Performance Analysis of Parallel Java Applications on Shared-memory Systems

Jordi Guitart, Jordi Torres, Eduard Ayguadé
European Center for Parallelism of Barcelona (CEPBA)
Computer Architecture Department,
Technical University of Catalonia
C/ Jordi Girona 1-3, Campus Nord UPC, Mòdul C6,
E-08034 Barcelona (Spain)
{jguitart, torres, eduard}@cepba.upc.es

J. Mark Bull
Edinburgh Parallel Computing Centre (EPCC)
University of Edinburgh,
Mayfield Road, Edinburgh EH9 3JZ, Scotland, U.K.
markb@epcc.ed.ac.uk

ABSTRACT
In this paper we describe an instrumentation environment for the performance analysis and visualization of parallel applications written in JOMP, an OpenMP-like interface for Java. The environment includes two complementary approaches. The first one has been designed to provide a detailed analysis of the parallel behavior at the JOMP programming model level. At this level, the user is faced with parallel, work-sharing and synchronization constructs, which are the core of JOMP. The second mechanism has been designed to support an in-depth analysis of the threaded execution inside the Java Virtual Machine (JVM). At this level of analysis, the user is faced with the supporting threads layer, monitors and conditional variables. The paper discusses the implementation of both mechanisms and evaluates the overhead incurred by them.

1. INTRODUCTION
Over the last few years, Java has emerged as an interesting language for the network programming community. This has largely occurred as a direct consequence of the design of the Java language. This design includes, among others, important aspects such as portability and architecture neutrality of Java code, and its multithreading facilities. The latter is achieved through built-in support for threads in the language definition. The Java library provides the Thread class definition, and Java runtimes provide support for thread, monitor and condition variable primitives. These characteristics, besides others like its familiarity (due to its resemblance with C/C++), its robustness, security and distributed nature also make it a potentially interesting language for parallel scientific computing.

However, the use of Java for scientific parallel programming has to face the large overheads caused by the interpretation of the bytecode that lead to unacceptable performances. Many current JVMs try to reduce this overhead by Just-in-Time compilation. This mechanism tries to compile JVM bytecodes into architecture-specific machine code at runtime (on the fly). In any case, the naive use of the threads support provided by Java may incur overheads that can easily offset the gain due to parallel execution. Other drawbacks include the lack of support for complex numbers and multi-dimensional arrays.

A lack of suitable standards for parallel programming in Java is also a concern. Most of the current proposals to support the specification of parallel algorithms in Java reflect the alternatives that have been proposed for other languages such as Fortran or C. For instance, there have been proposals to implement common message-passing standards, such as PVM or MPI, by means of Java classes [7, 9]. Other proposals [6] try to make Java a data-parallel language similar to HPF, in which parallelism could be expressed in a more natural way. The extensions allow the definition of data-parallel operations, non-rectangular or multi-dimensional arrays or to allow some kind of data locality. The emerging OpenMP standard for Fortran and C/C++ has lead to the proposal of a similar paradigm in the scope of Java (JOMP [4]) and the automatic restructuring of Java programs for parallelism exploitation based either on code annotations or compiler-driven analysis [1, 2]. The implementation of these extensions is done through runtime libraries and compiler transformations in order to avoid the overhead introduced by the intensive creation of Java Threads [4, 12].

As soon as the performance gap experienced between parallel applications written in Java (or a dialect of Java) and their counterparts written in other languages such as Fortran or C closes, the community is likely to accept Java as a valid approach. Performance analysis and visualization of such parallel applications is going to be necessary in order to tune their behavior on the target parallel systems and JVM.

In this paper we present two instrumentation mechanisms that are useful to support the analysis of parallel applications written in the JOMP shared-memory programming paradigm for Java. The first one has been designed to provide a detailed analysis of the parallel behavior at the JOMP programming model level. At this level, the user is faced with parallel, work-sharing and synchronization constructs, which are the core of JOMP. The second mechanism has been designed to support an in-depth analysis of the threaded execution inside the JVM. At this level of analysis, the user is faced with the supporting threads layer (for instance, pthreads, monitors and conditional variables). These two mechanisms are complementary and compose JIS, the Java Instrumentation Suite currently under development at the CEPBA. The two mechanims can be used at the same time to record the activity at the two levels of analysis. The instrumentation process currently targets Paraver [13]. Paraver is a flexible performance visualization and analysis tool based on traces generated from a real execution. Other modules specifically designed to gather performance statistics or visualization tools could also be used as a back-end of the whole instrumentation process.

The implementation of these two approaches uses two different mechanisms to instrument the parallel application. The JOMP instrumentation mechanism, which is built on top of the JOMP compiler, injects probes during the code generation phase. The instrumentation at the JVM level is done by dynamically interposing the instrumentation code at run time.

The rest of the paper is organized as follows: Section 2 presents the JOMP compiler, a Java to Java translator that interprets JOMP directives and generates parallel code for the JOMP supporting runtime. Section 3 describes the instrumentation library designed to support the tracing process. Section 4 describes the compiler-based instrumentation mechanism and evaluates the overhead introduced. Section 5 describes the mechanism offered to support the analysis of the threaded execution inside the JVM. Finally, Section 6 concludes the paper and outlines our current research directions.

2. JOMP compiler

In this section, we describe a simple portable compiler that implements a large subset of the JOMP specification for the Java language [4]. Additional implementation details about the API and implementation can be found in elsewhere [5, 10]. Currently, a few parts of the specification have yet to be implemented, such as nested parallelism and array reductions.

