Strategies for the efficient exploitation of loop-level parallelism in Java*

José Oliver, Eduard Ayguadé, Nacho Navarro
Jordi Guitart and Jordi Torres
Computer Architecture Department, Technical University Of Catalonia
C/ Jordi Girona 1-3, Campus Nord, 08034 Barcelona, Spain
{joseo,eduard,nacho,jguitart,torres}@ac.upc.es

ABSTRACT
This paper analyzes the overheads incurred in the exploitation of loop-level parallelism using Java Threads and proposes some code transformations that minimize them. Avoiding the intensive use of Java Threads and reducing the number of classes used to specify the parallelism in the application results in promising performance gains that may encourage the use of Java for exploiting loop-level parallelism. On average, the execution time for our synthetic benchmarks is reduced by 50% from the simplest transformation when 8 threads are used. The paper explores some possible enhancements to the Java threading API oriented towards improving the application–runtime interaction.

1. INTRODUCTION
Over the last years, Java has emerged as an interesting language for the Internet community. This fact has its basis in the design of the Java language. This design includes, among others, important aspects such as portability and architecture neutrality of Java code, or its multithreading facilities. The latter, is achieved through the 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 lock primitives. These characteristics, besides others like its familiarity (due to its resemblance with C/C++), its robustness and security or its distributed nature have made it an interesting language for scientific parallel computing.

However, the use of Java for scientific parallel programming has to face with the large overheads caused by the interpretation of the bytecodes, that leads to unacceptable performances. Many current JVM 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 in overheads that may easily offset the gain due to the parallel execution.

In this paper, we analyze the overheads introduced by the Java Threads when they are used to exploit loop-level parallelism (one of the most important found in scientific applications). We also present two transformations that can be applied to a Java program in order to efficiently exploit it. The evaluation of the proposals takes into account the overhead introduced in execution time and the increase in the number of classes needed for the application.

From the experimental evaluation of the proposed transformations, we conduct an analysis of the threaded execeution behaviour. This analysis has provided us come hints on how to modify the behaviour of the multithreaded runtime resulting in important performance gains. These results will probably end up in the proposal of API modifications and extensions.

The rest of the document is structured as follows: section 2 presents some related work. In section 3 we explain and compare three different techniques to exploit loop-level parallelism in Java. Section 4 evaluates the proposed transformations. Section 5 explores some possible enhancements to the Java threading API. Finally, section 6 concludes the paper, and states some future work.

2. RELATED WORK
Most of the current proposals to support the specification of parallel algorithms using Java mirror the large number of alternatives that have been proposed for other languages like FORTRAN or C. Some of them [11, 6] are based on the implementation of common message-passing standards, such as PVM or MPI [7, 18] by means of Java classes that, in turn, make use of Java communication classes [9] or some modified version of them [14, 15, 16]. These ideas and proposals are oriented to distributed processing, and do not attempt to deal with shared-memory parallelism.

*This work has been supported by the European Community under the ESPRIT project E21907 (NANOS) and the Ministry of Education of Spain (CICYT) under contracts TIC 98-0511 and TIC97-1445CE.
A preliminary version of this paper appears in the proceedings of the ACM 2000 Java Grande Conference.

There are also a number of proposals for making Java a data-parallel language, such as HPJava, Titanium or Spar [4, 5, 19, 20], in which parallelism could be expressed in a more natural way. These proposals, however, imply the modification of the Java language itself (in fact, these extensions become a Java superset or a Java dialect), in order to allow the definition of data-parallel operations, non-rectangular or multi-dimensional arrays or to allow some kind of data locality.

Finally, other authors propose the use of a shared-memory paradigm and the automatic restructuring of Java programs for parallelism exploitation based either on code annotations or compiler-driven analysis. For instance, Bik et al. [1] describe the restructuring process that should be carried out in order to exploit the parallelism found in loops or multi-way recursive methods. These works, however, make intensive use of Java Threads to exploit the parallelism available. As we will show in this paper there are other possibilities that allow the exploitation of some of this parallelism without having to pay the possible overhead introduced by the intensive use of the Java threading system.

3. EXPLOITING PARALLELISM WITH JAVA MULTITHREADING SUPPORT

This section presents and compares some transformations that can be applied to Java programs in order to exploit loop-level parallelism by means of the use of Java built-in multithreading support. This work does not try to deal with compiler optimizations or automatic detection of parallelism. Along these sections, we will assume the existence of some compiler or restructurer that is at least capable of transforming Java programs based on OpenMP-like annotations made by the user in the source code. Since we are not focusing into the restructurer itself, but in the transformations that the Java language does permit, we will not try to enter into discussions about the syntax or semantics of these annotations (for more information see [2, 3]). Although oriented towards code generated by a restructuring compiler, the transformations presented in this paper can be also applied manually.

