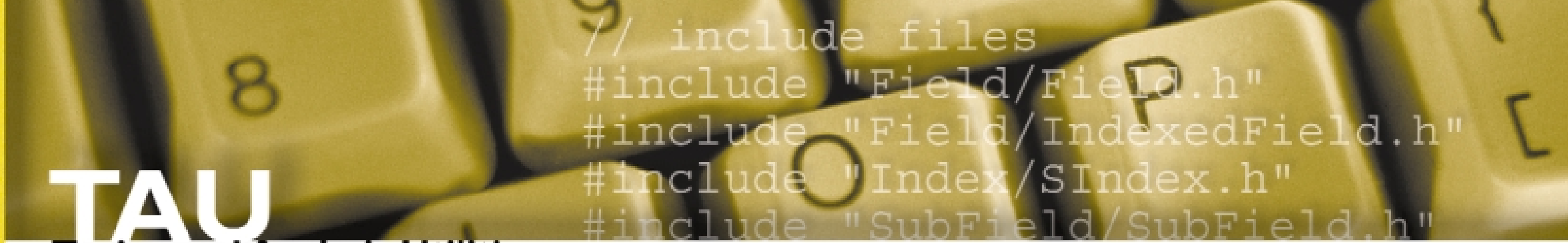


TAU
Tuning and Analysis Utilities

Overview

TAU (Tuning and Analysis Utilities) is a program and performance analysis tool framework for high-performance parallel and distributed computing. TAU provides a suite of tools for static and dynamic analysis of programs written in C, C++, FORTRAN 77/90, High Performance FORTRAN, and Java. In particular, TAU offers a state-of-the-art performance profiling and tracing facility that supports a general scalable parallel execution model based on nodes, contexts, and threads.

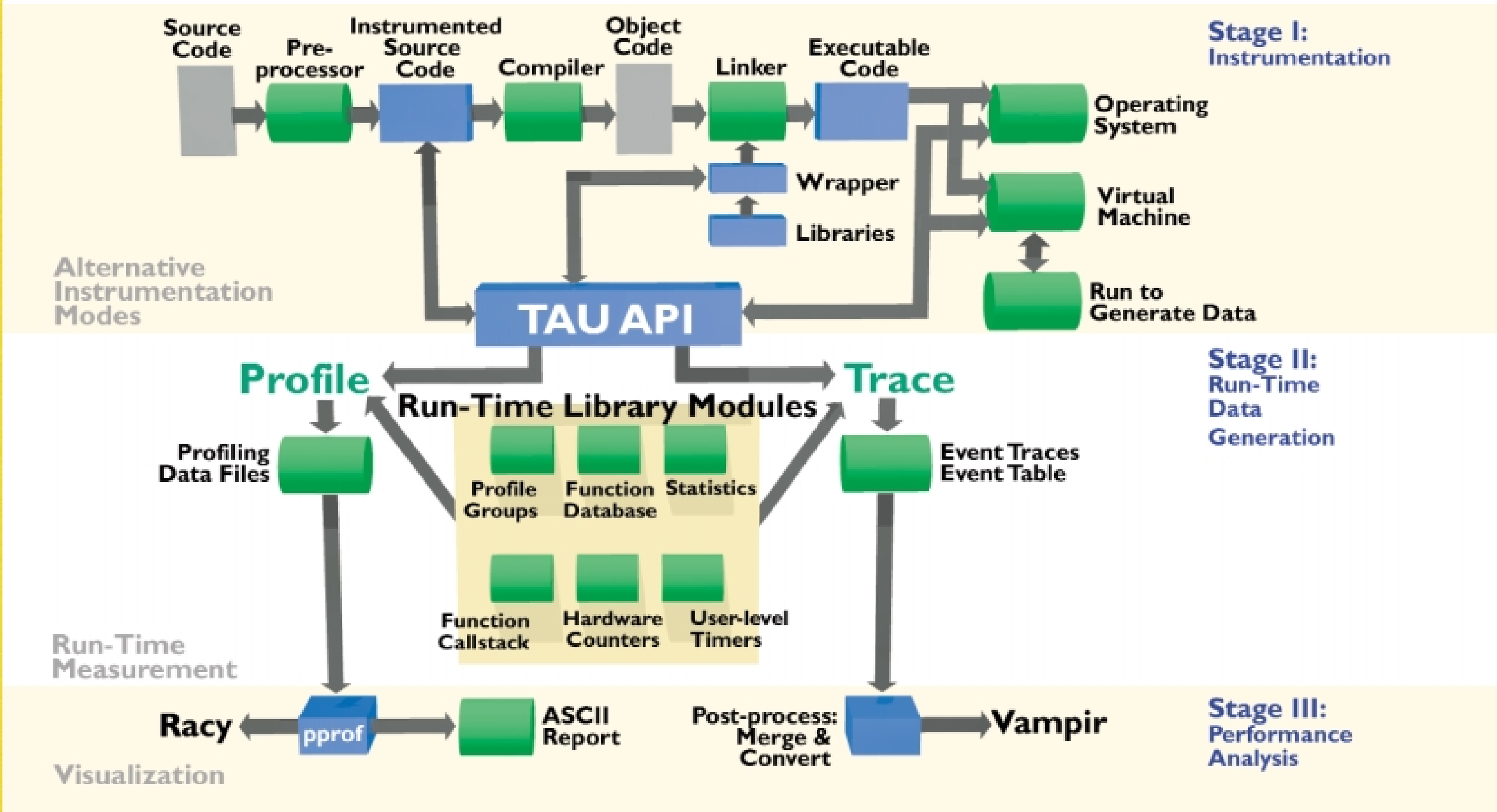
The TAU performance instrumentation and measurement package has been ported to all Accelerated Strategic Computing Initiative (ASCI) platforms and has been used extensively in the Advanced Computational Testing and Simulation (ACTS) toolkit. TAU works with a powerful code analysis system, Program Database Toolkit (PDT), to implement automatic source instrumentation and static program browsers linked to dynamic performance information.

