The Deterministic Parallel Java Tutorial
Version 1.1

Revised January 2013

Contents

1 Introduction

Deterministic Parallel Java (DPJ) is an extension to the Java programming language that guarantees *determinism* for programs written with fork-join parallelism. That is, if a DPJ program compiles without errors or warnings, then it will produce the same output on every execution of the program. DPJ achieves its determinism guarantee by enforcing *noninterference*: DPJ guarantees that for any two parallel tasks, there are no pairs of *conflicting accesses* (i.e., accesses to the same memory location, with at least one a write) performed by the tasks.

DPJ's determinism guarantee provides a significant benefit over traditional models such as Java threads. In such models, conflicting memory accesses and even data races can occur, so the behavior of the program can vary depending on the interleaving of operations from different threads on each execution. Such nondeterminism makes many parallel programs difficult to reason about and hard to debug.

The purpose of this tutorial is to bring you up to speed quickly using DPJ. Section 2 gives an overview of major DPJ concepts, so you can start writing programs as quickly as possible. It provides links to *The Deterministic Parallel Java Language Reference Manual* (e.g., Reference Manual § 1) so you can look up particular features there to get more detail. The rest of the sections show how to write common parallel patterns in DPJ. We believe that a very good way to get started with DPJ is to (1) look for patterns in this guide that are the same as or similar to patterns occurring in your parallel algorithm; (2) study and understand the relevant pattern or patterns in this guide; and (3) adapt the pattern or patterns to your needs.

Throughout this tutorial are DPJ code examples. Some are fragments of larger programs, but many are short, self-contained programs. We recommend that, before reading this tutorial, you download and install DPJ. See *The Deterministic Parallel Java Installation Manual* for instructions on how to do it. Then, as you read this tutorial, you can compile and run the examples. The examples in this tutorial are located in the directory `Documentation/v1.1/Source/Tutorial/Programs` of the DPJ release at `github.com/dpj/DPJ`. They may be compiled and run from that directory.

2 Overview of DPJ

This section provides an overview of DPJ concepts, to get you started quickly using the language. References to *The Deterministic Parallel Java Language Reference Manual* are provided throughout.

2.1 Basic DPJ Concepts

DPJ is based on Java. If you know Java, then you know most of DPJ already. In fact, every legal Java program is a legal DPJ program! However, the reverse is not true, because DPJ adds several features to Java to guarantee determinism for parallel programs. We'll start with the DPJ equivalent of "Hello, world" to explain the most basic DPJ features.

2.1.1 A Simple DPJ Point Class

At the most fundamental level, DPJ adds just three concepts to Java: explicit fork-join parallelism, regions, and effects. Figure 1 shows a simple DPJ class `Point` that illustrates the concepts.

Explicit fork-join parallelism: The DPJ language syntax shows explicitly where a parallel code section begins (a fork point) and ends (the join point). At a join point, execution is suspended until all the parallel tasks created at the corresponding fork point have finished executing. In Figure 1, the `cobegin` statement (lines 10–13) executes its component statements (lines 11 and 12) in parallel. The fork point is at the start of the `cobegin` statement, and the join point is at the end of the statement. Execution does not continue after the `cobegin` statement until lines 9 and 10 have both completed execution.

DPJ's fork-join model makes the structure of the parallelism explicit, in contrast to an unstructured model such as Java threads. Together with the determinism guarantee, this structure allows the programmer to reason about the behavior of the parallel program as if it were the equivalent sequential program with the parallel constructs elided. For example, DPJ guarantees that the code shown in Figure 1 will compute the same results as the code in which the `cobegin` statement in lines 10–13 were replaced with an ordinary sequential statement block. Thus we say that DPJ has a sequential *correctness model* and a parallel *performance model*.

Regions: To facilitate the analysis of reads and writes to the heap, DPJ groups memory locations into named sets called *regions*. The programmer explicitly partitions the heap into regions by marking class fields and array cells with region information. In Figure 1, line 2 declares region names `X` and `Y`. Lines 3 and 4 say that field `x` is located in region `X` and field `y` is located in region `Y`. Two or more variables may be grouped in the

```
1  class Point {
2      region X, Y;
3      double x in X;
4      double y in Y;
5      void setX(double x) writes X { this.x = x; }
6      void setY(double y) writes Y { this.y = y; }
7      void setXAndY(double x, double y)
8          writes X, Y
9      {
10         cobegin {
11             this.setX(x);
12             this.setY(y);
13         }
14     }
15 }
```

Figure 1: A simple DPJ class to illustrate basic concepts.

same region. For example, it would be permissible to add a field `double z in Y` to class `Point`. However, every variable always resides in exactly one region.

