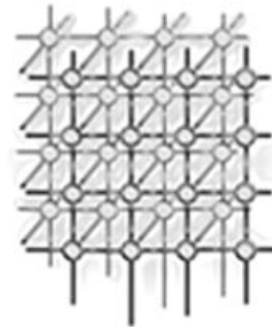


Strategies for the efficient exploitation of loop-level parallelism in Java[‡]



José Oliver¹, Jordi Guitart², Eduard Ayguadé^{2,*},
Nacho Navarro² and Jordi Torres²

¹*iSOCO, Intelligent Software Components, S.A.*

²*Computer Architecture Department, Technical University of Catalunya, Barcelona, Spain*

SUMMARY

This paper analyzes the overheads incurred in the exploitation of loop-level parallelism using Java Threads and proposes some code transformations that minimize them. The transformations avoid the intensive use of Java Threads and reduce the number of classes used to specify the parallelism in the application (which reduces the time for class loading). The use of such transformations results in promising performance gains that may encourage the use of Java for exploiting loop-level parallelism in the framework of OpenMP. On average, the execution time for our synthetic benchmarks is reduced by 50% from the simplest transformation when eight threads are used. The paper explores some possible enhancements to the Java threading API oriented towards improving the application–runtime interaction. Copyright © 2001 John Wiley & Sons, Ltd.

KEY WORDS: loop-level parallelism; Java Threads; program transformations

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

*Correspondence to: Eduard Ayguadé, Centre Europeu de Paral·lelisme de Barcelona (CEPBA), Dept. d'Arquitectura de Computadors, Univ. Politècnica de Catalunya, Jordi Cirona 1-3, Modul D6, Barcelona 08034, Spain.

†E-mail: eduard@ac.upc.es

‡Preliminary versions of the material presented in this paper appeared in the proceedings of the ACM 2000 Java Grande Conference and The Second Workshop on Java for High Performance Computing (ICS'2000).

Contract/grant sponsor: Ministry of Education of Spain (CICYT); contract/grant number: TIC 98-0511



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, when using Java for scientific parallel programming one is faced with the large overheads caused by the interpretation of the bytecodes, that leads 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 may easily offset the gain due to the parallel execution. Other issues that should be considered include the lack of support for complex numbers and multidimensional arrays.

A lack of suitable standards for parallel programming in Java is also a concern. The emerging OpenMP standard for Fortran and C/C++ has lead to the proposal of a similar paradigm in the scope of Java (JOMP [1]). Although it is, of course, possible to write shared memory parallel programs using Java's native threads model, it is clear that a directive-based system (as in OpenMP) has a number of advantages over the native threads approach.

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 could be applied by an OpenMP compiler for Java in order to efficiently exploit this parallelism.

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 (which reduces the time for class loading). After the experimental evaluation of the proposed transformations, we analyze the behavior of the threaded execution on a target machine. This analysis provides some hints on how to modify the behavior of the multithreaded runtime, which results in significant performance gains. These results will probably end up in the proposal of API modifications and extensions.

The document is structured as follows: Section 2 presents some related work. In Section 3 we describe three different techniques that could be used by the compiler to exploit loop-level parallelism in Java. Section 4 evaluates the transformations. Section 5 explores some possible enhancements to the Java threading API that may provide some kind of interaction between the application and the threading layer of the system. Finally, Section 6 concludes the paper.

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 [2,3] are based on the implementation of common message-passing standards, such as PVM or MPI [4,5] by means of Java classes that, in turn, make use of Java communication classes [6] or some modified version of them [7–9]. These ideas and proposals are oriented to distributed processing, and do not attempt to deal with shared-memory parallelism.

There are also a number of proposals for making Java a data-parallel language, such as HPJAVA, TITANIUM or SPAR [10–13], 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 multidimensional 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.* [14] 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 [1,14]). Although oriented towards code generated by a restructuring compiler, the transformations presented in this paper can also be 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. This strategy is similar to the one suggested in [14].