In this section we describe three different alternatives that could be used to restructure parallel loops written in Java in order to exploit their inherent parallelism. The parallelized loops are substituted with some scheduling code that is in charge of spawning parallelism, providing work to other threads and waiting for the termination of that work. The alternatives presented differ in where the parallelism is spawned and how work is supplied to other threads. Two of them require new packages that provide runtime support to the code generated by the compiler. The three alternatives could be summarized as follows:

Thread-based : Creates instances of a subclass of the Thread class, defined for each loop.

WorkDescriptor-based : Creates instances of a subclass defined for each loop that describes the work to be done (WorkDescriptor), and supplies these instances to previously pre-created instances of a subclass of the Thread class.

ReflectionWorkDescriptor-based : Combines the previous transformation with the use of the Java Reflection package to describe the work to be done and avoid the definition of a new class for each parallel loop.

Each one of these transformations is presented in detail in the following sections. Figure 1 presents the source code for a simple example (with only one parallel loop and using the directives proposed in [3]) that will be used in order to illustrate the transformations.

```
public class Loop {
  public static void
  main (String args[]) {
     // ...  }
  void foo() {
     // omp parallel for private(i)
     // schedule(static)
     for (int i=0;i<100;i++) {
        /* Do some work */
     }
  }
}
```

Figure 1: Source code for a simple example, with JOMP annotations

3.1 Thread-based transformation

The first transformation makes intensive use of threads for executing parallel loops (like [1]). The transformation includes the definition of one subclass of the Thread class for every parallelized loop.

The constructor of that class receives as parameters the information needed to execute the parallel loop. This information may include a reference to the instance where the parallel loop is located (we will call this its "target"), that might be null if the method is a static method. The run method of the new class invokes a concrete method of the target. The method invoked in the target contains the parallelized loop. The loop header is transformed so that each thread executes only in a subset of the whole iteration space, and some auto-scheduling code is added prior to the execution of the loop. The original loop is replaced with code that creates as many instances of the loop associated Thread subclass as indicated by the user by means of some command line arguments (by definition of properties) and waits for the completion of all them. Figure 2 presents the resulting code when this transformation is applied to the original example. The Thread-based transformation replaces the loop with scheduling and joining code in order to create Java Threads, supply work to them, and wait the completion of that work. Some initialization code is also inserted in the Main method of the application. A new method has been created in the sample class. This method contains a modified version of the original loop plus some code that is in charge of the modification of the iteration space of the loop (this step is common to all three transformations). Notice the definition of a new subclass of Thread that is in charge of executing the loop method with the necessary parameter to modify the iteration space: the thread number (assuming a static work distribution scheme). The definition of a new class is mandatory when using the Thread-based transformation or WorkDescriptor-based transformation, since the

```
public class Loop {
   static int NumThreads;
   public static void main (String args[]) {
      String sNumThreads =System.getProperty
                        ("JAVA_MP_THREADS");
      if (sNumThreads!=null)
         NumThreads = new
            Integer(sNumThreads).intValue();
      } else NumThreads = 1;
      //...
   }
   void foo() {
      //scheduling code
      int thNum=NumThreads;
      workerThread_0 threads[]=new
                        workerThread_0[thNum];
      for (int th=0;th<thNum;th++) {
         threads[th]=new
                        workerThread_0(this,th);
         threads[th].start();
      }
      //join code
      for (int th=0;th<thNum;th++) {
         try { threads[th].join();
         } catch (Exception e) {}
      }
    }
    //new code
    void parallelLoop_0 (int me){
       int chunk=(((100)-(0))/NumThreads);
       int rest=((100)-(0))-chunk *
                                NumThreads;
       int down=(0)+chunk*me;
       int up=down+chunk;
       if (me==NumThreads-1) up+=rest;
       for (int i=down;i<up;i++) {
          /* Do some work */
       }
    }
}
class workerThread_0 extends Thread {
   Loop target;
   int me;
   public workerThread_0(Loop t, int m) {
      target = t;
      me = m;
   }
   public void run(){
      target.parallelLoop_0(me);
   }
}
```

Figure 2: Transformed code using Java Threads

only starting point of a Java Thread is the run method, and each parallel loop is encapsulated in a separated function.