The JOMP specification for Java includes parallel, work-sharing and synchronization constructs. The `parallel` directive is used to specify the creation of a team of threads that will concurrently execute the code. Work-sharing directives are provided to allow the distribution of work among the threads in a team: `for` directive to distribute iterations in a parallel loop, `sections` directive to parcel out a sequence of statements and `master` and `single` directive to specify the execution by a single thread in the team. Parallel and work-sharing constructs also allow redefining the scope of certain variables in order to be `shared`, `private`, `firstprivate`, `lastprivate` or `reduction`. Synchronization directives provide the mechanisms to synchronize the execution of the threads in the team: `barrier` and `critical` regions. Work-sharing constructs assume a default barrier synchronization at the end of the work; this global synchronization may be removed with the `nowait` clause.

2.1 Basic Structure

The JOMP Compiler is built around a Java 1.1 parser provided as an example with the JavaCC [11] utility. JavaCC comes supplied with a grammar to parse a Java 1.1 program into a tree, and an `UnparseVisitor` class, which unparses the tree to produce code. The bulk of the JOMP compiler is implemented in the `OMPVisitor` class, which extends the `UnparseVisitor` class, overriding various methods that unparse particular nonterminals. Because JavaCC is itself written in Java, and outputs Java source, the JOMP system is fully portable, and requires only a JVM installation in order to run it.

These overriding methods output modified code, which includes calls to the supporting runtime library to implement appropriate parallelism.

2.2 The parallel directive

Upon encountering a `parallel` directive within a method, the compiler creates a new class. The new class has a single method, `go()`, which takes a parameter indicating an absolute thread identifier. For each variable declared to be `private`, `firstprivate` or `reduction`, the `go()` method declares a local variable with the same name and type signature. The local `firstprivate` variables are initialized from the corresponding field in the containing class, while the local `private` variables have the default initialization. The local `reduction` variables are initialized with the appropriate default value for the reduction operator. Private objects are allocated using the default constructor. The main body of the `go()` method contains the code to be executed in parallel.

In place of the parallel construct itself, code is inserted to declare a new instance of the compiler-created class, and to initialize the fields within it from the appropriate variables. The `OMP.doParallel()` method is used to execute the `go()` method of the inner class in parallel. Finally, any necessary values are copied from class fields back into local variables.

A simple example illustrating this process is shown in Figure 1.

```
public class Hello {
  public static void void main (String argv[]) {
    int myid;
    //omp parallel private (myid)
    {
      myid = OMP.getThreadNum();
      System.out.println("Hello from" + myid);
    }
  }
}
```

(a)

```
public class Hello {
  public static void main (String argv[]) {
    int myid;

    // OMP PARALLEL BLOCK BEGINS
    __omp_Class0 __omp_Object0 = new __omp_Class0();
    __omp_Object0.argv = argv;
    try {
      jomp.runtime.OMP.doParallel(__omp_Object0);
      } catch(Throwable __omp_exception) {
        System.err.println("OMP Warning:
                 Illegal thread exception ignored!");
        System.err.println(__omp_exception);
        }
    argv = __omp_Object0.argv;
    // OMP PARALLEL BLOCK ENDS
    }
// OMP PARALLEL REGION INNER CLASS DEFINITION BEGINS
private static class __omp_Class0 extends
jomp.runtime.BusyTask {
  String [ ] argv;

  public void go(int __omp_me) throws Throwable {
    int myid;

    // OMP USER CODE BEGINS
    myid = OMP.getThreadNum();
    System.out.println("Hello from" + myid);
    // OMP USER CODE ENDS
    }
  }
// OMP PARALLEL REGION INNER CLASS DEFINITION ENDS
}
```

(b)

Figure 1. Example of code injection: parallel directive.

2.3 Work-sharing directives

Upon encountering a `for`, `sections` or `single` directive, a new block is created. For each variable declared to be `private`, `firstprivate`, `lastprivate` or `reduction`, a local variable is declared and initialized if necessary. These newly created variables are used to communicate the values of variables to the enclosing block. In the case of the `for` and `sections` directives, it is also necessary to declare a boolean variable to hold information about whether the current thread is the one performing the sequentially last iteration of the loop or the sequentially last `section`.

Inside the newly allocated block, a second block is created. For each variable declared to be `private`, `firstprivate`, `lastprivate`, or `reduction`, a new variable with the same name is declared. Variables declared as `reduction` are initialized with the appropriate value. `Private` and `lastprivate` variables are initialized by calling the default constructor in the case of class type variables, or left uninitialized in the case of primitive or array type variables. `firstprivate` variables are initialized with the appropriate value from the original variable. A `clone()` method is called to initialize class or array type variables.

Next, the code to actually handle the appropriate work-sharing directive is inserted. At the end of the inner block, appropriate local variables associated to `lastprivate` and `reduction` variables are updated.

After the end of the inner block, a code to update the global copies of `lastprivate` and `reduction` variables is inserted. Only the thread performing the sequentially last iteration of the loop or the sequentially last section updates `lastprivate` variables. The master thread of the team updates `reduction` variables. Finally, the outer block is closed.

2.3.1 The for directive

Upon encountering a `for` directive, the compiler inserts code to create two `LoopData` structures. One of these is initialized to contain the details of the whole loop, while the other is used to hold details of particular chunks. The generated code then repeatedly calls the appropriate `getLoop` function for the selected schedule, executing the blocks it is given, until there are no more blocks. If a dynamic scheduling strategy was used, the `ticketer` is then reset. Any reductions are carried out, and if the `nowait` clause is not specified, the `doBarrier()` method is called.
If the `ordered` clause is specified on a `for` directive, then a call to `resetOrderer()` method is inserted immediately prior to the loop, at which point the value of the first iteration number is definitely known. Upon encountering an `ordered` directive, the compiler inserts a call to `startOrdered()` before the relevant block with the parameter being the current value of the loop counter. After the block is inserted a call to `stopOrdered()`, with the parameter being the next value the loop counter would take *after* its current value, during sequential execution.