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 [1]) 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 [14]). 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 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 [14] 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



```
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.



Table I. Overheads (in milliseconds).

Operation	SGI	Compaq	Sun
Thread creation	1.790	0.920	2.820
Integer creation	0.002	0.001	1.7×10^{-4}
WorkDescriptor creation	0.002	0.001	9.5×10^{-4}
Reflection use	0.030	0.020	0.127
Reflection invoke	0.003	0.001	0.045

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.

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 following 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. Table I shows the overhead for some basic actions considered in this paper. The first two rows in Table I compare the overhead for Thread and Integer class creations on three different architectures and JVMs. The other rows will be described later. The 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 were run with Just-in-Time compilation and native threads, and without asynchronous garbage collection. Notice that the overhead for creating a thread is several orders of magnitude larger than the overhead of creating an integer object.

This overhead, however, depends on the underlying native threads library that is supporting the JVM. The definition of the JVM does not state 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 ([15] identified the excessive object, and specially thread creation as an important source of overhead). 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. These kinds 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 those 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 creates instances of a class that acts as a work descriptor. There is one work descriptor class for each parallelized loop. Every one of these subclasses is a 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 also modifies the `Main` method of the application, inserting a call to a static method of the `LoopThread` class. This class spawns as many threads as specified by the user's command line parameters, and sets 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. A new class is also defined 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 it 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 this 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 one 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:



```

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 WorkDescriptors.

