Semi-Automatic Parallelization of Java Applications

Pascal A. Felber

Institut EURECOM
06904 Sophia Antipolis, France
felber@eurecom.fr

Abstract. *Some types of time-consuming computations are naturally parallelizable. To take advantage of parallel processing, however, applications must be explicitly programmed to use specific libraries that share the workload among multiple (generally distributed) processors. In this paper, we present a set of Java tools that allow us to parallelize some types of computationally-intensive Java applications* a posteriori*, even when the source code of these applications is not available. Our tools operate using techniques based on bytecode transformation, code migration, and distributed parallel method executions.*

1 Introduction

Motivations. The Java language is widely considered as inadequate for computationally-intensive tasks. The obvious reason lies in the "poor" performance of Java programs, which run significantly slower than their C or Fortran counterparts. There are a number of reasons, however, why Java may be used for such applications.

First, Java is easy to learn, safe, and scalable to complex programming problems. Its popularity and wide adoption have attracted significant interest from the scientific and engineering community and led to the development of tools and libraries adapted to high performance and parallel computing [1–3].

More importantly, Java's processor and operating system independence make it possible to deploy distributed parallel applications on heterogeneous platforms and harness the processing power of idle workstations in the Internet. This has the potential to extend the reach of parallel distributed applications far beyond specialized clusters of homogeneous machines traditionally used for high-performance computing.

Finally, Just-In-Time (JIT) compilers that translate Java bytecode into native instructions have made significant advances to improve performance, and modern JIT compilers have been estimated to reach up to two thirds of the speed of C code [4]. IBM's Ninja project has also demonstrated that, when compiling Java applications specifically for parallel architectures, one can achieve between 80 and 100% of the performance of highly optimized Fortran code [5]. Combined with quasi-static techniques [6, 7], Java code can be as efficient as C or Fortran code.

Overview and Contributions. The goal of the work presented in this paper is to provide mechanisms to seamlessly parallelize some kinds of Java applications and execute them on distributed processors, without requiring the application programmer to explicitly use dedicated message-passing libraries. These mechanisms can be applied to code that has not been programmed with parallelization in mind and whose source is not available. Parallelism is implemented at the coarse level of method invocations, by transforming a computationally-intensive operation into a set of shorter equivalent operations executed on multiple machines. Transformations are performed according to simple "rewriting" rules specified by the application deployer.

Our parallelization mechanisms consist of two major components: (1) a wrapper generator that instruments Java bytecode at load time and effectively wraps selected methods with user-specifier filters; and (2) a parallelization engine that instantiates application classes on multiple processes, dispatches method invocations to these processes, and finally collects and aggregate replies. Although the program does not need to be modified, some rewriting rules need to be specified by the application deployer. Parallelization is therefore transparent to the application, but not completely automatic (hence *semi-automatic*).

We would like to emphasize that our techniques can only be used with *some types* of Java applications with loosely-synchronous tasks. The focus and contributions of this work are less on raw speed or features, which may be better achieved using C and dedicated message-passing libraries, than on transparency and applicability of our parallelization techniques to legacy Java code. They provide an easy way to harness the processing power of idle workstation to increase the performance of applications with no built-in support for parallel processing.

To the best of our knowledge, this work is the first to address the problem of automatic parallelization of Java applications by instrumenting bytecode and transparently executing computationally-intensive programs on distributed processors.

Related Work. Automatic parallelization of a program is generally achieved using parallel compilers that generate code optimized for parallel architectures [8, 9]. In the context of the Java programming language, javar [10] is a source code transformation engine that makes implicit loop parallelism and multi-way recursive methods explicit by means of the multi-threading mechanism provided by the Java virtual machine. The resulting code can execute faster on parallel machines that run multiple threads on separate processors. Javab [11] performs essentially the same transformations, but on the program's bytecode rather than its source code. JOIE [12] is another toolkit for Java bytecode transformations, but it has been designed to modify the behavior of the code rather than optimizing its execution for a given target environment. IBM's Ninja project [5] includes a prototype Java compiler that performs high order loop transformations and parallelization completely automatically, resulting in runtime performance similar to Fortran code in a variety of benchmarks.

