	525	468	500.1	1.034	MPI_Startall() - Memory Headroom Available (MB)
21	525	491	523.4	7.241	MPI_UTILITY::BCAST_0C - Memory Headroom Available (MB)
42	525	489	523.3	7.457	MPI_UTILITY::BCAST_0I - Memory Headroom Available (MB)
1	491	491	491	0	MPI_UTILITY::BCAST_0R - Memory Headroom Available (MB)
2	509	496	502.5	6.5	MPI_UTILITY::GATHER_1C - Memory Headroom Available (MB)
128	491	441	459.3	12.39	MPI_UTILITY::GATHER_1D - Memory Headroom Available (MB)
11	508	461	479.5	14.33	MPI_UTILITY::GATHER_1I - Memory Headroom Available (MB)
1	526	526	526	0	CHECK_DATA_01 - Memory Headroom Available (MB)
1	526	526	526	0	CHECK_DATA_02 - Memory Headroom Available (MB)
1	502	502	502	0	ITERATE 1 - Memory Headroom Available (MB)
1	503	503	503	0	ITERATE 2 - Memory Headroom Available (MB)
1	503	503	503	0	ITERATE 3 - Memory Headroom Available (MB)
1	503	503	503	0	ITERATE 4 - Memory Headroom Available (MB)

Figure 1: **Performance data for memory headroom analysis**

name. This generates performance data that includes statistics such as the number of samples, maximum, minimum, mean, and standard deviation of the value. The user must configure TAU with the `-PROFILEMEMORY` configuration option to record heap memory utilization in this manner. TAU also supports tracking memory headroom, or the amount of memory available on the heap before the program runs out of memory. Again, an event correlates this value with the routine name, and generates these statistics as shown in Figure 1. To enable collection of this information, the user must choose the `-PROFILEHEADROOM` option during configuration. On the Cray XT3, TAU uses the *heap_info()* system call to extract the heap memory utilization and memory headroom available. On other systems, it computes the headroom available using a series of *malloc()* calls that allocate chunks of memory. The amount of memory requested at each call is twice the amount requested in the previous call. When the system runs out of memory, it requests for the smallest chunk and if it successful in allocating it, it requests twice as much until it cannot allocate even the smallest amount. At that point, all memory is exhausted and TAU computes the memory headroom in this manner. Thereafter, it frees up all the memory blocks allocated in this manner and associates this data with the currently executing routine. Clearly, the presence of the *heap_info()* system call helps us develop memory introspection tools and it would help tool developers if such library routines were available on other systems as well. TAU uses system specific interfaces where available and portable mechanisms elsewhere.

TAU also supports tracking heap and headroom using timer interrupts for a single global event. Headroom can be associated with the currently executing callpath as well. However, these calls need special annotations in the source code, unlike the two configuration options described above.

3.4 Integration in Application Build Environments

Each configuration and build of TAU results in a TAU library and a *stub makefile* that stores configuration parameters as makefile variables. Each stub makefile has a unique name that includes the compiler name (`-pgi`), the runtime system chosen (`-mpi or -shmem`), the measurement modules chosen (`-profile, -trace, -memory, -headroom, -callpath, -phase, -depth, etc.`) and the instrumentation mechanism (`-pdt`). TAU's automated source-level instrumentation using PDT is currently the most robust mechanism for portable performance evaluation and is available on most systems. To help integrate TAU in the build process, we have developed a compiler shell script