2.3.2 The sections directive

Upon encountering a `sections` directive, the compiler inserts code that repeatedly requests a ticket from the `ticketer`, and executes a different section depending on the ticket number. When there are no sections left, the `ticketer` is reset. If the `nowait` clause is not specified, the `doBarrier()` method is called.

2.3.3 The master directive

Upon encountering a `master` directive, the compiler inserts code to execute the relevant block if and only if the `OMP.getThreadNum()` method returns 0.

2.3.4 The single directive

Upon encountering a `single` directive, the compiler inserts code to get a ticket from the `ticketer`, execute the relevant block if and only if the ticket is zero, and then reset the ticketer. If the `nowait` clause is not specified, the `doBarrier()` method is called.

2.4 Synchronization directives

Upon encountering a `critical` directive, the compiler creates a synchronized block, with a call to `getLockByName()`. Upon encountering a `barrier` directive, the compiler inserts a call to the `doBarrier()` method.

3. INSTRUMENTATION METHODOLOGY

The analysis of JOMP applications in Java is done using traces from real executions in the parallel target architecture. These traces reflect the activity of each thread in the JOMP execution model. This activity is recorded in the form of a set of predefined state transitions (that are representative of the parallel execution of each parallel construct in JOMP) and the occurrence of some predefined events. Instead of providing a summary for the entire execution, this tracing environment provides a detailed view of the parallel application behavior. The visualization and analysis tool used afterwards (Paraver) helps the user in the process of understanding the application behavior and computing performance statistics and summaries.

The core of the environment is the instrumentation library (`ParaverLibrary`) that provides all the services required by the compiler to dynamically record the activity of the master and slave threads that compose the JOMP team of threads. The compiler inserts probes into the translated Java code in order to record this activity.

The library is implemented in C and offered to Java through the Java Native Interface JNI [15]. The instrumentation library offers the following services:

- ChangeState - Change the state of a thread.
- PushState - Store the current state of a thread in a private stack and change to a new one.
- PopState - Change the state of a thread to the one obtained from the private stack.

- UserEvent - Emit an event (type and associated value) for a thread.

The library also offers combined services to change the state and emit an event: ChangeandEvent, PushandEvent and PopandEvent. Two additional services are offered to initialize and finish the instrumentation process:

- InitLib - Initialize the library internal data structures to start a parallel trace receiving as parameters: 1) the maximum number of threads participating in the execution, 2) the maximum amount of memory that the library has to reserve for each thread buffer, and 3) the mechanism used to obtain timestamps.

- CloseLib - Stop the tracing; this call makes the library dump to disk all buffered data not yet dumped and write resulting sorted trace to a file.

For each action being traced, the tracing environment internally finds the time at which it was done and, if enabled, access to the hardware performance counters (that provide information about primary/secondary cache and TLB misses, number of fetched/issued/retired instructions, …) currently available in most microprocessors. Timestamps associated to transitions and events can be obtained using generic timing mechanisms (such as the `gettimeofday` system call) or platform-specific mechanisms (for instance the high-resolution memory-mapped clock). Hardware counters are accessed through machine specific mechanisms. All this data is written to an internal buffer for each thread (i.e. there is no need for synchronization locks or mutual exclusion inside the parallel tracing library). The data structures used by the tracing environment are also arranged at initialization time in order to prevent interference among threads (basically, to prevent false sharing). The user can specify the amount of memory used for each thread buffer. When the buffer is full, the runtime system automatically dumps it to disk.

Finally the instrumentation library generates a trace file with all the information that is required to analyze and visualize the execution of the threaded application. Paraver [13] serves as the visualization tool. It allows the user to navigate through the trace and to do an extensive quantitative analysis. A Paraver trace is a sequence of records, each with an associated timestamp. A record can be either a state transition or an event happening at a given instant in time. Next section details the contents of the trace.

4. CODE INJECTION

The JOMP compiler has been modified in order to enable the instrumentation of the transformed Java code. Calls to the parallel tracing library are inserted at specific points in the source code, where the state transitions occur and where flags are required to signal JOMP relevant actions.

4.1 Trace contents

Table 1 summarizes the different states that we consider relevant for a thread within the JOMP execution model. The `RUN` state corresponds to the execution of useful work, i.e. execution of work in the original source code. The `IDLE` state reflects the fact that a thread is waiting (outside a parallel region) for work to be executed. The JOMP runtime library creates threads at the first parallel region and keeps them alive until the end of the application. In the meanwhile, they check for new work to be executed, and if found, execute it. The `OVERHEAD` state shows that the thread is executing code associated with definition and initialization of `private`, `lastprivate`, `firstprivate` and `reduction` variables, or the determination of the tasks to be done in a work-sharing construct. The `SYNCHR` state refers to the situation in which a thread is waiting for another thread to reach a specific point in the program, or for access to a `ticketer` to guarantee specific ordered actions.

Table 1. Thread states in the JOMP instrumentation.

State	Description
IDLE	Thread is waiting for work to be executed
RUN	Thread is running
OVERHEAD	Thread is executing JOMP overhead
SYNCH	Thread is synchronizing with other threads in the team
TRACEFLUSH	Instrumentation library flushes internal buffer to disk

The trace can also contain events that provide additional information about the JOMP constructs being executed. Each event has two fields associated: *type* and *value*. The *type* is used to indicate:

- Entry/exit to/from a parallel, work-sharing or synchronization construct.

- Entry/exit to/from a procedure.

The *value* is used to relate the event type with the source code (for instance, line number in the source code and method name). Events are also used to capture the value of hardware counters.

The communication between the event types and values assigned by the compiler and Paraver is done through a configuration file generated by the compiler itself.

4.2 Examples

Figure 2 shows instrumented parallel code for the simple example in Figure 1.a. The compiler assumes the following association of numerical values to thread states: 0-`IDLE`, 1-`RUN`, 3-`SYNCR` and 7-`OVERHEAD`.