Effects: The DPJ compiler infers the read and write effects of each statement in the program in terms of regions. For example, in line 5 of Figure 1, the effect of the statement `this.x = x` is `writes X`, because the statement assigns a value to variable `x`, which is declared to be in region `X` (line 3).

To make the compiler analysis simple and modular, and to document the effects at API boundaries, the DPJ programmer writes *effect summaries* on methods that specify their read and write effects in terms of regions. Examples of effect summaries are shown in lines 5, 6, and 8 of Figure 1. The compiler checks that every method's summary includes all the actual effects of the method body (including the effects of methods called in the body, using those methods' summaries to compute the effects of the invocations). For example, in line 8, it would be a compile-time error to omit the effect `writes Y` from the summary of `setXAndY`, because of the write to region `Y` in line 12.

The compiler uses the inferred statement effects and method effect summaries to compute the effects of parallel tasks, and check that all pairs of parallel tasks are mutually noninterfering. For example, the effect of line 9 is `writes X`, and the effect of line 10 is `writes Y`. Because `X` and `Y` name different regions, the two writes are to disjoint memory locations, so the parallel tasks in the `cobegin` statement are safe to run in parallel.

Some effects don't need to be reported in a method summary. These are:

- Any read or write of a local variable (i.e., a variable declared inside method scope, including method formal parameters). For instance, the read of parameter `x` in line 10 doesn't generate any effect that needs to be reported. That's because Java doesn't allow you to take the reference of a local variable, so the compiler can confirm noninterference for local variables without any help from the programmer.
- Any effect on a `final` variable. That's because the value of a `final` variable never changes after it is initialized, so reading it cannot cause interference.
- Any initialization effect on fields of `this` by a constructor. That's because in DPJ, other parallel tasks can never see the constructed object until all the initialization is done.

2.1.2 Compiling the Point Class

Try compiling the `Point` class shown in Figure 1. If your DPJ environment is set up correctly, it should compile with no errors or warnings.

Now try modifying the example a bit to see how DPJ responds to the changes. Playing around with programs like this is one of the best ways to learn new language features.

First, delete `in X` at the end of line 3, and recompile. You should get an error that the effect summary `writes X` in line 5 doesn't cover the effects of the method `setX`. That's because when you take out the explicit declaration `in X`, the variable `x` gets put in a default region called `Root`. Now replace `writes X` in line 5 with `writes Root`. The code should compile again without errors.

Now delete the effect summary in line 5 entirely. This time the code should compile, but it gives a warning. Since you didn't give any explicit effect summary for `setX`, the compiler assumes the most conservative effect for that method — in effect, "writes the entire heap." The code compiles, because the assumed effect covers the effects of `setX`. However, the assumed effect conflicts with the effect `writes Y` of `setY`. So the compiler is warning you that there is potential interference between lines 11 and 12 in the `cobegin`.

Please notice three things about this example. First, there really isn't any interference; the compiler is just warning you because you haven't satisfied all its rules for *proving* there is no interference. This may seem like a minor point for this trivial class, but for a large and complex program, proving determinism can be quite tricky. So following the rules is important, because it gives a *guarantee* of correctness that would otherwise be hard to show.

Second, notice that the DPJ compiler gives an error when your declared method effects are too small, but a warning when your parallel effects are interfering. This behavior is so that you can write a DPJ program incrementally, without getting all the annotations right on the first go. For example, you could have written the class in Figure 1 by putting in the `cobegin` first, leaving out the region and effect annotations entirely. That code would compile, but it would give a warning. Then you could add the region and effect annotations incrementally, until the warnings go away. This is usually a good strategy for writing programs in DPJ. Writing bad method effects causes an error (not just a warning) because it is both easier to deal with (if you write nothing you get the default effect, which always works) and less localized — a bad method effect in one place could cause interference to be underreported in a different one.

Third, the defaults are designed so that you don't have to worry about region and effect annotations for code that is never called inside a parallel task. You only need to annotate code that is going to run in parallel.

