CONCURRENCY—PRACTICE AND EXPERIENCE

Concurrency: Pract. Exper. 2001; **00**:1–7 *Prepared using* cpeauth.cls *[Version: 2001/03/05 v2.01]*

Integration and Application of TAU in Parallel Java Environments

Sameer Shende and Allen D. Malony[†]

Department of Computer and Information Science
University of Oregon, Eugene, Oregon

SUMMARY

Parallel Java environments present challenging problems for performance tools because of Java's rich language system and its multi-level execution platform combined with the integration of native-code application libraries and parallel runtime software. In addition to the desire to provide robust performance measurement and analysis capabilities for the Java language itself, the coupling of different software execution contexts under a uniform performance model needs careful consideration of how events of interest are observed and how cross-context parallel execution information is linked. This paper relates our experience in extending the TAU performance system to a parallel Java environment based on mpiJava. We describe the complexities of the instrumentation model used, how performance measurements are made, and the overhead incurred. A parallel Java application simulating the game of Life is used to show the performance system's capabilities.

KEY WORDS: Parallel, Java, performance, tools

[†]E-mail: {`sameer,malony`}`@cs.uoregon.edu`

Received 31 August 2001

Revised ?? ??? 2001

1. INTRODUCTION

With the nascent use of Java for high-performance parallel and distributed computing comes the requirements that application developers and system managers have for performance measurement and analysis tools. These requirements are not new – performance is a dominant concern and the need for tools is fundamental. The Java language environment and how it is used for high-performance computing, however, pushes the state of performance technology in new respects. First, the Java Virtual Machine (JVM) presents a sophisticated shared memory execution platform that is multi-threaded, supports the mapping of user-level threads to system threads, allows just-in-time (JIT) compilation and dynamic loading of code modules, and interfaces with distributed systems middleware. The combination of these features is new. Second, the Java Native Interface (JNI) opens up the Java environment, making inter-language execution possible. While this allows access to high-performance application and communication libraries, it complicates the ability to track multi-level inter-language performance events across different execution contexts and to integrate those events in local and global performance views. Lastly, because the Java language system is portable, the facilities, tools, and interfaces that support performance measurement and analysis for Java need to be portable as well.

In this paper, we share our experiences developing a prototype performance measurement and analysis system for Java. The system is built upon our robust TAU (Tuning and Analysis Utilities) performance framework for scalable parallel and distributed computing. TAU has been designed to support performance analysis for a general model of parallel computation. It provides portable measurement interfaces and services, flexible instrumentation, the ability to observe multiple software layers and levels of execution, and certain provisions for mixed-language programming. However, in all of these areas, TAU had to be extended in new ways to accommodate Java software features and the hybrid execution model it allows. This experience has been valuable in that we believe such characteristics will be more the norm in the future, and the techniques we developed will contribute to the repertoire of methods applied to these new performance technology challenges.

In Section 2, we briefly describe the TAU framework and the general computation model it supports. We decided to focus our attention on a (cluster-oriented) style of high-performance computing that uses Java multi-threading for shared memory parallel computing on a symmetric multiprocessing (SMP) node and MPI message passing for communications between distributed nodes. Although not a comprehensive coverage of HPC Java environments [3], we feel this style of multi-level parallel Java programming is representative of current trends. In Section 3, we describe how the TAU framework has been adapted for this model. Following these sections, we show examples of performance analysis for a parallel Java application, highlighting the ability to capture performance information across execution levels and at different levels of parallelism. Sections 5 and 6 discuss recent features that enable more refined performance measurements. Section 7 addresses the issue of instrumentation overhead and quantifies the costs of TAU measurements. Conclusions and thoughts for future directions are given in Section 8.

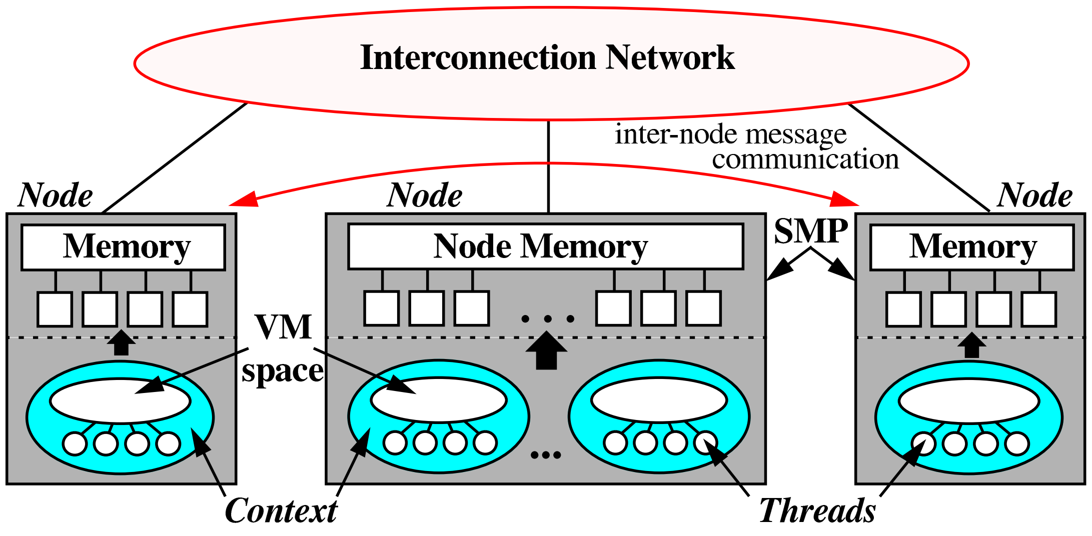


Figure 1. General parallel computation model

2. THE TAU PERFORMANCE SYSTEM

The TAU performance system [13] provides robust technology for performance instrumentation, measurement, and analysis for complex parallel systems [8]. It targets a general computation model initially proposed by the HPC++ consortium [5]. This model consists of shared-memory *nodes* where *contexts* reside, each providing a virtual address space shared by multiple *threads* of execution; see Figure 1. The model is general enough to apply to many high-performance scalable parallel systems and programming paradigms. Because TAU enables performance information to be captured at the node/ context/thread levels, this information can be flexibly mapped to the particular parallel software and system execution platform under consideration.