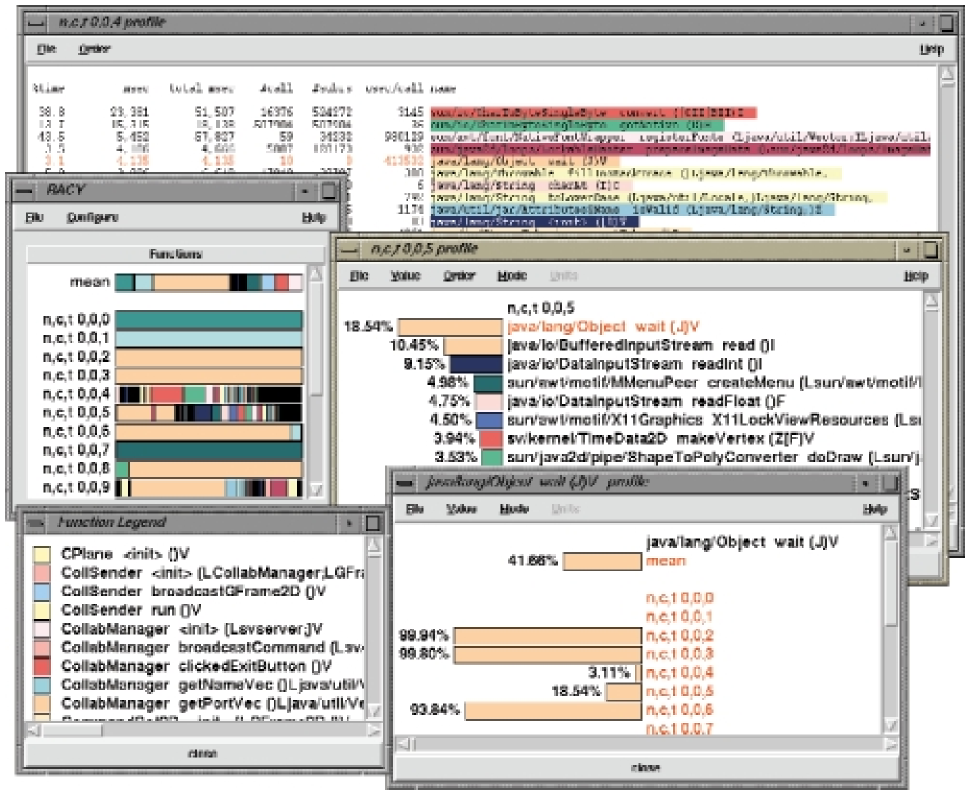
Figure 1. Architecture of TAU profiling and tracing toolkit.