2.1.3 Where To Learn More

The Reference Manual has more detailed information on the topics discussed above. In addition to `cobegin` for parallel statements, DPJ has a `foreach` construct for parallel loop iterations. See Reference Manual § 6 for more details. Reference Manual §§ 2.1 and 2.2 have more information on declaring and using regions. For further information about summarizing and checking effects, see the following sections of the Reference Manual:

- §§ 2.3.1, 2.3.2, and 6.2 explain how to write method effect summaries.
- § 6.4 explains how the compiler computes a method's actual effects.
- § 6.6 explains how the compiler compares a method's actual effects to its summarized effects.
- § 6.7 explains noninterference of effect.

It may be best to read at least the rest of this section of this tutorial before consulting these sections of the reference manual, because the full explanation of these features requires concepts (such as region path lists and region parameters) introduced in later subsections here. However, the Reference Manual contains plenty of cross references (hyperlinked in the HTML version), so you can also just start reading the Reference Manual, following cross references when you don't understand something, if you prefer learning that way.

2.2 Region Path Lists

Every region in DPJ is described by a *region path list*, or RPL. A basic region name, shown in § 2.1.1, is a particular kind of RPL. More generally, a *sequence of names* separated by colons is an RPL. For example, if `A` and `B` are declared region names, then `A`, `B`, `A:B`, `A:A:B`, etc., are RPLs. RPLs can be *partially specified*, by writing a `*` in place of some sequence of names. For example, `A:*:B` stands in for `A:B`, `A:A:B`, `A:B:B`, `A:A:A:B`, etc.

Nesting and partial specification are useful for describing *sets of regions* in effects. For example, in conjunction with region parameters (§ 2.3), RPLs let you write an effect that says "writes all nodes in the left subtree."

Here is a simple example of RPLs in action:

```
1  class RPLExample {
2      region A, B, C;
3      int x in A:B;
4      int y in A:C;
5      void method(int x, int y)
6          writes A:*
7      {
8          this.x = x;
9          this.y = y;
10     }
11 }
```

We have declared region names `A`, `B`, and `C` (line 2), and we use them to construct the RPLs `A:B` and `A:C`. We put field `x` in region `A:B` and field `y` in region `A:C` (lines 3–4). In line 6, we use the partially specified RPL `A:*` to cover both `A:B` and `A:C`. Note in particular that *`A` and `A:B` are different regions*. So `writes A` does *not* cover a write to `A:B`; the `*` is needed to get the inclusion. The reason is that to get more precision with RPLs, DPJ separates the concepts of *nesting* (`A:B` is nested under `A`) and *inclusion* (`A:B` is included in `A:*` but not in `A`). It may help to think of RPLs like a file system. For example, given the UNIX path `foo/bar`, the file system is a tree with `bar` nested under `foo`; but `rm foo` won't remove `bar`; you have to say either `rm foo/bar` or `rm foo/*`.

RPLs are useful in several ways:

- Covering effects in method effect summaries, as shown in the example above.
- Distinguishing different sets of regions from each other. For example, `A:*` is distinct from `B:*`, and `*:A` is distinct from `*:B`. This technique is useful for proving noninterference.
- Describing partially specified types, to allow flexible assignments to variables with region arguments in their types. See the next section of this tutorial for more information.

Later sections of this tutorial give more examples of using RPLs. For the full details, see §§ 3 (RPLs), 5 (Types), and 6 (Effects) of the Reference Manual.

2.3 Class and Method Region Parameters

Class region parameters: DPJ allows you to write classes with region parameters. This works similarly to Java generic type parameters and allows you to have different instances of the same class with the fields in different regions.

Here is a simple example to illustrate class region parameters:

```
1  class Data<region R> {
2      int x in R;
3  }
4  class DataPair {
5      region First, Second;
6      final Data<First> first;
7      final Data<Second> second;
8      // Constructor initialization effects don't have to be reported
9      // See Reference Manual s. 2.3.2
10     DataPair(Data<First> first, Data<Second> second)
11         pure
12     {
13         this.first = first;
14         this.second = second;
```

```
15      }
16      void updateBoth(int firstX, int secondX) {
17          cobegin {
18              // Effect is 'writes First'
19              first.x = firstX;
20              // Effect is 'writes Second'
21              second.x = secondX;
22          }
23      }
24  }
```

Lines 1–3 define a simple `Data` class with an integer field `x`. The class has one *region parameter*, `R`, and field `x` is located in `R`. That means that when an object of type `Data<r>` is created, for some actual region `r`, its `x` field will reside region `r`.