Notice that the compiler forces a state change to `OVERHEAD` (7) as soon as the master thread starts the execution of the block of code that encapsulates the parallel construct in the main method. The previous state is stored in an internal stack so that the master thread can restore it as soon as it finishes the execution of this block of code. Each thread in the team executing the `go()` method in class `__omp_Class0` changes to the `RUN` (1) state when it starts the execution of the user code. The previous state (`OVERHEAD` for the master thread or `IDLE` for the other threads) is stored in the internal stack. After executing the original user code, each thread restores its previous state. Then the master thread changes the state to `SYNCH` (3) in order to represent the implicit barrier synchronization at the end of the parallel construct. The following events types and values are used by the compiler: types `100` and `101` indicate the beginning and end of the parallel construct; value `500` indicates that this parallel construct is found at a certain line and method in the original source code.

```
public class Hello {
  public static void main (String argv[]) {
    paraverlib = new ParaverLibrary();
    int myid;

    paraverlib.InitLib(jomp.runtime.OMP.
                       getMaxThreads(),4096,1);
    // OMP PARALLEL BLOCK BEGINS
    paraverlib.PushandEvent(
              jomp.runtime.OMP.getThreadNum(),
              7,100,500);
    __omp_Class0 __omp_Object0 = new __omp_Class0();
    __omp_Object0.argv = argv;
    try {
      jomp.runtime.OMP.doParallel(__omp_Object0);
      } catch(Throwable __omp_exception) {
        System.err.println("OMP Warning:
                Illegal thread exception ignored!");
        System.err.println(__omp_exception);
        }
    argv = __omp_Object0.argv;
    paraverlib.PopandEvent(
            jomp.runtime.OMP.getThreadNum(),101,500);
    // OMP PARALLEL BLOCK ENDS
    paraverlib.CloseLib();
    }
// OMP PARALLEL REGION INNER CLASS DEFINITION BEGINS
private static class __omp_Class0 extends
jomp.runtime.BusyTask {
  String [ ] argv;

  public void go(int __omp_me) throws Throwable {
    int myid;

    // OMP USER CODE BEGINS
    paraverlib.PushState(
              jomp.runtime.OMP.getThreadNum(),1);
    myid = OMP.getThreadNum();
    System.out.println("Hello from" + myid);
    paraverlib.PopState(
              jomp.runtime.OMP.getThreadNum());
    // OMP USER CODE ENDS
    if (jomp.runtime.OMP.getThreadNum(__omp_me)==0)
         paraverlib.ChangeState(
              jomp.runtime.OMP.getThreadNum(),3);
    }
  }
// OMP PARALLEL REGION INNER CLASS DEFINITION ENDS
}
```

Figure 2. Example of code injection: parallel directive.

Figure 3 shows another example with the transformation of a JOMP `single` work-sharing construct (Figure 3.a) into instrumented parallel code (Figure 3.b). Each thread changes to the OVERHEAD (7) state when it is checking if it has to execute the user code. Only one of the threads in the JOMP team will do so, changing to RUN (1) state. A ticketing mechanism is used by the JOMP compiler and runtime to allow a single thread to execute the user code. The compiler also injects code to indicate when each thread in the team reaches the implicit barrier at the end of the work-sharing construct. All threads go into the `SYNCH` state to indicate this. The compiler generates the following event types and values: type `108` and `109` to indicate the begin and end of the `single` work-sharing construct; value `510` to indicate the source line and method name in the original source code.

```
for(int j=0; j<innerReps;j++) {
  //omp single
  {
  delay(delayLenght);
  }
}
```

(a)

```
for (int j=0; j<innerReps; j++){
  // OMP SINGLE BLOCK BEGINS
  paravelib.PushandEvent(
        jomp.runtime.OMP.getThreadNum(),
        7,108,510);
  if(jomp.runtime.OMP.getTicket(__omp_me) == 0) {
    // OMP USER CODE BEGINS
    paraverlib.PushState(
        jomp.runtime.OMP.getThreadNum(),1);
    delay(delayLength);
    paraverlib.PopState(
        jomp.runtime.OMP.getThreadNum());
    // OMP USER CODE ENDS
    }
  jomp.runtime.OMP.resetTicket(__omp_me);
  paraverlib.PushState(
        jomp.runtime.OMP.getThreadNum(),3);
  jomp.runtime.OMP.doBarrier(__omp_me);
  paraverlib.PopState(
        jomp.runtime.OMP.getThreadNum());
  paraverlib.PopandEvent(
        jomp.runtime.OMP.getThreadNum(),109,510);
  // OMP SINGLE BLOCK ENDS
}
```

(b)

Figure 3. Example of code injection: single directive.

4.3 Visualization of traces

Figure 5 (top) shows a Paraver window displying the execution trace for one iteration of the LUAppl kernel, a LU reduction kernel over a two-dimensional matrix of double-precision elements. The source code for the kernel is shown in Figure 4. The horizontal axis represents execution time in microseconds. The vertical axis shows the four JOMP threads that compose the team. Each thread evolves though a set of states, each one represented with a different color (as indicated with the legend on the right). Flags appearing on top of each thread bar are used to visualize the events mentioned in Section 4.1. For instance, all the threads start executing the body of the parallel construct, and distribute themselves the work (`OVERHEAD` state, yellow color in the visualization) as indicated by the two `for` work-sharing directives. After determining the chuck of iterations, each thread executes them (`RUN` state, dark blue color in the visualization). A barrier synchronization happens at the end of second work-sharing construct (`SYNCH` state, red color in the visualization), which forces all the threads to wait. Notice that the `nowait` clause in the first work-sharing construct omits the implicit barrier synchronization.

