An Annotation-Based Framework for Parallel Computing*

C. A. Cunha[1], J. L. Sobral[2]
[1]Departamento de Informática, Instituto Politécnico de Viseu, Portugal
[2]Departamento de Informática, Universidade do Minho, Braga, Portugal
cacunha@di.estv.ipv.pt jls@di.uminho.pt

Abstract

This paper presents a programming language for parallel computing based on code annotations. It has similar goals and philosophy as OpenMP but it is more tightly coupled to the object oriented paradigm. We include annotations for most common concurrency patterns and mechanisms, namely, one-way, futures, barriers, reads/writers and thread-local. Our current prototype is implemented using Java 5 annotations and AspectJ and provides a feasible and efficient alternative to the Java thread model.

1. Introduction

Multi-core and multiprocessor machines are becoming mainstream architectures. However, to take advantage of this type of architectures, applications should go through several modifications, as to specify tasks that can run in parallel and to perform synchronisation among tasks. This incurs in an additional burden to the programmer and may involve non-reversible changes to applications.

Traditional thread programming can be used to convert a sequential application to a parallel counterpart, however it lacks of suitable abstractions to help the programmer to structure parallel applications and may require a considerable amount of code rewrite to fit into the thread parallelisation model.

OpenMP [1] provides a more structured way to introduce parallelism into a sequential application. It includes parallel blocks and parallel loops (e.g. *for*), as well as a set of synchronisation directives for shared variables. Parallelism related concerns in OpenMP are specified through a set of directives that can be ignored by a non-compliant compiler, achieving a valid sequential program. This approach makes parallel code closer to sequential versions, softening the transition from sequential to parallel programming. OpenMP is currently supported by important compilers, namely Intel, Microsoft and GNU compilers.

There is still no official binding of OpenMP for Java and, most important, OpenMP is targeted for traditional structured programming and not for object-oriented programming models. As consequence, it can not take advantage of the structure of object oriented applications (e.g., classes and method calls).

In this article we propose a framework comprising a set of annotations intended to replace OpenMP directives in the context of object oriented applications. In this approach we annotate classes, methods and instance variables instead of generic blocks of code, resulting in a more high level and object-based reasoning. The provided annotations include: @Oneway, which spawns a method call in a new thread; @Barrier, which blocks a set of threads until all have reached a certain point in program execution and @ThreadLocal, which creates a local instance variable per thread.

The framework was implemented in Java using Java 5 annotations [2] and AspectJ [3]. This approach provides a fully integration into the Java language, including compile time check of annotation syntax and direct generation of Java bytecodes, without intermediate compilation steps. This avoids a common problem, in pre-processor based approaches, of tracing an error to the correct point in source code.

This work differs from other research works in the way it uses programmer based annotations to specify parallel execution and synchronisation, in a philosophy similar to OpenMP. It differs substantially from OpenMP, since annotations are provided at instance variable/method level and are fully integrated in an object oriented model.

The remainder of this paper is organised as follows. Section 2 presents related work. Section 3 describes the proposed annotations from a programmer perspective. Section 4 overviews the current implementation of these annotations by means of Java 5 annotations and AspectJ. Section 5 shows several code examples and section 6 presents performance evaluation. The last section discusses obtained results and future work.

* This work was supported by PPC-VM project (Portable Parallel Computing Based on Virtual Machines, POSI/CHS/47158/2002) and by SeARCH (Services & Advanced Computing with HTC/HPC, CONC-REEQ/443/EEI/2005) both funded by Portuguese FCT (POSI) and European funds (FEDER).

2. Related Work

Early work on objects and concurrency introduced several high level patterns and mechanisms to address the complexity of concurrent programs. One of these early efforts was concurrency annotations [4]. In this approach Eiffel programs are annotated to introduce a parallel semantic, i.e., an alterative semantic that supports parallel execution. Provided annotations include compatibility annotations (a generalisation of readers and writers methods), delayed acceptance based on Eiffel preconditions, autonomous objects (i.e., active) and asynchronous invocations. This model was recently ported to Java [5].

Another relevant effort was COOL [6], an extension to C++ that provides parallel methods that run on a separate thread and event synchronisation, a mechanism similar to Java monitors. Although this model has a philosophy similar to concurrency annotations it introduces new keywords to C++ to provide a parallel semantic.