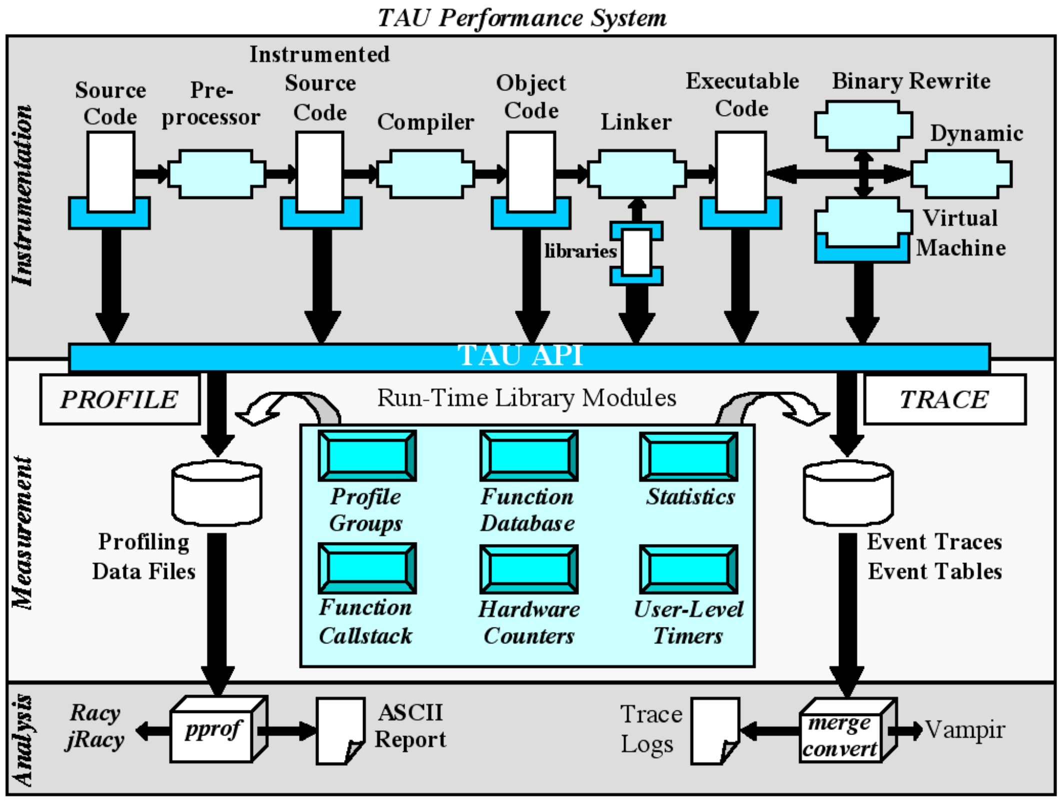


Figure 2. TAU performance system

The TAU performance system is shown in Figure 2 TAU supports a flexible instrumentation model that allows access to a measurement API at several stages of program compilation and execution. The instrumentation identifies code segments, provides for mapping of low-level execution events to high-level computation entities, and works with multi-threaded and message passing parallel execution models. It interfaces with the TAU measurement model that can capture data for function, method, basic block, and statement execution. Profiling and tracing form the two measurement choices that TAU provides. Performance experiments can be composed from different measurement modules, including ones that access hardware performance monitors. The TAU data analysis and presentation utilities are open; they offer text-based and graphical tools to visualize the performance data as well as bridges to third-party software, such as Vampir [9, 12] for sophisticated trace analysis and visualization.

3. PERFORMANCE INSTRUMENTATION FOR PARALLEL JAVA

Scientific applications written in Java are often implemented using a combination of languages such as Java, C++, C and Fortran. While this defies the pure-Java paradigm, it is often necessary since needed numerical, system, and communication libraries may not be available in Java, or since the use of compiled native versions can offer significant performance improvements [3]. Analyzing such hybrid multi-language programs requires a performance measurement strategy that leverages instrumentation alternatives and APIs at several levels of compilation, linking, and execution. To illustrate this point, we consider instrumentation mechanisms employed for profiling and tracing Java programs that communicate with each other using the Message Passing Interface (MPI) [4].

3.1. mpiJava

While there are several design issues that determine how a message communication interface for Java is implemented[6], we considered mpiJava [2] for our work. mpiJava is an object-oriented interface to MPI that allows a Java program to access MPI entities such as objects, routines, and constants. mpiJava relies on the existence of native MPI libraries, but its API is implemented as a Java wrapper package using C bindings for MPI routines. In contrast, the reference implementation for MPJ [1], the Java Grande Forum's MPI-like message-passing API, will rely heavily on RMI and Jini for finding computational resources, creating slave processes, and handling failures; user-level communication will be implemented efficiently, directly on top of Java sockets, not a native MPI library. For mpiJava, when a Java application creates an object of the MPI class, mpiJava loads a native dynamic shared object (`libmpijava.so`) in the address space of the Java Virtual Machine (JVM). This Java package is layered atop the native MPI library using the Java Native Interface (JNI) [14]. There is a one-to-one mapping between Java methods and C routines. Applications are invoked using a script file `prunjava` that calls the `mpirun` application for distributing the program to one or more nodes.

3.2. Instrumentation Problems

The Java execution environment with mpiJava poses several challenges to a performance tool developer. The performance model implemented by the tool must embed the hybrid-execution model of the system where multiple Java threads within a virtual machine and multiple MPI (native) processes execute concurrently. One faces two major problems instrumenting a hybrid system consisting of MPI contexts and Java threads within each of those contexts. The first involves how to expose the thread information to the MPI interface. The second involves how to provide MPI context information to the Java interface. It is necessary to address these problems so events can be tracked in the correct context and thread.

However, different events occur in the different software components (e.g., routine transitions, inter-task message communication, thread scheduling, and user defined events), and performance data should be collected to highlight the different execution modes and the inter-relationship of the software layers. For instance, the event representing a Java thread invoking a message send operation occurs in the JVM, while the actual communication send and receive events take place in compiled native C modules. Ideally, we want the instrumentation inserted in the application, virtual machine, and native language libraries to gather performance data for these events at their origin in a uniform and consistent manner. This involves maintaining a common API for performance measurement as well as a common database for multiple sources of performance data within a context of execution.

Below, we present our multi-level instrumentation approach for this parallel Java system using the TAU performance framework. TAU applies instrumentation at both the Java virtual machine level and the MPI library level to capture performance data and associate performance events.

3.3. JVMPI

Instrumenting Java and the JVM poses several difficulties. Conveniently, Java 2 (JDK1.2+) incorporates the Java Virtual Machine Profiler Interface (JVMPI) [16, 15] which we have used for our work. 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