The bottom part in Figure 5 shows the kind of information reported by Paraver when the user clicks on a specific point of the trace. Observe that, in addition to timing measurements and thread state, Paraver also relates the visualization with the original JOMP code.

```
for (k=0;k<SIZE;k++) {
    //omp parallel
    {
      //omp for schedule(static) nowait
      for (int i=k+1;i<SIZE;i++) {
         matrix[i][k] = matrix[i][k] / matrix[k][k];
      }
      //omp for schedule(static)
      for (int i=k+1;i<SIZE;i++) {
         for (int j=k+1;j<SIZE;j++) {
              matrix[i][j] = matrix[i][j]
                      - matrix[i][k] * matrix[k][j];
         }
      }
    }
}
```

Figure 4. JOMP code for the LUAppl kernel (main loop)

4.4 Evaluation

Our experimental environment is based on a 64-processor Origin 2000 architecture with MIPS R10000 processors at 250 MHz [14] running the JVM version Sun Java 1.2.2.

4.4.1 Instrumentation microbenchmarks

In order to evaluate the instrumentation library overhead, we have designed a microbenchmark that measures the individual overhead incurred by each kind of service in the library. For instance, to measure the overhead due to changing the state of a thread using the `PushState` and `PopState` routines, we measure the time taken to execute

```
for (int j=0; j<innerReps; j++){
  lib.PushState(0,1);
  delay(delayLength);
  lib.PopState(0,1);
}
```

and subtract the reference time, that is the time taken to execute

```
for (int j=0; j<innerReps; j++){
  delay(delayLength);
}
```

on a single thread, and dividing by twice the number of iterations executed (`2*innerReps`). The `delay` method contains a dummy loop of length `delayLength`; this length is chosen so that the overhead of the routines and the time taken for by this routine are the same order of magnitude. The value of `innerReps` is chosen so that the execution time is significantly larger that the clock resolution.

Table 2: Results from the instrumentation microbenchmark

Function pair	Overhead (microseconds)
ChangeState	6
PushState / PopState	6.5
UserEvent	6
PushandEvent / PopandEvent	7

Table 2 summarizes the overhead measured for the different services available in the instrumentation library. The invocation of these services is done through the Java Native Interface. The version evaluated uses the `gettimeofday` system call. Notice that it is in the order of 6-7 microseconds per call.

4.4.2 JOMP microbenchmarks

The JOMP microbenchmarks are based on a set of microbenchmarks developed for the Fortran OpenMP interface, and described in [3]. In this paper we only use the subset designed

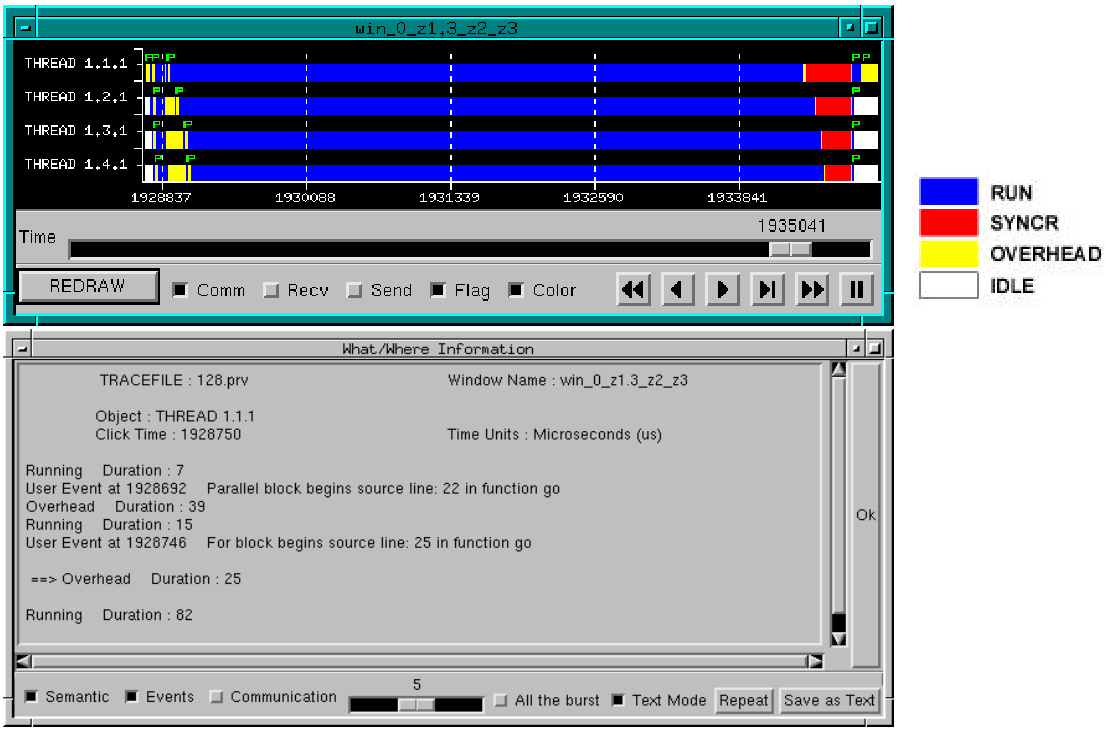

Figure 5. Two Paraver windows with the execution trace for one iteration of the LUAppl kernel (top) and additional information related to events at a specific point (bottom).

to measure the overhead of the JOMP directives implementation. The technique used is to compare the time taken for a section of code executed sequentially, to the time taken for the same code executed in parallel enclosed in a given directive. For example, we measure the overhead of the `for` directive by measuring the time taken to execute

```
//omp parallel
{
  for (int j=0; j<innerReps; j++){
    //omp for
    for (int i=0; i<nThreads ;i++) {
      delay(delayLength);
    }
  }
}
```

subtracting the reference time, that is the time taken to execute the same reference loop described in 4.1. Similar comparisons are used to measure the overheads of other directives.