There may be different variations on this transformation, but we have tried to present here the simplest one. Some implementations like [1] define additional classes that give a more structured view of the transformation (for example, a class that represents the loop, a class that implements the scheduling policy to divide the iteration space among threads, a class that provides synchronization facilities, and so on). However, the excessive overhead due to the massive creation of objects or the intensive use of synchronized methods may reduce the gain due to the parallel execution itself.

Table 1: Miscellaneous overheads (in milliseconds)

Operation	*SGI*	*Compaq*	*Sun*
Thread Creation	1.790	0.920	2.820
Integer Creation	0.002	0.001	1.7E-4
WorkDescriptor Creation	0.002	0.001	9.5E-4
Reflection Use	0.030	0.020	0.127
Reflection Invoke	0.003	0.001	0.045

This transformation may lead to an undesired high overhead due to the intensive creation of Java threads. In order to support the proposals in the next sections, we first try to quantify the overhead incurred in the creation of a Thread object and compare it with the creation of other kinds of objects. The two first rows in table 1 compare the overhead for Thread and Integer class creations on three different architectures and JVMs. SGI column presents times obtained in a SGI Origin 2000 with MIPS R10000 processors at 250 MHz and JVM version 3.1.1 (Sun Java 1.1.6). The SUN column presents times obtained in a Sun Ultra 170E with a UltraSPARC processor running at 167 MHz, and JVM 1.2.1_03 (Java version 1.2.1). The Compaq column presents times obtained in a Compaq DEC Alpha Server 8400 with Alpha 21264 processors running at 525 MHz, and JVM 1.1.6_03. All JVM where run with Just-in-Time compilation and native threads, and without asynchronous garbage collection.

This overhead, however, depends on the underlying native threads library that is supporting the JVM. The definition of the JVM does not states how Java threads are mapped into kernel entities nor into the JVM threading system, so there is no control, from a Java application, of how Java threads are mapped onto kernel threads. In the worst case, a Java thread creation implies the creation of a kernel thread and, therefore, a large overhead.

3.2 WorkDescriptor-based transformation

The second transformation tries to cope with the overhead due to intensive creation of Thread objects. This objective is approached by the implementation of an application-level work dispatching mechanism. Threads are pre-created and remain alive until they are not needed for any parallel work (Klemm identified the excesive object, and specially Thread, creation as an important source of overhead in [12]). In our case, we create them at the beginning of the application, and they remain alive until the end of the execution. But the creation and destruction points might be moved to some other points, for example, the creation of threads could be moved to the start of a code block that contains lot of parallel loops, and the destruction of the threads could be inserted at the end of that block. This kind of decisions might be made by the user, by a parallelizing compiler or even by the class that implements the application-level work dispatching mechanism, in order to make efficient use of system resources.

The modifications performed on the source program differ from the explained in the previous section. The scheduling code that replaces the parallelized loop is not creating instances of a Thread subclass; instead, the scheduling code

```
public class Loop {
   public static void main (String args[]) {
      LoopThread.initPackage();
      // ...
   }
   void foo() {
      // scheduling code
      LoopThread.supplyGlobalWork(new
                     workDescriptor_0(this));
      // join code
      LoopThread.joinGlobalWork();
   }
   //new code
   void parallelLoop_0 (int me){
      int chunk=(((100)-(0)) /
                  LoopThread.threadsTeam());
      int rest=((100)-(0))-chunk *
                   LoopThread.threadsTeam();
      int down=(0)+chunk*me;
      int up=down+chunk;
      if (me==LoopThread.threadsTeam()-1)
         up+=rest;
      for (int i=down;i<up;i++) {
         /* Do some work */
      }
   }
}
class workDescriptor_0 extends
                              workDescriptor{
   Loop target;
   public workDescriptor_0(Loop t) {
      target = t;
   }
   public void run(int me){
      target.parallelLoop_0(me);
   }
}
```

Figure 3: Transformed code using Work Descriptors

creates instances of a class that acts as a work descriptor. There is one work descriptor class for each parallelized loop. Every one of those subclasses is descendant of an abstract class that defines a constructor and a run method. Actually, this approach splits the transformation described in the previous section into two parts: the creation of the threads themselves and the supply of work to them. Figure 3 presents the resulting code when this transformation is applied to the original example. This transformation does also modify the Main method of the application, inserting a call to an static method of the LoopThread class. This class spawns as many threads as specified by the user's command line parameters, and set them to start looking for work. The original loop is also replaced with code for creating work, scheduling it to slave Threads and waiting the completion of that work. There is also defined a new class that is in charge of executing the method that encapsulates the modified loop body.

