PERFORMANCE TECHNOLOGY FOR COMPLEX PARALLEL AND DISTRIBUTED SYSTEMS

ALLEN D. MALONY* AND SAMEER SHENDE

Abstract. The ability of performance technology to keep pace with the growing complexity of parallel and distributed systems will depend on robust performance frameworks that can at once provide system-specific performance capabilities and support high-level performance problem solving. The TAU system is offered as an example framework that meets these requirements. With a flexible, modular instrumentation and measurement system, and an open performance data and analysis environment, TAU can target a range of complex performance scenarios. Examples are given showing the diversity of TAU application.

Key words. performance tools, complex systems, instrumentation, measurement, analysis

1. Introduction. Modern parallel and distributed computing systems present both a complex execution environment and a complex software environment that target a broad set of applications with a range of requirements and goals, including high-performance, scalability, heterogeneous resource access, component interoperability, and responsive interaction. The execution environment complexity is being fueled by advances in processor technology, shared memory integration, clustering architectures, and high-speed inter-machine communication. At the same time, sophisticated software systems are being developed to manage the execution complexity in a way that makes available the potential power of parallel and distributed platforms to the different application needs.

Fundamental to the development and use of parallel and distributed systems is the ability to observe, analyze, and understand their performance at different levels of system implementation, with different performance data and detail, for different application types, and across alternative system and software environments [7]. However, the growing complexity of parallel and distributed systems challenge the ability of performance technologists to produce tools and methods that are at once robust and ubiquitous. On the one hand, the sophistication of the computing environment demands a tight integration of performance observation (instrumentation and measurement) technology optimized to capture the requisite information about the system under performance access, accuracy, and granularity constraints. Different systems will require different observation capabilities and technology implementations specific to system features. Otherwise restricting technology to only a few performance observation modes severely limits performance problem solving in these complex environments.

On the other hand, application development environments present programming abstractions that hide the complexity of the underlying computing system, and are mapped onto layered, hierarchical runtime software optimized for different system platforms. While providing application portability, a programming paradigm also defines an implicit model of performance that is made explicit in a particular system context. System-specific performance data must be mapped to abstract, high-level views appropriate to the performance model. The difficult problem is to provide such a performance abstraction uniformly across the different computing systems where the programming paradigm may be applied. This requires not only a rich set of

*Department of Computer and Information Science, University of Oregon, Eugene, OR, USA, {malony,sameer}@cs.uoregon.edu.

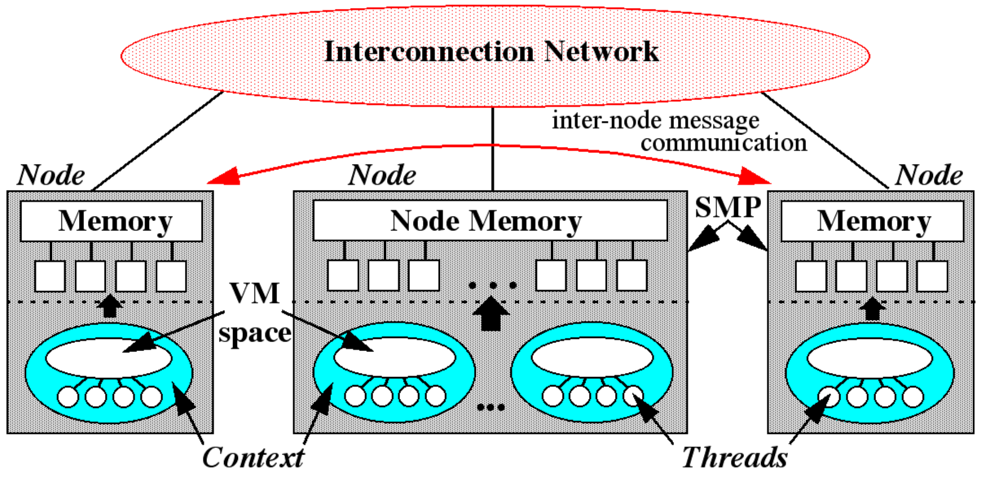


FIG. 2.1. *Execution model supported by the TAU Performance System*

observation capabilities that can provide consistent relevant performance information, but a high degree of flexibility in how tools are configured and integrated to access and analyze this information. Without this ability, common performance problem solving methodologies and tools that support them will not be available.

In this paper, we propose an approach to performance technology development for complex parallel and distributed systems. This approach is based on a general complex systems computation model and a modular performance observation and analysis framework. The computation model, discussed in §2, defines a hierarchical execution architecture reflecting dominant features of modern systems and the layers of software available. The TAU performance framework is presented in §3 as an example of a flexible, configurable, and extensible performance tool system for instrumentation, measurement, and analysis. TAU's ability to address complex system performance requirements is demonstrated in §4 using examples drawn from MPI, multi-threading, mixed-mode parallelism, and combined task/data parallelism performance studies. We conclude the paper with an outlook towards open performance technology as a plan for developing next-generation performance tools.

2. A General Computation Model. To address the dual goals of performance technology for complex systems – robust performance capabilities and widely available performance problem solving methodologies – we need to contend with problems of system diversity while providing flexibility in tool composition, configuration, and integration. One approach to address these issues is to focus attention on a sub-class of computation models and performance problems as a way to restrict the performance technology requirements. The obvious consequence of this approach is limited tool coverage. Instead, our idea is to define an abstract computation model that captures general architecture and software execution features and can be mapped straightforwardly to existing complex system types. For this model, we can target performance capabilities and create a tool framework that can adapt and be optimized for particular complex system cases.

Our choice of general computation model must reflect real computing environments. The computational model we target was initially proposed by the HPC++

consortium [4] and is illustrated in Figure 2.1. Two combined views of the model are shown: a physical (hardware) view and an abstract software view. In the model, a *node* is defined as a physically distinct machine with one or more processors sharing a physical memory system (i.e., a shared memory multiprocessor (SMP)). A node may link to other nodes via a protocol-based interconnect, ranging from proprietary networks, as found in traditional MPPs, to local- or global-area networks. Nodes and their interconnection infrastructure provide a hardware execution environment for parallel software computation. A *context* is a distinct virtual address space within a node providing shared memory support for parallel software execution. Multiple contexts may exist on a single node. Multiple *threads* of execution, both user and system level, may exist within a context; threads within a context share the same virtual address space. Threads in different contexts on the same node can interact via inter-process communication (IPC) facilities, while threads in contexts on different nodes communicate using message passing libraries (e.g., MPI) or network IPC. Shared-memory implementations of message passing can also be used for fast intra-node context communication. The bold arrows in the figure reflect scheduling of contexts and threads on the physical node resources.

3. The TAU Performance System Framework.

The computation model above is general enough to apply to many high-performance architectures as well as to different parallel programming paradigms. Particular instances of the model and how it is programmed defines requirements for performance tool technology. For any performance problem, a performance framework to address the problem should incorporate:

- an *instrumentation model* defining how and when performance information is made available;
- a *performance measurement model* defining what performance information is captured and in what form;
- an *execution model* that relates measured events with each other;
- a *data analysis model* specifying how data is to be processed;
- a *presentation model* for performance viewing; and
- an *integration model* describing how performance tool components are configured and integrated.

The performance framework and the models therein must be realized by tools implemented in the particular computational environment where the performance problem solving will be done. We have developed the TAU performance framework as an integrated toolkit for performance instrumentation, measurement, and analysis for parallel, multithreaded programs that attempts to target the general complex system computation model while allowing flexible customization for system-specific needs.

