APJava: An Aspect-Oriented Parallel Programming Model in Java

Hyeyeon Hong
Konkuk University
1 Hwayang-dong Gwangjin-gu
Seoul, Korea
+82-10-4559-1022
hyhong@konkuk.ac.kr

Hanku Lee
Konkuk University
1 Hwayang-dong Gwangjin-gu
Seoul, Korea
+82-2-2049-6082
hlee@konkuk.ac.kr

ABSTRACT
The paper introduces the APJava programming environment, called APJava. APJava is an aspect-oriented parallel dialect of Java that imports HPJava-like arrays – in particular the distributed arrays – as new data structures. The main purpose of APJava is to provide an easy-to-use aspect-oriented parallel programming environment to engineers and scientists unfamiliar with parallel programming. The paper discusses an overview of APJava and pre-translation scheme and basic translation scheme adopted in a translator for the APJava language.

Categories and Subject Descriptors
D.3.3 [Programming Techniques]: Concurrent Programming – Parallel Programming

General Terms
Languages

Keywords
Parallel Programming, Aspect-Oriented Programming, Distributed Computing, Translation Scheme, Java.

1. INTRODUCTION
In complex parallel programs it is easy to seek many crosscutting concerns, called aspect, since classical procedural programming and object-oriented programming are unable to modularize all concerns even though they are very useful. For instance parallel attributes (e.g. a number of processors, dimensions of the grid of processors, attributes of distributed arrays, collective communication libraries, etc.) are crosscutting concerns. These aspects are very useful information for parallel programs but independent on compile-time and run-time information for parallel compilers. Whenever these aspects are modified (e.g. a number of processors are changed), complex parallel programs are re-compiled since most of classical parallel programming techniques are not interested in these useful aspects. Aspects make it possible to implements crosscutting concerns in a modular way and easy to develop and maintain.

This paper proposes an easy-to-use programming model for aspect-oriented parallel programming environments, especially suitable for programming massively parallel and distributed memory computers. A Java implementation of the model is Aspect-Oriented Parallel Java, called APJava. Several of the ideas in APJava are lifted from the High Performance Java (HPJava) programming language [1, 4, 5, 7, 8, 9, 10, 11, 12]. Also the programming model of APJava adopts the aspect-oriented programming (AOP) model.

The main purpose of APJava is to provide an easy-to-use aspect-oriented parallel programming environment to engineers and scientists unfamiliar with parallel programming. Even an entry level parallel programmer can easily implement a parallel code in APJava since APJava programs look like a sequential Java code with some grammar extensions such as distributed arrays, on constructs, and forall construct. The programmer also implements an XML-based aspect code to separate aspects from the parallel program.

The paper introduces pre-translation scheme and basic translation scheme in order to build well-defined an APJava compiler. For the moment the APJava compiler is a source-to-source translator and fully dependent upon the HPJava compiler. That is, the compiler translates a source code of APJava into an HPJava code.

Section 3 describes an overview of APJava. Section 4 depicts pre-translation scheme and basic translation scheme adopted in a translator for the APJava language and section 5 concludes.

2. RELATED WORKS

2.1 HPJava
HPJava [1, 4, 5, 7, 8, 9, 10, 11, 12] is an environment for parallel and scientific programming with Java, developed by the present authors. It provides a programming model with various similarities to—and some notable differences from—the programming model of High Performance Fortran, (HPF) [2, 3]. In comparison with HPF, HPJava has a stronger emphasis on the role of libraries that operate on distributed data. This emphasis sits naturally with the object-oriented features of the base
language—Java rather than Fortran. In other respects HPJava provides a lower-level programming model than HPF—with multiple SPMD threads of control. But it retains important distributed data abstractions from HPF. It is most specifically aimed at distributed memory architectures, although of course HPJava can also run on shared memory computers. We call our general programming model the HPspmd model—so HPJava is a realization of the HPspmd model.

An initial release of the HPJava environment was placed in the public domain in 2003. This includes a reasonably sophisticated translator implementing the full HPJava language (which includes the full Java language as a subset), and a set of supporting libraries. In earlier publications we argued that HPJava should eventually provide acceptable performance, making it a practical tool for HPC [4]. We showed in [5] that HPJava node performance is quite reasonable, compared with C, Fortran, and ordinary Java: in particular on many platforms Java is no longer much slower than C or Fortran. Thus we verified our library-based HPspmd programming language extensions can be implemented quite efficiently in the context of Java.