3.2.1 The LoopThread class

The LoopThread class is the class that we have developed to implement the application-level work dispatching mechanism. It is a very simple example of a class that provides the basic operations to spawn threads (initPackage), to distribute work among them, either globally or individually (supplyWork, supplyGlobalWork), and to wait the completion of that work (joinWork, joinGlobalWork). The package also offers an additional service to ask for the number of threads that are taking part in the execution of the parallel loop (threadsTeam). Notice that this is a very simple class utilized as an example, and that has some limitations (for example, only one level of parallelism can be spawned, synchronization is done by busy-waiting mechanisms, among others).

The run method of the LoopThread class is an infinite loop that looks for work by calling the doWork method. Instances of the LoopThread class are marked as daemons as they are created, in order to point to the JVM that it must not wait for the completion of these threads.

Notice that there exists the possibility of supplying the same WorkDescriptor to all the threads. The code we have shown in figure 3 makes use of that facility. This is of importance in the case of loop-level parallelism, because, in the assumption of N slave threads, we only have to create a WorkDescriptor and supply it to all the Threads, avoiding the creation of N-1 WorkDescriptors. Other work distribution schemes may need the individual supply of work using different WorkDescriptors for every of them.

3.2.2 The WorkDescriptor classes

The basic WorkDescriptor class is an abstract class. The transformation defines one subclass of the WorkDescriptor class for each loop being parallelized. These subclasses define a run method that only performs a call to the method that contains the transformed loop in the target (the instance or class where the original parallel loop was located).

The main differences between this transformation and the previous one are:

- Only one object (WorkDescriptor) is created for each loop, and can supply work to as many threads as needed.
- The created object is not a Thread, and its creation is faster than the creation of a Thread object (as shown in the third row in Table 1).

3.3 ReflectionWorkDescriptor-based transformation

The last transformation we consider makes use of the *reflect* package, that provides classes and interfaces for obtaining reflective information about Java classes and objects.

The two previous transformations enforce the definition of a new class for each parallelized loop. This new class can have as its ancestor either the Thread class or the WorkDescriptor class. This is because the only starting point for a Java Thread is the run method of its target object or the run method of the object itself if it is an instance of a subclass of the Thread class [10]. Other languages, such as C, allow us to access the address of a function, and make use of that address to invoke it, but this is not possible in Java.

The java.lang.reflect package, however, makes us able to adopt a similar approach. This package can be used in order to obtain an object that represents a method of a given class, and to invoke it. With this mechanism in our hands, we can

```
import java.lang.reflect.Method;

public class Loop {
   public static void main (String args[]) {
      ReflectionLoopThread.initPackage();
      // ...
   }
   void foo() {
      Class formalArgs[] = {int.class};
      Object o[] = {null};
      try {
      // scheduling code
      Method m=Loop.class.getDeclaredMethod
              ("parallelLoop_0",formalArgs);
      ReflectionLoopThread.supplyGlobalWork(
               new ReflectionWorkDescriptor(
                                 this,m,o));
      // join code
      ReflectionLoopThread.joinGlobalWork();
      } catch (Exception e) {
      System.err.println(e);
      System.exit(-1);
      }
   }
   //new code
   void parallelLoop_0 (int me){
      int chunk=(((100)-(0))/
        ReflectionLoopThread.threadsTeam());
      int rest=((100)-(0))-chunk *
         ReflectionLoopThread.threadsTeam();
      int down=(0)+chunk*me;
      int up=down+chunk;
      if (me==
       ReflectionLoopThread.threadsTeam()-1)
         up+=rest;
      for (int i=down;i<up;i++) {
         /* Do some work */
      }
  }
}
```

Figure 4: Transformed code using Reflection

apply a different transformation to our Java programs, in order to avoid the definition of a new class for each parallelized loop, and thus avoid the associated overhead

As in the Descriptor-based transformation, this one also makes use of a user-level work dispatching mechanism, defined in the ReflectionLoopThread class, that is quite similar to the one used for the work-descriptor transformations, but it makes use of a ReflectionWorkDescriptor class instead of a WorkDescriptor class.