The TAU performance framework [16] is shown in Figure 3.1. It is composed of instrumentation, measurement, and analysis and visualization phases. TAU implements a flexible instrumentation model that allows the user to insert performance instrumentation calling the TAU measurement API at several levels of program compilation and execution stages. The instrumentation identifies code segments, provides mapping abstractions, and supports multi-threaded and message passing parallel execution models. Instrumentation can be inserted manually, or automatically with a source-to-source translation tool, such as implemented by the Program Database Toolkit (PDT) [22] program analysis facility. When the instrumented application is compiled and executed, profiles or event traces are produced. TAU can use wrapper libraries to perform instrumentation when source code is unavailable for instrumen-

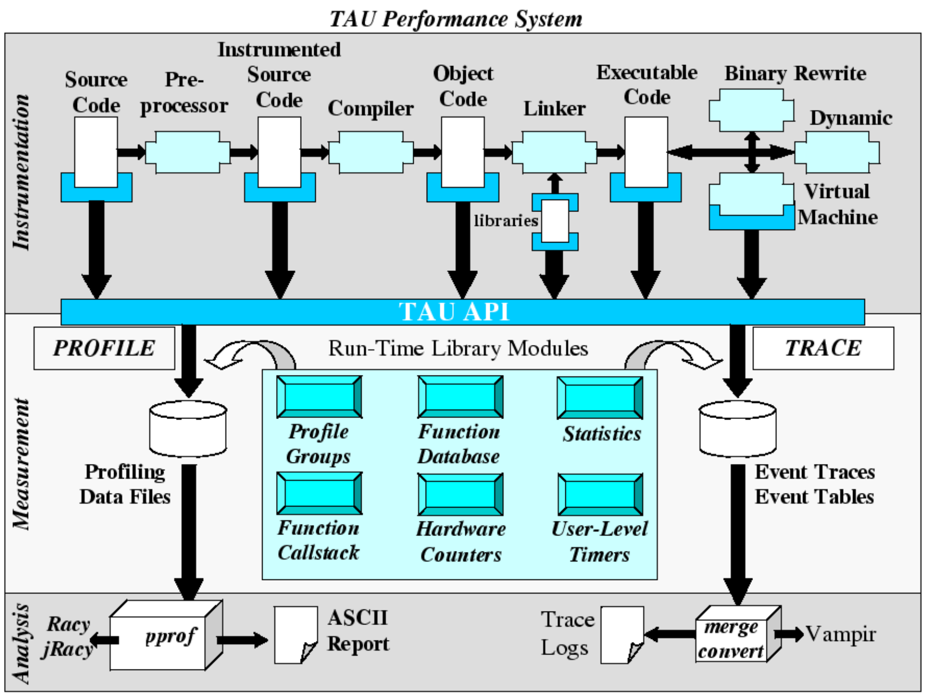


FIG. 3.1. *Architecture of TAU Performance System*

tation. TAU uses existing wrapper capabilities when possible, such as in the case of MPI's profiling interface. Instrumentation can also be inserted at runtime, prior to execution, using the dynamic instrumentation system DyninstAPI [3, 11] or at the virtual machine level, using language supplied interfaces such as the Java Virtual Machine Profiler interface [19, 20].

The instrumentation model interfaces with the measurement model. TAU's measurement model is sub-divided into a high-level performance model, that determines how events are processed, and a low-level measurement model, that determines what system attributes are measured. The measurement captures data for functions, methods, basic blocks, and statement execution. Profiling and tracing are the two measurement choices that TAU allows. The measurement API lets measurement *groups* be defined for organizing and controlling instrumentation. The measurement library also supports the mapping of low-level execution measurements to high-level execution entities (e.g., data parallel statements) so that performance data can be properly assigned. Performance experiments can be composed from different measurement modules, including ones that can measure the wall-clock time, the CPU time, or processor specific activity using non-intrusive hardware performance monitors available on most modern processors; TAU can access both Performance Counter Library [14] and Performance API [23] portable hardware counter interfaces. Based on the composition of modules, an experiment could easily be configured to measure the profile that shows the inclusive and exclusive counts of secondary data cache misses associated with basic blocks such as routines, or a group of statements. By providing a flexible measurement infrastructure, a user can experiment with different attributes of the system and iteratively refine the performance characterization of a parallel application.

The TAU data analysis and presentation models are open. Although TAU comes with both text-based and graphical tools to visualize the performance data collected [21], it provides bridges to other third-party tools (e.g., Vampir [12]) for more sophisticated analysis and visualization. The performance data format is documented and TAU provides tools that illustrate how this data can be converted to other formats [21].

An important component of the performance model presented in a tool is how its integration model provides composition and integration of its different components. The modules must provide well-defined interfaces that are easy to extend. The nature and extent of cooperation between modules that may be vertically and horizontally integrated in the distinct layers defines the degree of flexibility of the measurement system. The integration support in TAU has enabled the performance system to be ported to a diverse set of machine platforms, languages, runtime systems, thread and communication libraries, and application frameworks. It has also allowed TAU to incorporate performance technology of other groups, leveraging functionality to give TAU added capabilities (e.g., using the DyninstAPI [11] for dynamic instrumentation) or access to performance events that can be merged with TAU's mechanisms (e.g., using PAPI [23] to get to hardware performance data and high-resolution timing data). The configuration of available TAU capabilities is the final integration aspect to emphasize. Applied performance investigation depends on creating experiments that capture the type and amount of performance data needed for analysis during performance problem solving. The TAU performance system offers configuration and selection throughout, and this will continue to be important in its evolution and future application.

3.1. TAU Status. The success of the TAU performance system thus far has been primarily due to the benefits of targeting the TAU performance technology to the general computation model and integrating the technology within the framework architecture. TAU functionality is governed, therefore, primarily by how performance observation requirements are addressed at an abstract computation level, allowing system-dependent implementation to be concerned with how efficiently the observation of abstract performance events is supported for a specific platform. The outcome has been broad availability of TAU technology across:

- **system platforms** - including IBM, Sun, SGI, Compaq, HP, Intel/Compaq Linux clusters, IA-32, IA-64, Cray T3E, Windows;
- **languages** - C, C++ [16], Fortran 77/90, Java [17];
- **thread packages** - Pthreads, OpenMP, Windows, Java, SMARTS [24];
- **communications libraries** - MPI [8], PVM, Tulip [2];
- **compilers** - KAI, PGI, IBM, Sun, SGI, Compaq, HP, Cray, Fujitsu, Microsoft, GNU; and
- **application libraries and frameworks** - SMARTS [24], POOMA [15], PETE [10], A++/P++ [13].

In addition, TAU has been able to integrate effectively technology from other projects: DyninstAPI [3], PCL [14], PAPI [23], Vampir [12], and PDT [22]. The successes of TAU also represent its opportunity for enhancement. TAU's architecture will likely need extending, especially to better support dynamic performance measurement and adaptive performance control. Plus, the general computational model may need further sophistication. Nevertheless, we believe the TAU system as a whole captures general observational and analysis techniques and components that are foundational in general performance technology evolution.