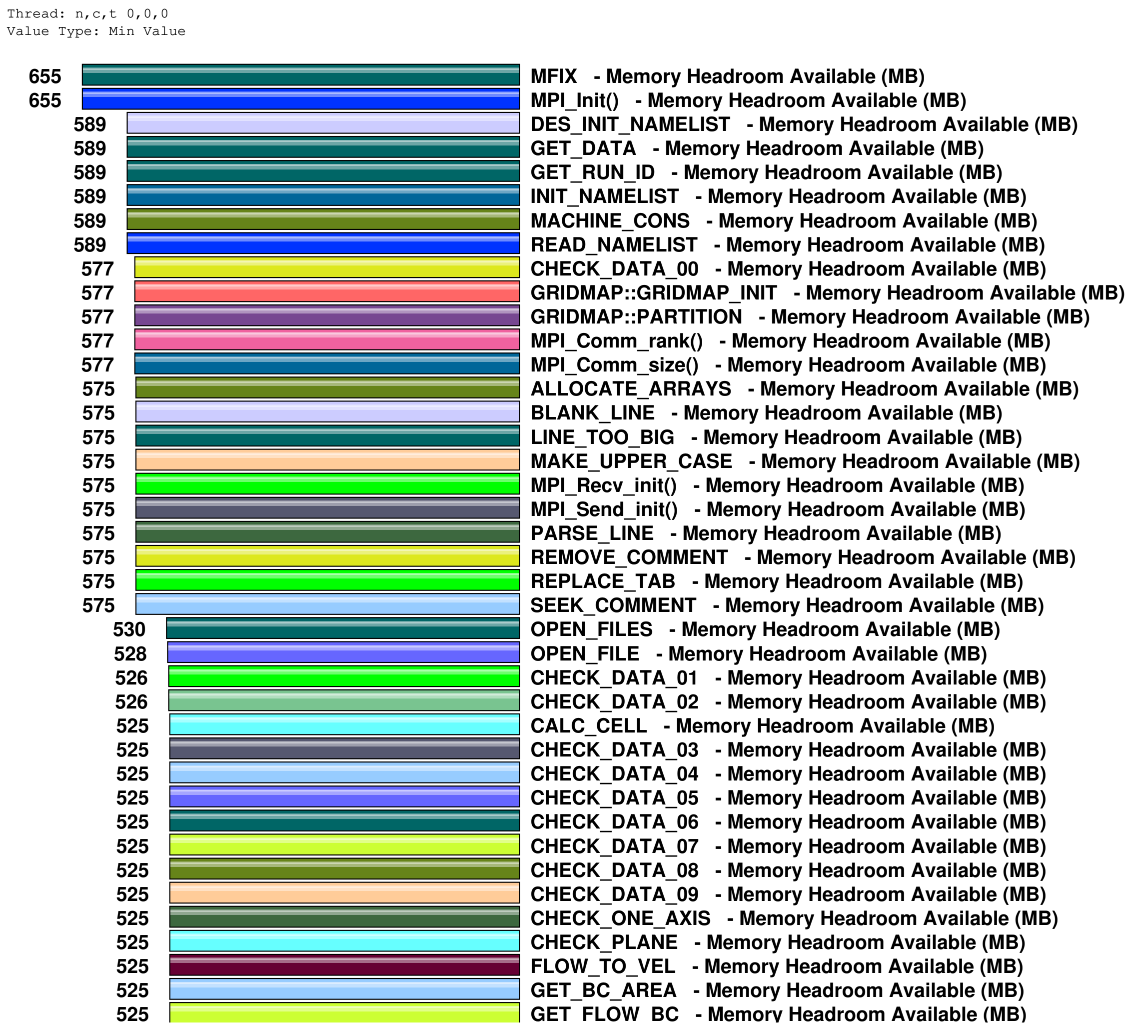

Figure 2: **Memory headroom available on node 0**

(`tau_compiler.sh`) that invokes a series of tools that finally create an instrumented object code. These include the C pre-processor, an optional OpenMP directive rewriter (`tau_ompcheck` followed by `opari`), the PDT parsers (`gfparse, f95parse, cparse,` and `cxxparse`) that emit a common PDB format, the TAU instrumentor (`tau_instrumentor`) that examines the PDB file, the source file, an optional instrumentation specification file, and writes the instrumented source file, and finally the compiler that compiles the instrumented source code to generate the object code. While linking, it takes care of supplying the appropriate TAU and MPI wrapper libraries along with system specific libraries such as the Fortran or C++ runtime libraries to create the executable. We provide the compiler wrapper scripts `tau_f90.sh`, `tau_cc.sh`, and `tau_cxx.sh` that can replace full fledged compilers in any build system. These scripts hide the multi-stage compilation process. By setting a pair of environment variables (`TAU_MAKEFILE` and `TAU_OPTIONS` as shown in Figure 3, the user may specify the measurement options chosen, and the optional parameters that are propagated to the source transformation phases, respectively.

```
xterm
# Step 1: Set the compiler names
F90    = tau_f90.sh
CXX    = tau_cxx.sh
CC     = tau_cc.sh

# instead of mpif90, mpicxx, and mpicc

# Step 2: Set environment variables before invoking make:
# 2a) set TAU stub makefile name
# setenv TAU_MAKEFILE /usr/local/tau-2.15.3/xt3/lib/Makefile.tau-mpi-pdt-pgi
#
# 2b) and optional parameters to pass to tau_compiler.sh:
# setenv TAU_OPTIONS -optVerbose -optTauSelectFile=select.tau -optPdtGnuFortranParser
# See tau_compiler.sh  for a complete list of options!

# Use the same compilation rules
OBJS = f1.o f2.o c1.o c2.o
app: $(OBJS)
        $(F90) $(OBJS) -o app $(LIBS)
.f90.o:
        $(F90) -c $(INCLUDE) $< -o $@
.cpp.o:
        $(CXX) -c $(INCLUDE) $< -o $@
clean:
        /bin/rm -f $(OBJS) app
"Makefile" 171L, 4335C written                              25,6          0%
```