The reflection-based transformation does not need to define an additional WorkDescriptor class for each parallelized loop, since the general description of all methods that encapsulate a parallelized loop can be expressed with Reflection as a single work-descriptor that contains a target Object, a Method to invoke in that object, and a vector of arguments. So, the number of classes defined by the parallel application remains constant independently of the number of parallelized loops. Figure 4 presents the resulting code when this transformation is applied to the original example. This last transformation, does not need to define any new class. The Main method of the application is also modified in order to insert a call to the ReflectionLoopThread class initialization code, that works pretty much like in the LoopThread case. The transformation includes, again, the definition of a new method that encapsulates the modified loop body, which has been replaced with scheduling/join code. This scheduling code makes use of the java.lang.reflect package to obtain information about the method that encapsulates the corresponding loop in order to fill a generic work descriptor that is supplied to the slave threads. Notice that there is no need to define a new class for each thread, since java.lang.reflect gives also to ReflectionLoopThread the capability of invoking the loop method by the use of the "invoke" method on the reflective information that represents the method that contains the loop (an instance of the Method class).

4. EXPERIMENTS

In this section, we evaluate the performance of the described transformations on three Java programs:

- LUAppl: kernel that performs an LU reduction over a matrix of 512x512 double precision numbers.
- Diamond: This is a synthetic benchmark that iterates 800 times over a single parallelized loop that performs one million of multiplications.
- Stress: This is also a synthetic benchmark that contains different parallel loops that perform one million of square roots each one. We run experiments with number of loops varying between 40 and 256.

These programs have been specifically prepared to evaluate the behavior of the code transformations proposed. However, their structure reflects in some way the structure of the parallel computation found in numerical applications. All the results presented here were obtained in the SGI system described in section 3.1. The speed-up is calculated relative to the sequential version.

In the following performance plots "JTH" corresponds to the Thread-based transformation, "WD" corresponds to the WorkDescriptor-based transformation and "ReflectionWD" corresponds to the ReflectionWorkDescriptor-based transformation.

4.1 LUAppl

Figure 5 shows the speed-up obtained for the LUAppl kernel. For this kernel, the use of threads in the "JTH" transformation reduces the execution time as the number of Threads utilized increases. However, transformations "WD" and "ReflectionWD" produce better results due to a considerable reduction of the overhead for spawning parallelism. On the average, "WD" improves the execution time by 32% (48% when 8 threads are used), and "ReflectionWD" can also do it by 37% (49% when 8 threads are used).

4.2 Diamond

Figure 6 shows the speed-up obtained for the Diamond benchmark. This graph is quite similar to the one shown for the LUAppl kernel (actually, the structure of both benchmarks is quite similar). We can observe again how the two Descriptor-oriented transformations work better than the

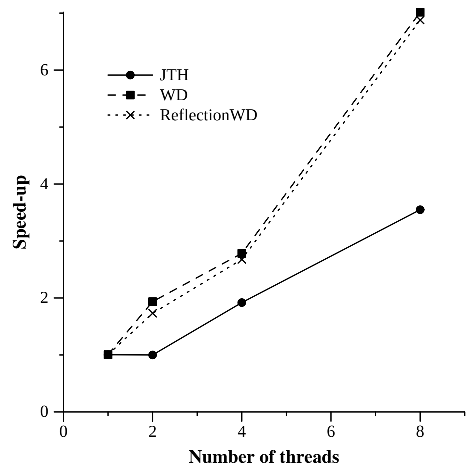

Figure 5: Speed-up for the LU kernel (512x512 matrix)

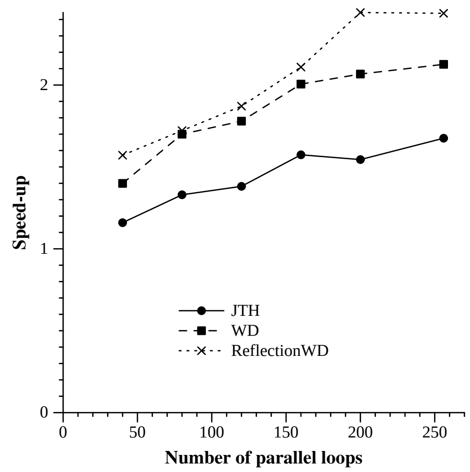

Figure 7: Speed-up for the Stress benchmark (4 Threads)