4. Performance Scenarios. When one considers the robustness and availability goals for any performance technology it should necessarily be done with respect to the performance observation and analysis requirements of the performance problems being addressed. In general, one can view these requirements with respect to three performance axes:

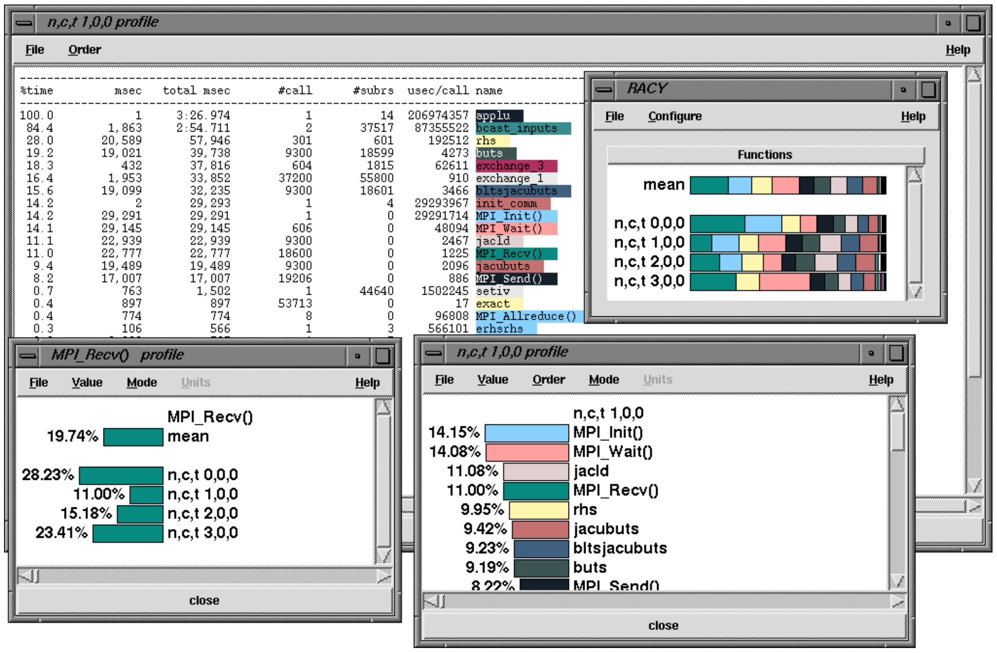
- **Where** do we want to observe performance and where is it possible to do so (in the program, in the system software, and in the hardware)?[1]
- **When** do we want to generate or access performance data, and when is it possible or necessary to do so?
- **How** do we choose performance observation alternative based on what performance data needed?

Fundamental to this *where/when/how* perspective is the concept of performance *events* and their associated performance semantics. Performance events can be common (low-level) events with simple performance semantics, such as routine entry and exit events, message communication events, or threading events, or most abstract (higher-level) events that have more complex semantics, requiring more sophisticated performance observation techniques. Associated with performance events are application-level metrics, such as routine execution time, message size, and synchronization counts, as well as system-specific performance data that might be obtained from hardware performance monitors or OS services. Given the diversity of performance problems, evaluation methods, and types of events and metrics, the instrumentation and measurement mechanisms needed to support performance observation should be flexible, to give maximum opportunity for configuring performance experiments to meet where/when/how objectives, and portable, to allow consistent cross-platform performance problem solving.

Our claim is that TAU provides both a robust and a widely applicable performance technology framework for complex parallel and distributed systems. This section presents selected performance scenarios that demonstrate that TAU can offer effective technology across complex systems types. Underlying our discussion are the general issues raised above. The main point to highlight is TAU's ability to support different high-level performance problem solving requirements via system specific instrumentation, measurement, and analysis.

4.1. SPMD Parallelism and Message Passing. Parallel programming on distributed memory computer systems is commonly of a SPMD style supported by portable message passing libraries. While the SPMD model provides a "single program" view towards application-level performance events, inter-node performance interactions are captured by message communication events. Application instrumentation in this case is facilitated by TAU's macro-based soure-level instrumentation and compile-time measurement configuration. However, instrumentation at the source level is not possible without access to the source code. A convenient mechanism to get around this problem with libraries (e.g., a message communication library) is in

[1]The **where** axis can be regarded as relating to issues of *spatial visibility*, while the **when** axis involves issues of *temporal visibility*.



```
---------------------------------------------------------------------------------
%time        msec   total msec       #call      #subrs  usec/call name
---------------------------------------------------------------------------------
100.0           1    3:26.974            1          14  206974357 applu
 84.4       1,863    2:54.711            2       37517   87355522 bcast_inputs
 28.0      20,589      57,946          301         601     192512 rhs
 19.2      19,021      39,738         9300       18599       4273 buts
 18.3         432      37,816          604        1815      62611 exchange_3
 16.4       1,953      33,852        37200       55800        910 exchange_1
 15.6      19,099      32,235         9300       18601       3466 bltsjacubuts
 14.2           2      29,293            1           4   29293967 init_comm
 14.2      29,291      29,291            1           0   29291714 MPI_Init()
 14.1      29,145      29,145          606           0      48094 MPI_Wait()
 11.1      22,939      22,939         9300           0       2467 jacld
 11.0      22,777      22,777        18600           0       1225 MPI_Recv()
  9.4      19,489      19,489         9300           0       2096 jacubuts
  8.2      17,007      17,007        19206           0        886 MPI_Send()
  0.7         763       1,502            1       44640    1502245 setiv
  0.4         897         897        53713           0         17 exact
  0.4         774         774            8           0      96808 MPI_Allreduce()
  0.3         106         566            1           3     566101 erhsrhs
```

```
         MPI_Recv()
19.74%   mean

28.23%   n,c,t 0,0,0
11.00%   n,c,t 1,0,0
15.18%   n,c,t 2,0,0
23.41%   n,c,t 3,0,0
```

```
         n,c,t 1,0,0
14.15%   MPI_Init()
14.08%   MPI_Wait()
11.08%   jacld
11.00%   MPI_Recv()
 9.95%   rhs
 9.42%   jacubuts
 9.23%   bltsjacubuts
 9.19%   buts
 8.22%   MPI_Send()
```

FIG. 4.1. *TAU profile browser displays for NAS Parallel Benchmark LU running on 4 processors*

the use of a wrapper interposition library. Here, the library designer provides alternative entry points for some or all routines, allowing a new library to be interposed that reimplements the standard API with routine entry and exit instrumentation, calling the native routine in between.

Requiring that such profiling hooks be provided in a standardized library before an implementation is considered ”compliant” forms the basis of an excellent model for developing portable performance profiling tools for the library. Parallel SPMD programs are commonly implemented using a message passing library for inter-node communication, such as MPI. The MPI Profiling Interface [8] provides a convenient mechanism to profile message communication. This interface allows a tool developer to interface with MPI calls without modifying the application source code, and in a portable manner that does not require a vendor to supply the proprietary source code of the library implementation. A performance tool can provide an interposition library layer that intercepts calls to the native MPI library by defining routines with the same name (e.g., *MPI_Send*). These routines can then call the name-shifted native library routines provided by the MPI profiling interface (e.g., *PMPI_Send*). Wrapped around the call is performance instrumentation. The exposure of routine arguments allows the tool developer to also track the size of messages, identify message tags or invoke other native library routines, for example, to track the sender and the size of a received message, within a wild-card receive call.

TAU uses the MPI profiling interface for performance profiling and tracing of message communication events; several other tools also use the interface for tracing (e.g., Upshot [1] and Vampir [12]). Below is the interposition wrapper for the *MPI_send* routine with TAU entry and exit instrumentation:

```
int  MPI_Send( buf, count, datatype, dest, tag, comm )
void * buf;  int count;  MPI_Datatype datatype;
int dest;    int tag;    MPI_Comm comm;
{
  int  retval, typesize;
  TAU_PROFILE_TIMER(tautimer, ‘‘MPI_Send()’’,  ‘‘ ‘‘,
                    TAU_MESSAGE);
  TAU_PROFILE_START(tautimer);
  if (dest != MPI_PROC_NULL) {
    PMPI_Type_size( datatype, &typesize );
    TAU_TRACE_SENDMSG(tag, dest, typesize*count);
  }
  retval = PMPI_Send(buf, count, datatype, dest, tag, comm);
  TAU_PROFILE_STOP(tautimer);
  return returnVal;
}
```

Notice the TAU instrumentation for the start and stop events surrounding the call to *PMPI_Send*.