ProActive [7] includes active objects and automatic future method calls (a mechanism called wait-by-necessity). ProActive does not rely on programmer annotation to express parallelisation issues, as it is based on a more implicit parallelisation model.

A large amount of Java based approaches are concerned to Java distributed machines aimed to address distributed thread execution, supporting a single system image. Examples are cJVM [8], Hyperion [9], Jakal [10] and JESSICA2 [11]. These approaches rely on the Java thread model to introduce concurrency into sequential applications.

An OpenMP bind for Java is presented in [12]. Directives are introduced through special Java comments. An external tool converts an annotated program standard to Java code. In [13] a similar bind is presented, better fitted into Java language philosophy and [14] presents an implementation on distributed memory systems by means of a DSM middleware.

Our proposal differs from these previous efforts as we provide a richer set of annotations, fully integrated into an object oriented model. Additionally, we provide a complete implementation, based on Java 5 annotations, built as an AspectJ library[§].

3. Annotations for Concurrent Execution

Annotations are a metadata facility introduced in J2SE 5.0 (Tiger) [15] implemented as modifiers that can be added to the code, namely to classes, interfaces, methods and fields. Java Tiger includes built-in annotations and supports the creation of new custom annotations. Annotations describe the elements behaviour which contributes to a better code comprehension about the program behaviour.

Several annotations were implemented to describe concurrent behaviour. Such annotations are considered in the context of many widely known high-level constructs for concurrency [16][17], namely One-way, Future, Active Object, Barrier, Synchronised, Readers-Writer, Scheduler and Thread local. Each construct comprises one or more annotations, which can be used to describe specific element types, e.g. fields, methods or classes (Figure 1).

```
@ClassAnnotation
public class <class_name> {
  @FieldAnnotation int x;

  //...
  @MethodAnnotation
  public void execute(){}
}
```

Figure 1. Annotate elements

Occurrences of annotated elements are intercepted by aspects to add the corresponding concurrent behaviour at those points (see section 4).

We classify annotations into concurrency generation annotations and synchronisation annotations. Most annotations can be applied to a specific thread group, supplied as an annotation parameter. Concurrency generation annotations can associate the spawned thread to a specific group, while synchronisation annotations can be restricted to a specific thread group. This feature allows multiple annotations to coexist in the same point in the code.

Synchronisation annotations also feature an optional lock id name. This allows several locks to coexist for the same object or sharing of locks among objects.

3.1. One-way

One-way mechanism [16] applies to methods that run on a thread of their own: the client never blocks, waiting for some result. One-way pertains only to asynchronous void method calls – when the method does return a value, a future (3.2) should be used.

One-way usage follows the syntax:

```
@Oneway(threadGroup=[thread group id],
     saveState=[Yes|No])
```

@Oneway annotate methods which should be executed in parallel. A new thread will be created to execute each annotated method, associated to thread

[§]The AspectJ library can be downloaded from the PPC-VM web page at http://gec.di.uminho.pt/ppc-vm

group specified in *threadGroup* parameter. When omitted, the new thread is associated with the thread group of the current thread. If the thread reference is required latter – e.g. to perform a *join* or *sleep* operation – the *saveState* parameter should be specified as *True*.

The traditional fork and join algorithms require, in addition to the thread spawning mechanism, a way to synchronise (i.e., join) the main thread and spawned threads at some point in the application. Join operations are allowed by using @JoinBeforeExecution and @JoinAfterExecution annotations to force the current thread to wait for all threads created by it to terminate, before or after the annotated method execution. Similarly, @SleepBeforeExecution and @SleepAfterExecution forces the current thread to sleep during an amount of time specified by parameter *time*. Interruption of threads are also allowed using @Interrupt to interrupt all threads created by the current thread and @InterruptAll to interrupt all threads created in the annotation context.

@OnewayExecutor is a less flexible One-way implementation, but more efficient for fine-grained, non-blocking methods as long as threads present in the thread pool are reused several times, reducing thread spawning overhead and avoiding the unrestrained creation of threads. It has the following syntax:

```
@OnewayExecutor(executorId="x",
    tasksGroup=[thread group id],
    poolSize=[size],
    chunkSize=[size])
```

OnewayExecutor uses *java.util.concurrent.Executor* service to execute one way method calls, which is based on Thread Pools. @OnewayExecutor supports some additional parameters. Each *Executor* is identified by an *executorId* and the number of threads in the thread pool is specified by *poolSize*. If omitted, the number of threads is the same as the number of processors (e.g., cores). Tasks submitted to the executor service can belong to a specific group (*tasksGroup* parameter) and can by agglomerated to reduce the number of tasks submitted to the executor. *chunkSize* parameter specifies the number of one way invocations agglomerated per each submitted task. By default chunk size is 1. Additional mechanisms deal with incomplete chunks (e.g., time out mechanisms).