In part because the SPMD programming model is relatively loosely coupled (compared with HPF), and in part because of its foundation in a ubiquitous Internet programming language (Java), we can make the case that our HPspmd programming model provides promising programming support for high-performance grid-enabled environments.

The Range class describes a distributed index range. There are subclasses describing index ranges with different properties. In this example, we use the BlockRange class, describing block distributed indexes. The first argument of the constructor is the global size of the range; the second argument is a process dimension—the dimension over which the range is distributed. Thus, ranges x and y are distributed over the first dimension (i.e. \( p.\text{dim}(0) \)) and second dimension (i.e. \( p.\text{dim}(1) \)) of \( p \), and both have \( N \) elements.

The most important feature HPJava adds to Java is the multiarray. There are two kinds of multiarray in HPJava: distributed arrays and sequential multidimensional arrays. A distributed array is a collective array, shared by a number of processes. Like an ordinary array, a distributed array has some index space and stores a collection of elements of fixed type. Unlike an ordinary array, the index space and associated elements are scattered across the processes that share the array. A sequential multi-dimensional array is similar to an ordinary Fortran array. Like in Fortran (but unlike ordinary Java arrays) one can form a regular section of a multiarray.

2.2 Aspect-Oriented Programming

Aspect-oriented programming (AOP) is a technique for improving separation of concerns in software. AOP is based on classic programming skills such as procedural programming and object-oriented programming that have already made significant improvements in software modularity. The main idea of AOP is the hierarchical modularity mechanisms of object-oriented languages are inherently unable to modularize all concerns of interest in complex systems while they are extremely useful. Instead, in the implementation of any complex software, there will be concerns that crosscut the natural modularity of the rest of the implementation.

The main functionality of AOP is crosscutting concerns (while OOP has done for object encapsulation and inheritance). This makes it possible to program crosscutting concerns in a modular way, and achieve the usual benefits of improved modularity: simpler code that is easier to develop and maintain, and that has greater potential for reuse. A well modularized crosscutting concern is called an aspect. AspectJ [6] is a simple and practical aspect-oriented extension to Java.

3. THE APJAVA LANGUAGE

APJava is a Java implementation of what we call the aspect-oriented parallel model. This is a flexible hybrid of HPJava-like data parallel features and the popular SPMD style. The main purpose of APJava is to provide an easy-to-use aspect-oriented parallel programming environment to engineers and scientists unfamiliar with parallel programming. Even an entry level parallel programmer can easily implement a parallel code in APJava since APJava programs look like a sequential Java code with some grammar extensions such as distributed arrays, on
constructs, and forall construct. The programmer also implements an XML-based aspect code to separate aspects from the parallel program.

```java
Block p = new Block("basename.xml");
on(p) {
    int[[-,-]] a = new int[[p.dim(0),p.dim(1)]];
    int[[-,-]] b = new int[[p.dim(0),p.dim(1)]];
    int[[-,-]] c = new int[[p.dim(0),p.dim(1)]];
    ... initialize 'a', 'b'
    forall(int i=0; ; ; p.dim(0))
        forall(int j=0; ; ; p.dim(1))
            c[i, j] = a[i, j] + b[i, j];
}
```

Figure 2. A Parallel Matrix Addition in APJava

```
<parallel name = "p">
    <dimension> 6 <dimension>
    <proc kind = "procs2">
        <num> 8 </num>
        <dim0> 2 </dim0>
        <dim1> 3 </dim1>
        <dist>
            <range> BlockRange </range>
        </dist>
    </proc>
</parallel>
```

Figure 3. Aspects in XML

Figure 2 is a basic APJava program for a matrix addition. It includes much of the APJava special syntax. The program starts by creating an instance p of the class Block. This is a class that gets aspects from a XML-based file. There is an example in Figure 3. Aspects of the above program are dimensions of the grid of processors, a number of processors, attributes of distribution for distributed array. When the instance of Block is created, a number of processes in which the parallel program is executing.