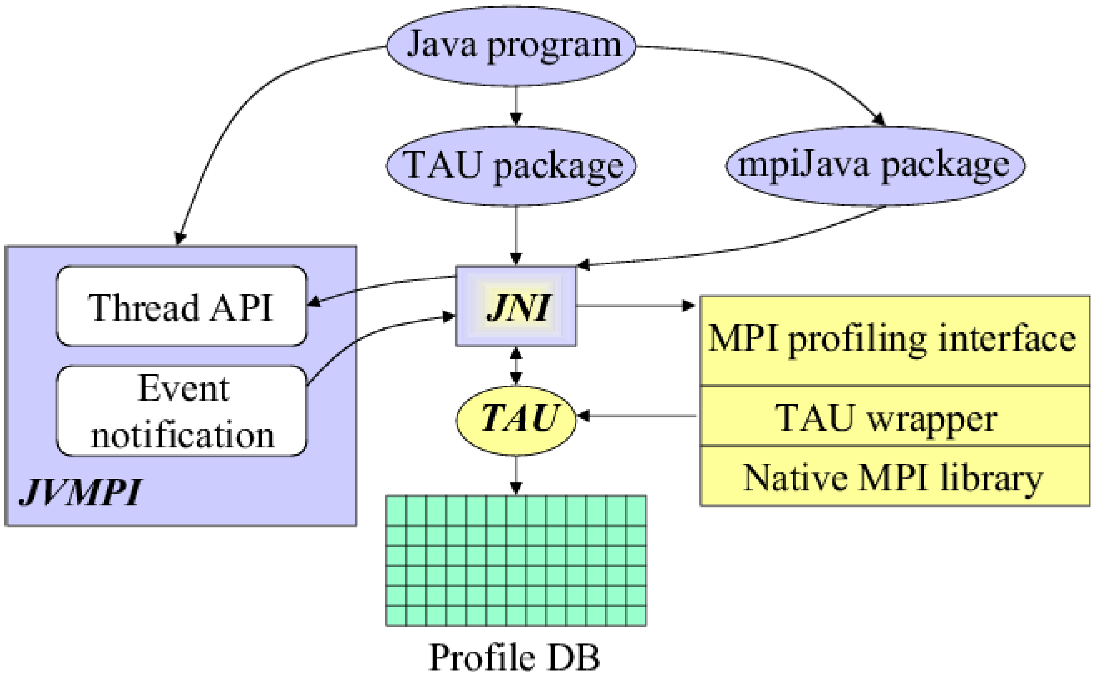


Figure 3. TAU instrumentation for Java source, virtual machine and mpiJava packages

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 3 describes how JVMPI is use by TAU for performance measurement. Consider a single context of a distributed parallel MPI Java program. At start-up, the Java program loads the mpiJava package as a shared object and the JVM loads the TAU performance measurement library as a shared object, which acts as a JVMPI profiling agent. A two-way function call interface between the JVM and the TAU profiler agent is established. The JVM notifies TAU of events and TAU can, in turn, obtain information about and control the behavior of the virtual machine threads using the JVMPI thread primitives (e.g., for mutual exclusion).

When the TAU agent is loaded in the JVM as a shared object, a TAU initialization routine is invoked. 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 [11]). 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. When a thread is created, it creates a top-level routine that corresponds to the name of the thread, so the lifetime of each user and system level thread can be tracked.

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.

3.4. MPI PROFILING INTERFACE

Given a means to capture Java-level execution events, we now consider MPI events. MPI provides an interface [4] that allows a tool developer to intercept MPI calls in a portable manner without requiring a vendor to supply proprietary source code of the library and without requiring the application source code to be modified by the user. This is achieved by providing hooks into the native library with a name-shifted interface and employing weak bindings. Hence, every MPI call can be accessed with its name shifted interface as well. Library-level instrumentation can be implemented by defining a wrapper interposition library layer that inserts instrumentation calls before and after calls to the native routines.

We developed a TAU MPI wrapper library that intercepts calls to the native library by defining routines with the same name, such as *MPI_Send*. These routines then call the native library routines with the name shifted routines, such as *PMPI_Send*. Wrapped around the call, before and after, is TAU performance instrumentation. An added advantage of providing such a wrapper interface is that the profiling wrapper library has access to not only the routine transitions, but also to the arguments passed to the native library. This allows TAU to track the size of messages, identify message tags, or invoke other native library routines. This scheme helps a performance tool track inter-process communication events. For example, it is possible to track the sender and the size of a received message in completion of a wild-card receive call. Whereas JVMPI-based instrumentation can notify the profiling agent of an event such as an mpiJava method entry, it does not provide the agent with arguments that are passed to the methods. However, this information can be obtained using the TAU MPI wrapper library.

To expose thread information to the MPI interface, we decided to have the TAU instrumentation access its runtime thread API layer within the MPI wrapper. As shown in Figure 3, the MPI and Java modules within the TAU system use JNI 1.2 routines to gain access to the Java virtual machine environment associated with the currently executing thread within the JVM. It does so by using the virtual machine information stored by TAU when the in-process profiling agent is loaded by the virtual machine during initialization, as described in the previous section. Using the thread environment, the thread layer can invoke routines to access thread-local storage to access the current thread identifier, and invoke mutual exclusion routines from the JVMPI interface to maintain consistency of the performance data. This scheme allows events generated at the MPI or the Java layer to uniformly access the thread API.

To allow the Java instrumentation to access the correct node and context information, we instrument the *MPI_Init* routine to store the rank of the MPI process in a globally accessible data structure. The TAU instrumentation triggered by JVMPI event notification (see Figure 3) then accesses this MPI information in the same manner as instrumentation requests from any layer from any language. By giving access to the execution model information to all measurement and instrumentation modules in a well-defined, uniform manner, the performance framework can be extended with a minimal effort to additional libraries and new evolving execution models. A combination of instrumentation at multiple levels in TAU helps us solve the hybrid execution model instrumentation problem.