@OnewayExecutor does not support the *sleep* functionality, but is supports *join* operations (in specific executors and/or *tasksGroups*) and a cancel annotation, similar to @Interrupt.

3.2. Futures

Futures [16] allow two-way asynchronous invocations of methods that return a value to the client. Futures are join-based mechanisms based on data objects that automatically block when clients try to use their values before the corresponding computation is complete. During execution of methods, the Future is a placeholder for the value that has not yet computed.

In typical situations, Futures are used when a variable stores the result of a computation, which will be used later. Consider the following code:

```
a = someobject.compute();
...// other statements
a.doSomething();
```

The *compute* method is executed by the new thread. Instead of blocking at the computation phase – i.e. during the execution of *compute* – the thread blocks when the variable is actually accessed – i.e. when the method *doSomething* is executed.

Our annotation-based version of Future uses @Future annotation to annotate non-void methods invoked asynchronously and @FutureClient to annotate methods that use values returned in consequence of asynchronous method invocations. In addition, when the return type cannot be instantiated automatically (e.g., it requires specific parameters) the user should provide the method *getFakeObject* – which implements the code that instantiate the fake object.

3.3. Active Object

Active object decouples the invocation of methods from their execution [17]. Each object runs into its own thread of control. Whenever a client object invokes a method from an active object, the thread associated with the active object carries out the execution. Active object is an abstraction that merges objects with concurrency. One of the early systems to use active objects was ABCL [18].

Several applications can benefit from the use of active objects. Client applications such as windowing systems and network browsers employ active objects to simplify concurrent, asynchronous network operations. Multi-threaded servers, producer/consumer and reader/writer applications are also well suited for the use of active objects.

The creation of an active object involves the creation of several support structures, including a proxy object and a scheduler [17]. Our @ActiveObject annotation completely hides this complex structure behind a reusable aspect.

3.4. Barrier

Barrier establishes a set of synchronisation points where threads should synchronise. Each thread reaching one of that points blocks waiting for the other threads of the group.

Methods where threads synchronise can be annotated either with @BarrierBeforeExecution or @BarrierAfterExecution, depending if it blocks before or after method execution. Element-pair values in annotations allow definition of parameter values that can be used in this case to store the number of blocking threads and the target thread group name. Element-pair values, namely *nThreads* and *threadGroup* should be assigned with the number of threads and the target thread group where the barrier should apply. For instance, for five threads blocking after method execution associated to thread group *calculus*, the annotation is:

```
@BarrierAfterExecution (nThreads = 5,
            threadGroup = "calculus")
```

When the number of threads is not specified in barrier annotation the barrier waits for all spawned threads.

3.5. Synchronised

Synchronisation is implemented in Java language using the *synchronized* modifier on method declaration or using the block construct

```
synchronized(<object>){ //... code }
```

Each object holds its own lock, which is used to assure the exclusive access to methods or regions declared as synchronised. Although *synchronized* methods share the same object lock, *synchronized* block construct enables the use of other object lock, in order to have a single lock shared among multiple type-unrelated objects.

Synchronised methods would be annotated with *@Synchronized*. As a consequence, each intercepted method or variable access uses the respective object lock to synchronise access to it.

Optionally, the synchronised mechanism would use other lock instances to synchronise access to methods defined in different class instances. In order to identify a particular lock in the application, the parameter *id* should be settled. An *id* is associated univocally to a specific lock, which can be specified in the following manner:

```
@Synchronized(id = "lockName")
```

Contrasted with Java constructs, *@Synchronized* disallows fine-grain synchronisation, as long as annotations cannot be used to annotate single instructions or block of instructions. By resorting to refactoring we can move instructions to methods in order to annotate such code and then enable the same expressivity of Java constructs.