The rest of the program is a simple class `DataPair` which stores two `Data` objects in fields `first` and `second` (lines 6–7). Note that we have made the fields `final`. That is because we are not going to assign into the fields themselves, as in Figure 1; instead we are going to read the fields and assign into the objects they refer to. We do this in method `updateBoth`. In line 19, the effect is `writes First`. That's because the write is to field `x`, `x` is in region `R` (line 2), and `R=First` in the type of the selector `first`. See Reference Manual § 2.4.1 for more information on defining class region parameters, and Reference Manual § 4 for more information on using parametric classes as types. See Reference Manual § 6.4 for more information on how the compiler computes the effect of a field access through a parametric type.

Method region parameters: Methods can also have region parameters in DPJ. Method region parameters allow the same method to be invoked with different regions. See Reference Manual § 2.4.2.

Disjointness constraints on parameters: You can put disjointness constraints on class and method parameters. This allows you, for example, to require that two parameters coming into a class or method be different names. The requirement is checked by the compiler at the point where the region arguments are given to the parameters. Inside the class or method definition, the compiler can use the disjointness constraint to prove noninterference. For example, if two parameters are constrained to be disjoint, then effects on them are disjoint. See Reference Manual § 2.4.3 for more details.

Assignment restrictions: In order to ensure sound reasoning about types, some assignments aren't allowed when two types with the same class have different region arguments. The rules are in Reference Manual § 5.1.2. Sometimes this means you can't do an assignment you want to do. For example, if variable `a` has type `C<A>` and `b` has type `C<B>`, you can't say `b=a`. In this case, you have three options:

1. Use partially specified RPLs (Reference Manual § 3.4) and/or additional region parameters to express the assignment. DPJ's subtyping features are fairly powerful, so often you can do this. For example, if you give `b` the type `C<*>`, now you can assign `a` to it, as well as `C<B>` objects. However, the `*` causes the compiler to infer coarser information about effects. So this strategy doesn't always work.

2. Clone an object with the new type. For example, `b = a.<B>clone()`. Here `clone` is a generic method that takes as an arguments the region parameter arguments of the outgoing object, and clones `this` with the new type:

   ```
   <region Out>C<Out>clone() {
       C<Out> result = new C<Out>();
       // Copy fields from this to result
       ...
       return result;
   }
   ```

   This strategy adds some overhead, which is a price of DPJ's determinism guarantee in this case.

3. As a last resort, use a type cast (Reference Manual § 5.1.3). For example, `b = (C<B>) a`. This is usually not a good option, because it breaks DPJ's determinism guarantee.

2.4 Arrays

Arrays are important in parallel programming, so DPJ has several features to support them.

2.4.1 Array Classes

DPJ v1.1 introduces a feature called an *array class* to support parallel programming with arrays. An array class is like a Java class with a single field, except the field has no name and it defines the type of an array cell. For example, the following array class describes an array of `int` cells all located in region `R`:

```
arrayclass ArrayInt<region R> {
    int in R;
}
```

Array classes are used similarly to Java arrays. Array class instances are created by calling a constructor and specifying the length, e.g., `new ArrayInt<r>(10)`. They are accessed using the `[...]` dereference operator, just like normal Java arrays (and they are in fact translated to normal Java arrays during compilation). See Reference Manual § 4 for more details.

Sometimes it's useful to put different array cells in different regions, in particular if you want to update different parts of the array in parallel. To do this, you use a DPJ feature called an *index-parameterized array class*. For example, the following array class defines an array of `int` such that cell `i` is in region `[i]`:

```
arrayclass IPArrayInt<region R> {
    int in R:[index];
}
```

An RPL `[`*e*`]`, where *e* is an integer expression, is called an *array index RPL*; it refers to the array index given by the runtime value of *e*. Every array class has an `index` field that refers to the index of a cell.

The two strategies can also be mixed. For example, you can give array `A` the type `IPArrayInt<rA>` and array `B` type `IPArrayInt<rB>`. Then if you want to write in parallel to the two arrays, you can express the effects on `A` as `writes rA:[?]` and the effects on `B` as `writes rB:[?]`. The `[?]` is an RPL element that stands in for any array index element (see Reference Manual § 3.4.2). However, you can also express disjoint effects on different parts of `A` or `B`. For example, writing to cells 0 and 1 of `A` generates effects `writes rA:[0]` and `writes rA:[1]`, which are disjoint.

Index-parameterized arrays are also useful with arrays of class objects and arrays of arrays. See Reference Manual § 4 for more information. Later sections of this tutorial also provide concrete examples.