Figure 4.1 shows the profile of the NAS Parallel Benchmark LU suite written in Fortran using TAU's MPI profiling wrapper. The TAU graphical profile display tool, *Racy*, shows the execution of four processes and the timing of MPI events on each process. (Here, a "process" maps to a single node with one context and one thread of execution.) Notice the integration of communication events with routine performance information. Routine profiles can be shown for each process (e.g., process 1 $\equiv$ *n,c,t 1,0,0*) and the performance of individual routines (e.g., *MPI_Recv*) can be listed for all processes.

Because the MPI wrapper instrumentation targets TAU's measurement API, it is possible to configure the measurement system to capture various types of performance data, including system and hardware data, as well as switch between profiling and tracing. In addition, TAU's performance grouping capabilities allows MPI event to be presented with respect to high-level categories such as send and receive types. These performance configurations can done without change to the source- and wrapper-level instrumentation.

4.2. Multi-Threaded Systems and Java. Multi-threaded systems and applications present a more complex environment for performance tools due to the different forms and levels of threading and the greater need for efficient instrumentation. How to determine thread identity, how to store per-thread performance data, and how to provide synchronized and consistent update and access to the data are some of the questions that must be addressed. TAU provides modules that interface with system-specific thread libraries and member functions for thread registration, thread identification, and mutual exclusion for locking and unlocking runtime performance data structures. This allows the measurement system to work with different thread packages (e.g., pthreads, Windows threads, and Java threads), as well as special-purpose thread libraries (e.g., SMARTS [24] and Tulip [2]) while maintaining a common measurement model. Because TAU targets a general threading model, it can extend its common thread layer to provide well-defined core functionality for each new thread system.

We chose the Java language to demonstrate TAU's application in multi-threaded systems since it utilizes both user-level and system-level threads and involves the

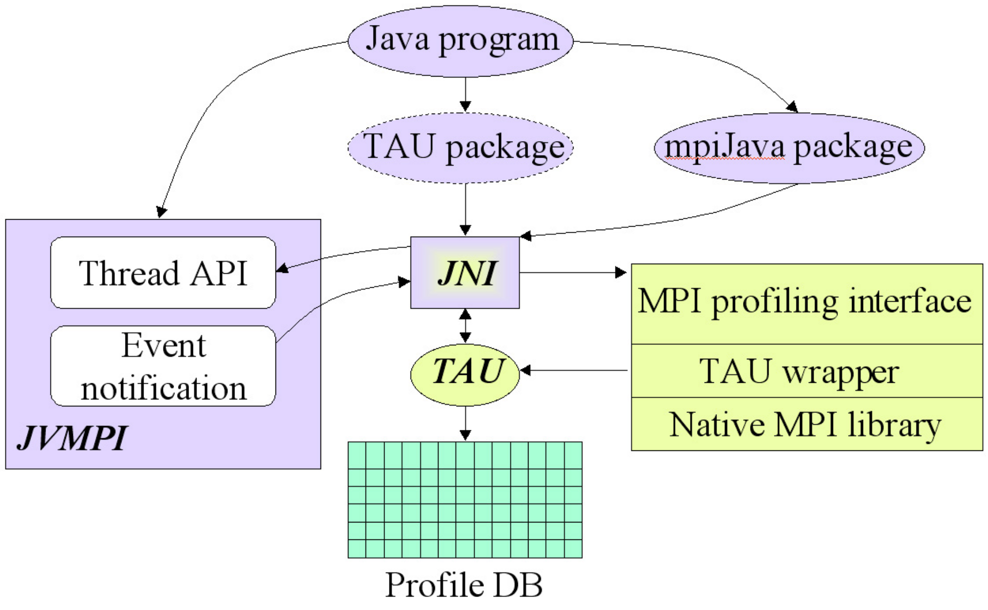


FIG. 4.2. *TAU instrumentation for Java source, virtual machine, and mpiJava packa ges*

additional complexity of virtual machine execution. Performance instrumentation and measurement of multi-threaded interpreted programs such as Java pose several difficulties. Because Java programs are compiled to a platform independent byte-code that is interpreted by a Java Virtual Machine (JVM), a performance system must interface to the JVM to capture performance events, but still make measurements as efficiently as possible. This may be difficult to do portably in the presence of just-in-time (JIT) compilation and runtime adaptive optimizations, as realized by state-of-the-art JVM implementations, such as realized in the Sun Hot-Spot Virtual Machine[25]. Furthermore, it can become difficult to associate virtual machine state with actual system state to record performance measurements accurately.

Conveniently, Java 2 (JDK1.2+) incorporates the Java Virtual Machine Profiler Interface (JVMPI) [19, 20] which we have used for our work in TAU [17]. JVMPI provides profiling hooks into the virtual machine and allows a profiler agent to instrument the Java application without any changes to the source code, bytecode, or the executable code of the JVM. JVMPI provides a wide range of events that it can notify to the agent, including method entry and exit, memory allocation, garbage collection, and thread start and stop; see the Java 2 reference for more information. When the profiler agent is loaded in memory, it registers the events of interest and the address of a callback routine to the virtual machine using JVMPI. When an event takes place, the virtual machine thread generating the event calls the profiler agent callback routine with a data structure that contains event specific information. The profiling agent can then use JVMPI to get more detailed information regarding the state of the system and where the event occurred.

Figure 4.2 describes how JVMPI is used by TAU for performance measurement.[2]

[2]The figure shows both how JVMPI is used by TAU for performance measurement of Java events, as well as how performance measurement of native libraries (e.g., the MPI library) is integrated in