The development of parallel applications targeted for execution on distributed processors traditionally requires parallelism to be dealt with explicitly. These applications are traditionally implemented using specialized message-passing libraries such as PVM [13] and MPI [14], which manage communications between sets of collaborating processes executing on multiple machines. PVM and MPI have mappings for several programming languages, and have recently been extended to support Java [15, 16]. Although powerful and robust, theses message-passing libraries are also complex to program, even in their Java incarnation. JavaParty [17] simplifies this process by introducing language constructs for the development of distributed and parallel Java applications, but the programmer still needs to deal explicitely with parallelism. COPS [18] goes one step further by using parallel design patterns to automatically generate the structural code necessary for a Java application to run in parallel.

Organization. The rest of this paper is organized as follows. Section 2 presents an overview of our Java parallelization framework. Section 3 describes the wrapper generator used to transparently instrument Java bytecode and intercept selected method invocations. Section 4 presents the parallelization engine responsible for managing communications between distributed processors. Section 5 illustrates our tools using a concrete example, and Section 6 elaborates on their performance. Finally, Section 7 concludes the paper.

2 Program Parallelization

There are several approaches to make a program execute faster using parallel processing. For instance, a multi-threaded program can benefit from parallel architectures by having each thread run on a distinct processor. In this paper, we focus on coarse-grain parallelization, where multiple *distributed* computers work together to perform time-consuming computations. As the time necessary for communication between collaborating computers is not negligible, this approach works well when computations require significant processor resources, in the order of seconds, and each processor can compute its share independently of other processors (loose synchronization).

Several types of applications can benefit from distributed parallel processing. For instance, complex database queries can be executed by having each processor looking through part of the data, or executing part of the query. Complex computations, such as cryptographic key discovery, or synthesis image generation, can also be parallelized by having each processor explore part of the space of input values (using “divide-and-conquer” algorithms).

The idea underlying our Java application parallelization framework is to instrument the classes responsible for time-consuming computations, instantiate them on multiple machines, and re-direct the invocations to computationally-intensive methods to all the instances for parallel execution. Method interception is achieved by the means of a wrapper generator toolkit, which constitutes the lowest layer of our parallelization framework. At the next level, the Java parallelization engine takes care of load sharing and communication with distributed processors. Finally, the deployer has to provide application-specific adapters that

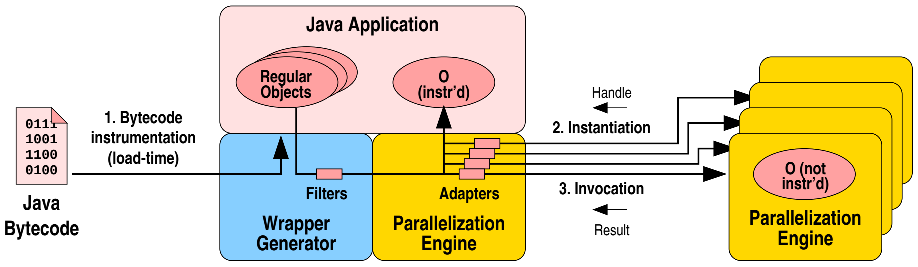

Fig. 1. Semi-Automatic Parallelization of a Java Application.

essentially define "rewriting rules" for splitting requests and merging replies. The overall system architecture is shown in Fig. 1.

Rewriting rules are application-specific. They specify for each application how a computationally-intensive request can be split into multiple simple sub-requests that can execute in parallel. In addition, they specify how the results of these individual sub-requests can be combined to produce the complete result expected from the initial request. A typical rule for the computation of a synthesis image would rewrite request arguments and assign non-overlapping areas of the image to each target processor; the results from each processor would later be combined into a single image by appending them in the right sequence.

Our approach is transparent to the application being parallelized, as it does not require source-code modifications, but it is not fully automatic, in the sense that the application deployer has to specify the rewriting rules. All the information pertaining to the rewriting rules, the classes to instrument, the addresses of the distributed processors, etc. are specified using Java properties and configuration files.