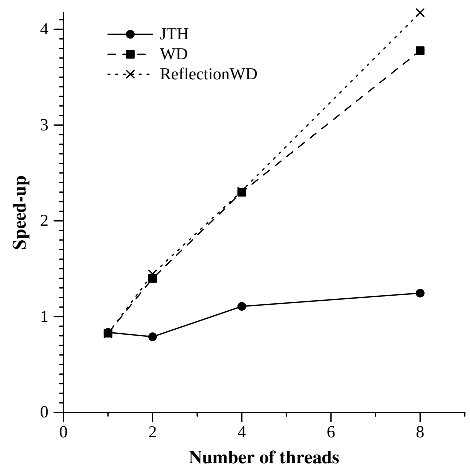

Figure 6: Speed-up for the Diamond benchmark

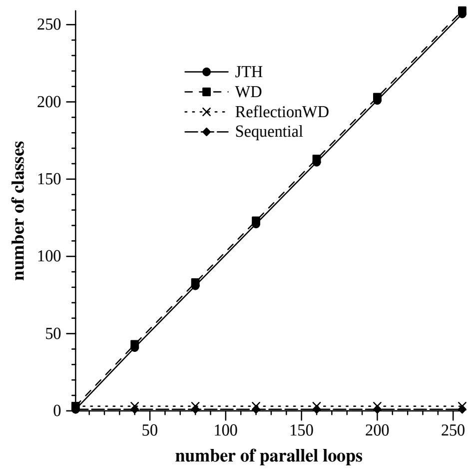

Figure 8: Number of classes needed depending on the number of parallel loops

Thread-oriented transformation. The use of "WD" and "ReflectionWD" reduces the execution time by 40% and 51% on average (68% and 70% for the 8 threads case), respectively.

4.3 Stress

Figure 7 presents the speed-up for Stress. In this plot the number of threads is fixed to 4, and the number of parallel loops in the application goes from 40 to 256. Notice that both "WD" and "ReflectionWD" outperform "JTH". In particular, "WD" reduces the execution time by 11% on average (21% with 8 threads), and "ReflectionWD" reduces the execution time by 21% on average (31% with 8 threads).

Notice that as the number of parallel loops increases the difference between "WD" and "ReflectionWD" become noticeable (for 256 loops, the difference between them is 10%). This effect is because of the fact of the definition of a new class for each parallel loop. In the case of use of the "WD" transformation we are speeding-up the creation of work, since we are not creating Thread instances, but we cannot avoid the loading of the class that represents the WorkDescriptor corresponding to that parallel loop. This class-loading is done in the critical path of the application and, as can be deduced from the graphs, influences the execution time of the application. The same may apply to JIT engines: if these engines compile all methods the first time they are executed, then they are compiling code that will never be reused, and they are enforced to compile new code for each parallel loop. The "ReflectionWD" transformation does not imply a class-load for every parallel loop, since the number of different classes needed to execute the application remain constant independently of the number of parallel loops (this transformation makes use of the same class for every one of them). Figure 8 illustrates that fact.

5. RUNTIME POLICIES AND MULTITHREADING PERFORMANCE

The results shown in the previous section expose a large performance improvement between the basic transforma-

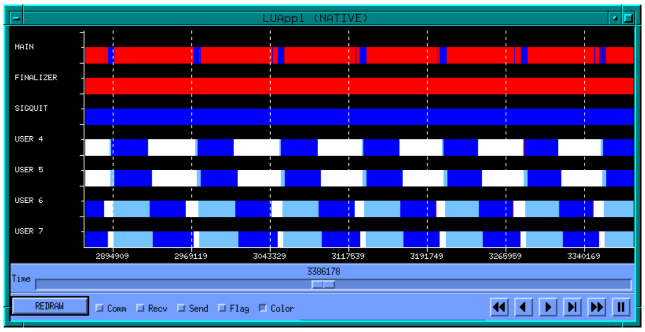

Figure 9: Visualization of the LUAppl running with the Java Threads transformation (4 threads)

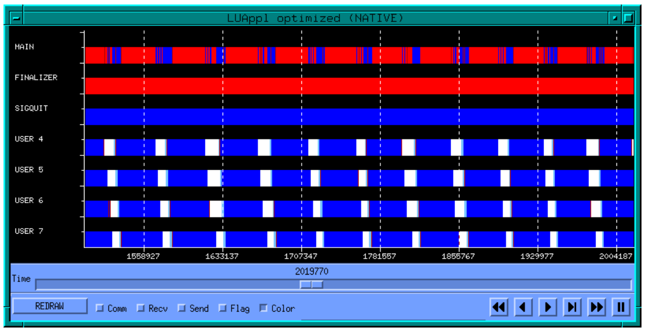

Figure 10: Visualization of the LUAppl running with the Java Threads transformation (4 threads) and the pthread_setconcurrency service

tion (using Java Threads) and the two advanced transformations (WorkDescriptor and ReflectionWorkDescriptors). However, these results show performance gains due to the use of the two later transformations, but also due to a better behaviour of the underlying thread system indirectly incurred by the transformations themselves.

In order to analyze these effects and discover performance bottlenecks, the behaviour of the LU kernel is studied using JIS [8]. JIS is an instrumentation framework for Java programs based on the DITools [17] code interposition tool and the Paraver [13] trace visualization and analysis tool.