3.6. Readers/Writer

Synchronisation mechanism allows a single thread to enter in a critical section of code for reading or writing. Readers/Writer lock differentiates accesses that change object state from the ones that just read state, allowing multiple simultaneous readers or one writer. Access for reading is allowed when no writers are executing or waiting to access object state whereas writers can access when there are no writing or reading operations executing ([16]). Readers/Writer lock fits well when access is predominantly for reading purposes. This assumes that methods in a class can be cleanly semantically separated into those that read variables values - i.e. *Readers* - and those that change such values – i.e. *Writers*.

@Reader and @Writer annotations describe reader and writer methods, respectively. Any of these annotations receives an optional *id* value, representing univocally the lock identity. A single lock is created per each *id*, enabling the association of methods from several classes to a specific lock in a similar way of synchronised mechanism. If *id* is not specified, the aspect uses a different lock per each object.

3.7. Scheduler

Usually, synchronisation mechanism implemented in OO languages - e.g. Java - is inflexible, as each monitor associated to an object restricts scheduling of threads in waiting state to monitor implementation. Scheduler pattern is independent of any scheduling policy, which is specified in a per case basis. Specification of a scheduling order can be determined dynamically accordingly to the state of the object and/or the method parameters passed on the call.

Scheduler can be engaged into applications by the use of annotations. Hence, @Scheduled can be used with enumerated parameter values, in the form:

```
@Scheduled(
    order=Order.[FIFO | LIFO |
            ordered| master | single],
    threadGroup=[thread-group name])
```

@Scheduled annotation receives the scheduling order by parameter. Using *FIFO*, threads are selected

to execute accordingly to the order they are queued, whilst *LIFO* selects threads in the inverse order. Using *ordered*, threads are selected in the order they were created. Ordered scheduling applies only to threads created by Oneway or Future annotations.

Master and single scheduling are similar to the equivalent OpenMP directives and can be used to force a block of code to be executed by a single thread or by the master thread (first spawned thread). This can be particularly useful in applications where concurrency annotations introduce speculative execution as a means to avoid additional synchronisation.

3.8. Thread Local

Thread local variables are instantiated in a per thread basis. Consequently the variable scope is always the thread. Each thread local variable should be annotated with @ThreadLocal with the following syntax:

```
@ThreadLocal(deepCopy=[yes|no],
     reduce=[operationId])
```

Each access to an object variable annotated with @ThreadLocal will be forward to a thread specific local variable. Each thread local object is initialised with the value of the object outside the thread local context (i.e., thread context), if the first thread access is a read operation. Otherwise the thread local value is not initialised, since the first thread access is for writing. If *deepCopy* parameter is specified with *yes*, each thread local object is a deep copy of the original object.

Some algorithms require the reduction of thread local variables in order to compute the object original variable. To perform reduction of variables after thread termination, the parameter *reduce* should be specified or a reduction function must be defined. In the former, *operationId* represents the preset reduction operation name, which can assume values like *sum*, *avg*, *min* or *max*. In the latter, the object to be reduced should define the operation that reduces that value with the following signature:

```
Object reductionOperation(
     Object tLocal, Object objVar)
```

Such method receives as parameter the thread local variable and the original object variable and returns the new object to be stored in the object variable.

An alternative to @ThreadLocal that may be more efficient is the @ThreadLocalObject, which applies to classes. In this case, threads use a local copy of the target object on each method invocation. It is equivalent to the use of the thread local annotation in all object fields. This renders into less execution overhead that @ThreadLocal since the thread local value is implemented by changing the target of each method call.

4. Implementation Overview

AspectJ 5 enables aspects to quantify over Java 5 annotations. In light of this, annotations have two roles: describe the elements (concurrent) behaviour and provide a hook for the aspects to compose the concurrent behaviour into the base program. Annotated elements are intercepted by the aspects which add the related behaviour at those points.

Mechanisms associated with annotations are implemented in abstract aspects. Each abstract aspect holds the code required to add the behaviour associated with each annotation. Further information about creating reusable code in AspectJ (e.g., AspectJ libraries) can be found in [19][20].