3 The Java Wrapper Generator

We have developed a tool, called the Java Wrapper Generator (JWG), which uses load-time reflection to transparently insert pre- and post-filters to any method from a Java class. These generic filters allow developers to add crosscutting functionality to compiled Java code and extend it with various features, such as debugging, profiling, proxying, runtime validation, security, or aspect-oriented extensions. In this section, we briefly present the major features of our wrapper generator, used by the parallelization engine to add parallel distributed behavior to sequential centralized Java programs.

3.1 Filters

The Java wrapper generator allows Java developers to transparently insert pre- and post-filters to any method of a Java class. Filters are custom classes written by the user and attached to specific methods at the time the class is loaded by the Java virtual machine, by instrumenting the bytecode (1. in Fig. 1). It is therefore not necessary to access the source code of the Java class.

Pre- and post-filters can be installed at the following levels, from the most to the least specialized: an *instance method* filter applies to a specific method of a given instance; an *instance* filter applies to all methods of a given instance; a *class method* filter applies to a specific method of a given class; a *class* filter applies to all methods of a given class; and finally, a *global* filter applies to all methods of all classes. Upon invocation of an instrumented method, the wrapper generator runtime searches all installed filters in decreasing order of specialization, until it finds a valid filter. If no filter is found, then no filtering takes place.

Pre-filters are invoked at the beginning of each instrumented method. In the special case of constructors, pre-filters are invoked after the call to the constructor of the superclass or the class itself, i.e., after the constructor of the `Object` base class has been called. Pre-filters receive as parameters the target object (for non-static methods) or class (for static methods), and the method name, signature, and arguments. A pre-filter can modify the values of the method arguments, but the number and types of the arguments must remain consistent with the method's signature. A pre-filter can terminate in three different manners: (1) the filtered method continues normally after execution of the filter (normal termination); (2) the filtered method terminates immediately with the return value provided by the filter, which must be of a type consistent with the method signature (short-circuit); and (3) the filtered method terminates immediately by throwing the exception provided by the filter, which must be consistent with the exceptions declared by the method (exceptional termination).

Post-filters are invoked at the end of each instrumented method. They are also invoked upon abrupt completion (`return` statement in the middle of a method) or when an exception occurs during method execution. Post-filters receive as parameters the target object (for non-static methods) or class (for static methods), the method name, signature, and arguments, and the return value or exception resulting from the method's execution. A post-filter can modify the arguments (which may be used to return data to the caller) and the return value or exception. In addition, a return value can be replaced by an exception, and vice versa, as long as the type of the return value or exception remains consistent with the method's signature.

The association of filters with specific classes, objects and methods can be performed declaratively (via a configuration file) or programmatically. Furthermore, the wrapper generator provides a simple API to dynamically install and remove filters during program execution.

3.2 Bytecode Instrumentation

Bytecode instrumentation is performed using the BCEL bytecode engineering library [19] and a custom "class loader", with overrides the default behavior of the Java class loading mechanism. The code of selected methods is modified to include calls to user-specified pre- and post-filters. The functionality added to the bytecode is minimal: it includes parameter transformation (simple types are transformed in their equivalent object types), filter invocation, and exception and result management. Additional logic is implemented in regular Java libraries.

The names of the classes and methods to be instrumented can be specified at deployment time via a configuration file. Although instrumented methods can have no filter attached to them, unmodified code executes faster and it is therefore desirable to restrict the scope of instrumentation to only those classes that need it. Methods that are not instrumented during class loading are not filterable.

4 The Java Parallelization Engine

The Java parallelization engine builds on top of the wrapper generator described in the previous section. It is responsible for sharing the workload and managing communication with the distributed processors.

4.1 Architectural Overview

The parallelization engine consists of two major components (see Fig. 1). A client-side library that attaches itself transparently to the application being parallelized, and a server-side daemon program that provides its processing power to the application. Both of these components are independent of the target application. Application-specific functionality is specifies by the means of "adapters", which act as the glue that binds the client-side parallelization engine and the target application.