5.1 LU behaviour

Figure 5 reports an speed–up for the "JTH" transformation close to 2 and 3 when 4 and 8 threads are used, respectively. Figure 9 shows a Paraver window in which the behaviour 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 `public static void main` method), FINALIZER and SIGQUIT are two JVM internal threads, and USER4 to USER7 are slave threads created by the MAIN thread, as result of the "JTH" code transformations. Each thread evolves through a set of states (INIT, RUNNING, BLOCKED and STOPPED). For example, light blue in the trace reflects that the thread is being created, dark blue reflects that the thread is running, red indicates that the thread is blocked and white indicates that the thread has finished.

As can be deduced from the graphical representation, the number of threads with dark blue color (RUNNING state) at a given time gives us the parallelism level achieved by the application. So notice that, although four slave Java Threads are created for each loop, only two of them are running simultaneously. This is due to the fact that the multithreading runtime system used (pthreads in SGI's JVM) is only providing two virtual processors (kernel threads) to support the execution of the four slave Java Threads. This explains the poor performance gains in the LU application.

The observations obtained from the LUAppl instrumentation were utilized to perform some modifications in the behaviour of the JVM and its interface with the multithreading runtime. By default, the threads library adjust the level of concurrency itself as the application runs. We made use of JIS in order to give the library a hint about the concurrency level needed by the application. With the use of JIS, we automatically insert a call to the `pthread_setconcurrency (int level)` service of the threads library. Argument `level` is used to inform about the ideal number of kernel threads needed to schedule the available Java threads. Figure 10 shows the execution trace after seting the `level` value to the maximum parallelism degree of the application. Notice that, in this execution, 4 pthreads and kernel threads are used to schedule the 4 slave Java threads, with the consequent performance improvement. This results in a reduction of the execution time close to 50%. Table 2 shows the execution time for different problem sizes.

Problem Size	*Original*	*Set_concurrency*
64X64	916	715 (22 %)
128X128	4473	2813 (37 %)
256X256	53319	17652 (66 %)
512X512	215525	110128 (49 %)

Table 2: Execution time (milliseconds) for the LUAppl using JTH and after setting pthread_setconcurrency to 4

Figure 11 shows the speed–up achieved in the execution of the LU application (size 512x512) for different numbers of threads when using the JTH and WD code transformations, after setting pthread_setconcurrency to the number of threads. Compared to Figure 5, notice that the `pthread_setconcurrency` call improves the behaviour in the two versions. However, the improvement is more significative in the JTH version due to the inability of the multithreading runtime system to determine the required number of kernel threads for this code transformation. The WD code transformation creates the Java threads at the beginning and therefore gives more chances to the multithreading runtime system to determine this number.

5.2 Application–runtime communication

As can be deduced from the previous results, a good cooperation between the application and the multithreading

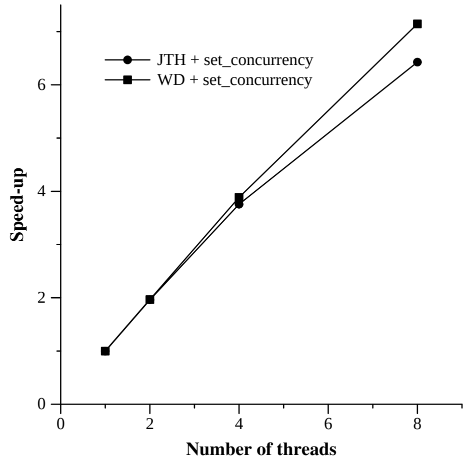

Figure 11: Speed-up for the LU kernel (512x512 matrix) when using the JTH and WD code transformations and after setting pthread_setconcurrency to the number of threads

runtime could speedup the application execution time. But the Java specification does not considers the interaction between the runtime and the application. For instance, the application is not able to specify, for example, the concurrency level or force a specific mapping of the Java threads into kernel threads.

This observation drive us to propose new extensions to the Java API in order to provide these services. These modifications includes, among others:

- `System.setConcurrency(int value)` method to set the concurrency level of the application.
- `System.getMPConfig()` method in order to inform the application about the underlying architecture: number of nodes and processing elements per node, latencies in NUMA/UMA memory organizations, ...
- KernelThread class including, for instance, services to control the binding of Java threads to kernel threads (`KernelThread.bind( Thread t)` method).

These proposals and their implementation are subject of our current research.