TAU Design

The goal of the TAU project is to develop program and performance analysis technology that meets both the challenges of evolving scalable parallel computing systems and the needs of programming methodologies used for next-generation scientific applications. TAU should be able to target the diversity of computing paradigms and machines while offering a framework of portable and reconfigurable measurement and analysis components that can be optimized and extended. While the tools and techniques implemented may address specific needs of a language or execution environment, they should be coherent, based on a unified analysis model and able to interoperate with other framework components.

The TAU environment achieves these aims. TAU targets a computing model for programming scalable, parallel, multi-threaded programs based on abstractions of nodes, contexts, and threads. (This computing model was defined by the High Performance C++ consortium.) A node is a shared-memory multiprocessor (SMP), having a coherent shared-address space that can be read and modified by any of its processors. Nodes can range from laptop computers to large-scale cache-coherent nonuniform memory architecture





```
%time      msec   total msec   #call   #subrs  usec/call name
 38.8    23,381      51,507    16376   534272      3145 [illegible]
 13.7    15,315      15,138   507986   507984        36 [illegible]
 48.5     5,452      67,827       59    34232    980129 [illegible]
  3.5     4,106       4,666     5087   101173       930 [illegible]
  3.1     4,135       4,135       10        0    413532 java/lang/Object wait (J)V
                                                    380 java/lang/Throwable fillInStackTrace ()Ljava/lang/Throwable;
                                                      6 java/lang/String charAt (I)C
                                                    792 java/lang/String toLowerCase (Ljava/util/Locale;)Ljava/lang/String;
                                                   1174 java/util/jar/Attributes$Name isValid (Ljava/lang/String;)Z
                                                     81 java/lang/String <init> ([C)V
```

Functions

mean

n,c,t 0,0,0
n,c,t 0,0,1
n,c,t 0,0,2
n,c,t 0,0,3
n,c,t 0,0,4
n,c,t 0,0,5
n,c,t 0,0,6
n,c,t 0,0,7
n,c,t 0,0,8
n,c,t 0,0,9

n,c,t 0,0,5

18.54%	java/lang/Object wait (J)V
10.45%	java/io/BufferedInputStream read ()I
9.15%	java/io/DataInputStream readInt ()I
4.98%	sun/awt/motif/MMenuPeer createMenu (Lsun/awt/motif/
4.75%	java/io/DataInputStream readFloat ()F
4.50%	sun/awt/motif/X11Graphics X11LockViewResources (Ls
3.94%	sv/kernel/TimeData2D makeVertex (Z[F)V
3.53%	sun/java2d/pipe/ShapeToPolyConverter doDraw (Lsun/j

Function Legend

🟨	CPlane <init> ()V
🟥	CollSender <init> (LCollabManager;LGFr
🟦	CollSender broadcastGFrame2D ()V
🟨	CollSender run ()V
⬜	CollabManager <init> (Lsvserver;)V
🟥	CollabManager broadcastCommand (Lsv
🟥	CollabManager clickedExitButton ()V
🟦	CollabManager getNameVec ()Ljava/util/V
🟧	CollabManager getPortVec ()Ljava/util/Ve

java/lang/Object wait (J)V

41.68%	mean
	n,c,t 0,0,0
	n,c,t 0,0,1
99.94%	n,c,t 0,0,2
99.80%	n,c,t 0,0,3
3.11%	n,c,t 0,0,4
18.54%	n,c,t 0,0,5
93.84%	n,c,t 0,0,6
	n,c,t 0,0,7

Figure 2. TAU generates profiling data for each user- and system-level thread in Java.

(NUMA) systems. Nodes can be connected locally or distributively by a network. A context refers to a virtual address space on a node that may be accessible to several threads of control. Multiple contexts can exist on a node.