- only one object (WorkDescriptor) is created for each loop, and it 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. The third row in Table I shows the overhead incurred in the creation of a WorkDescriptor, which includes the creation and supply (what is identified by the `scheduling code` comment in Figure 3).



```
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.

3.3. ReflectionWorkDescriptor-based transformation

The last transformation we consider makes use of the *reflect* package, which provides classes and interfaces to obtain 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 [16]. 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, enables us 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 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, it is quite similar to that used for the `WorkDescriptor` 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 `WorkDescriptor` 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` also gives 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).

The last two rows in Table I show the overheads associated with the use and invocation of Reflection. The *Reflection use* overhead includes the creation of the work descriptor and supply (i.e. all the code below the `scheduling` code comment in Figure 4). The *Reflection invoke* is the additional overhead incurred in the library due to the invocation.

4. EXPERIMENTS

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

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

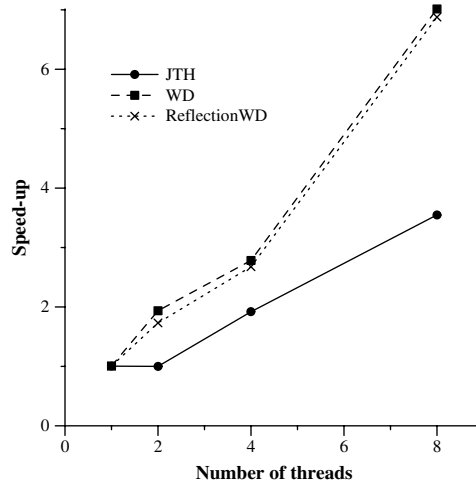


Figure 5. Speed-up for the LU kernel (512×512 matrix).

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 average, 'WD' improves the execution time by 32% (48% when eight threads are used), and 'ReflectionWD' improves the execution time by 37% (49% when eight 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 Thread-oriented

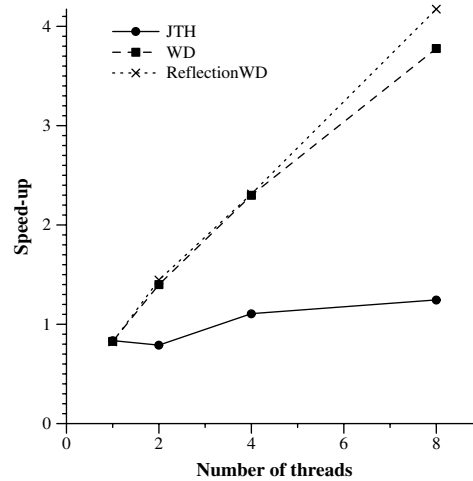


Figure 6. Speed-up for the Diamond benchmark.

transformation. The use of 'WD' and 'ReflectionWD' reduces the execution time by 40% and 51% on average (68% and 70% for the eight threads case), respectively.

4.3. Stress

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

As the number of parallel loops increases, the difference between 'WD' and 'ReflectionWD' becomes noticeable (for 256 loops, the difference between them is 10%). This is due to the definition of a new class for each parallel loop. When the 'WD' transformation is used, the overhead due to work creation is reduced; however, 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 remains constant independently of the number of parallel loops (this transformation makes use of the same class for all the loops). Figure 8 illustrates this fact.

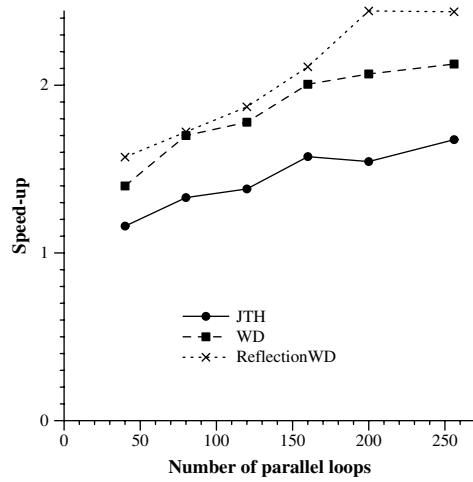


Figure 7. Speed-up for the Stress benchmark (four threads).

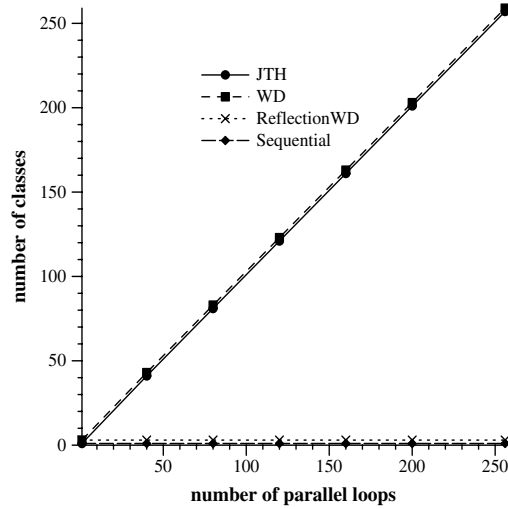


Figure 8. Number of classes needed depending on the number of parallel loops.

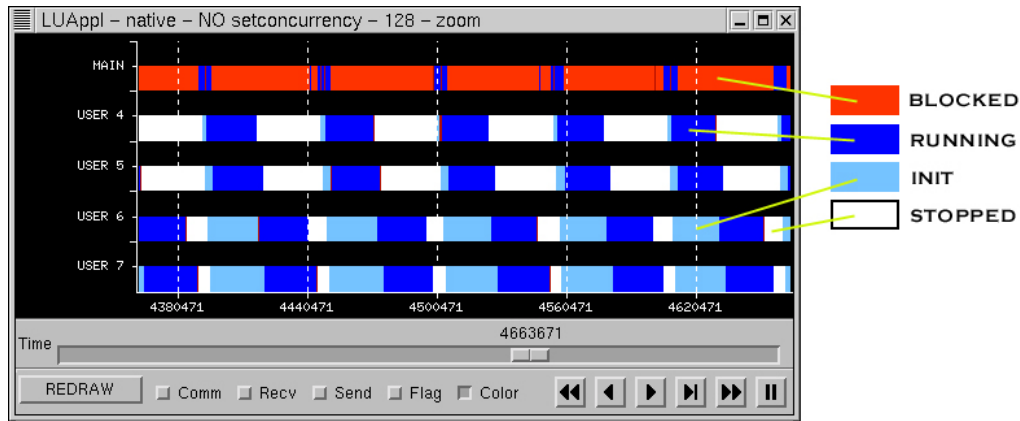


Figure 9. Visualization of the LUAppl running with the Java Threads transformation (four threads).

5. RUNTIME POLICIES AND MULTITHREADING PERFORMANCE

The results shown in the previous section expose a large performance improvement between the basic transformation (using Java Threads) and the two advanced transformations (WorkDescriptor and ReflectionWorkDescriptors). However, these results show performance gains due to the use of the latter two transformations, but also due to a better behavior of the underlying thread system indirectly incurred by the transformations themselves.