6. CONCLUSIONS AND FUTURE WORK

In this paper, we have presented an overview of some transformations available to efficiently exploit loop-level parallelism of Java applications running on shared-memory multiprocessors. We have analyzed three different transformations that might be applied by a restructuring compiler in order to exploit that parallelism based either on:

1. Intensive use of Thread creation for each parallel loop.
2. Conservative use of Thread creation combined with the creation of an object that describes work to be done for each parallel loop.
3. Conservative use of Thread creation combined with the creation of an object that describes work to be done for each parallel loop, and avoiding the definition of one class for each parallel loop by means of the utilization of the java.lang.reflect package.

The proposed transformations are evaluated using a set of synthetic applications. We have concluded that the use of the two latter transformations (i.e. avoiding the massive creation of Java Threads for each parallel loop) outperforms the performance obtained by the utilization of the first one. The evaluation includes a comparison taking into account the number of classes utilized by each transformation. We conclude that the "ReflectionWD" transformation can reduce the overhead introduced by the need of class-loading (and possible Just-in-time compilation) for each parallel loop in the two former transformations (JTH, WD), and it reduces the size of the resulting bytecodes.

Finally we have explored some possible enhancements to the Java threading API (such as the ability to give hints about concurrency level to the runtime system) can improve the performance of parallel applications. As a future work, we will further investigate how to improve Java support for Threads, and how to give users more control on how application threads map into kernel threads. At the moment, users have to blindly rely in the run-time libraries that give multithreading support to the JVM. Its our thought that, currently, the JVM hides too much information to the user and does not permit a powerful user-level scheduling (for example, a user cannot decide where a Java thread is going to run, and the Java API does not have any standard mechanism, for example, to expose to the application the underlying architecture). These decisions ease application development, but it may reduce the performance that can be obtained in certain kind of applications.

7. 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 and M. E. Kambites. Jomp – an openmp-like interface for java. In *Proceedings of the 2000 ACM Java Grande Conference*. ACM, june 2000.

[4] B. Carpenter, Y. Chang, G. Fox, D. Leskiw, and X. Li. Experiments with hp java. *Concurrency, Practice and Experience*, 9(6):633–648, June 1997.

[5] 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.

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

[7] MPI Forum. Document for a standar message passing interface. *University of Tenesse Technical Report CS-93-214*, November 1993.

[8] J. Guitart, J. Torres, E. Ayguadé, J. Oliver, and J. Labarta. Java instrumentation suite: Accurate analysis of java threaded applications. In *Proceedings of the Second Annual Workshop on Java for High-Performance Computing, ICS'00*, May 2000.

[9] Sun Microsystems Inc. Rmi specification. *http://java.sun.com/products/jdk/1.2/docs/guide/rmi/*.

[10] B. Joy, J. Gosling, and G. Steele. *The Java Language Specification*. Addison-Wesley, 1996.

[11] 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.

[12] R. Klemm. Practical guideline for boosting java server performance. In *Proceedings of the 1999 ACM Java Grande Conference*, 1999.

[13] J. Labarta, S. Girona, V. Pillet, T. Cortes, and L. Gregoris. Dip: A parallel program development environment. In *Proceedings of 2th International Euro-Par Conference*, August 1996.

[14] C. Nester, M. Philippsen, and B. Haumacher. A more efficient rmi for java. In *Proceedings of the 1999 ACM Java Grande Conference*, 1999.

[15] M. Philippsen and M. Zenger. Javaparty – transparent remote objects in java. *Concurrency, Practice and Experience*, 9(11):1225–1242, November 1997.

[16] R. R. Raje, J. I. Williams, and M. Boyles. Asynchronous remote method invocation (armi) mechanism for java. *Concurrency, Practice and Experience*, 9(11):1207–1211, November 1997.

[17] A. Serra, N. Navarro, and T. Cortes. Ditools: Application-level suport for dynamic extensions and flexible composition. In *Proceedings of the USENIX Annual Technical Conference*, June 2000.

[18] V. S. Sunderam. Pvm: A framework for parallel distributed computing. *Concurrency, Practice and Experience*, 2(4), December 1990.

[19] K. van Reeuwijk, A. J. C. van Gemund, and H. J. Sips. Spar: A programming language for semi-automatic compilation of parallel programs. *Concurrency, Practice and Experience*, 9(11):1193–1205, November 1997.

[20] K. Yelick, L. Semenzato, G. Pike, C. Miyamoto, B. Liblit, A. Krishnamurthy, P. Hilfinger, S. Graham, D. Gay, P. Colella, and A. Aiken. Titanium: a high-performance java dialect. In *Proceedings of the ACM 1998 Workshop on Java for High-Performance Network Computing*, September 1998.