Figure 2 shows a simplified One-way implementation in AspectJ. *OnewayProtocol* aspect defines the pointcut *onewayAnnotation* which intercepts the execution of methods annotated with @Oneway. The *around* statement wraps the execution of each intercepted method (by means of the *proceed* keyword) inside a *Runnable* object and creates a new thread to execute such method.

```
public abstract aspect OnewayProtocol {
   protected pointcut onewayAnnotation():
         execution(@Oneway * *.*(..));

   void around(): onewayAnnotation(){
      Thread t = new Thread( new Runnable(){
            public void run(){ proceed(); }
         }
      );
      registerThread(t); // save to join later
      t.start();
   }
   ...// other advices and auxiliary methods
}
```

Figure 2. One-way implementation

Iterative algorithms can perform fine-grained method invocations. As the number of threads spawned by one-way is equal to the number of method invocations, the performance would be drastically reduced in such fine-grained methods. One-way executor presents considerable lower overheads, which can be further reduced by using task agglomeration (e.g., the chunk size). Figure 3 shows a simplified implementation of One-way Executor. Each method invocation is wrapped inside a *Runnable* object and inserted into a vector of objects created inside a

CompositeTask object (a composite of *Runnable* objects, see [21]). When the number of calls reaches the chunk size, the composite is submitted for execution by the executor service.

```
void around(): onewayChunk(){
   //...
   Runnable t = new Runnable(){
      public void run(){   proceed();  }
   };
   composite.add(t); // add call to composite
   currentCall++;
   if (currentCall==chunkSize) {  //
     registerTask(executor.submit(composite));
     currentCall=0;
     composite = new CompositeTask();
   }
}
```

Figure 3. Implementation of the OnewayExecutor.

5. Case Studies and Evaluation

This section presents several case studies and performance results obtained by parallelising several JGF benchmarks [22] using the proposed annotations.

The first case study is a financial simulation using Monte Carlo techniques to price products. A skeleton of the original code is presented in Figure 4. The application mainly consists on a *for* loop, where each iteration can be performed in parallel. At the end of each loop the result is accumulated in a *Vector*.

```
public void runSerial() {
   results = new Vector(nRunsMC);
   PriceStock ps;
   for(int iRun=0; iRun < nRunsMC; iRun++) {
      ps = new PriceStock();
      ps.setInitAllTasks(initAllTasks);
      ps.setTask(tasks.elementAt(iRun));
      ps.run();
      results.addElement(ps.getResult());
   }
}
```

Figure 4. Original Monte Carlo Simulation code.

To annotate the previous code we need to refactor the loop into a separate method, as well as the line that accumulates the results in the Vector. Figure 5 presents the resulting code, including annotations required to parallelise this application. Method *runIter* is executed in a separate thread, method *addResult* is declared synchronised to provide thread-safe access to the vector. At the end of the *runSerial* method, the main thread waits for the completion of all spawned threads.

```
@OnewayExecutor
public void runIter(Object allTasks,
                Object task, Vector results)
   PriceStock ps = new PriceStock();
   ps.setInitAllTasks(initAllTasks);
   ps.setTask(tasks.elementAt(iRun));
   ps.run();
   addResult(ps.getResult(),results);
}

@Synchronised
public void addResult(Object o, Vector res){
   res.addElement(o);
}

@JoinAfterExecution
public void runSerial() {
   results = new Vector(nRunsMC);
   for(int iRun=0; iRun < nRunsMC; iRun++) {
      runIter(initAllTasks,
            tasks.elementAt(iRun),results);}
}
```

Figure 5. Monte Carlo Simulation code after refactoring, including parallelism annotations

In the previous example the final result does not depend on the order in which results are added to the Vector. However, it is possible enforce the original sequential order with *ordered* scheduling (see 3.7).

The second case study from the JGF benchmarks is the ray tracer, which renders an image of sixty four spheres. The refactored code for this benchmark has a similar structure to code in previous case study, the main differences (see Figure 6) is that each thread that executes the *renderLine* call uses a different clone of the original RayTracer object (@ThreadLocalObject annotation) and that the checksum value must be reduced to a single value at the end of one-way calls.

```
@ThreadLocalObject(reduce=Reductions.SUM)
public class RayTracer {
  long checksum=0;

  @OneWay
  public void renderLine(/* … */)
    /*… */
  }

  @JoinAfterExecution
  public void render(Interval in) {
    /* … */
    for(int y=0; y < in.height; y++)
      renderLine(/* … */);
    /* …*/
  }
}
```

Figure 6. RayTracer code with parallelism annotations

Our last code example is an implementation of the classic Fibonacci recursive function (Figure 7). In this case study method *compute* is performed in a new

thread (e.g., Future method call) and local variables *r1* and *r2* are used to save future values.

```
public class Fib {

   @Future
   @FutureClient
   public long compute(long value) {
      if (value <=1) return(value);
      else{
         Fib f1 = new Fib();
         Fib f2 = new Fib();
         Long r1 = f1.compute(value-1);
         Long r2 = f2.compute(value-2);
         return (r1+r2);
      }
   }
```