A Block object p can parameterize an on construct. The on construct limits control to processes. The code in the on construct is only executed by processes that belong to p. The on construct fixes p as the active process group within its body.

The most important feature APJava adds to Java is the distributed array\(^1\), adopted from HPJava. A distributed array is a collective array, shared by a number of processes. Like an ordinary array, a distributed array has some index space and stores a collection of elements of fixed type. Unlike an ordinary array, the index space and associated elements are scattered across the processes that share the array. The type signature of a distributed array is clearly distinguished by double brackets. In the type signature of a distributed array each slot holding a hyphen, -.

The forall construct is another control construct of APJava. It represents a distributed parallel loop, sharing some characteristics with the overall construct of HPJava. The integer, i, scoped by the forall construct is called a distributed index. Moreover the forall construct needs the size of each dimension, steps, and an attribute of distribution, which are optional.

4. TRANSLATION SCHEME

4.1 Pre-translation

The APJava translator has two translation phases. The first pre-translation decrease the input APJava program to some other, equivalent APJava program, coded in a restricted subset of the full APJava language. The pre-translator reduces complex expression to simpler expressions.

A program is in restricted form if it respects certain limits on the complexity of expressions involving distributed arrays.

Simple expressions are

1) Constant expressions

2) Local variable references

4) The keyword this

The restrictions on expressions are

1) A composite distributed array expression may appear as the right-hand-side of a top-level assignment. Also note that a variable declaratory with an initializer is not an assignment expression, so the restricted form does not allow composite distributed array expressions to appear as initializers in declarations.

2) All of the following must be simple expressions:
   A. The target object in accesses to distributed array-valued fields
   B. The array operand in array-restriction expression.
   C. The range and group parameters of distributed array creation expressions
   D. The range expression in the header offorall

In principle reducing a program to restricted form is a straightforward matter of precomputing subexpressions that break the rules above, and saving their values in temporary variables introduced by the pre-translator. Figure 4 illustrate the process of expression simplification.

---

\(^1\) A sequential multidimensional array is NOT supported yet in APJava.
Prior to pre-translation:

```java
float [[-,-]] v = new float [[b.dim(0), b.dim(1)]]
```

After pre-translation:

```java
float [[-,-]] v;
v = new float [[b.dim(0), b.dim(1)]]
```

where `b` is an expression and `S` is a statement.

INITS and `b'` are the results of applying the simplify algorithm to `b`. If `b' = b`, i.e. no simplification is required. The transformed version is

```java
{ INITS on (b') S'
}
```

4.1.4 forall statements
Consider the statement

```java
forall (int i = e0; e1; e2; x) S
```

The `forall` statement follows the same pattern. `x` is an expression. Again let INITS and `x'` be the result of applying the simplify algorithm to `x`. `S'` is the pre-translated version of `S`. The transformed version is

```java
{ INITS
  forall (int i = e0; e1; e2; x') S'
}
```

4.2 Basic Translation

4.2.1 Translation variable declarations

Consider the Variable declarations

```
T  V0  I0 , … , Vn-1  In-1;
```

where `T` is a Java type, each term `Vi` is a simple identifier, `vi`, followed by zero or more bracket pairs, `[]`, and each optional initializer clause `Ii` is either empty or a term `" = hi"`, where `hi` is an expression or an array initializer. We must have `n ≥ 1`.

If `T` is a distributed array type, we unconditionally break the declaration into a series of transformed, single-variable declarations,

```
Di (D0 … Dn-1)
T  Vi
```

When `h` is an array initializer, `e` is just equal to `h`. If `h` is an array initialize, `e` is the array creation expression:

```
new Ui hi
```

The enhancement brought forth by APJava concerns the distributed arrays syntax. These are distinguished in HPJava by double-brackets, so they are not confused with classic Java arrays.
variable \( a \) is a name of array in the source program. Each \( a_n \) are the extent of dimension \( n \), for all integers \( n > 0 \), and expressed in a single hyphen,-. A function, \( T[e] \), on expression terms returns the result of translating an expression \( e \). Each \( attr_n \) are expressed in a single hyphen.