The benefit of targeting this model is that it covers all common modes of execution, including "multithreaded, shared memory" and "single program, multiple data" (SPMD). Because TAU identifies the three abstractions in the model, it can perform analyses particular to each, using knowledge of the mapping of the model to the specific execution context.

Requirements for program and performance analysis arise from differences not only in computing systems but also in how the systems are programmed. TAU applies the concept of mapping at different levels within a programming hierarchy to build analysis abstractions that capture the important behavioral and semantic characteristics of the software. The mapping concept extends to languages where compile-time code manipulation can take place. TAU's support for analysis mapping is found in careful implementation of techniques consistent with the software level where they are applied.

TAU supports an integrated, extensible analysis framework through modular component design, published data formats, and programs to interface to third-party tools. This has made it possible for TAU to be retargeted to new language, run-time, and system contexts and extended with new analysis functionality.

TAU Implementation

There are five different parts to TAU's implementation: program code analysis, performance instrumentation, performance measurement, performance analysis and visualization, and online monitoring.

The program code analysis part is being built on PDT and will offer a rich set of tools to view program information, including class/function/template browsers and graphical displays of calling and class hierarchies. Performance instrumentation, performance measurement, and performance analysis and visualization constitute TAU's portable profiling and tracing package which is being released as part of Los Alamos National Laboratory's 1999 Advanced Computing Laboratory CD. This package is discussed in detail below. Online monitoring is an extension of the TAU measurement system which supports run-time access to application and system performance information.

The TAU implementation has been primarily targeted to requirements of the DOE ACTS toolkit and ASCI computing platforms. This has forced TAU to deal with the difficult challenges of working with evolving language standards, large software frameworks, and multiple machine platforms, operating systems, run-time libraries, and language compilers. As a result, TAU provides one of the most portable and robust analysis systems for parallel scientific applications that exist today.

TAU Profiling and Tracing

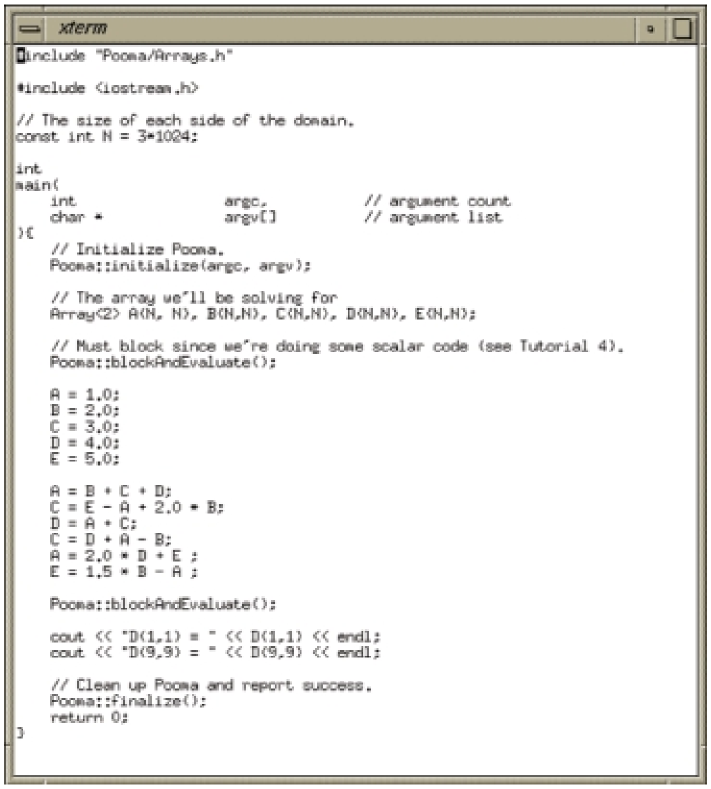
The heart of TAU is its profiling and tracing environment, an integrated toolkit for performance instrumentation, measurement, and analysis of parallel, multithreaded programs. The architecture of the TAU profiling and tracing environment is shown in Figure 1. The environment supports the High Performance C++ model of computation (threads, contexts, and nodes), capturing performance data for these levels of execution and applying analysis tools for different performance views. TAU's implementation targeted C++ since the advanced object-oriented features of this language, such as templates and namespaces, presented difficult challenges in instrumentation and measurement, beyond those present in other language targets. All C++ language features are supported in the TAU instrumentation.