Figure 7. Fibonacci code with parallelism annotations

6. Performance Evaluation

The performance evaluation aims to show that the benefits of the framework do not impose significant performance degradation, when compared with traditional thread-based programming. We compare the performance of our framework against reference implementations from the Java Grande Forum (Java multithreaded benchmarks [22]). Presented results were collected on a 2-way machine with Xeon 5130 2.0 GHz processors (Core2 processors with 4MB L2 cache), 4 GB RAM DDR2 533MHz, running CentOS 4.0, SUN JDK 1.5.0_3 in client mode and AspectJ Development Tool (AJDT) 1.4.0. All presented values are median of 5 runs.

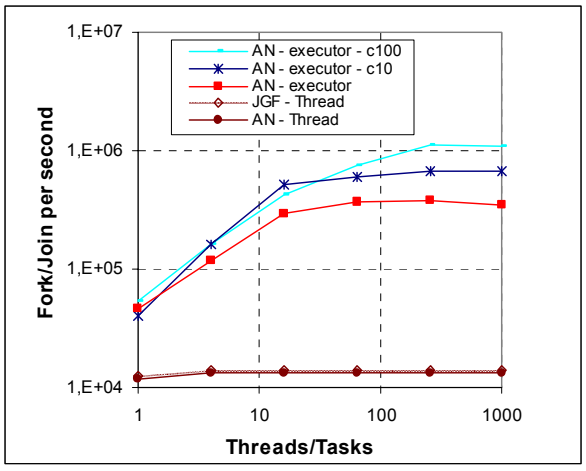

Figure 8. Annotations (AN) versus Threads (JGF) fork/joins per second

Figure 8 compares the number of spawned tasks per second in several implementations. The lowest two curves compare the traditional Java thread fork and join (using *start* and *join* methods of class Thread) and the *@Oneway* and *@JoinAfterExecution* annotations (*AN – Thread* curve). The results present the number of spawned tasks per second for a range of simultaneous threads (X-axis). The overhead, in this case, of the annotation framework is around 5% (74 µs versus 71 µs per fork/join).

When using the Java 5 task scheduling framework the overhead of thread creation drops more than one order of magnitude (3 µs, *AN executor* curve). Using chunk sizes (*AN executor c10* and *c100* curves) shows performance benefits for a high number of parallel tasks (respectively, 2 µs and 1 µs per tasks).

Table 1 presents high level benchmarks for several JGF applications: Monte Carlo (size B), Ray Tracer (size B) and SOR (size C).

	1	2	4	8
JGF - RayTracer	54.0	30.01	16.21	16.86
AN - RayTracer		30.24	16.29	17.46
AN - Overhead (%)	-	1	1	4
JGF - Monte Carlo	39.0	20.69	11.41	11.78
AN - Monte Carlo		20.89	11.49	11.85
AN - Overhead (%)	-	1	1	1
JGF - SOR	15.8	09.19	07.78	20.14
AN - SOR		10.00	08.14	08.17
AN - Overhead (%)	-	9	5	-

Table 1. Annotations (AN) versus JGF multithreaded benchmarks (MT) execution time

In Monte Carlo and RayTracer benchmarks the framework presents an overhead less than 5%. In both versions the Monte Carlo benchmark scales better due to higher memory locality (the Ray Tracer allocates many objects for intermediate calculations).

The SOR application presents fine grained OneWay method calls. The execution time is around 16 seconds for 100 iterations on a matrix of 2000x2000 (generating 400 000 one way calls, with an average execution of 40 microseconds per call). This justifies higher overhead of the annotation framework in this case (less than 10%) and the lower speed-up in this application. The JGF multithreaded version presents an unusual high overhead with more than 4 threads, due to the fine grained parallelism.

The small overhead of the annotation framework is due to aspect overheads to implement annotation semantics. The AspectJ compiler can, in most cases, inline the concurrency code into the original classes, replacing the annotation. However, in several cases the code must be placed into a separate class that is called

in the execution points were the annotation applies. Another source of overhead, more noticed in fine-grained methods, is the creation of a *Runnable* object to represent each one-way call. This second source of overhead is the main responsible for the overhead in the SOR benchmark.