### 4.2.2 Translation distributed array creation

SOURCE:

\[
a = \text{new} \ T[[e_0, \ldots, e_{n-1}]] \ bras;
\]

TRANSLATION:

\[
\begin{align*}
\text{Range } x_0 &= \text{new} \ \text{Range}_\text{type} \ (i, b.\text{dim}(0)); \\
\text{Range } x_{n-1} &= \text{new} \ \text{Range}_\text{type} \ (i, b.\text{dim}(n-1)); \\
T[a] &= \text{new} \ T[ [T[e_0], \ldots, T[e_{n-1}]]] \ bras;
\end{align*}
\]

where:
- \( T \) is a Java type,
- \( n \) is the rank of the created array,
- each \( e_i \) is a simple expression in source program,
- the expression \( a \) is the assigned array variable in the source program,
- the term \( bras \) is a string of zero or more bracket pairs, [],
- \( \text{Range}_\text{type} \) is some distribution format of HP Java,
- the expression \( x_n \) is the Range variable,
- \( i \) is the total number of processors,
- \( b.\text{dim()} \) is the range of a dimension.

Figure 6 gives a scheme for translating a distributed array creation in the source APJava program.

Each \( e_n \) is the extent of dimension \( n \). These extents are returned by a method call on an object of the special class Block, representing the grid of processors. They are either the range of a dimension or the full size of the distributed array being defined. In the former case, \( e_n \) has the shape \( b.\text{dim}(n) \). \( i \) is the total number of processors. Range variables \( x_n \) are fresh, i.e., they are not bound in the scope at their point of definition. \( \text{Range}_\text{type} \) is some distribution format of HP Java, as BlockRange, CyclicRange, ExtBlockRange, etc.

### 4.2.3 Translation on constructs

A translation for the on construct is given in Figure 7. The basename.xml is an XML file, called the aspect file, containing the specification of the grid of processors. The latter case of extent is \( p.\text{size} \).

The processor kind is \( P \), \( d_n \) are the dimensions of the grid of processors, all these numbers and processor kind being found in the accompanying XML file aspect.xml. \( S \) is a statement in the source program. The translation function for statements, \( T[S/b] \), translates the statement or block \( S \) in the context of \( b \). \( i \) is the total number of processor.

It is possible to declare distributed arrays both outside on construct and inside on construct, but distributed array was created only inside on construct.

For instance, we can write

Block \( p = \text{new} \ \text{block} \ (\text{"aspect"}); \)

\[
\begin{align*}
\text{on} \ (p) \ {\{ \ \\
\quad \text{float} \ [\ldots] \ v &= \text{new} \ \text{float} \ [p.\text{dim}(0), p.\text{dim}(1)]; \\
\quad \text{int} \ [\ldots] \ w &= \text{new} \ \text{int} \ [p.\text{dim}(0), p.\text{dim}(1)]; \\
\} \}
\end{align*}
\]

The semantics of this new construct is specified by means of a source-to-source translation to HP Java. The previous example actually means

Procs \( p = \text{new} \ \text{Procs2} \ (d_0, \ldots, d_{n-1}); \)

\[
\begin{align*}
\text{on} \ (p) \ {\{ \ \\
\quad \text{Range} \ x &= \text{new} \ \text{BlockRange} \ (N, p.\text{dim}(0)); \\
\quad \text{Range} \ y &= \text{new} \ \text{BlockRange} \ (N, p.\text{dim}(1)); \\
\quad \text{float} \ [\ldots] \ v &= \text{new} \ \text{float} \ [x, y]; \\
\quad \text{int} \ [\ldots] \ w &= \text{new} \ \text{int} \ [x, y]; \\
\} \}
\end{align*}
\]

SOURCE:

\[
\text{Block } b = \text{new} \ \text{Block} \ (\text{basename}); \\
\text{on} \ (b) \ S
\]

TRANSLATION:

\[
\begin{align*}
\text{Procs } b &= \text{new} \ P \ (d_0, \ldots, d_{n-1}); \\
\text{on} \ (b) \ T[S/b]
\end{align*}
\]

where:
- \( b \) is the name of Block class object,
- \( d_n \) is the dimensions of the grid of processors,
- \( p \) is a kind of processor,
- \( S \) is a statement in the source program.