The instrumentation captures data for functions, methods, basic blocks, and statement execution at all execution levels. An instrumentation library and application programming interface (API) are provided that allow the user to select between profiling and tracing, measurement alternatives (e.g., timers, counters, hardware monitors), and application level. The API also provides selection of measurement groups for organizing and controlling instrumentation. The instrumentation supports the mapping of low-level execution measurements to high-level execution entities, such as data parallel statements or expression templates, so that performance data can be properly assigned.

Performance measurements have very low overhead and utilize high-resolution timers for accuracy.

From the data collected (profiles or traces), TAU's analysis procedures can generate a wealth of performance information for the user. Profile analysis can show the exclusive and inclusive time spent in each function with nanosecond resolution. For templated entities, it shows the breakdown of time spent for each instantiation. Other data includes how many times each function was called, how many profiled functions each function invoked, and what the mean inclusive time per call was. Time information can also be displayed relative to nodes, contexts, and threads. Instead of time, hardware performance data can be shown. TAU's profile visualization tool, "Racy," provides graphical displays of all the performance analysis results, in aggregate and per node/context/thread form as shown in Figure 2. The user can quickly identify sources of performance bottlenecks in the application using the graphical interface.

Figure 3. POOMA 2 application.



```
#include "Pooma/Arrays.h"

#include <iostream.h>

// The size of each side of the domain.
const int N = 3*1024;

int
main(
    int                 argc,           // argument count
    char *              argv[]          // argument list
){
    // Initialize Pooma.
    Pooma::initialize(argc, argv);

    // The array we'll be solving for
    Array<2> A(N, N), B(N,N), C(N,N), D(N,N), E(N,N);

    // Must block since we're doing some scalar code (see Tutorial 4).
    Pooma::blockAndEvaluate();

    A = 1.0;
    B = 2.0;
    C = 3.0;
    D = 4.0;
    E = 5.0;

    A = B + C + D;
    C = E - A + 2.0 * B;
    D = A + C;
    C = D + A - B;
    A = 2.0 * D + E ;
    E = 1.5 * B - A ;

    Pooma::blockAndEvaluate();

    cout << "D(1,1) = " << D(1,1) << endl;
    cout << "D(9,9) = " << D(9,9) << endl;

    // Clean up Pooma and report success.
    Pooma::finalize();
    return 0;
}
```

Trace analysis utilizes the Vampir trace visualization tool. (Please see http://www.pallas.de for more information on Vampir.) Vampir can be used to show detailed event history on a timeline display and execution statistics computed across time intervals. Users can view the trace data at different levels of detail, helping them to identify performance bottlenecks in local regions and globally.

The TAU profiling and tracing environment is highly robust and works in the following cases:

- Platforms: SGI Power Challenge and Origin 2K, IBM SP2, Intel Teraflop, Cray T3E, HP 9000, Sun, Windows 95/98/NT, Compaq Alpha Linux cluster, Intel Linux cluster
- Languages: C, C++, FORTRAN 77/90, HPF, HPC++, Java
- Thread packages: pthreads, Tulip threads, SMARTS threads, Java threads, Windows threads
- Communications libraries: MPI, Nexus, Tulip, ACLMPL
- Application libraries: Blitz++, ACLVIS
- Application frameworks and codes: POOMA, POOMA 2, MC++, Conejo, PARP
- Compilers: KAI, PGI, GNU, Fujitsu, Sun, Microsoft, SGI, Cray