For mutual exclusion directives, we take a similar approach. For example, to measure the overhead of the `critical` directive, we measure the execution time for

```
//omp parallel
{
  for (int j=0; j<innerReps/nThreads; j++){
    //omp critical
    {
      delay(delayLength);
    }
  }
}
```

then subtract the same reference time as above, and divide by `innerReps`.

To obtain accurate results, we have to be careful in our choice of clock routines. Second differences of raw clock values are used to compute overheads, and the run time for each measurement cannot be too large, since we need to repeat the measurements many times for statistical stability. We therefore have to ensure that the values returned by the clock routine are not only sufficiently accurate (typically to the nearest microsecond), but also sufficiently precise. In particular, clock routines returning 32-bit floating-point clock values in seconds can result in significant loss of precision. Since Java provides only a millisecond clock (`System.currentTimeMillis()`), we use the Java Native Interface to call a system-specific high resolution clock.

To obtain statistically meaningful results, we repeat each overhead measurement a number of times within each run, and for a number of different runs. The reason for this is that we observe a much larger variability *between* different runs that between different measurements *within* the same run. The cause of this is not clear---it may be due to the physical memory locations of synchronization variables altering from run to run.

Table 3 shows the overhead incurred by the instrumentation for the different set of JOMP directives evaluated in the microbenchmark. The first column shows the overhead in the execution of the code injected by the JOMP compiler when instrumentation is disabled. The second one corresponds to the overhead introduced by the instrumentation calls. Half of this overhead is caused by the invocation of the `OMP.getThreadNum` method. This method returns the identifier of the thread that is going to execute the instrumentation probe. The compiler could optimise that (by keeping this identifier whenever possible in a local variable) in order to reduce this overhead.

Table 3: Overhead in the instrumentation of JOMP constructs (microseconds)

Directive	JOMP overhead	Instr. overhead
parallel	3.45	47.75
for	18.35	99.35
parallel for	27.16	105,98
barrier	0.63	30.24
single	5.32	87.85
critical	2.51	47.15

4.4.3 LUAppl kernel

The instrumentation process of a JOMP application introduces some additional overhead due to the execution of `InitLib` and `CloseLib`. For instance, when the application is going to finish, the instrumentation library joins the per-thread buffers into a single trace (ordered in time). This adds an extra overhead to the whole execution time of the job that does not have any impact in the trace. For the LUAppl kernel mentioned in Section 4.3, the overhead introduced for the whole instrumentation process is presented in Table 4. The table reports the execution time in milliseconds when the kernel is executed with 4 processors and with different problem Notice that the overhead is reasonable (less than 3%).

Table 4: Execution Time of the LUappl (in milliseconds) and instrumentation overhead.

Matrix size	Original	Instrumented	Overhead
128x128	1899	1949	2.63%
256x256	15842	16222	2.4%
512x512	105962	108092	2%

5. JVM THREADS INSTRUMENTATION

The JOMP runtime uses Java Threads as the supporting threading model. In this section we describe a second instrumentation approach designed to support an in-depth analysis of the threaded execution inside the JVM. At this level of analysis, the user is faced with the supporting threads layer (for instance pthreads), monitors and conditional variables. This approach is complementary to the one described in Section 4 and can be used simultaneously for the same application run.

The instrumentation at the level of the JVM is done by dynamically interposing the instrumentation code at run time, avoiding recompilation of the original code or the JVM. This implementation is based on the mechanism already presented in [8]. The reader is referred to this publication for additional implementation details.

5.1 Dynamic code interposition

Dynamic linking is a feature available in many modern operating systems. Program generation tools (compilers and linkers) support dynamic linking via the generation of linkage tables. Linkage tables are redirection tables that allow delaying symbol resolution to run time. At program loading time, a system component fixes each pointer to the right location using some predefined resolution policies. Usually, the format of the object file as well as these data structures are defined by the system Application Binary Interface (ABI). The standardization of the ABI makes possible to take generic approaches to dynamic interposition.

Code interposition is used in this approach to instrument the code of the JVM. References to specific functions of the dynamically linked libraries used by the JVM are captured and probed. This instrumentation does not require any special compiler support and makes it unnecessary to rebuild either the bytecode of the application or the executable of the JVM. The probes executed with the intercepted calls generate a trace file with all the information that is required to analyze and visualize the execution of the threaded application.
The required knowledge about the execution environment is expressed using a state transition graph, in which each transition is triggered by a procedure call and/or a procedure return. In order to drive this transition graph, we use dynamic code interposition, avoiding instrumentation and recompilation of the source code and/or the JVM.

5.2 Trace contents

Table 5 summarizes the different states that we consider for a thread using this approach. As we mentioned before, the trace can also contain events that provide additional information about the behavior of the threaded application. The following events could be recorded in the trace:

- Information relative to the monitor in which a thread is blocked (either in the queue of threads waiting to enter into the monitor or in the queue of threads waiting in this monitor). Some of these monitors are registered by the JVM, in which case the event reports the name; otherwise, it reports a numerical identifier.
- Information relative to actions performed by a thread, for instance processor yield, entry to and exit from code protected by a mutual exclusion or notify on a monitor (to wake up a previously blocked thread on that monitor).
- In general, the invocation of any method in the Java Thread API, for instance sleep or interrupt.

Table 5. Thread states in the JVM instrumentation.

State	Description
INIT	Thread is being created and initialized
RUN	Thread is running
BLOCKED IN CONDVAR	Thread is blocked waiting on a monitor
BLOCKED IN MUTEX	Thread is blocked waiting to enter in a monitor
STOPPED	Thread has finalized

5.3 State transition graph

The monitoring methodology is based on the fact that the JVM invokes a set of run-time services at key places in order to use threads or to synchronize them. Figure 6 shows the state transition graph, in which nodes represent states, and edges correspond to procedure calls (indicated by a + sign) or procedure returns (indicated by a - sign) causing a state transition. We can also monitor internal threads of the JVM (such as the finalizer and garbage collector). In this case, the invocation of other procedures (such as `InitializeGCThread`) are used to trigger the transition to the `INIT` state instead of the invocation of procedure `java_lang_Thread_start`.