Although we could have re-used specialized toolkits such as PVM [13] and MPI [14] to implement the parallelization engine and handle our communications, we have rather chosen to develop lightweight mechanisms adapted to our specific requirements. Server applications, also called "workers", listen to incoming TCP connections from parallelized clients. (Note that we could have used Java RMI instead, but raw TCP has less overhead and makes it easier to quickly detect and recover from worker failures.) Each worker can service multiple clients concurrently, using Java's multi-threading features. Once a connection is established between a client and a worker, the client can send requests to be processed by the worker. There are three types of requests: object creation, object invocation, and object deletion. These requests control all interactions between the parallelized Java program and the remote processors utilized by the parallelization engine.

4.2 Distributed Invocations

The parallelization engine filters the constructors of each parallelized object, and issues a remote object creation request to multiple workers when such a constructor is called (2. in Fig. 1). The bytecode of the classes to instantiate (or alternatively a URL/URI to that bytecode) is sent together with the creation request, which leads to the instantiation of a non-instrumented copy of the object in each worker process. The worker returns a handle—a string identifying the object in the server process—to the client; this handle is subsequently used in client requests to identify objects on the server. The parallelization engine transparently keeps track of the object handles associated with each parallelized object at each worker. An instrumented copy of the object is also created locally; this copy can be used for serving requests that do not need to be parallelized.

Regular invocations to a parallelized object are also intercepted by the parallelization engine. The method invocation is first passed to an adapter for rewriting (to be described shortly), and then sent to each worker along with the handle of the object on that worker (3. in Fig. 1). Once the request has been processed by each worker, replies are sent to the client, combined using the adapters, and returned to the invoker.

Finally, when a parallelized object is no longer needed and its `finalize` method is called, the parallelization engine sends a deletion request to each worker along with the handle of the object to delete. All objects and resources allocated for a given client are also automatically reclaimed when the connection to that client is closed.

4.3 Adapters

Adapters are regular Java objects that implement the `Adapter` interface. They implement a `split` method, which takes the original request targeted to the non-parallelized version of an object, and transforms it into a request to be sent to a single worker in the parallel version of the program. Likewise, the `join` method combines the replies sent back by individual workers into a single reply returned to the original program. Both the `split` and `join` methods are given the total number of workers, as well as the index of the worker concerned by the current request/reply; this information enables adapters to deterministically determine which part of the request must be processed by each worker. For instance, given n workers, the workload can be split into n equals parts, with the worker at index i being responsible for the i^{th} part. It is also possible to configure the parallelization engine to call each worker multiple times in the context of a given client invocation, and to implement more dynamic scheduling algorithms, such as guided self scheduling [20].

The parallelization engine creates one adapter per worker. An adapter is assigned to a single worker during its lifetime, and is guaranteed that each invocation to `join` directly follows the matching invocation to `split`. These properties enable adapter objects to maintain consistent state information about the workers they are responsible for.

The `split` method is given information about the method being invoked, as well as its parameters. A simple rewriting rule would change the arguments to specify the part of the workload affected to a given worker. An adapter may have to consistently rewrite the arguments of several methods, including the constructor of the parallelized object, to ultimately achieve the desired partial computation.

The `join` method is given information about the target method, the parameters that were received as part of the non-parallel invocation, and the results from the execution on the worker. A typical rewriting rule would copy the relevant portion of the data received from the worker into the parameters/return value associated with the original invocation. Examples of `join` and `split` method implementations are given in the next section, in Listing 1.2.

4.4 Failures

When the client application fails, all TCP connections with the workers are closed and the resources associated to that client are automatically reclaimed. When one of the worker fails, the client runtime will transparently re-submit the partial request assigned to that worker to another worker. Optionally, it can also recursively split and share the aborted partial request among all non-failed workers. By default, no new connections are opened at runtime and failed workers are not replaced during the lifetime of the client. This does not pose a problem in practice as the lifetime of clients is generally much shorter than that of the workers and failures are expected to be rare events. In addition, clients can initially connect to a larger number of workers that they actually need, to account for possible failures.