Figure 3: **Modifications to the application makefile to use TAU are minimal**

4 Case Study: MFIX

To illustrate the use of memory headroom analysis and phase-based profiling, we instrumented the Multiphase Flow with Interphase eXchanges (MFIX [21, 22, 23]) application. MFIX is developed at the National Energy Technology Laboratory (NETL) and is used for studying hydrodynamics, heat transfer and chemical reactions in fluid-solid systems. We ran a simulation modeling the Ozone decomposition in a bubbling fluidized bed [24].

After instrumenting and running the application on 128 processors of the Cray XT3 at the Pittsburgh Supercomputing Center, we analyzed the performance data using the ParaProf profile browser. Figure 4 shows the aggregate profile of the application across all nodes. Figure 5 shows the average time spent in different routines and phases for all nodes. Figure 2 shows the minimum memory headroom available to each routine on node 0. This data reveals that when the `MFIX` application is loaded, 655 MB of memory headroom is available, this shrinks to 525 MB at the entry of the `CALC_CELL` routine. Figure 1 also highlights the memory headroom available. Figure 6 shows the relationship of the exclusive time spent in four routines for all processes in a four dimensional scatter plot. Each dimension in this scatter plot corresponds to a routine, metric pair. Each process is plotted with co-ordinates based on its values in these four routine metric pairs. We use this display to identify basic process clustering and patterns among processes. Using a combination of callpath profiles and memory headroom analysis, we are able to identify where the memory availability shrinks or grows to its lowest and the highest value. The parent of the given routine is responsible for memory allocation or de-allocation.

5 Conclusion

The integration of TAU in the Cray XT3 environment was a successful endeavor and demonstrated the ability to incorporate system-specific features for performance observation in TAU's portable and configurable measurement facility. In particular, the leverage we gained from the memory introspec-