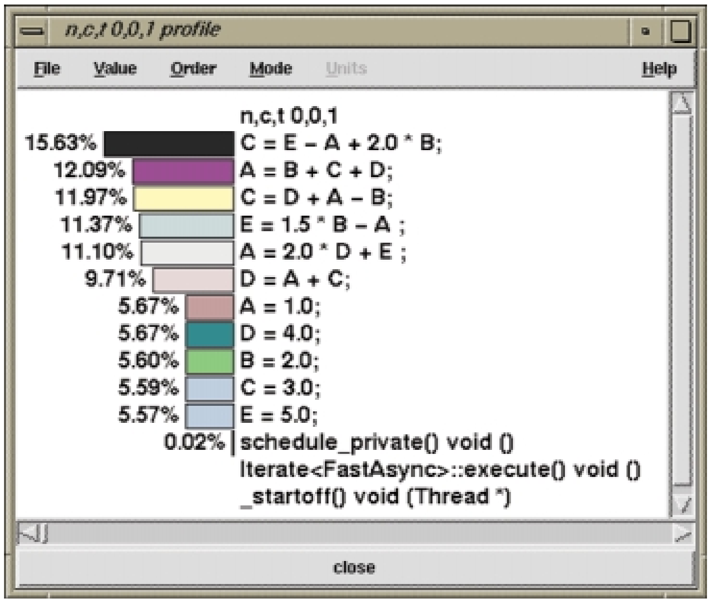
TAU has been applied extensively in the ACTS toolkit.

TAU Applications

The TAU framework has been applied in several contexts, in different languages, across application libraries and frameworks, and on multiple platforms. Below, we highlight applications that demonstrate TAU's versatility.

Shown in Figure 3 is a sample program using the POOMA (Parallel Object-Oriented Methods and Applications) framework. POOMA is a C++ library that includes data-parallel array and particle classes. The original POOMA implemented parallelism in a lock-step fashion using

Figure 4. TAU profiles asynchronous execution of array expressions in POOMA 2.



```
                n,c,t 0,0,1
15.63%  C = E - A + 2.0 * B;
12.09%  A = B + C + D;
11.97%  C = D + A - B;
11.37%  E = 1.5 * B - A ;
11.10%  A = 2.0 * D + E ;
 9.71%  D = A + C;
 5.67%  A = 1.0;
 5.67%  D = 4.0;
 5.60%  B = 2.0;
 5.59%  C = 3.0;
 5.57%  E = 5.0;
 0.02%  schedule_private() void ()
        Iterate<FastAsync>::execute() void ()
        _startoff() void (Thread *)
```