4.5 Limitations

As previously discusses, one of the major contributions of our work lies in the automatic parallelization of binary Java applications. The steps involved in the parallelization process are the discovery of the classes and methods to parallelize, the specification of rewriting rules in the form of adapter objects, and the deployment of the parallelized application with multiple worker processes.

Because of its transparency feature, our parallelization framework has several limitations. First, it only applies to applications that are naturally parallelizable, and for which the gain of parallelization exceeds its cost. Note that this is also true of distributed parallel application deployed with specialized message-passing libraries such as PVM [13] and MPI [14]. Our transparency goals also limit our scope to applications that can be parallelized by intercepting and rewriting selected invocations. This is more often the case with well-engineered object-oriented applications, which have a modular structure and encapsulate functionality (e.g., compute-intensive tasks) inside objects with a well-defined interface.

Programs that have a complex structure, for instance because they extensively use callbacks or exchange complex objects that are not serializable as part of invocation arguments, may also not be parallelizable without modifications to their source code. Note again that such program would also need major reengineering to be deployed on top of PVM or MPI. Finally, classes that use native libraries and are not written in 100% pure Java cannot be instrumented by the wrapper generator.

5 Example

In this section, we illustrate the use of our framework by showing the steps necessary to parallelize a computationally-intensive Java program that generates fractal images.

5.1 The Mandelbrot Set

An example of an time-consuming, easily-parallelizable application is the computation of the Mandelbrot set. The Mandelbrot set is a fractal structure defined in the complex plane by the following equation: $z_n = (z_{n-1})^2 + z_0$. The set itself is the area where $\lim_{n\to\infty} z_n < \infty$.

It is demonstrated that if $|z_i| > 4$, then z_n will eventually reach ∞. An approximation of the set can be computed by iterating the formula. Points where $|z_i| > 2$ are not part of the set, and the remaining points *may* be part of the set. The resulting set is traditionally displayed in a two-dimensional picture.

This computation is time-consuming: for each point the formula is iterated until $|z_i| > 2$, or a constant number of iterations have been performed. Because the adherence of each point to the set is determined only by the point's position, the computation is easy to parallelize.

5.2 The Application

We have taken an existing Mandelbrot application written in Java [21]. Roughly-speaking, this application consists of several classes responsible for the graphical user interface, and a `MandelComputer` class responsible for computing a region of the Mandelbrot set. The application has been programmed with no parallelism in mind: the computation of the complete region displayed on the screen is performed by a single instance of the `MandelComputer` class.

```java
public class MandelComputer {
  public MandelComputer(int w, int h /* ... */);
  public final void computeRegion(short[] buf, int l, int w, int t , int h);
}
```

Listing 1.1. Structure of the class responsible for the computation of the Mandelbrot set. This class will be instrumented for parallelization.

Although the actual parallelization of a Java application does not require access to its source code, one needs to understand enough of the application structure to determine which classes and methods to instrument and how to define the rewriting rules. To that end, one can use `javap`, a tool provided with Sun's Java compiler that lists the signatures of the methods defined in a class file. Note that it can be difficult to understand the actual semantics of the methods and their parameters when source code is not available, without adequate documentation or a decompiler. The structure of the `MandelComputer` class is shown in Listing 1.1 (for the sake of clarity, we have slightly modified the method signatures).

A `MandelComputer` object is instantiated with a given width and height, as well as parameters such as the maximal number of iterations. Once instantiated, the object computes a region of the Mandelbrot set upon invocation of its `computeRegion`. The top-left corner, the width, and the height of the region are given as parameters. The result of the computation is stored in an array, also handed as parameter to the method.

5.3 The Adapter

Once the classes to parallelize have been defined, one needs to write the adapter object responsible for the rewriting of the requests and replies sent to and received from individual workers. Listing 1.2 shows the code of the adapter for our Mandelbrot application. Note that this code has less that 40 lines.