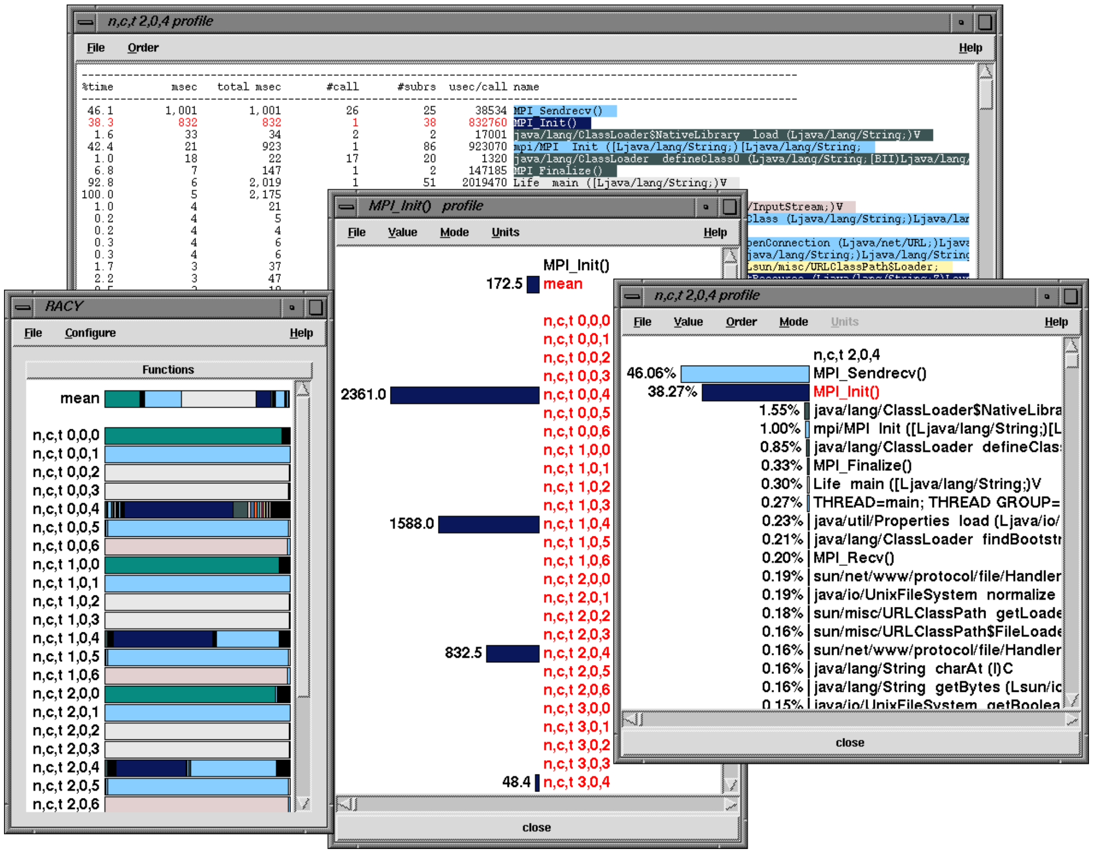
3.5. TRACING HYBRID EXECUTION

Instrumentation of multi-threaded MPI programs poses some challenges for tracking inter-thread message communication events. MPI is unaware of threads (Java threads or otherwise) and communicates solely on the basis of rank information. Each process that participates in synchronization operations has a rank. However, all threads within the process share the same rank. For a message send operation, we can track the sender's thread by querying the underlying thread system (in this case, through JVMPI) and we can track the receiver's thread likewise.

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, we decide to delay matching sends and receives 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). During trace conversion, each record is examined and converted to the target trace format (such as Vampir, ALOG, SDDF or Dump). 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. This technique is used later on in our example to produce Figure 5.

4. PERFORMANCE ANALYSIS FOR A PARALLEL JAVA APPLICATION

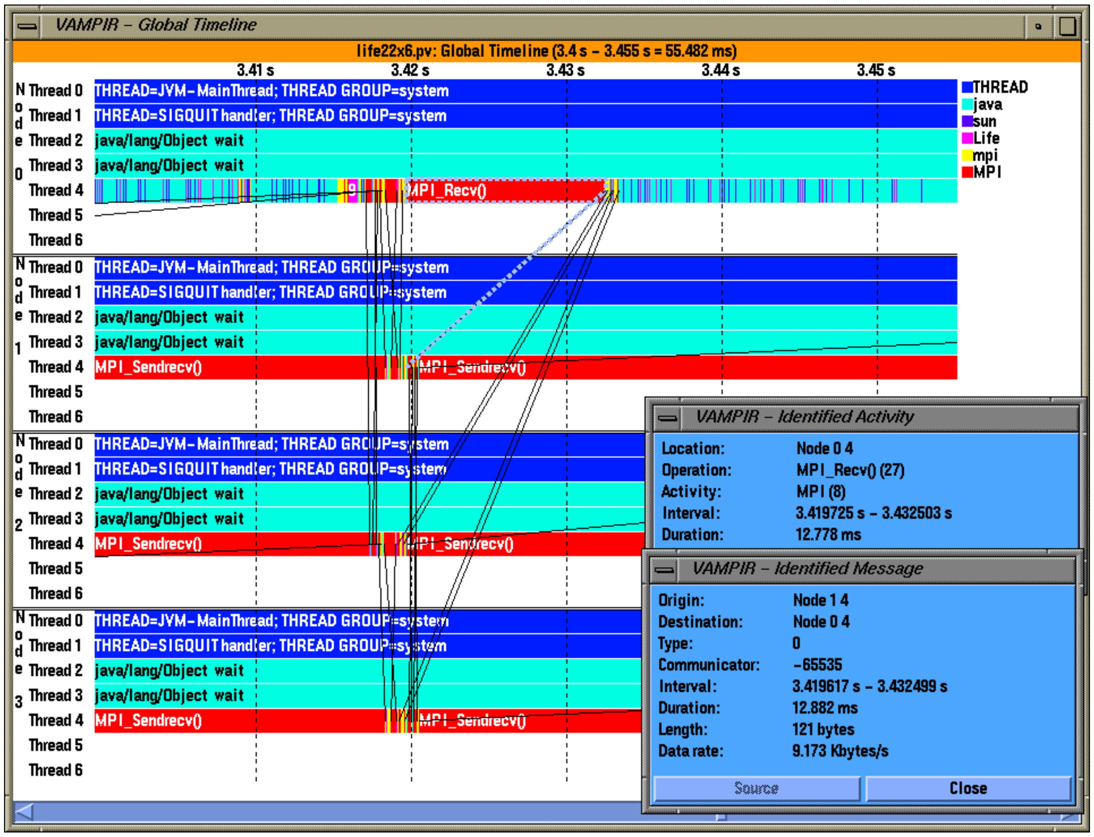
TAU supports both profiling and tracing performance analysis methodologies. Profiling presents the user with summary statistics of performance metrics while tracing highlights the temporal aspect of performance behavior, showing when and where events took place. To provide a sense of how TAU's capabilities can be applied to parallel Java applications, we present performance analysis of an mpiJava benchmark application that simulates the game of Life. We use a simple application and run it on four processors mainly for purposes of brevity and clarity in our discussion. However, it should be understood that TAU's capabilities can extend and scale in respect to the complexity and requirements of applications and system environments, including larger numbers of Java contexts and processors.



```
---------------------------------------------------------------------------------------
%time         msec   total msec       #call      #subrs  usec/call name
---------------------------------------------------------------------------------------
 46.1        1,001        1,001          26          25      38534 MPI_Sendrecv()
 38.3          832          832           1          38     832760 MPI_Init()
  1.6           33           34           2           2      17001 java/lang/ClassLoader$NativeLibrary  load (Ljava/lang/String;)V
 42.4           21          923           1          86     923070 mpi/MPI  Init ([Ljava/lang/String;)[Ljava/lang/String;
  1.0           18           22          17          20       1320 java/lang/ClassLoader  defineClass0 (Ljava/lang/String;[BII)Ljava/lang/
  6.8            7          147           1           2     147185 MPI_Finalize()
 92.8            6        2,019           1          51    2019470 Life  main ([Ljava/lang/String;)V
100.0            5        2,175
  1.0            4           21
  0.2            4            5
  0.2            4            4
  0.3            4            6
  0.3            4            6
  1.7            3           37
  2.2            3           47
```

Figure 4. TAU's profile browser RACY shows per thread performance data

In Figure 4, we see the profile of the mpiJava Life application obtained from TAU measurement, as described in the previous sections. It shows seven Java threads running on each node. Notice that events across different levels and components of execution are being observed. Thread 4 in each context is executing MPI calls for communication between the four processes. Of particular interest is the well-known cascading behavior of the mpich *MPI_Init* routine seen in the *MPI_Init profile* window. This illustrates how tasks are spawned off successively by MPICH. The performance of individual MPI routines is shown across each



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

Figure 5. Vampir global time display shows activities and inter-thread message communication

context and thread, as in the *MPI_Init profile* window. A detailed performance profile for each thread can be displayed graphically and textually, as shown in the two *n,c,t 2,0,4* profile windows for (*t*)hread 4 in (*c*)ontext 0 on (*n*)ode 2. Some of the other threads are performing background JVM and mpiJava module tasks that the application developer would not directly see.