In order to analyze these effects and discover performance bottlenecks, the behavior of the LU kernel is studied using JIS [17]. JIS is an instrumentation framework for Java programs based on the DITTOOLS [18] code interposition tool and the Paraver [19] trace visualization and analysis tool.

5.1. LU behavior

Figure 5 reports a speed-up for the 'JTH' transformation close to of 2 and 3 when four and eight threads are used, respectively. Figure 9 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 `public static void main` method), 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

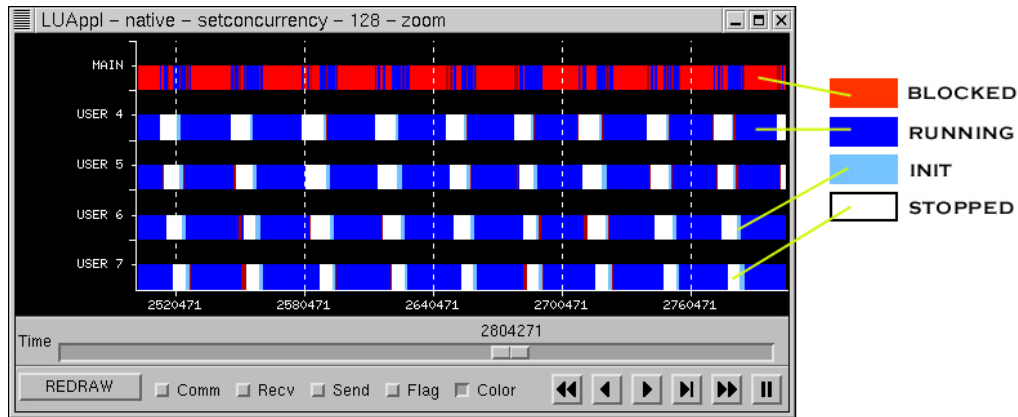


Figure 10. Visualization of the LUAppl running with the Java Threads transformation (four threads) and the `pthread_setconcurrency` service.

simultaneously. This is due to the fact that the multithreading runtime system used (pthreads in SGI's JVM) only provides 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 behavior of the JVM and its interface with the multithreading runtime. By default, the threads library adjusts 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 setting the `level` value to the maximum parallelism degree of the application. Notice that, in this execution, four pthreads and kernel threads are used to schedule the four slave Java Threads, with the consequent performance improvement. This results in a reduction of the execution time close to 50%. Table II shows the execution time for different problem sizes.

Figure 11 shows the speed-up achieved in the execution of the LU application (size 512×512) 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 behavior in the two versions. However, the improvement is more significant 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.



Table II. Execution time (in milliseconds) for the LUAppl using JTH and after setting `pthread_setconcurrency` to 4.

Problem size	Original	Set_concurrency
64 × 64	916	715 (22%)
128 × 128	4473	2813 (37%)
256 × 256	53 319	17 652 (66%)
512 × 512	215 525	110 128 (49%)

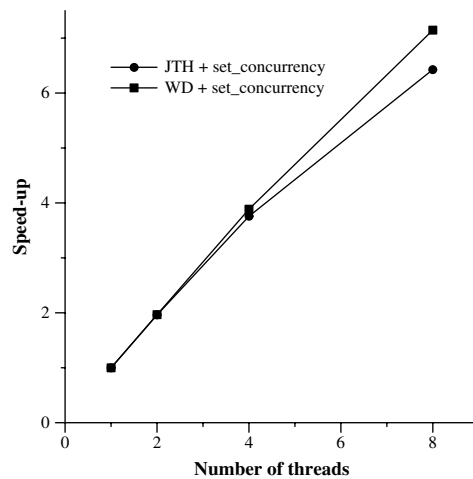


Figure 11. Speed-up for the LU kernel (512×512 matrix) when using the JTH and WD code transformations and after setting `pthread_setconcurrency` to the number of threads.

5.2. Application–runtime communication

As can be deduced from the previous results, a good cooperation between the application and the multithreading runtime could speed-up the application execution time. However, the Java specification does not consider 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:

- the `System.setConcurrency(int value)` method to set the concurrency level of the application;



- the `System.getMPCConfig()` 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, etc.;
- the `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 the 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. This reduces the time needed for class loading, thus improving the final performance.