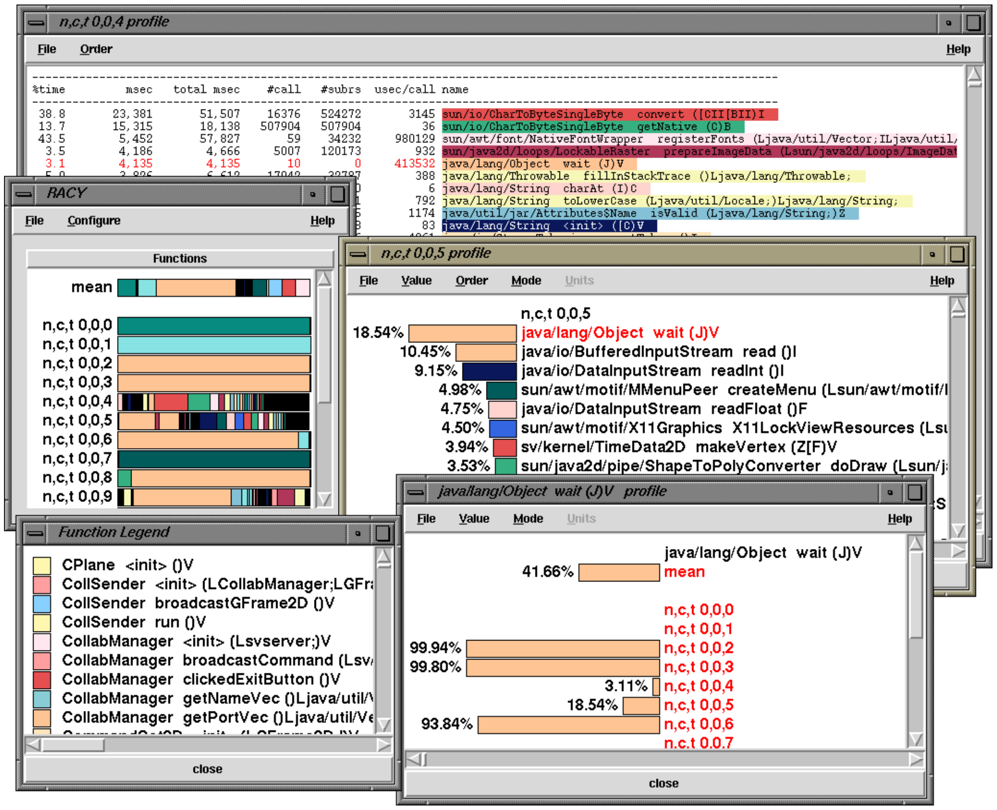
The TAU measurement library is compiled into a dynamic shared object which is loaded in the address space of the virtual machine. An initialization routine specifies a mapping of events that are of interest to the performance system and registers a TAU interface that will be called when the events occur. It stores the identity of the virtual machine and requests the JVM to notify it when a thread starts or terminates, a class is loaded in memory, a method entry or exit takes place, or the JVM shuts down. When a class is loaded, TAU examines the list of methods in the class and creates an association of the name of the method and its signature, as embedded in the TAU object, with the method identifier obtained, using the TAU Mapping API (see the TAU User's Guide [21]). When an event is triggered, event specific information is passed to the TAU interface routine by the virtual machine. When a method entry takes place, TAU performs measurements and correlates these to the TAU object corresponding to the method identifier that it receives from JVMPI. TAU identifies the thread in which the event takes place and uses the Java thread interface to maintain per-thread performance data. TAU classifies all method names and their signatures into higher level profile group names, such as for different Java packages (*/lang*, */io*, */awt*, etc.).

To deal with Java's multi-threaded environment, TAU uses a common thread layer for operations such as getting the thread identifier, locking and unlocking the performance database, getting the number of concurrent threads, etc. This thread layer is then used by the multiple instrumentation layers. When a thread is created, TAU registers it with its thread module and assigns an integer identifier to it. It stores this in a thread-local data structure using the JVMPI thread API described above. It invokes routines from this API to implement mutual exclusion to maintain consistency of performance data. It is important for the profiling agent to use the same thread interface as the virtual machine that executes the multi-threaded Java applications. This allows TAU to lock and unlock performance data in the same way as application level Java threads do with shared global application data. TAU maintains a per-thread performance data structure that is updated when a method entry or exit takes place. Since this is maintained on a per thread basis, it does not require mutual exclusion with other threads and is a low-overhead scalable data structure. When a thread exits, TAU stores the performance data associated with the thread to stable storage. When it receives a JVM shutdown event, it flushes the performance data for all running threads to the disk.

To demonstrate the efficacy of TAU's use of JVMPI for Java, we downloaded a collaborative client-server scientific visualization system, *Scivis* [5], written entirely in Java. With no modification to the Java source code, we ran the Scivis server with TAU performance measurements enabled, generating the per-thread execution profile shown in Figure 4.3 for different methods across different Java packages. A total of twenty-four threads executed in this run of Scivis. Notice that some of the threads (0-3) are performing system functions for the JVM while others (4, 5, and 9) are performing user tasks. As before, it is a simple matter of loading a different measurement library to capture a performance trace instead of statistical profiles.

4.3. Mixed-Mode Parallelism. Increasingly, scalable parallel systems are being designed as clusters of shared memory multi-processors (SMPs). These systems support what is referred to as "mixed-model parallelism" where multi-threaded shared

a Java execution environment by TAU. This is discussed in more detail in [17] and demonstrated in §4.3.2.



```
---------------------------------------------------------------------------------------------------
%time        msec   total msec    #call    #subrs  usec/call name
---------------------------------------------------------------------------------------------------
 38.8      23,381      51,507     16376    524272       3145 sun/io/CharToByteSingleByte  convert ([CII[BII)I
 13.7      15,315      18,138    507904    507904         36 sun/io/CharToByteSingleByte  getNative (C)B
 43.5       5,452      57,827        59     34232     980129 sun/awt/font/NativeFontWrapper  registerFonts (Ljava/util/Vector;ILjava/util,
  3.5       4,186       4,666      5007    120173        932 sun/java2d/loops/LockableRaster  prepareImageData (Lsun/java2d/loops/ImageDat
  3.1       4,135       4,135        10         0     413532 java/lang/Object  wait (J)V
                                                         388 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
                                                          83 java/lang/String  <init> ([C)V
```

```
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/I
 4.75%   java/io/DataInputStream  readFloat ()F
 4.50%   sun/awt/motif/X11Graphics  X11LockViewResources (Lsu
 3.94%   sv/kernel/TimeData2D  makeVertex (Z[F)V
 3.53%   sun/java2d/pipe/ShapeToPolyConverter  doDraw (Lsun/j
```

```
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
41.66%   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
```

FIG. 4.3. *TAU profiles the multi-threaded Java visualization application using JVMPI*

memory programming is used within the SMP and message passing (MPI or some other inter-process communication package) is used for communication between SMP nodes. A key problem for performance tools in mixed-mode environments is the need to maintain multiple performance views (shared memory and message communication) and relationships between views. Performance instrumentation specific to both models can be combined from their individual implementations, but there must be a transposition (i.e., mapping) of each model's performance information into a joint composite observation. Not only is access needed to events in the thread runtime systems and communication layers, but it is necessary at times to associate the occurrence of these events to one another.

4.3.1. OpenMP and MPI. Because TAU supports a general parallel computation model, it can configure the measurement system to capture both thread and communication performance information. We have demonstrated the ability to form an integrated performance measurement for applications that use OpenMP for shared memory parallel programming and MPI for cross-node message-based parallelism. Figure 4.4 shows a performance trace of a ocean circulation application based on a 2D Stommel model using Jacobi iteration on a 5-point stencil. Notice the integrated identification of OpenMP and MPI events. Also, we can see parallel thread execution ("Process *i*" in the figure) interposed between regions of message communication conducted by the main threads ("Process 0" in the figure). The Vampir timeline display