To observe dynamic performance behavior, TAU can also generate event traces that are visualized here using a third-party commercial trace visualization program called Vampir

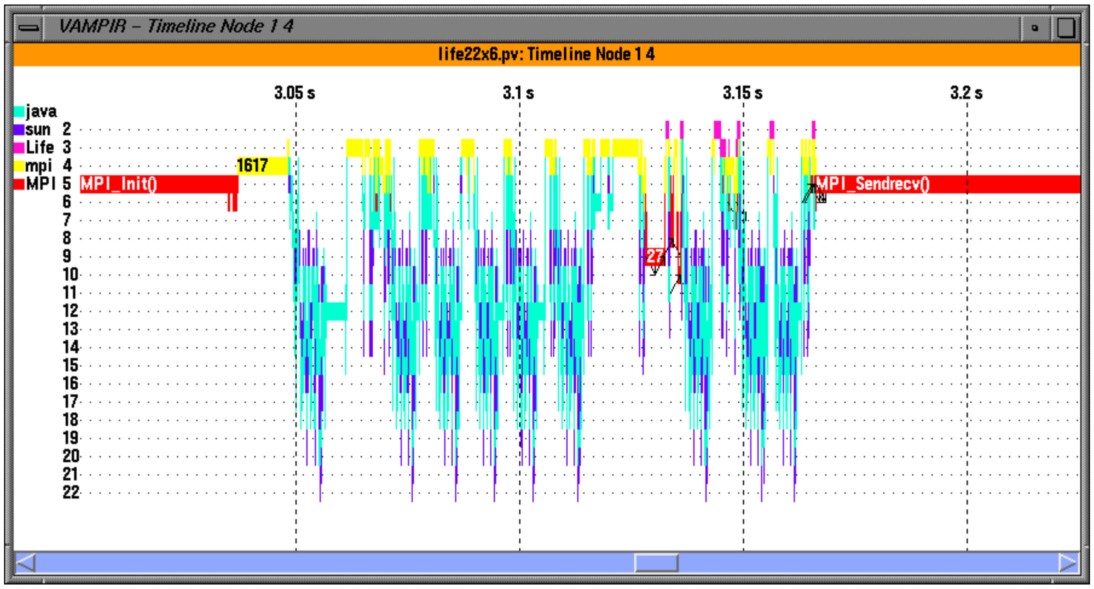


Figure 6. Vampir timeline display can show the depth of routine nesting of the callstack on a particular thread

[9, 12]. Figure 5 illustrates how we can group threads within a node and show inter-thread, inter-node message communication events as line segments that connect the send and receive events within a global timeline. The user can zoom into interesting portions of the timeline and can click on a message or a segment to get more detailed information (e.g., the node where the events took place, the message tag, length, and bandwidth). Vampir provides a rich set of views for exploring different aspects of performance behavior. Figure 6 shows levels of nesting along a timeline in each thread. Figure 7 shows a summary of performance data grouped in higher level semantic groups (mpi, java, sun, and so forth) in the form of pie charts on a set of threads within each node. Each thread could be an application or a virtual machine level thread. Figure 8 shows a dynamic calltree on a selected thread. It shows the calling order of