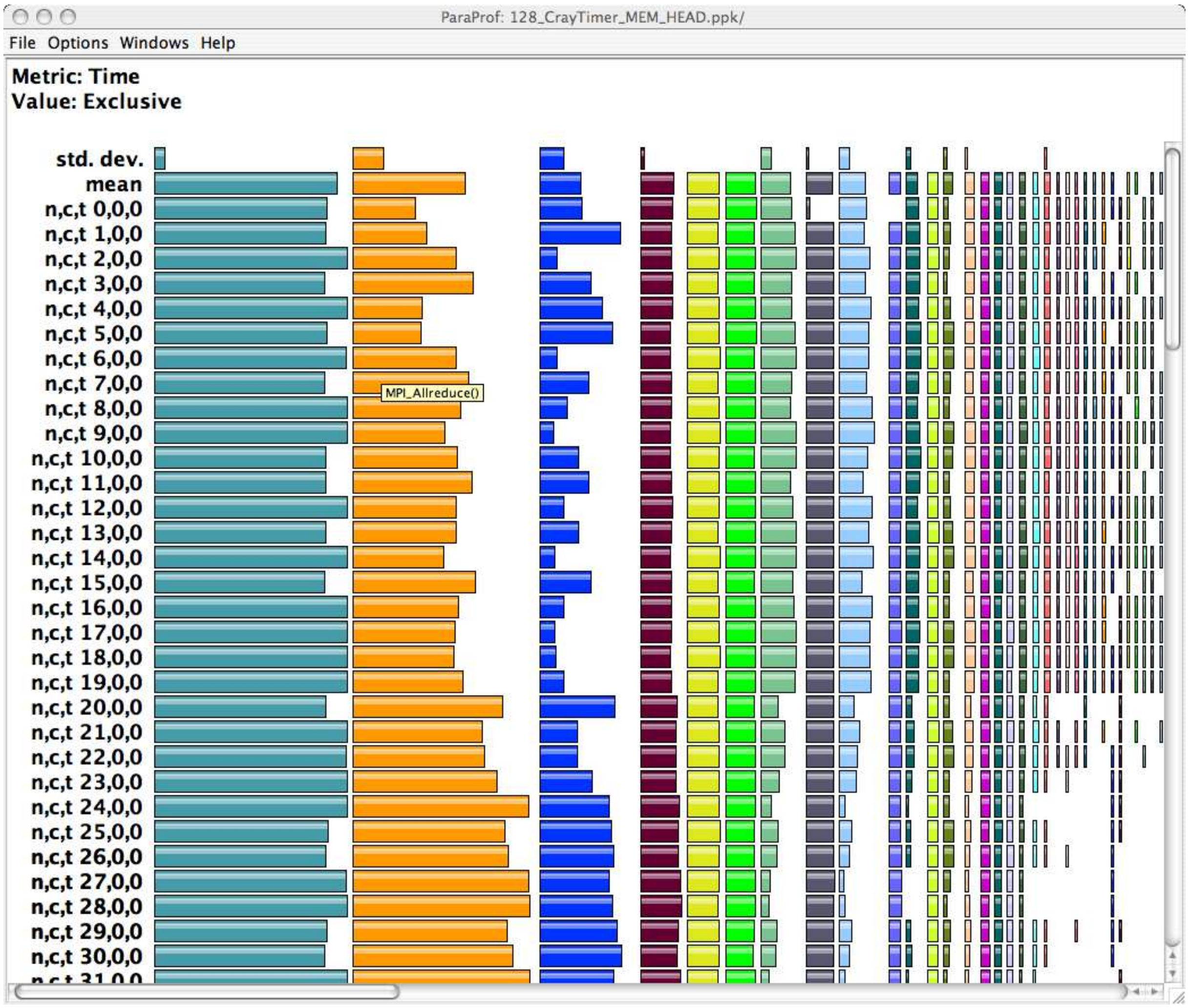

Figure 4: **ParaProf main window shows the time spent in different routines**

tion mechanisms in Cray's Catamount node kernel was significant and highlights how portable techniques provided by TAU can be improved by system-provided support. Certainly, we have witnessed such integration benefits on other platforms as well, but are especially satisfied with the range of integration specialization we achieved for the XT3. The work we have discussed is available as part of the latest TAU performance system distribution.

6 Acknowledgments

Research at the University of Oregon is sponsored by contracts (DE-FG02-05ER25663, DE-FG02-05ER25680) from the MICS program of the U.S. Dept. of Energy, Office of Science. We wish to thank Aytekin Gel of Aeolus Research and NETL for his work on the MFIX application, and the Pittsburgh Supercomputing Center for providing us access to the Cray XT3 system.

7 About the Authors

Sameer Shende is a postdoctoral research associate in the Performance Research Laboratory at the University of Oregon, and the president and director of ParaTools, Inc. Allen D. Malony is the director of the Neuroinformatics Center and a professor in the