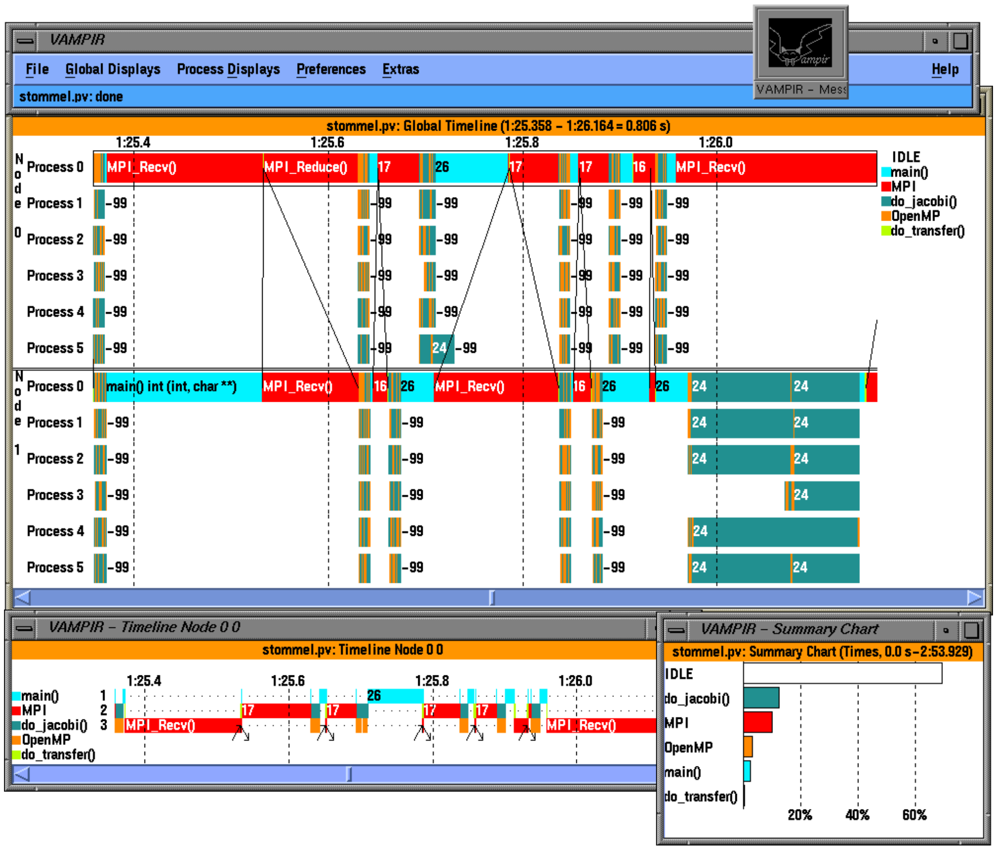
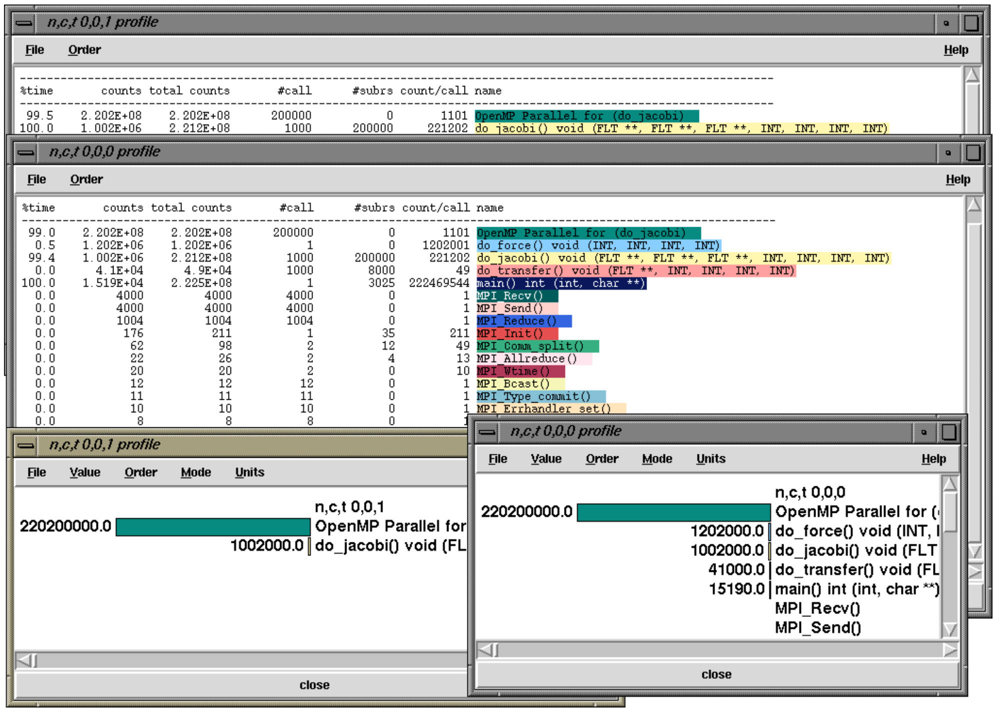


FIG. 4.4. *Mixed-mode OpenMP / MPI execution trace of ocean circulation application*

at the bottom shows that the main thread on Node 0 spends the majority of its time in communication.

To observe hardware performance for the parallel OpenMP sections of the computation, we can switch the TAU measurement system without change to the instrumentation. Figure 4.5 shows the performance profile of floating-point instructions. In comparing the main thread (labeled *n,c,t 0,0,0*) with thread 1 (labeled *n,c,t 0,0,1*), we can see that the floating-point operations are the same for the *OpenMP Parallel for* region and the *do_jacobi* routines, but the main thread calculates *do_force* alone as well as performs all communication.

4.3.2. Java and MPI. We have also demonstrated TAU's use for mixed-mode parallelism with multi-threaded Java programs using the mpiJava [9] package. While mpiJava relies on the existence of native MPI libraries, its API is implemented as a Java wrapper package that uses C bindings for MPI routines. The integrated instrumentation for this scenario is portrayed in Figure 4.2. However, instrumentation of multi-threaded MPI programs poses some challenges for tracking inter-thread message communication events, especially in the case where threads are managed by a virtual machine. MPI is unaware of threads (Java threads or otherwise) and communicates solely on the basis of rank information. Each process (i.e., context) that participates in synchronization operations has a rank. However, all threads within the process



```
---------------------------------------------------------------------------------------------
%time       counts total counts       #call      #subrs count/call name
---------------------------------------------------------------------------------------------
 99.5    2.202E+08    2.202E+08      200000           0       1101 OpenMP Parallel for (do_jacobi)
100.0    1.002E+06    2.212E+08        1000      200000     221202 do_jacobi() void (FLT **, FLT **, FLT **, INT, INT, INT, INT)
```

```
%time       counts total counts       #call      #subrs count/call name
---------------------------------------------------------------------------------------------
 99.0    2.202E+08    2.202E+08      200000           0       1101 OpenMP Parallel for (do_jacobi)
  0.5    1.202E+06    1.202E+06           1           0    1202001 do_force() void (INT, INT, INT, INT)
 99.4    1.002E+06    2.212E+08        1000      200000     221202 do_jacobi() void (FLT **, FLT **, FLT **, INT, INT, INT, INT)
  0.0     4.1E+04      4.9E+04        1000        8000         49 do_transfer() void (FLT **, INT, INT, INT, INT)
100.0    1.519E+04    2.225E+08           1        3025  222469544 main() int (int, char **)
  0.0        4000         4000        4000           0          1 MPI_Recv()
  0.0        4000         4000        4000           0          1 MPI_Send()
  0.0        1004         1004        1004           0          1 MPI_Reduce()
  0.0         176          211           1          35        211 MPI_Init()
  0.0          62           98           2          12         49 MPI_Comm_split()
  0.0          22           26           2           4         13 MPI_Allreduce()
  0.0          20           20           2           0         10 MPI_Wtime()
  0.0          12           12          12           0          1 MPI_Bcast()
  0.0          11           11          11           0          1 MPI_Type_commit()
  0.0          10           10          10           0          1 MPI_Errhandler_set()
  0.0           8            8           8           0
```

```
                        n,c,t 0,0,1
220200000.0             OpenMP Parallel for
              1002000.0 do_jacobi() void (FL
```