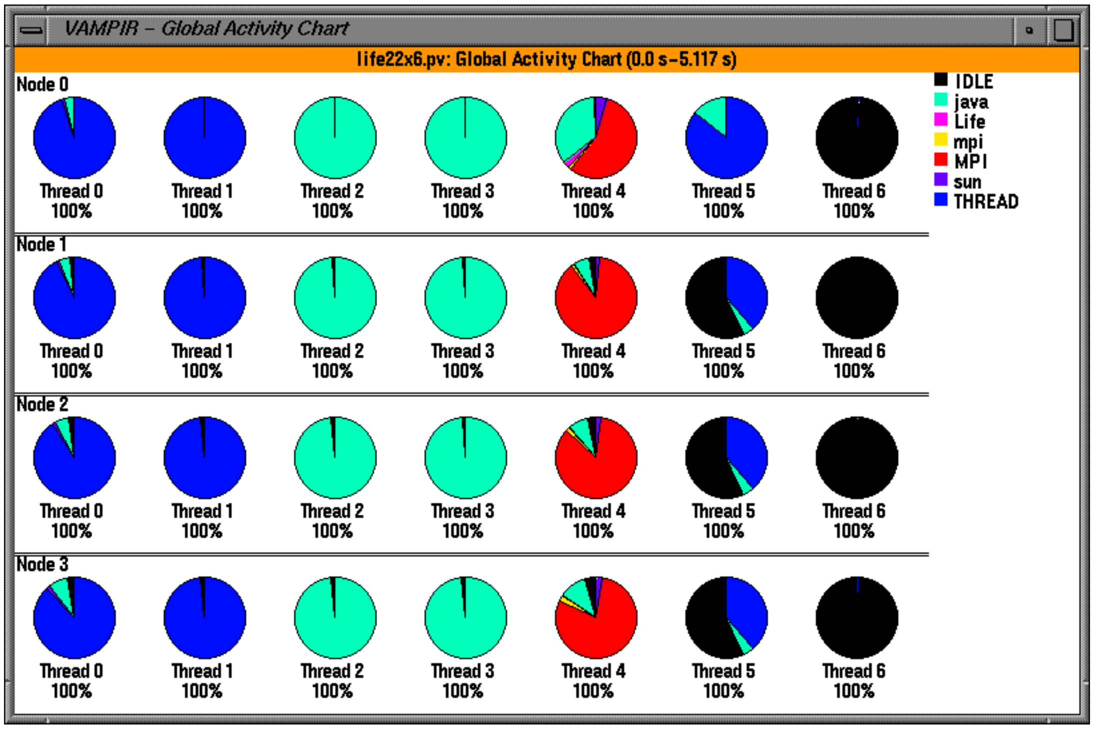


Figure 7. Vampir global activity chart illustrates thread grouping

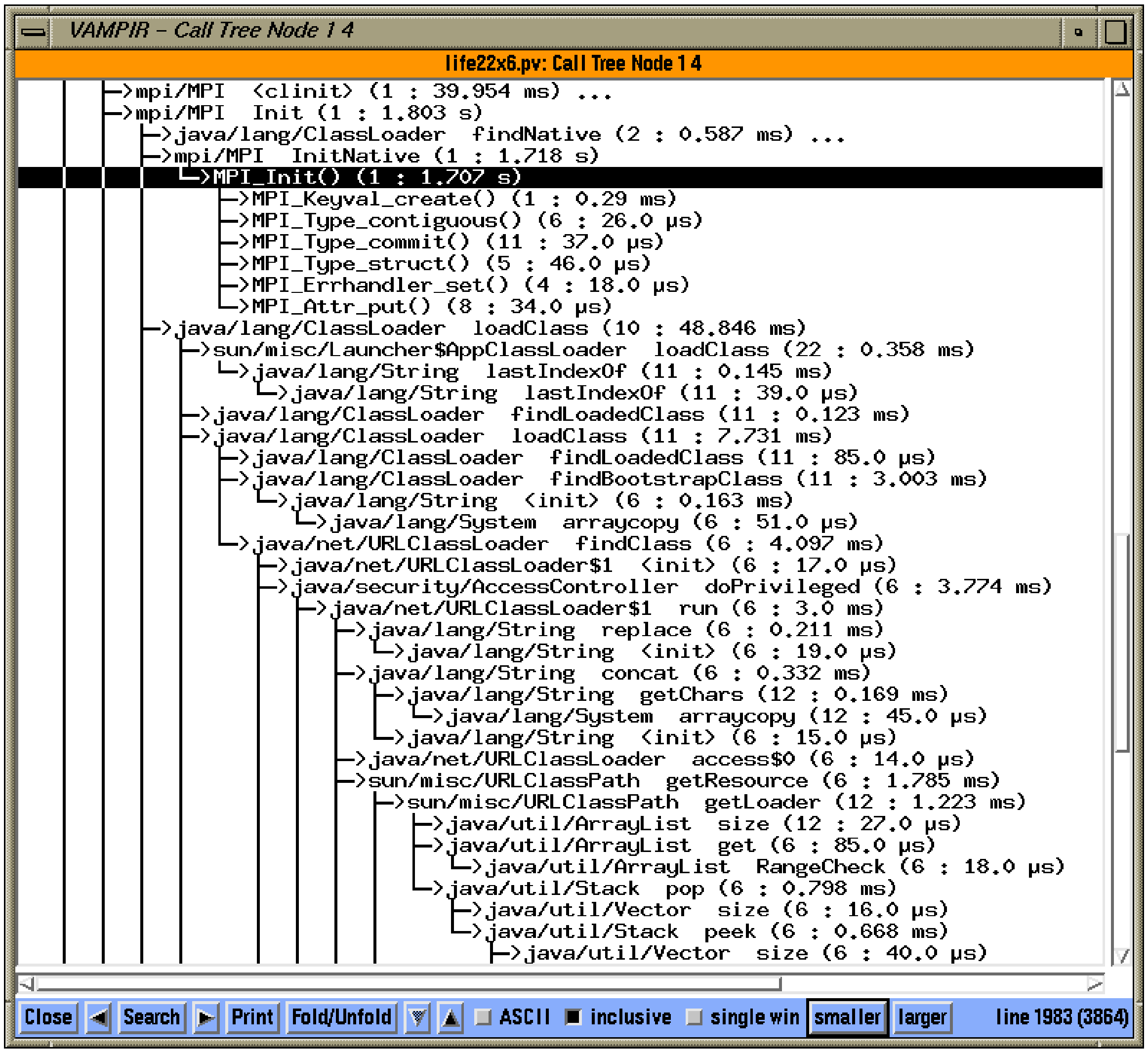
routines annotated with performance metrics (inclusive, exclusive times, and number of calls). A user can fold or unfold a segment of the tree to gain better insight. In Figure 9, we see a communication matrix display with nodes and threads along the rows and columns marking the senders and receivers, and the color-coded values in the matrix that show the extent of inter-thread message communication.

Grouping performance data according to virtual machine and application level entities is not new. It has been successfully demonstrated in Paradyn-J [10], a tool for detecting performance bottlenecks in interpreted, just-in-time compiled Java programs, where data is separately grouped in two distinct trees (one for the application, and another for the virtual machine).

This approach allows both application developers as well as virtual machine developers to gain valuable information regarding the interaction between the two groups. In contrast, as illustrated in the performance displays, TAU gathers performance data from MPI and Java layers in a seamlessly integrated fashion, showing the precise thread where MPI calls execute and allowing data to be grouped in two hierarchies according to nodes and threads and semantic groups. While providing a set of displays for profiling and tracing data, we can see the need for other customized, user-defined multi-dimensional displays that may show data in more effective ways. To accomplish this, TAU provides an open, documented interface for accessing performance data that it generates and illustrates with examples how a user could transform the data to commonly used performance data formats.

5. SELECTIVE INSTRUMENTATION

In examining the data output of a performance instrumented Java application, we notice that there is a significant amount of data about the internal workings of the JVM (e.g., see Figure 8). While this may provide a wealth of useful information for the JVM developer, it could inundate the application developer with superfluous details when a more selective focus is desired. To avoid making performance measurements for all system classes, the TAU Java instrumentation system must be extended to selectively disable certain events from measurement. How is TAU informed of which events to disable? Since Java classes are packaged in a hierarchical manner, our approach is to allow the user to specify a list of classes to be excluded, on the TAU instrumentation command line. For instance, if the user specified `java/lang, sun` in the exclude list, TAU should then eliminate all `java/lang/*` classes and `sun/*` classes from consideration.



```
│  ├->mpi/MPI  <clinit> (1 : 39.954 ms) ...
│  ├->mpi/MPI  Init (1 : 1.803 s)
│  │  ├->java/lang/ClassLoader  findNative (2 : 0.587 ms) ...
│  │  ├->mpi/MPI  InitNative (1 : 1.718 s)
│  │  │  └->MPI_Init() (1 : 1.707 s)
│  │  │     ├->MPI_Keyval_create() (1 : 0.29 ms)
│  │  │     ├->MPI_Type_contiguous() (6 : 26.0 µs)
│  │  │     ├->MPI_Type_commit() (11 : 37.0 µs)
│  │  │     ├->MPI_Type_struct() (5 : 46.0 µs)
│  │  │     ├->MPI_Errhandler_set() (4 : 18.0 µs)
│  │  │     └->MPI_Attr_put() (8 : 34.0 µs)
│  │  ├->java/lang/ClassLoader  loadClass (10 : 48.846 ms)
│  │  │  ├->sun/misc/Launcher$AppClassLoader  loadClass (22 : 0.358 ms)
│  │  │  │  └->java/lang/String  lastIndexOf (11 : 0.145 ms)
│  │  │  │     └->java/lang/String  lastIndexOf (11 : 39.0 µs)
│  │  │  ├->java/lang/ClassLoader  findLoadedClass (11 : 0.123 ms)
│  │  │  ├->java/lang/ClassLoader  loadClass (11 : 7.731 ms)
│  │  │  │  ├->java/lang/ClassLoader  findLoadedClass (11 : 85.0 µs)
│  │  │  │  ├->java/lang/ClassLoader  findBootstrapClass (11 : 3.003 ms)
│  │  │  │  │  └->java/lang/String  <init> (6 : 0.163 ms)
│  │  │  │  │     └->java/lang/System  arraycopy (6 : 51.0 µs)
│  │  │  │  └->java/net/URLClassLoader  findClass (6 : 4.097 ms)
│  │  │  │     ├->java/net/URLClassLoader$1  <init> (6 : 17.0 µs)
│  │  │  │     ├->java/security/AccessController  doPrivileged (6 : 3.774 ms)
│  │  │  │     │  ├->java/net/URLClassLoader$1  run (6 : 3.0 ms)
│  │  │  │     │  │  ├->java/lang/String  replace (6 : 0.211 ms)
│  │  │  │     │  │  │  └->java/lang/String  <init> (6 : 19.0 µs)
│  │  │  │     │  │  ├->java/lang/String  concat (6 : 0.332 ms)
│  │  │  │     │  │  │  ├->java/lang/String  getChars (12 : 0.169 ms)
│  │  │  │     │  │  │  │  └->java/lang/System  arraycopy (12 : 45.0 µs)
│  │  │  │     │  │  │  └->java/lang/String  <init> (6 : 15.0 µs)
│  │  │  │     │  │  ├->java/net/URLClassLoader  access$0 (6 : 14.0 µs)
│  │  │  │     │  │  ├->sun/misc/URLClassPath  getResource (6 : 1.785 ms)
│  │  │  │     │  │  │  ├->sun/misc/URLClassPath  getLoader (12 : 1.223 ms)
│  │  │  │     │  │  │  │  ├->java/util/ArrayList  size (12 : 27.0 µs)
│  │  │  │     │  │  │  │  ├->java/util/ArrayList  get (6 : 85.0 µs)
│  │  │  │     │  │  │  │  │  └->java/util/ArrayList  RangeCheck (6 : 18.0 µs)
│  │  │  │     │  │  │  │  └->java/util/Stack  pop (6 : 0.798 ms)
│  │  │  │     │  │  │  │     ├->java/util/Vector  size (6 : 16.0 µs)
│  │  │  │     │  │  │  │     └->java/util/Stack  peek (6 : 0.668 ms)
│  │  │  │     │  │  │  │        ├->java/util/Vector  size (6 : 40.0 µs)
```