7. Conclusion

This article presents an annotation framework to build parallel programs. In this framework it is possible to built parallel applications that are still valid sequential applications, if annotations are ignored (a goal similar to OpenMP). The provided annotations are fully integrated into the object oriented paradigm However, since many current applications do not fully exploit the object oriented paradigm, re-factorings may be required to make the base code amenable for annotations.

The framework provides a rich set of annotations to build efficient parallel applications. The current implementation, using Java 5 annotations and AspectJ, presents very low overheads, when compared with Java thread-based applications.

Current work includes the support for distributed memory systems and new annotations to support efficient execution on this type of machines.

8. References

[1] OpenMP architecture review board, OpenMP Application Program Interface, Version 2.5, May 2005, http://www.openmp.org

[2] Sun Microsystems, Inc. Java Specification Requests JSR 175: A Metadata Facility for the Java Programming Language.

[3] http://www.eclipse.org/aspectj/

[4] K. Löhr, Concurrency Annotations for Reusable Software, Communications of the ACM, vol. 36, no. 9, September 1993.

[5] M. Haustein, K. Löhr, JAC: Declarative Java Concurrency, Concurrency and Computation: Practice and Experience, vol. 18, no. 5, April 2006.

[6] R. Chandra, A. Gupta, J. Hennesy, COOL: An Object Based Language for Parallel Programming, IEEE Computer, vol. 27, no. 8, August 1994.

[7] F. Baude, L. Baduel, D. Caromel, A. Contes, F. Huet, M. Morel and R. Quilici, Programming, Composing, Deploying for the Grid, GRID COMPUTING: Software Environments and Tools, Jose C. Cunha and Omer F. Rana (Eds), Springer Verlag, January 2006.

[8] Y. Aridor, M. Factor, A. Teperman, cJVM: A Single System Image of a JVM on a Cluster, International Conference on Parallel Processing, Wakamatsu, Japan, September 1999.

[9] G. Antoniu, L. Bougé, P. Hatcher. , M. MacBeth, K. McGuigan, R. Namyst, The hyperion system: Compiling multi-threaded java bytecode for distributed execution, Parallel Computing, vol. 27, no. 10, September 2001.

[10] R. Veldema, R. Bhoedjang, H. Bal, Jackal, a compiler based implementation of java for clusters of workstations, ACM PPoPP'01, Utah, USA, June 2001.

[11] W. Zhu, C. Wang, F. Lau, JESSICA2: Distributed Java Virtual Machine with Transparent Thread Migration Support, IEEE Cluster 2002, Chicago, USA, September 2002.

[12] J. Bull, M. Kambites, JOMP—an OpenMP-like interface for Java, Proceedings of the ACM 2000 conference on Java Grande, California, June 2000.

[13] M. Klemm, R. Veldema, M. Bezold M. Philippsen, A Proposal for OpenMP for Java, Second International Workshop on OpenMP (IWOMP 2006), Reims, France, June 2006.

[14] M. Klemm, M. Bezold, R. Veldema, M. Philippsen, JaMP: An Implementation of OpenMP for a Java DSM, Proceedings of the 12th Workshop on Compilers for Parallel Computers, La Coruna, Spain, January 2006.

[15] Java 1.5 Specification, http://java.sun.com/j2se/1.5.0/docs/api/index.html

[16] D. Lea, Concurrent Programming in Java, Second edition, Addison-Wesley, 1999.

[17] D. Schmidt, M. Stal, H. Rohnert, F. Buschmann, Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects, Wiley & Sons 2000.

[18] A. Yonezawa, ABCL: an Object-Oriented Concurrent System, MIT Press, 1990.

[19] C. Cunha, J. Sobral, M. Monteiro, Reusable Aspect-Oriented Implementations of Concurrency Patterns and Mechanisms, AOSD'06, Bonn, Germany, March 2006.

[20] J. Hannemann, G. Kiczales, Design Pattern implementation in Java and in AspectJ, OOPSLA 2002, Seattle, USA, November 2002.

[21] E. Gamma, R. Helm, R. Johnson and J. Vlissides, Design Patterns – Elements of Reusable Object-Oriented Software. Addison-Wesley, 1994.

[22] J. Smith, J. Bull, J. Obdrzálek, A Parallel Java Grande Benchmark Suite, Supercomputing 2001 (SC'01), Denver, November 2001.

0-7695-2784-1/07 $20.00 © 2007 IEEE