2.4.2 DPJ Runtime Classes

The DPJ runtime provides several classes that are useful for manipulating arrays. First, it provides array classes similar to `ArrayInt` and `IPArrayInt` shown above for each of the Java primitive types; there are also generic versions. Second, the DPJ runtime provides classes `ArraySlice` and `Partition` for manipulating subsections (i.e., "slices") of arrays. These classes are useful for divide and conquer traversals that update disjoint parts of the same array.

Reference Manual § 8 gives an overview of how these classes work. They are fully documented in the HTML documentation included with the DPJ release. Later sections of this tutorial also give concrete examples for how to use the classes.

Please note that, as described in § 8 of the Reference Manual, it is not sufficient to put the DPJ runtime classes in your class path when compiling DPJ code that uses the runtime classes. Instead, you must compile the runtime together with your code that depends on the runtime. This is because the DPJ bytecode does not yet properly support separate compilation of DPJ annotations, so the compiler needs all the source code to process the annotations.

```
1  class DisjointArrayUpdate {
2      static class Data<region R> {
3          int x in R;
4          static arrayclass Array {
5              Data<[index]> in [index];
6          }
7      }
8      Data.Array arr;
9      void initialize() {
10         arr = new Data.Array(10);
11         foreach (int i in 0, 10)
12             arr[i] = new Data<[i]>();
13     }
14     void compute() {
15         foreach (int i in 0, 10)
16             ++arr[i].x;
17     }
18     public static void main(String[] args) {
19         DisjointArrayUpdate example = new DisjointArrayUpdate();
20         example.initialize();
21         example.compute();
22         for (Data<[?]> data : example.arr)
23             System.out.print(data.x + " ");
24         System.out.println();
25     }
26 }
```

Figure 2: Disjoint Array Update.

3 Computations on Arrays

This section explains how to write common parallel patterns for array processing, using DPJ's features for supporting arrays:

- § 3.1, Disjoint Array Update, shows how to create an array whose cells all point to different objects, then traverse the array in parallel and update the objects.
- § 3.2, Blocked Array Update, shows how to divide an array into segments and process the segments in parallel using a loop.
- § 3.3, Divide-and-Conquer Array Update, shows how to partition and recurse on arrays in parallel.

3.1 Disjoint Array Update

Here we explain how to use DPJ to create an array of objects, put a different object into every element of the array, and then iterate over the array in parallel and update the objects. We call this pattern Disjoint Array Update. Because objects are stored as references in Java, in general, multiple array cells could point to the same object, causing a race. We'll show how to use the DPJ type and effect system to ensure that no such race can occur.

How to write the pattern: Figure 2 illustrates the Disjoint Array Update pattern in DPJ, for a simple computation (creating an array of objects, then incrementing a field of each one).

Class `Data` (lines 2–7) has an integer field `x` and one region parameter `R` (Reference Manual, § 2.4.1). The field `x` is in region `R`; that means that when `Data` is used as a type, the field of the object instance will be in whatever region is supplied as an argument to the type. See Reference Manual, § 5.1.1. The `Data` class also

```
1  import DPJRuntime.*;
2
3  class BlockedArrayUpdate {
4      public static void main(String[] args) {
5          ArraySliceInt array = new ArraySliceInt(100);
6          final PartitionInt segs =
7              PartitionInt.stridedPartition(array, 10);
8          foreach (int i in 0, segs.length) {
9              ArraySliceInt<segs:[i]:*> seg = segs.get(i);
10             for (int j = 0; j < seg.length; ++j) {
11                 seg.put(j, 10*i+j);
12             }
13         }
14     }
15 }
```

Figure 3: Blocked Array Update.

defines a static inner array class `Array` representing an array of `Data` objects (lines 4–6). The `Data.Array` class is index-parameterized, as discussed in § 2.4.1.

The rest of the code consists of an array `arr` of `Data` objects and three methods that operate on the array:

1. `initialize` assigns a new array to `arr` and uses a `foreach` loop (Reference Manual, § 7.2) to assign a fresh `Data` object to every element of `arr` in parallel.
2. `compute` increments the `x` field of each `Data` object in `arr` in parallel.
3. `main` creates a new `DisjointArrayUpdate` object, calls `initialize` on it, then calls `compute` on it, prints out the array, and exits.

The compiler can prove that the parallelism is safe because:

- In line 12, the update to `arr[i]` has effect `writes [i]`. The value of `i` is different in each loop iteration, so the updates are all to disjoint regions, i.e., there are no conflicting writes.
- In line 16, each iteration of the loop reads cell `arr[i]` (in region `[i]`) and writes field `arr[i].x`. Again, the operations are to disjoint regions in different iterations, so there is no interference.