Figure 8. Vampir dynamic calltree display on each thread shows the calling order annotated with performance metrics

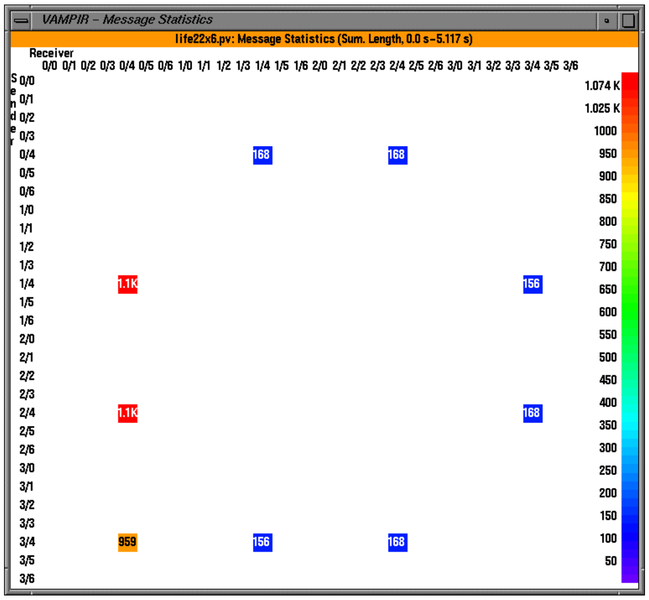

VAMPIR – Message Statistics
life22x6.pv: Message Statistics (Sum. Length, 0.0 s–5.117 s)
Receiver
Sender
168
168
1.1K
156
1.1K
168
959
156
168
1.074 K
1.025 K
1000
50

Figure 9. Vampir communication matrix shows the extent of inter thread communication

Shown below are the command line statements to run a java application, run a java application with TAU instrumentation fully enabled, and run a java application with TAU instrumentation selectively enabled:

```
% java <app> <args>

% java -XrunTAU <app> <args>

% java -XrunTAU:exclude=java/lang,sun <app> <args>
```

The implementation of this selective instrumentation in TAU is complicated by JVMPI processing. JVMPI allows the in-process profiling agent to enable and disable the notification of events using its event API. However, if the agent tries to disable the notification of the method entry or exit event, it affects all methods of a class, and not just methods that belong to a certain class. That is, the disabling has to do with events of type "method," not particular method events. This makes selective instrumentation difficult. Instead, TAU leaves method events enabled, but implements selection by comparing the name of a class with the "exclude list" specified by the user. This comparison is done when a class is loaded at runtime. If the class name is excluded, a flag is then maintained in each class method timer, indicating that instrumentation is disabled. At runtime, when the method executes, JVMPI informs TAU about method entry and exit events, and TAU in turn checks to see if its instrumentation is disabled by examining this flag and processes the event accordingly. Currently, the level of instrumentation granularity is the class, but we are looking into ways to refine the granularity of selection to class methods.

6. SOURCE-LEVEL INSTRUMENTATION

For other programming languages that the TAU performance system supports (C, C++, Fortran), standard routine entry/exit instrumentation is supplemented by the ability to specify "user-defined" performance events. These events can be associated with any code points the user desires. TAU provides an API to define the events, and to start and stop event profiling around code sections, including individual statements. However, in the first version of our Java JVMPI-based instrumentation, we were only able to see Java method invocation events. The definition and profiling of user events at the Java source level was not possible.

To accomplish this in TAU's current implementation for Java, we developed a source-level API in the form of a TAU Java package for creating user-level event timers. This API is consistent with similar capabilities TAU provides for other languages. The user can define events timers of a `TAU.Profile` class and then annotate the source code at desired places to start and stop the timers. Below is a example code segment demonstrating the API's use:

```
import TAU.*;


// Create timer
static TAU.Profile t= new TAU.Profile("Tau Timer",
  "test", "TAU_DEFAULT", TAU.Profile.TAU_DEFAULT);


t.Start();
// Code segment here
t.Stop();
```

The TAU Java package provides the API, but utilizes JNI to interface with the TAU profiling library. This library is implemented as a dynamic shared object that is loaded by the JVM or the TAU Java package. It is within the TAU profiling library that the performance measurements are made. However, TAU captures performance data with respect to nodes and threads of execution. What makes Java source-level instrumentation interesting is that node identification and JVM thread information is not accessible at the Java language level. Where does TAU get this information?

To maintain a common performance data repository in which performance data from multiple "streams" comes together and presents a consistent picture, we need instrumentation at various levels to co-operate. As shown in Figure 3, the TAU profiling library uses JNI to

Table I. TAU overhead for the parallel Java application Life

Operation		Mean (μsec)	Std. Deviation	Samples	Range (μsec)
Method Loading	profiling	30.28	7.12	123	20.14 - 70.14
	profiling + tracing	33.76	9.01	123	21.81 - 93.14
Method Entry	profiling	2.67	2.01	12860	1.14 - 50.14
	profiling + tracing	4.71	2.82	12860	3.14 - 190.14
Method Exit	profiling	1.16	0.31	12860	0.14 - 15.14
	profiling + tracing	2.85	1.29	12860	2.14 - 25.14

interface with the JVMPI layer to determine which JVM thread of execution is associated with Java method events and with MPI events. In the same manner, TAU uses this mechanism to determine thread information for user-defined events at the source level. To determine node information, TAU queries the MPI library to find out its process rank.

Thus, TAU instrumentation occurs at the Java source level, at the MPI wrapper library level, and at the virtual machine level. These different layers together form a consistent view of the execution model and thus must synchronize effectively to maintain the multi-threaded performance data in a consistent state.