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 for 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 foresee some possible enhancements to the Java threading API to improve the performance of parallel applications. For instance, the possibility of providing hints to the runtime system about the concurrency level of the application. 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 on the runtime libraries that give multithreading support to the JVM. It is our thought that, currently, the JVM hides too much information from 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 kinds of applications.

ACKNOWLEDGEMENT

This work has been supported by the Ministry of Education of Spain (CICYT) under contract TIC 98-0511.



REFERENCES

1. Bull JM, Kambites ME. JOMP—An OpenMP-like interface for Java. *Proceedings of the 2000 ACM Java Grande Conference*. ACM, 2000.
2. Judd G, Clement M, Snell Q, Getov V. Design issues for efficient implementation of MPI in Java. *Proceedings of the 1999 ACM Java Grande Conference*, 1999.
3. Ferrari A. JPVM network parallel computing in Java. *Proceedings of the 1998 ACM Workshop on Java for High-Performance Network Computing*, March 1998.
4. MPI Forum. Document for a standard message passing interface. *University of Tennessee Technical Report CS-93-214*, November 1993.
5. Sunderam SV. Pvm: A framework for parallel distributed computing. *Concurrency, Practice and Experience* 1990; **2**(4):315–339.
6. Sun Microsystems Inc. RMI specification. <http://java.sun.com/products/jdk/1.2/docs/guide/rmi/>.
7. Nester C, Philippsen M, Haumacher B. A more efficient RMI for Java. *Proceedings of the 1999 ACM Java Grande Conference*, 1999.
8. Philippsen M, Zenger M. Javaparty—Transparent remote objects in Java. *Concurrency, Practice and Experience* 1997; **9**(11):1225–1242.
9. Raje RR, Williams JI, Boyles M. Asynchronous remote method invocation (ARMI) mechanism for Java. *Concurrency, Practice and Experience* 1997; **9**(11):1207–1211.
10. Carpenter B, Chang Y, Fox G, Leskiw D, Li X. Experiments with HPJava. *Concurrency, Practice and Experience* 1997; **9**(6):633–648.
11. Carpenter B, Zhang G, Fox G, Li X, Wen Y. HPJava: Data parallel extensions to Java. *Concurrency, Practice and Experience* 1998; **10**(11-13):873–877.
12. van Reeuwijk K, van Gemund AJC, Sips HJ. SPAR: A programming language for semi-automatic compilation of parallel programs. *Concurrency, Practice and Experience* 1997; **9**(11):1193–1205.
13. Yelick K, Semenzato L, Pike G, Miyamoto C, Liblit B, Krishnamurthy A, Hilfinger P, Graham S, Gay D, Colella P, Aiken A. Titanium: A high-performance Java dialect. *Proceedings of the ACM 1998 Workshop on Java for High-Performance Network Computing*, September 1998.
14. Bik AJC, Villancis JE, Gannon DB. Javar: A prototype Java restructuring compiler. *Concurrency, Practice and Experience* 1997; **9**(11):1181–1191.
15. Klemm R. Practical guideline for boosting Java server performance. *Proceedings of the 1999 ACM Java Grande Conference*, 1999.
16. Joy B, Gosling J, Steele G. *The Java Language Specification*. Addison-Wesley, 1996.
17. Guitart J, Torres J, Ayguadé E, Oliver J, Labarta J. Java instrumentation suite: Accurate analysis of Java threaded applications. *Proceedings of the Second Annual Workshop on Java for High-Performance Computing, ICS'00*, May 2000.
18. Serra A, Navarro N, Cortes T. Ditoools: Application-level support for dynamic extensions and flexible composition. *Proceedings of the USENIX Annual Technical Conference*, June 2000.
19. Labarta J, Girona S, Pillet V, Cortes T, Gregoris L. DiP: A parallel program development environment. *Proceedings of 2nd International Euro-Par Conference*, August 1996.