In the `split` method, we only rewrite the parameters for invocations to `computeRegion`.[1] This is achieved by splitting the area into horizontal bands of equal sizes, with each worker being responsible for one such band. Arguments are modified to update the new coordinates of the top of the band, its height, as well as to provide a properly-dimensioned array for storing the computed data. For performance reasons, we also store the location in the full region where the band assigned to the current worker will be stored; this index will not have to be re-computed in the `join` method.

In the `join` method, we simply copy the data computed by the worker at the right position in the array originally provided by the client application.

5.4 Deployment

Deployment of the parallelized application merely consists of starting multiple workers on distributed processors, and launching the Mandelbrot application. The addresses of the workers, the maximum number of workers to use, the name of the adapter classes, and the name of classes and methods to filter, are all specified using configuration files and Java properties.

As bytecode modification is performed using a custom class loader, the Java application must be started using a special launcher program that makes sure that the application classes are effectively loaded by our class loader. This is achieved by invoking the Java application as follows:

```
java [properties...] jwg.Launcher Mandelbrot [arguments...]
```

[1] Note that, for space efficiency, we could also have rewritten the arguments of the constructor to reduce the dimensions of the Mandelbrot set instantiated on each worker. However, this strategy would require non-trivial changes to the rewriting rules for `computeRegion`.

```
public class MandelAdapter implements Adapter {

  int idx_;  // Start index for this worker in returned array

  public void split(int id, int nb,
                    String classname, String method, String signature,
                    Object[] args)
  {
    if(method.equals("<init>")) {
      // MandelComputer(int w, int h, ...)
      // May rewrite constructor (not required for this application)
    } else if(method.equals("computeRegion")) {
      // computeRegion(short[] buf, int l, int w, int t, int h)
      int w = ((Integer)args[2]).intValue();
      int h = ((Integer)args[4]).intValue();
      int t = id * h / nb;
      idx_ = t * w;  // Pre-compute index since we have t and w anyway
      h = (id == nb - 1 ? h - t : ((id + 1) * h / nb) - t);
      args[0] = new short[w * h];
      args[3] = new Integer(t);
      args[4] = new Integer(h);
    }
  }

  public void join(int id, int nb,
                   String classname, String method, String signature,
                   Object[] in_args, Object in_result,
                   Object[] out_args, ResultHolder out_result)
  {
    if(method.equals("computeRegion")) {
      // computeRegion(short[] buf, int l, int w, int t, int h)
      short[] dst = (short[])out_args[0];
      short[] src = (short[])in_args[0];
      int idx = idx_;  // Index has been pre-computed in split()
      for(int i = 0; i < src.length; i++)
        dst[idx++] = src[i];
    }
  }
}
```

Listing 1.2. Adapter object for parallelizing the Mandelbrot application.

Other than instrumenting the bytecode, this command has virtually the same effect as invoking directly the Java application using the following command:

```
java [properties...] Mandelbrot [arguments...]
```

6 Performance Evaluation

6.1 Experimental Setup

We have run tests with the Mandelbrot program on a set of 18 identical Sun Ultra 10 workstation, with a 440 MHz processor and 256 MB of memory, running Solaris 2.8. We have computed images with a resolution of 720×512 pixels and up to 1200 iterations for two distinct regions of the Mandelbrot set. We have run experiments with the non-instrumented application (centralized), and with semi-automatic parallelization using from 1 to 15 worker processors. Each worker was

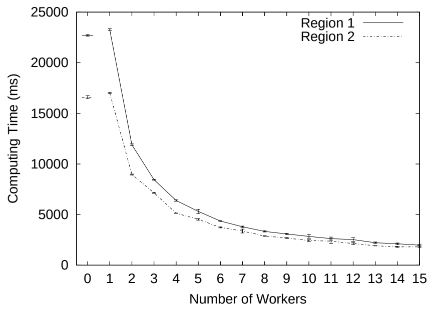

Fig. 2. Performance improvements with parallel processing.