This transition graph is then used to derive the interposition routines used to keep track of the state in the performance monitoring backend. These routines are simple wrappers of functions that change the thread state, emit an event and/or save thread information in the internal per-thread buffers. These wrappers can perform instrumentation actions before and/or after the call being interposed.

The interposition of other functions leads to the generation of an event in the trace and/or to an update of the internal structures of the instrumentation system (e.g. `pthread_mutex_unlock`, `pthread_cond_signal`, `pthread_cond_broadcast`, `sched_yield`). These functions do not provoke a direct change in the state of any other thread.

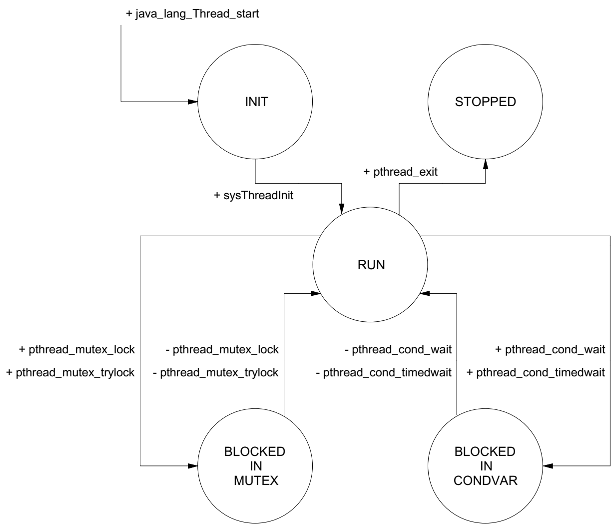

Figure 6. State transition graph for native threads.

5.4 Visualization of traces

In order to show the information collected by the instrumentation mechanism, we use a version of the LUAppl programmed directly using Java Threads. Figure 7 (top) shows a Paraver window in which the behavior of the LU application with 4 threads is shown. The horizontal axis represents execution time (in microseconds). The vertical axis shows the different threads used by the application: `MAIN` stands for the main thread of the Java application (the one executing the `main` method), and `USER4` to `USER7` are slave threads created by the `MAIN` thread. Each thread evolves through a set of states (`INIT`, `RUNNING`, `BLOCKED`

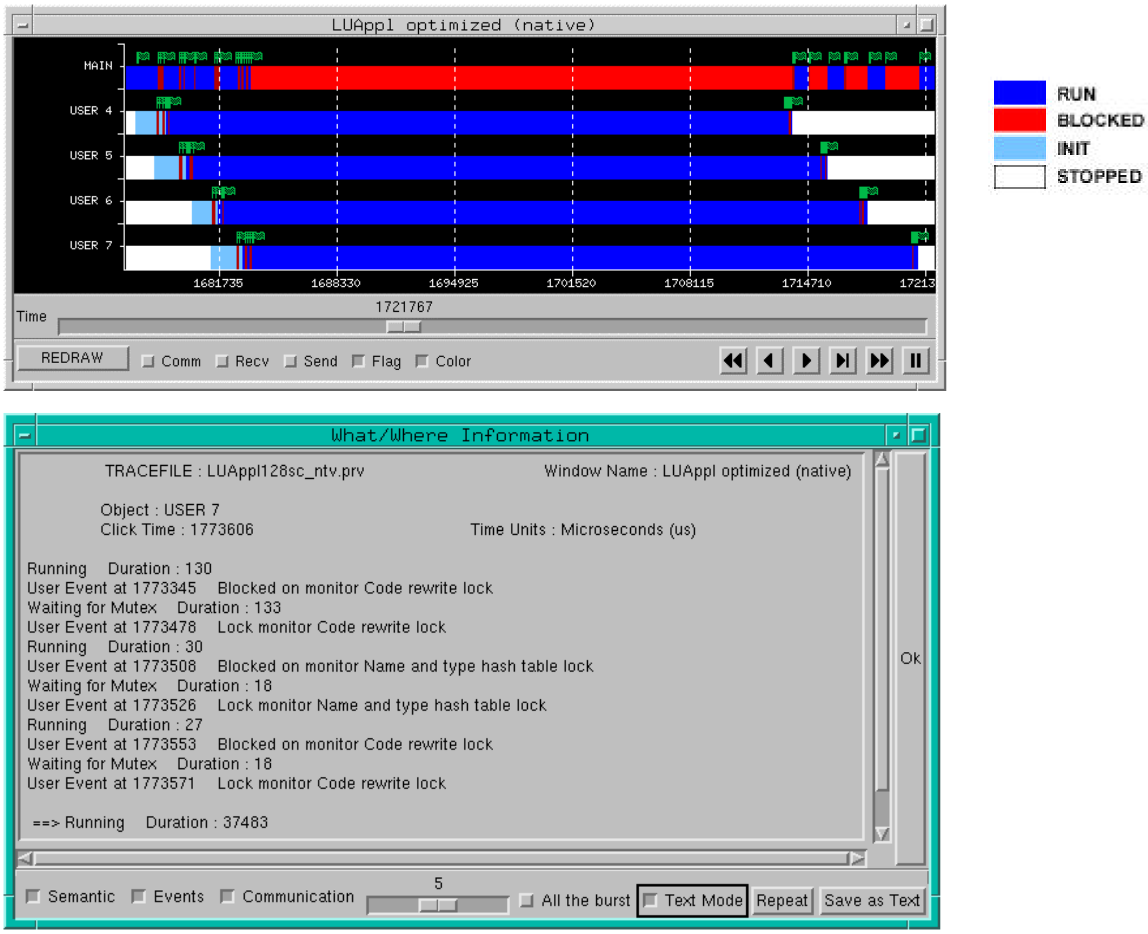

Figure 7. Two Paraver windows with the execution trace for one iteration of the LUAppl kernel (top) and additional information related to events at a specific point (bottom).

and STOPPED). Colors are used to identify each state. For instance, light blue reflects that the thread is being created (INIT), dark blue reflects that the thread is in the RUN state, red indicates that the thread is blocked either on a mutual exclusion or conditional variable, and white indicates that the thread has finished (STOPPED). Flags are used to visualize the events captured during the instrumentation.