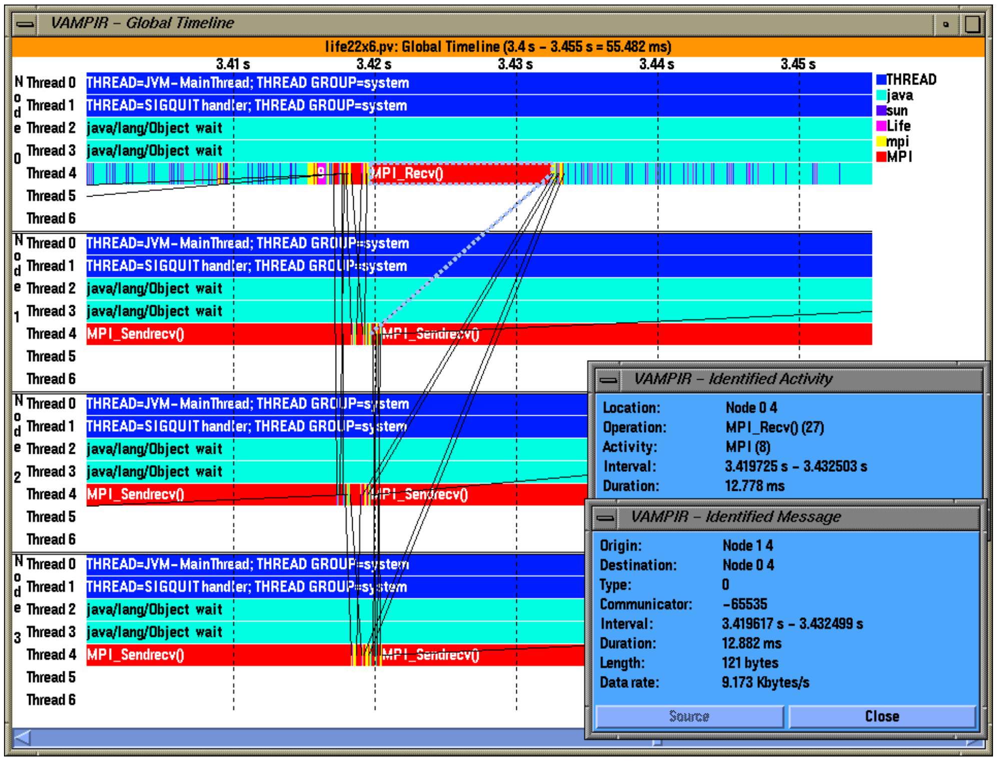
```
                        n,c,t 0,0,0
220200000.0             OpenMP Parallel for (
              1202000.0 do_force() void (INT, I
              1002000.0 do_jacobi() void (FLT
                41000.0 do_transfer() void (FL
                15190.0 main() int (int, char **)
                        MPI_Recv()
                        MPI_Send()
```

FIG. 4.5. *Mixed-mode OpenMP / MPI execution profile of ocean circulation application with floating-point counts*

share the same rank. For a message send operation, we can track the sender's thread by querying the underlying thread system and we can track the receiver's thread likewise. For the JVM, this requires TAU to call into JVMPI across the Java Native Interface (JNI) boundary.

Unfortunately, there still exists a problem with MPI communication between threads in that the sender doesn't know the receiver's thread id and vice versa. To accurately represent a message on a global timeline, we need to determine the precise node and thread on both sides of the communication, either from information in the trace file or from semantic analysis of the trace file. To avoid additional messages to exchange this information at runtime or to supplement messages with thread ids, matching sends and receives is best reserved to the post-mortem trace conversion phase. Trace conversion takes place after individual traces from each thread are merged. The merged trace is a time ordered sequence of events (such as sends, receives, routine transitions, etc.). Each event record has a timestamp, location information (node, thread) as well as event specific data (such as message size, and tags). When a send is encountered, we search for a corresponding receive operation by traversing towards the end of the trace file and matching the receiver's rank, message tag and message length. When a match is found, the receiver's thread id is obtained and a trace record containing the sender and receiver's node, thread ids, message length, and a message tag can be generated. The matching works in a similar fashion when we encounter a receive record, except that we traverse the trace file in the opposite direction, looking for the corresponding send event.

In Figure 4.6 we see a performance trace of a mixed-mode Java/mpiJava appli-



VAMPIR – Identified Activity

Location:	Node 0 4
Operation:	MPI_Recv() (27)
Activity:	MPI (8)
Interval:	3.419725 s – 3.432503 s
Duration:	12.778 ms

VAMPIR – Identified Message

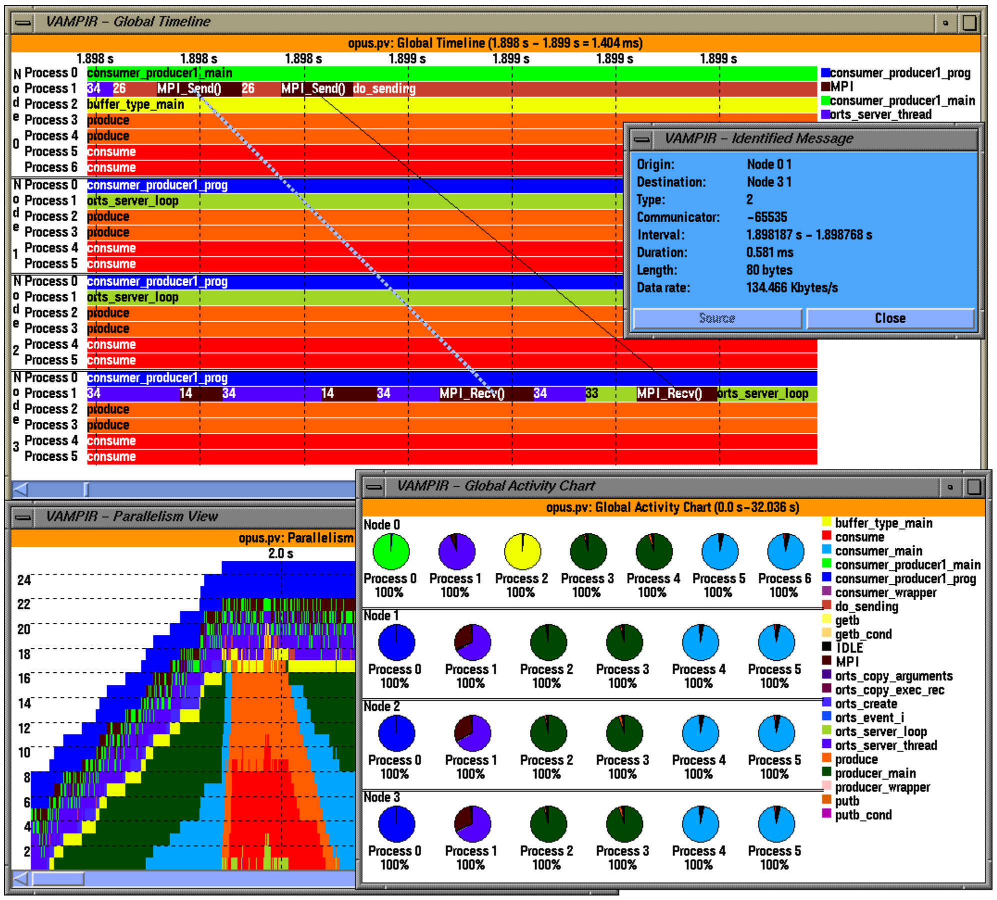
Origin:	Node 1 4
Destination:	Node 0 4
Type:	0
Communicator:	-65535
Interval:	3.419617 s – 3.432499 s
Duration:	12.882 ms
Length:	121 bytes
Data rate:	9.173 Kbytes/s

FIG. 4.6. *Mixed-mode Java / MPI execution profile of a game of life application*

cation simulating the game of Life. A total of twenty-eight threads are executing across four nodes. The integrated events are seen as before, as well as the grouping of events. Our thread message matching algorithm was applied to correctly visualize the message pairing.