7. MEASUREMENT OVERHEAD

Software-based instrumentation schemes have a runtime overhead that intrudes on application execution, possibly perturbing its performance [7]. It is impossible to completely eliminate this overhead, but it can be quantified and its effects evaluated to some extent. We have attempted to characterize the overhead that TAU generates in the execution of the Java application. Since TAU instrumentation is typically triggered at entry, exit, and initialization of methods, we break up the overhead in these three categories. We also consider the overhead when only profiling is enabled, and when profiling and tracing is selected.

As described earlier, TAU requires the use of JVMPI for performance measurement for two reasons. First, it gives a convenient mechanism for observing method entry/exit events and other JVM actions. Second, even if an alternative instrumentation approach was used, such as directly in the Java source or in JNI- linked libraries, JVMPI is the only current mechanism to obtain thread information and JVM state data. In evaluating TAU overhead, we are concerned with both the absolute overhead as well as the relative overhead in contrast to the JVMPI overhead. Although a full characterization of JVMPI overheads is beyond the scope of this paper, our experience is that a JVMPI-enabled application (without any performance measurement) can see performance delays. Because TAU executes native code in the JVM address space, its efficiency should be high save for JVMPI interactions. If, in the future, the JVMPI capabilities that TAU utilizes are offered by some other, more efficient means, the overhead of having JVMPI enabled may be avoided.

The experimental apparatus to quantify TAU measurement overhead is based on how classes are instrumented. Java supports dynamic loading of class bytecode in the virtual machine during program execution. This allows TAU to instrument only those classes that are loaded in the virtual machine, as opposed to all the classes. When a class is loaded, TAU examines the names of methods and creates timers for each method. To determine this cost of instrumenting a class, we can divide the time for loading a class by the number methods it contains to give an estimate of the fixed cost of method initialization. We measure all costs in terms of elapsed wall-clock time obtained by the system call `gettimeofday`. In a similar fashion, we measured overheads for method entry and method exit. All measurements take place after JVMPI calls the TAU profiler agent. Here we consider the standard time measurement where profile information is updated and trace data is optionally generated.

Table I shows the profiling overhead measurements in association with the overhead when tracing is also enabled. The overhead seen in this table includes disk I/O for storing the profile information at the end of the application or for saving per-thread trace buffers. We compute the cost of the `gettimeofday` call on the system and compensate for it while measuring the overhead associated with method loading, entry, and exit. The TAU overhead for each method is different and is influenced by the time spent looking up the mapping table, string operations that depend upon the length of a method name, load on the system, and other platform specific parameters. However, we can compute average costs and give an estimate for a specific platform. From the table, we see that method loading costs 30.28 microseconds on the average, and it costs 2.67 microseconds for method entry and 1.16 microseconds for method exit during profiling. The costs are a little higher when we generate both profiles and event-traces. The measurements were made on a quad Pentium III Xeon/550 MHz, 3GB RAM symmetric multiprocessor machine with the following software environment:

- TAU version 2.8.11
- RedHat Linux 6.1 operating system with 2.3.40 Linux kernel,
- GNU gcc 2.95.2 C++ compiler that used the -O2 optimization flag, and

- Blackdown JDK 1.2.2 Java runtime environment (version Linux_JDK_RC3) that used the native threads package and the Sunw JIT compiler.

TAU currently does not employ any means for compensating for the perturbation caused by the instrumentation. General techniques for compensating for instrumentation perturbation are addressed in [7].

8. CONCLUSIONS

As more applications for parallel and distributed systems are developed using portable hierarchical software frameworks, layered runtime modules, and multi-language software components, the requirements for integrated portable performance analysis will grow more complex. In particular, it becomes a challenge to observe performance events that occur throughout the software hierarchy and across language components and then relate those events to high-level execution abstractions and associated performance views.

Some of the challenges performance technologists face became apparent in our work with Java and its use in a MPI-based parallel execution environment. The extensions we made to the TAU system for unifying JVM versus native execution performance measurement, managing multi-level multi-threading, utilizing different instrumentation mechanisms for Java and MPI, and providing source-level instrumentation, all demonstrate TAU's robust capabilities. However, in the future, we also expect that new techniques for Java code parallelization will introduce new requirements for integrated performance instrumentation.

9. Acknowledgments

This work was supported by a DOE 2000 ACTS grant (No. DEFC 0398 ER 259 986), a DOE ASCI Level 3 grant (No. 03588-001-99 4R), and an NSF National Young Investigator award.

REFERENCES

1. M. Baker and B. Carpenter. Thoughts on the structure of an MPJ reference implementation. http://www.npac.syr.edu/projects/pcrc/HPJava/ mpiJava.html.
2. M. Baker, B. Carpenter, G. Fox, S. Ko, and S. Lim. mpiJava: An Object-Oriented Java Interface to MPI. In *Proc. International Workshop on Java for Parallel and Distributed Computing, IPPS/SPDP 1999*, April 1999.
3. R. Boisvert, J. Moreira, M. Philippsen, and R. Pozo. Java and Numerical Computing. *Computing in Science & Engineering*, 3(2):18–24, 2001.
4. M. P. I. Forum. MPI: A Message Passing Interface Standard. *International Journal of Supercomputer Applications (Special Issue on MPI)*, 8(3/4), 1994.
5. HPC++ Working Group. HPC++ White Papers. Technical Report TR 95633, Center for Research on Parallel Computation, 1995.
6. G. Judd. Design Issues for Efficient Implementation of MPI in Java. In *Proc. ACM JavaGrande Conf.*, pages 37–46, 1999.
7. A. Malony. *Performance Observability.* PhD thesis, University of Illinois, Urbana-Champaign, September 1990.
8. A. Malony and S. Shende. *Distributed and Parallel Systems: From Concepts to Applications*, chapter Performance Technology for Complex Parallel and Distributed Systems, pages 37–46. Kluwer, Norwell, MA, 2000.
9. W. Nagel, A. Arnold, M. Weber, H.-C. Hoppe, and K. Solchenbach. VAMPIR: Visualization and Analysis of MPI Resources. *Supercomputer*, 12(1):69–80, 1996.
10. T. Newhall. *Performance Measurement of Interpreted, Just-in-Time compiled, and Dynamically Compiled Executions.* PhD thesis, University of Wisconsin, Madison, August 1999.

11. U. of Oregon. TAU User's Guide. http://www.cs.uoregon.edu/research/paracomp/tau.

12. Pallas GmbH. VAMPIR: Visualization and Analysis of MPI Resources. http://www.pallas.de/pages/vampir.htm.

13. S. Shende, A. D. Malony, J. Cuny, K. Lindlan, P. Beckman, and S. Karmesin. Portable Profiling and Tracing for Parallel Scientific Applications using C++. In *Proceedings of the SIGMETRICS Symposium on Parallel and Distributed Tools (SPDT'98)*, pages 134–145, 1998.

14. SUN Microsystems Inc. Java Native Interface (JNI). http://java.sun.com/products/jdk/1.3/docs/guide/jni/index.html.

15. SUN Microsystems Inc. Java Virtual Machine Profiler Interface (JVMPI). http://java.sun.com/products/jdk/1.3/docs/guide/ jvmpi/jvmpi.html.

16. D. Viswanathan and S. Liang. Java Virtual Machine Profiler Interface. *IBM Systems Journal*, 39(1):82–95, 2000.