The bottom part in Figure 7 shows the kind of information reported by Paraver when the user clicks on a specific point of the execution trace. Notice that the information is related with the threaded execution inside the JVM and not with the high-level programming model. For instance, in the example Paraver informs about actions on a set of predefined JVM locks.

5.5 Instrumentation overhead

Table 6 shows the execution for a version of the LUAppl kernel programmed using Java threads. Notice that the overhead is higher than the overhead introduced by the instrumentation injected by the JOMP compiler (Table 4). However the overhead is kept reasonably low (below 8%) and considered acceptable taking into account the level of detail provided by the process.

Table 6: Instrumentation overhead for LUAppl with 4 processors (execution times in milliseconds).

Matrix size	Java Threads	Instrumented version	% Overhead
128x128	2795	2996	7,19 %
256x256	17542	17975	2,47 %
512x512	109976	110857	0,80 %

6. CONCLUSIONS AND FUTURE WORK

In this paper we have described the two main approaches available in the Java Instrumentation Suite (JIS) for the performance analysis and visualization of parallel applications written in JOMP, an OpenMP-like interface for Java.

The environment includes two complementary approaches. The first one has been designed to provide a detailed analysis of the parallel behavior at the JOMP programming model level. At this level, the user is faced with parallel, work-sharing and synchronization constructs, which are the core of the JOMP programming model. The second mechanism has been designed to support an in-depth analysis of the threaded execution inside the JVM. At this level of analysis, the user is faced with the supporting threads layer, monitors and conditional variables. The paper described the implementation of the two approaches and evaluated the overhead using a set of microbenchmarks and application kernel. The overhead introduced is reasonably small (below 3% and 8%, respectively) and does not distort the execution of the parallel application.

This instrumentation is in fact a first step in the design of a platform for doing research on scheduling mechanisms and policies oriented towards optimizing the execution of multithreaded Java applications on parallel servers. The whole effort is part of the Barcelona Java Suite (BJS) project at the CEPBA (http://www.cepba.upc.es/BJS).

7. ACKNOWLEDGMENTS

We acknowledge the European Center for Parallelism of Barcelona (CEPBA) and the Edinburgh Parallel Computing Center (EPCC) for supplying the computing resources for our experiments. This work is partially supported by the Ministry of Education of Spain under contract TIC98-511.

8. REFERENCES

[1] A.J.C. Bik and D.B. Gannon. Automatically exploiting implicit parallelism in Java. Concurrency: Practice and Experience, 9(6):579--619, June 1997.

[2] A.J.C. Bik, J.E. Villancis, and D.B. Gannon. Javar: A prototype Java restructuring compiler. UICS Technical Report TR487, 1997.

[3] J.M. Bull. Measuring Synchronisation and Scheduling Overheads in OpenMP. In Proceedings of First European Workshop on OpenMP, Lund, Sweden, September 1999.

[4] .M. Bull and M.E. Kambites. JOMP -- an OpenMP-like interface for Java. In Proceedings of the 2000 ACM Java Grande Conference. June 2000.

[5] J.M. Bull, M.D. Westhead, M.E. Kambites and J.Obdrvzalek. Towards OpenMP for Java. In Proceedings of Second European Workshop on OpenMP, Edimburgh, UK, September 2000.

[6] B. Carpenter, G. Zhang, G. Fox, X. Li, and Y. Wen. HPJava: data parallel extensions to Java. Concurrency: Practice and Experience, 10(11-13):873--877, September 1998.

[7] A. Ferrari. JPVM: network parallel computing in Java. In Proceedings of the 1998 ACM Workshop on Java for High-Performance Network Computing, March 1998.

[8] J. Guitart, J. Torres, E. Ayguadé and J. Labarta, Java Instrumentation Suite: Accurate Analysis of Java Threaded Applications. 2nd Workshop on Java for High Performance Computing (part of the 14th ACM International Conference on Supercomputing ICS'00), Santa Fe, New Mexico (USA). May 7, 2000.

[9] G. Judd, M. Clement, Q. Snell, and V. Getov. Design issues for efficient implementation of MPI in Java. In Proceedings of the 1999 ACM Java Grande Conference, 1999.

[10] M.E. Kambites. Java OpenMP: Demonstration implementation of a compiler for a subset of OpenMP for Java, EPCC Tech. Report EPCC-SS99-05, September 1999, www.epcc.ed.ac.uk/ssp/1999/ProjectSummary/kambites.html

[11] Metamata Inc. JavaCC: The Java Parser Generator, www.metamata.com/JavaCC.

[12] J. Oliver, E. Ayguadé and N. Navarro. Towards an efficient exploitation of loop-level parallelism in Java. In Proceedings of the 2000 ACM Java Grande Conference. June 2000.

[13] Paraver: Parallel Program Visualization and Analysis Tool. European Center for Parallelism of Barcelona. www.cepba.upc.es/paraver.

[14] Silicon Graphipcs Inc. Origin200 and Origin2000 Technical Report, 1996.

[15] Sun Microsystems, “Java Native Interface”, Mar 2000, http://java.sun.com/products/jdk/1.3/docs/guide/jni/index.html