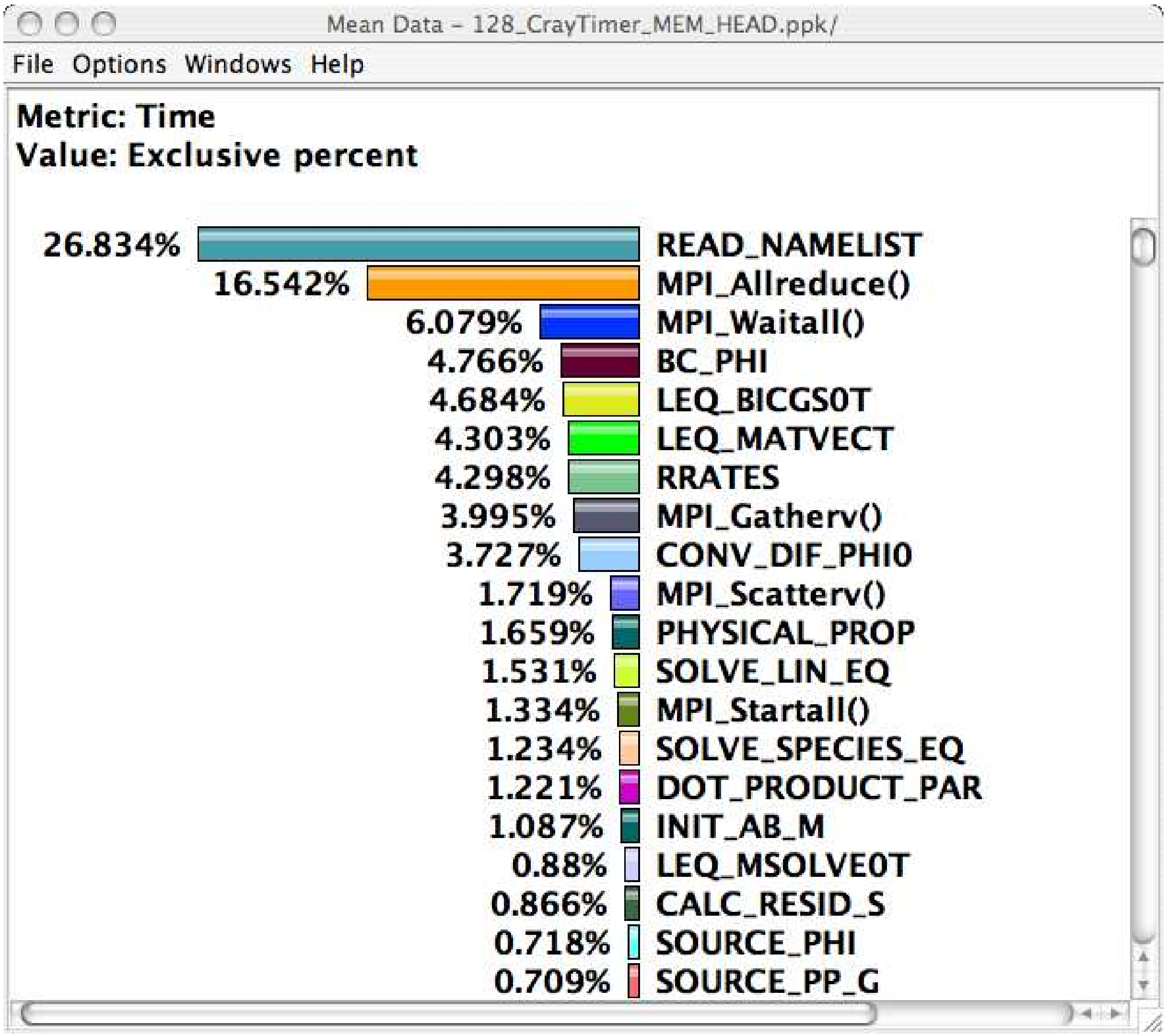

Figure 5: **The ParaProf mean profile window shows the average time spent in all routines**

Department of Computer and Information Science at the University of Oregon. He is also the CEO and director of ParaTools, Inc. Alan Morris is a software engineer in the Performance Research Laboratory at the University of Oregon. Peter H. Beckman is a computer scientist at the Mathematics and Computer Science Division at the Argonne National Laboratory.

References

[1] S. Shende, and A. Malony, "The TAU Parallel Performance System," In International Journal of High Performance Computing Applications, ACTS Collection Special Issue, Summer 2006.

[2] A. Malony, S. Shende, "Performance Technology for Complex Parallel and Distributed Systems," In G. Kotsis, P. Kacsuk (eds.), *Distributed and Parallel Systems, From Instruction Parallelism to Cluster Computing, Third Workshop on Distributed and Parallel Systems (DAPSYS 2000)*, Kluwer, pp. 37–46, 2000.

[3] A. D. Malony, S. Shende, and A. Morris, "Phase-Based Parallel Performance Profiling," In Proceedings of the PARCO 2005 conference, 2005.