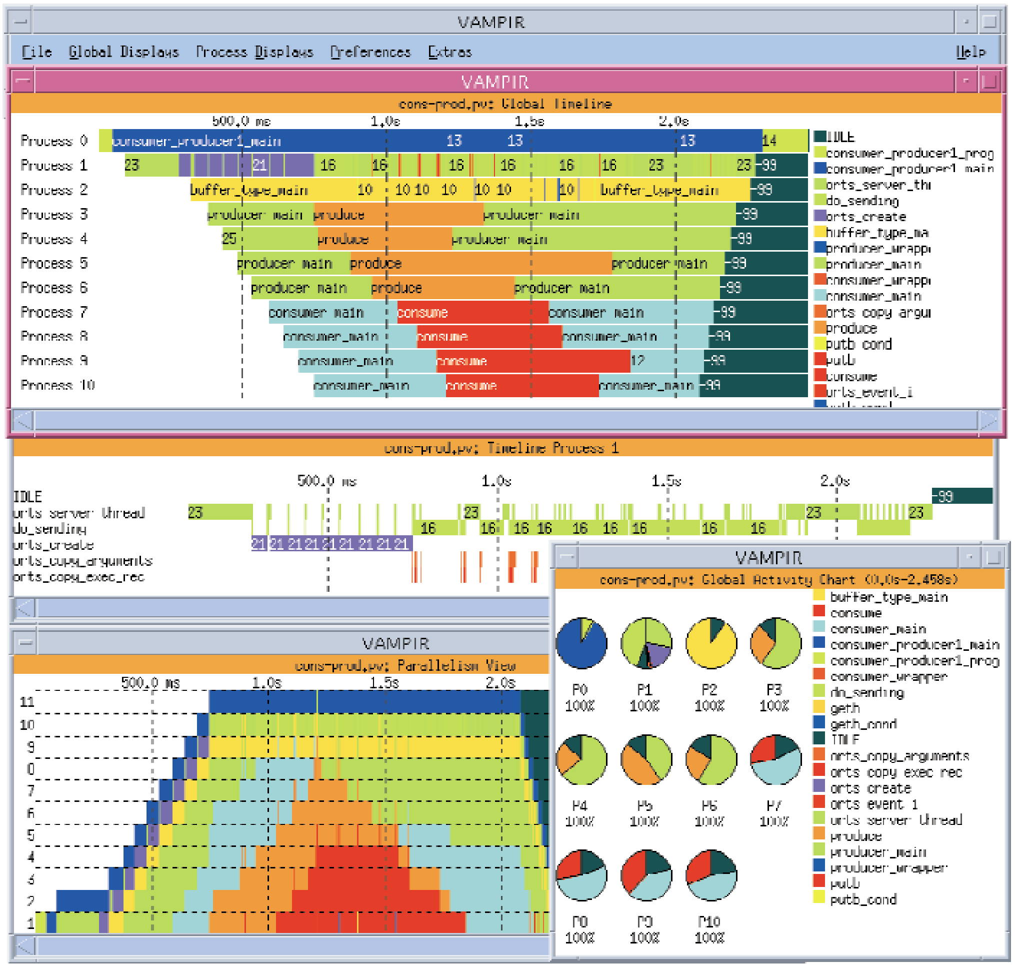

VAMPIR
File Global Displays Process Displays Preferences Extras Help
cons-prod.pv: Global Timeline
cons-prod.pv: Timeline Process 1
cons-prod.pv: Global Activity Chart (0.0s-2.458s)
cons-prod.pv: Parallelism View

Figure 5. Vampir displays TAU traces for an Opus/HPF application.

message passing. POOMA 2 includes thread-based evaluation and the ability to use SMARTS (Scalable Multithreaded Asynchronous Run-Time System).

POOMA 2 and SMARTS present several problems to a performance analysis system. First, being a class library with data-parallel semantics, POOMA-level expressions will be mapped to parallel computations, either an SPMD code with message passing or a multithreaded asynchronous code. The performance system has to be able to track this mapping and associated performance data with the framework-level abstraction. TAU does this through its mapping API and its support for tracking asynchronous execution. As shown in Figure 4, TAU is able to produce performance profiles of applications objects, such as expressions, instead of only routine profiles of object methods. Notice, too, that TAU can report performance data at different software levels, showing the overhead of asynchronous iterate scheduling.

An important feature of the TAU system is its ability to interface with software at compile time and at run time. In particular, TAU supports different modes of instrumentation: source code, library, statically and dynamically linked, and run time. This has proven valuable in TAU's recent support for Java performance measurement. Java visualization application SciVis was developed at the Northeast Parallel Architectures Center at Syracuse University. Using the TAU mapping API, we were able to observe entry and exit events of dynamically loaded Java modules and measure the performance of SciVis execution in detail. In Figure 2, performance data from the SciVis application is displayed with Racy, Tau's profile visualization tool.

A final example illustrates TAU's ability to work with multiple languages, run-time systems, and threads. Figure 5 shows a Vampir display of an application written using High Performance FORTRAN (HPF) for data parallelism and Opus for task parallelism. The HPF compiler produces FORTRAN 90 data parallel modules which execute on multiple processors. The processors interoperate using the Opus run-time system which is built on MPI (message passing interface) and pthreads. In systems of this type, it is important to be able to see the influence of different software levels. TAU is able to capture performance data at different parts of the Opus/HPF system, exposing the bottlenecks within and between levels.

Acknowledgments

TAU is being developed jointly by the University of Oregon, Los Alamos National Laboratory, and Research Centre Juelich in Germany.

More information about TAU…
contact: Allen D. Malony
e-mail: malony@cs.uoregon.edu
web: www.acl.lanl.gov/tau/

Get TAU and other Advanced Computing Laboratory Software…
web: www.acl.lanl.gov/software/
cd: 1999 Advanced Computing Laboratory Software

This work supported by the US Department of Energy.

Los Alamos
NATIONAL LABORATORY

advanced computing laboratory

LALP-99-205 November 1999