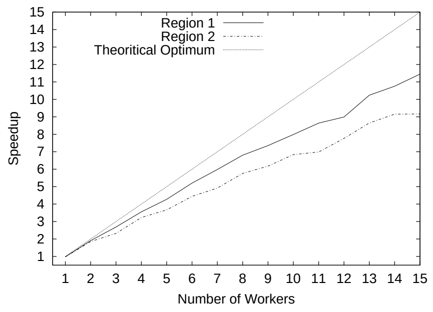

Fig. 3. Speedup factor with parallel processing.

running on a separate machnie, in its own Java virtual machine (version 1.4.1). For each configuration, we have run the program 12 times on random subsets of the workstations and taken the mean of the measurements.

6.2 The Gain of Parallelization

The computation times as a function of the number of workers are shown in Fig. 2 (error bars correspond to the 95% confidence intervals). The value for 0 worker corresponds to the execution of the non-parallelized version of the program. The graph clearly shows that performance increases as a function of the logarithm of the number of workers.

We have also computed the speedup factor gained from parallelization, in comparison with the non-parallelized version of the program. The speedup, shown in Fig. 3, remains within 25% of the optimum for the first region, and within 35% for the second region. Two main reasons prevent the speedup to remain closer to the optimum. First, the time necessary for communications, and for request and reply rewriting is not negligible and must be accounted for. Second, our program splits the Mandelbrot set in equal regions sent to each worker; as some regions require more computations than others, the load is not distributed equally and speedup cannot be maximized. In our experiments, the second region appears to be more affected by this problem as its speedup is smaller than for the first region.

6.3 The Cost of Parallelization

The overhead of the parallelization framework can be observed in Fig. 2 as the difference between the non-parallelized version of the program and the parallel version with 1 worker. In our tests, this cost was 501 *ms* on average. It can be broken down into the cost of method interception, the cost of request and reply processing, and the cost of remote invocation. These various sources of overhead are detailed in Table 4.

Method interception	$< 1\ \mu s$
Request processing	288 ms
Reply processing	44 ms
Remote invocation	169 ms
Total cost	501 ms

Fig. 4. Cost of Parallelization

Method interception only introduces a small performance overhead. In our experiments, adding empty pre- and post-filters to a method costs less that 1 μs with a JIT compiler. In addition to processing overhead, instrumented classes also incur a size penalty due to the extra code added during instrumentation. For our Mandelbrot application, the size of the `MandelComputer` class grows from $1,702$ to $2,646$ bytes with two instrumented methods, i.e., an increase of less than 1 kB. If only a fraction of the classes of an application are instrumented, we can safely ignore both the time and space penalty of method interception.

Request processing is clearly the most expensive operation during parallelization, because it includes the creation of one thread per worker,[2] in addition to the rewriting and marshaling of the request. Reply processing is significantly cheaper, as it only includes the cost of unmarshaling and rewriting. Finally, the cost of remote invocation, which includes both the round-trip communication time and the data un/marshaling on the server, also adds non-negligible overhead to the parallelized application.

Despite the overhead introduced by the parallelization process, it appears clearly that the performance improvements resulting from parallelization well exceed its cost, even when using as few as two workers. Other applications that can be parallelized using a divide-and-conquer strategy should exhibit similar performance gains.

7 Conclusions

In this paper, we have presented mechanisms to parallelize certain types of Java applications, without modifications to their source code. Once parallelized, applications execute their time-consuming computations on multiple distributed processors.

Bytecode is first instrumented by our Java wrapper generator, which controls the flow of Java applications by inserting filters in selected methods. At runtime, invocations to instrumented methods are intercepted by the Java parallelization engine, which is responsible for rewriting the requests, sharing the workload among multiple workers, and aggregating the return values. A small piece of code—called an adapter—must be provided to the parallelization framework for handling application-specific request and reply transformations.

[2] Thread creation is a costly operation in Java. As an obvious improvement, we could avoid the cost of thread creation by using a pool of threads.

We have illustrated our mechanisms with an existing Java application that computes regions of the Mandelbrot set, and we have evaluated the performance of the resulting parallelized application. Experimental results demonstrate that the speedup of parallelization increases almost linearly with the number of processors, while its cost remains reasonably small.