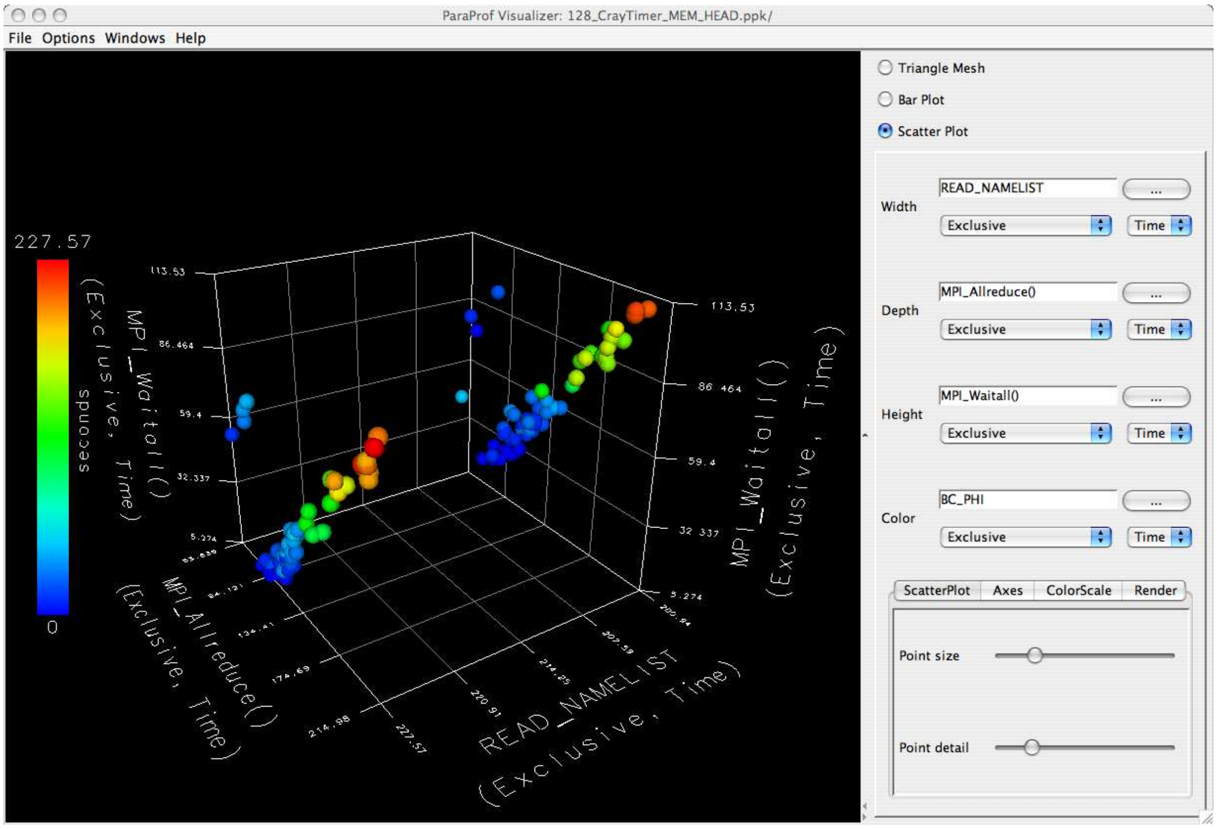

Figure 6: **ParaProf's scatter plot shows the relationship of upto four routines**

[4] S. Shende, A. D. Malony, and A. Morris, "Optimization of Instrumentation in Parallel Performance Evaluation Tools," Proc. of PARA 2006 conference, June 2006.

[5] R. Bell, A. D. Malony, and S. Shende, "A Portable, Extensible, and Scalable Tool for Parallel Performance Profile Analysis", Proc. EUROPAR 2003 conference, LNCS 2790, Springer, Berlin, pp. 17-26, 2003.

[6] A. D. Malony, and S. S. Shende, "Overhead Compensation in Performance Profiling," Proc. Europar 2004 Conference, LNCS 3149, Springer, pp. 119-132, 2004.

[7] S. Shende, A. D. Malony, A. Morris, and F. Wolf, "Performance Profiling Overhead Compensation for MPI Programs," in Proc. EuroPVM/MPI 2005 Conference, (eds. B. Di. Martino et. al.), LNCS 3666, Springer, pp. 359-367, 2005.