Figure 7. Translation of on construct

### 4.2.4 Translation forall constructs

The forall construct is control construct of APJ. A translation for the forall construct is given in Figure 8.
The initial value of loop is expressed as integer \( e_0 \). The upper bound \( e_1 \) is extent of the range. \( e_2 \) is step value. The term \( e_1 \) and \( e_2 \) are optional. \( x \) is distributed Range variable. The variable \( i \) is global index in the source program. \( S \) is a statement in the source program. Terms like \( T[e] \) represent the translated form of expression \( e \).

**SOURCE:**

\[
\forall (\text{int } i = e_0; e_1; e_2; x) \ S
\]

**TRANSLATION:**

\[
\text{overall } (i = T[x] \mid T[e_0] : T[e_1] : T[e_2])\]

\[
T[S/b]
\]

where:
- \( i \) is a global index variable name in the source program,
- \( e_0, e_1, e_2 \), and \( x \) are simple expressions in the source,
- \( S \) is a statement in the source program.

**Figure 8. Translation of forall construct**

### 4.2.5 Translation single element access

**SOURCE:**

\[
[a[e_0, ... , e_{n-1}], e]
\]

**TRANSLATION:**

\[
[a[e_0, ... , e_{n-1}], e] = T[e]
\]

where:
- the expression \( a \) is the assigned distributed array variable in the source program,
- each \( e_n \) is an simple expression in source program,
- the expression \( e \) is an optional initialize in the source program,
- \( i_n \) is index name in the source program.

**Figure 9. Translation of single element access**

We need to update or access a single element of a distributed array (rather than accessing a whole set of elements in parallel). A translation for the single element access is given in Figure 9.

\( a \) is distributed array name. This is distinguished between classic Java array and the distributed array of APJava by form of brackets ([ , ]). Each \( e_i \) is index of array. If the array is distributed array and the indexes are integers, like \( a[1,2] \), then keyword ‘at’ is add. After the value of Range variable \( x, y \) corresponding to element of array \( a \) assigned to \( i_n \), update or access the value of distributed array \( a \).

For instance, we can write

\[
\text{int } [[-,-]] a = \text{new int } [[b.dim(0), b.dim(1)]]; \\
\text{...} \\
\text{a [1,2] = 7;}
\]

The semantics of this new construct is specified by means of a source-to-source translation to HP Java. The previous example actually means

\[
\text{at } (i = x [1]) \\
\text{at } (j = y [2]) \\
\text{a [i , j] = 7;}
\]

**4.2.6 Translation distributed element access**

**SOURCE:**

\[
a[e_0, ... , e_{n-1}] = e
\]

**TRANSLATION:**

\[
a[e_0, ... , e_{n-1}] = T[e]
\]

where:
- the expression \( a \) is the assigned distributed array variable in the source program,
- each \( e_n \) is an simple expression in source program,
- the expression \( e \) is an optional initialize in the source program.

**Figure 10. Translation of distributed element access**

Figure 10 gives a scheme for translating distributed element access.

If \( e_n \) are global index variables and element access exist in forall construct, like a \([i,j]\) , then expression \( e \) translated into \( T[e] \).

### 4.2.7 Translation global index expression

**SOURCE:**

\[
e = i
\]

**TRANSLATION:**


Figure 11. Translation of global index

Figure 11 gives a scheme for translating a global index expression. 

\( T[e] \equiv i' \)

where:

- \( i \) is an global index variable name in the source program.

5. CONCLUSION

In parallel programs it is easy to find many crosscutting concerns, called aspect, since classical procedural programming and object-oriented programming are unable to modularize all concerns even though they are very useful. These aspects are very useful information for parallel programs but independent on compile-time and run-time information for parallel compilers.

Thus, in this paper, we proposed an easy-to-use programming model for aspect-oriented and parallel programming environments, especially suitable for programming massively parallel and distributed memory computers, called the APJava language.

The paper introduced pre-translation scheme and basic translation scheme in order to build well-defined an APJava compiler. For the moment the APJava compiler is a source-to-source translator and fully dependent upon the HPJava compiler. That is, the compiler translates a source code of APJava into an HPJava code.

6. ACKNOWLEDGMENTS

This paper is supported by Seoul R&BD program in 2009. Professor Hanku Lee is the corresponding author. hlee@konkuk.ac.kr, +82-2-2049-6082.

7. REFERENCES