The major contribution of this work lies in the semi-automatic parallelization and distributed deployment of legacy Java code. Our parallelization framework provide an easy way to harness the processing power of idle, heterogeneous workstation on the Internet to increase the performance of applications with no built-in support for parallel processing.

References

1. Fox, G., ed.: Special Issue on Java for Computational Science and Engineering–Simulation and Modeling II. Volume 9 (11) of Concurrency: Practice and Experience. John Wiley & Sohn Ltd. (1997)
2. Fox, G., ed.: Special Issue on Java for High-performance Network Computing. Volume 10 (11-13) of Concurrency: Practice and Experience. John Wiley & Sohn Ltd. (1998)
3. Lobosco, M., de Amorim, C., Loques, O.: Java for High-performance Network-based Computing: A Survey. Concurrency: Practice and Experience **14** (2002) 1–31
4. Hsieh, C.H., Gyllenhaal, J., Hwu, W.M.: Java Bytecode to Native Code Translation: The Caffeine Prototype and Preliminary Results. In: Proceedings of the 29th International Symposium on Microarchitectures. (1996)
5. Artigas, P., Gupta, M., Midkiff, S., Moreira, J.: Automatic Loop Transformations and Parallelization for Java. In: Proceedings of the International Conference on Supercomputing (ICS 2000). (2000) 1–10
6. Serrano, M., Bordawekar, R., Midkiff, S., Gupta, M.: Quicksilver: A Quasi-Static Compiler for Java. In: Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA). (2000) 66–82
7. Yu, D., Shao, Z., Trifonov, V.: Supporting Binary Compatibility with Static Compilation. In: Proceedings of the 2nd Java Virtual Machine Research and Technology Symposium (JVM'02). (2002) 165–180
8. Polychronopoulos, C.: Parallel Programming and Compilers. Kluwer (1988)
9. Wolfe, M.: High Performance Compilers for Parallel Computers. Addison-Wesley (1996)
10. Bik, A., Gannon, D.: Automatically Exploiting Implicit Parallelism in Java. Concurrency: Practice and Experience **9** (1997) 579–619
11. Bik, A., Gannon, D.: JAVAB–A Prototype Bytecode Parallelization Tool. Technical Report TR489, Indiana University (1997)
12. Cohen, G., Chase, J., Kaminsky, D.: Automatic Program Transformation with JOIE. In: Proceedings of the 1998 USENIX Annual Technical Conference. (1998)
13. Sunderam, V.: PVM: A Framework for Parallel Distributed Computing. Concurrency: Practice and Experience **2** (1990) 315–339
14. Hempel, R.: The MPI standard for message passing. In Gentzsch, W., Harms, U., eds.: High-Performance Computing and Networking, International Conference and Exhibition, Proceedings, Volume II: Networking and Tools. Volume 797 of Lecture Notes in Computer Science., Springer-Verlag (1994) 247–252

15. Yalamanchilli, N., Cohen, W.: Communication Performance of Java-Based Parallel Virtual Machines. Concurrency: Practice and Experience **10** (1998) 315–339
16. Baker, M., Carpenter, D., Fox, G., Ko, S., Lim, S.: mpiJava: An Object-Oriented Java interface to MPI. In: Proceedings of the 1st Java Workshop at the 13th IPPS & 10th SPDP Conference. Lecture Notes in Computer Science, Springer-Verlag (1999)
17. Philippsen, M., Zenger, M.: JavaParty: Transparent remote objects in Java. Concurrency: Practice and Experience **9** (1997) 1225–1242
18. MacDonald, S.: From Patterns to Frameworks to Parallel Programs. PhD thesis, University of Alberta (2002)
19. The Apache Software Foundation: BCEL: Byte Code Engineering Library. http://jakarta.apache.org/bcel (2003)
20. Polychronopoulos, C., Kuck, D.: Guided Self Scheduling. IEEE Transactions on Computers **36** (1987) 1425–1439
21. Ziring, N.: JManEx: Java Mandelbrot Explorer. http://users.erols.com/ziring/-mandel.html (2001)