[8] S. Shende, A. D. Malony, A. Morris, and F. Wolf, "Performance Profiling Overhead Compensation for MPI Programs," in Proc. EuroPVM/MPI 2005 Conference, (eds. B. Di. Martino et. al.), LNCS 3666, Springer, pp. 359-367, 2005.

[9] F. Song, F. Wolf, "CUBE User Manual," ICL Technical Report, ICL-UT-04-01, February 2, 2004.

[10] S. Graham, P. Kessler, and M. McKusick, "gprof: A Call Graph Execution Profiler," SIGPLAN Symposium on Compiler Construction, pp. 120–126, June 1982.

[11] S. Graham, P. Kessler, and M. McKusick, "An Execution Profiler for Modular Programs," Software–Practice and Experience, Volume 13, pp. 671–685, August 1983.

[12] B. Mohr, and F. Wolf, "KOJAK - A Tool Set for Automatic Performance Analysis of Parallel Applications," Proc. of the European Confer-

ence on Parallel Computing, Springer-Verlag, LNCS 2790, pp. 1301-1304, August 26-29, 2003.

[13] A. Malony, S. Shende, N. Trebon, J. Ray, R. Armstrong, C. Rasmussen, and M. Sottile, “Performance Technology for Parallel and Distributed Component Software,” Concurrency and Computation: Practice and Experience, Vol. 17, Issue 2-4, pp. 117-141, John Wiley & Sons, Ltd., Feb - Apr, 2005.

[14] A. D. Malony, S. Shende, R. Bell, K. Li, L. Li, N. Trebon, “Advances in the TAU Performance System,” Chapter, “Performance Analysis and Grid Computing,” (Eds. V. Getov, M. Gerndt, A. Hoisie, A. Malony, B. Miller), Kluwer, Norwell, MA, pp. 129-144, 2003.

[15] N. Trebon, A. Morris, J. Ray, S. Shende, and A. Malony, “Performance Modeling of Component Assemblies with TAU,” Proc. Workshop on Component Models and Frameworks in High Performance Computing (CompFrame 2005).

[16] K. Lindlan, J. Cuny, A. Malony, S. Shende, B. Mohr, R. Rivenburgh, C. Rasmussen, “A Tool Framework for Static and Dynamic Analysis of Object-Oriented Software with Templates,” SC 2000 conference, 2000.

[17] K. A. Huck, A. D. Malony, R. Bell, and A. Morris, “Design and Implementation of a Parallel Performance Data Management Framework,” In Proceedings of International Conference on Parallel Processing (ICPP 2005), IEEE Computer Society, 2005.

[18] K. A. Huck, and A. D. Malony, “PerfExplorer: A Performance Data Mining Framework for Large-Scale Parallel Computing,” In Proceedings of SC 2005 conference, ACM, 2005.

[19] 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, Fall 2000.

[20] B. Buck and J. Hollingsworth, “An API for Runtime Code Patching”, *Journal of High Performance Computing Applications*, pp. 317–329, 14(4), 2000.

[21] M. Syamlal, W. Rogers, T. O’Brien, “MFIX documentation: Theory Guide, Technical Note,” DOE/METC-95/1013, 1993.

[22] M. Syamlal, “MFIX documentation: Numerical Technique,” EG&G Technical Report DE-AC21-95MC31346, 1998.

[23] MFIX, URL: http://www.mfix.org, 2006.

[24] C. Fryer and O.E. Potter, “Experimental Investigation of models for fluidized bed catalytic reactors,” AIChE J., 22, 38-47, 1976.

[25] GNU, “GNU Fortran 95,” URL: http://gcc.gnu.org/fortran/, 2006.

[26] G. Watson, “Debug Malloc Library,” URL: http://www.dmalloc.com, 2006.

[27] Silicon Graphics Inc., “Open SpeedShop For Linux”, URL: http://oss.sgi.com/projects/openspeedshop/, 2006.

[28] Luiz DeRose, “Performance Visualization on the Cray XT3,” URL:http://www.psc.edu/training/XT3_Oct05/lectures/CrayXT3Apprentice.pdf, 2006.

[29] V. Herrarte and E. Lusk, “Study parallel program behavior with Upshot,” Technical Report ANL-91/15, Mathematics and Computer Science Division, Argonne National Laboratory, Aug. 1991.