4.4. Hybrid Parallelism. More sophisticated forms of mixed-mode parallelism are possible when software layers are built to hide the intricacies of efficient communication or data distribution, presenting a compiler backend or an application programmer with a set of well-defined, portable interfaces for general parallel programming. In some cases, these layers involve a mixing of languages, libraries, runtime software, and application components to create a "hybrid" parallel development environment that enables higher-level or hierarchical parallel programming abstractions to be used. This poses a challenge to performance tools to not only work with the different parts involved, but also to map performance data to the parallel execution abstractions and user-level performance views.

We've used TAU to investigate task and data parallel execution in the Opus/HPF programming system [6]. Figure 4.7 shows a Vampir display of TAU traces generated from an application written using HPF for data parallelism and Opus for task parallelism. The HPF compiler produces Fortran 90 data parallel modules which can execute on multiple processes. The processes interoperate using the Opus runtime system built on MPI 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



VAMPIR – Identified Message

Origin:	Node 0 1
Destination:	Node 3 1
Type:	2
Communicator:	–65535
Interval:	1.898187 s – 1.898768 s
Duration:	0.581 ms
Length:	80 bytes
Data rate:	134.466 Kbytes/s

FIG. 4.7. *Vampir displays for TAU traces of an Opus/HPF application using MPI and pthread*

between levels.

5. Conclusions. To be at once robust and ubiquitous, TAU attempts to solve performance technology problems at levels where performance analysis system solutions can be configured and integrated to target specific performance problem solving needs. TAU has been developed based on the principle that performance technology should be open, easy to extend, and able to leverage external functionality. The complex system case studies presented here is but a small sample of the range of TAU's potential application [21].

In rapidly evolving parallel and distributed systems, performance technology can ill-afford to stand still. A performance technologist always operates under a set of constraints as well as under a set of expectations. While performance evaluation of a system is directly affected by what constraints the system imposes on performance instrumentation and measurement capabilities, the desire for performance problem solving tools that are common and portable, now and into the future, suggests that performance tools hardened and customized for a particular system platform will be short-lived, with limited utility. Similarly, performance tools designed for constrained parallel execution models will likely have little use to more general parallel and distributed computing paradigms. Unless performance technology evolves with system technology, a chasm will remain between the users expectations and the capabilities that performance tools provide. The challenge for the TAU system in the future is to maintain a highly configurable tool architecture while not arbitrarily enforcing constraining technology boundaries.

Acknowledgments. This work was supported in part by a U.S. Department of Energy DOE 2000 / ACTS grant (No. DEFC 0398 ER 259986), a DOE ASCI Level 3 grant (No. 03588-001-99 4R), and an NSF National Young Investigator award.

REFERENCES

[1] Argonne National Laboratory, "The Upshot program visualization system," URL: http://www-fp.mcs.anl.gov/˜lusk/upshot/.
[2] P. Beckman and D. Gannon, "Tulip: A Portable Run-Time System for Object Parallel Systems," Tenth International Parallel Processing Symposium, August 1996.
[3] B. Buck and J. Hollingsworth, "An API for Runtime Code Patching," Journal of High Performance Computing Applications, Vol. 14, No. 4, pp. 317–329, Winter 2000.
[4] HPC++ Working Group, "HPC++ White Papers," Technical Report TR 95633, Center for Research on Parallel Computation, 1995.
[5] B. Ki and S. Klasky, "Scivis," ACM Workshop on Java for High-Performance Network Computing, February 1998.
[6] E. Laure, P. Mehrotra, H. Zima, "Opus: Heterogeneous Computing With Data Parallel Tasks," Parallel Processing Letters, Vol. 9, No. 2, pp. 275–289, June 1999.
[7] A. Malony, "Tools for Parallel Computing: A Performance Evaluation Perspective," in J. Blazewicz, K. Ecker, B. Plateau, D. Trystram (Eds.), Handbook on Parallel and Distributed Processing, Springer, pp. 342–363, 2000.
[8] Message Passing Interface Forum, "MPI: A Message Passing Interface Standard," International Journal of Supercomputer Applications, Special issue on MPI, 1994.
[9] M. Baker, B. Carpenter, G. Fox, S. Ko, and S. Lim, "mpiJava: An Object-Oriented Java Interface to MPI," International Workshop on Java for Parallel and Distributed Computing, IPPS/SPDP 1999, April 1999.
[10] S. Haney, J. Crotinger, S. Karmesin, and S. Smith, "PETE, the Portable Expression Template Engine," Los Alamos National Laboratory, Technical Report, LA-UR-99-777, published in Dr. Dobbs Journal, October 1999.
[11] B. Miller, M. Callaghan, J. Cargille, J. Hollingsworth, R. Irvin, K. Karavanic, K. Kunchithapadam, and T. Newhall. "The Paradyn Parallel Performance Measurement Tools," IEEE Computer, Vol. 28, No. 11, pp. 37–46, November 1995.
[12] Pallas, "VAMPIR - Visualization and Analysis of MPI Resources," 2000. URL: http://www.pallas.de/pages/vampir.htm.
[13] R. Parsons and D. Quinlan, "A++/P++ Array Classes for Architecture Independent Finite Difference Computations," OONSKI, pp. 408-418, 1994.
[14] Research Center Juelich GmbH, "PCL - The Performance Counter Library," URL: http://www.fz-juelich.de/zam/PCL/.
[15] J. Reynders et al., "POOMA: A Framework for Scientific Simulation on Parallel Architectures," in G.V. Wilson and P. Lu (Eds.), Parallel Programming using C++, pp. 553-594, MIT Press, 1996.
[16] S. Shende, A. Malony, J. Cuny, K. Lindlan, P. Beckman, and S. Karmesin, "Portable Profiling and Tracing for Parallel Scientific Applications using C++," Proc. of the SIGMETRICS Symposium on Parallel and Distributed Tools, pp. 134–145, ACM, August 1998.
[17] S. Shende and A. Malony, "Integration and Application of the TAU Performance System in Parallel Java Environments," Joint ACM Java Grande - ISCOPE 2001 Conference, to appear, June 2001.
[18] SUN Microsystems Inc., "Java Native Interface (JNI)," URL: http://java.sun.com/products/jdk/1.3/docs/guide/ jni/index.html.
[19] Sun Microsystems Inc., "Java Virtual Machine Profiler Interface (JVMPI)," URL: http://java.sun.com/products/jdk/1.3/docs/guide/jvmpi/jvmpi.html.
[20] D. Viswanathan and S. Liang, "Java Virtual Machine Profiler Interface," IBM Systems Journal, Vol. 39, No. 1, pp. 82–95, 2000.

[21] University of Oregon, "TAU User's Guide," URL: http://www.cs.uoregon.edu/research/paracomp/tau/.

[22] 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," Supercomputing 92000. See URL: http://www.cs.uoregon.edu/research/paracomp/pdtoolkit.

[23] University of Tennessee, "PerfAPI - Performance Data Standard and API," URL: http://icl.cs.utk.edu/projects/papi/.

[24] S. Vajracharya, S. Karmesin, P. Beckman, J. Crotinger, A. Malony, S. Shende, R. Oldehoeft, and S. Smith, "SMARTS: Exploiting Temporal Locality and Parallelism through Vertical Execution," Los Alamos National Laboratory, Technical Report LA-UR-99-16, 1999.

[25] Sun Microsystems Inc. "The JAVA HotSpot Performance Engine Architecture," Sun Microsystems White Paper, April 1999. http://java.sun.com/products/hotspot/whitepaper.html