Further examples: This pattern appears in the parallel Barnes-Hut force computation in method `computegrav` of file `Tree.java` in directory `Benchmarks/Applications/BarnesHut/dpj` of the DPJ release. The force computation iterates in parallel over a disjoint array of `Body` objects and writes into the `force` field of each one.

3.2 Blocked Array Update

When you write a `foreach` loop in DPJ, the runtime repeatedly divides the iteration space in half, until a programmer-specified depth is reached. See Reference Manual, § 7.2.2. In conjunction with an index-parameterized array that has each cell in its own region (Reference Manual, § 4.3), this is usually the easiest and best way to write a loop that traverses an array in parallel and updates its elements. However, sometimes you may want to control the partition directly, by dividing the array into blocks (or tiles) and having each task operate on a block. We call this pattern Blocked Array Update.

How to write the pattern: Figure 3 illustrates the Blocked Array Update pattern in DPJ, for a simple computation (initializing an array of integers so that cell `i` contains the value `i`).

The code uses the `ArraySliceInt` and `PartitionInt` classes discussed in ReferenceManual, § 8, and the DPJ runtime documentation. (`ArraySliceInt` and `PartitionInt` are specializations of the `ArraySlice` and `Partition`

classes to the primitive type `int`.) Line 5 creates a fresh `ArraySliceInt` called `array` with 100 elements. A `ArraySliceInt` wraps an ordinary Java array (of type `int[]`) and behaves much like a `java.util.ArrayList`: for example, it has `put` and `get` methods for accessing the array elements. Lines 6–7 create a *strided partition* of `array` called `segs`. The partition has stride 10, which means that the array is divided into 10 segments of length 10 each. The variable `segs` is declared `final` so that it can be used as a region (Reference Manual, § 3.1.5).

Lines 8–13 represent the parallel computation. The outer parallel `foreach` loop iterates over the segments in `segs`. Line 4 pulls segment `i` out of the partition. Segment `i` is itself a `ArraySlice`, representing the segment consisting of indices $10 * i$ through $10 * i + 9$ of the original array. Its type is `ArraySlice<segs:[i]:*>`. As explained in Reference Manual, § 7.2, the `[i]` in the type allows different segments of the same array to be treated as disjoint, and the variable `segs` in the type ensures that different partitions of the same array are not treated as disjoint. For now, all you really need to know to write the pattern is that you have to declare the partition variable `segs` `final`, as shown in line 6, and you have to write the type of a segment as shown in line 9.

The inner sequential loop iterates over the elements of a segment and writes values into its elements. Notice that in iteration `i` of the outer loop, the inner loop is zero-indexed, even though it is accessing elements $10 * i, 10 * i + 1, \ldots$ of the underlying array. That's because when you create a `Partition` out of a `ArraySlice`, each segment provides a zero-indexed *view* of some segment of the original `ArraySlice`. The `ArraySlice` class takes care of the index translation for you, which is handy.

Further examples: This pattern appears in the International Data Encryption Algorithm (IDEA), in method `Do` of file `IDEATest.java` in directory `Benchmarks/Applications/IDEA/dpj` of the DPJ release. The pattern is useful for IDEA, because the IDEA algorithm operates on fixed-size blocks of the input data.

This pattern is also useful for implementing reductions. See § 6 of this tutorial for more details.

3.3 Divide-and-Conquer Array Update

A common pattern in parallel array algorithms is to divide an array in pieces, and work independently on the pieces, then recursively divide again, etc., until a base case is reached. This pattern is often called *divide and conquer*, because it recursively divides the problem into smaller subproblems, then "conquers" the subproblems by solving the base case.

How to write the pattern: Figure 4 illustrates the Divide-And-Conquer Array Update pattern in DPJ, for a simple computation (initializing an array of integers so that every cell contains the same value).

The code uses the `ArraySliceInt` and `PartitionInt` classes also used for Blocked Array Update (§ 3.2). The method `recursiveInit` (line 4) takes as input a `ArraySliceInt` parameterized by a method region parameter `R` (Reference Manual, § 2.4.2). Its effect is `writes R:*`, i.e., it reads and writes regions under the parameter. If the array is sufficiently small, the initialization is done sequentially (lines 8–12). Otherwise the program creates a `Partition` that splits the array in half. Lines 17–22 recursively and in parallel call `recursiveInit` on the halves. The effects of the recursive calls are as shown, for the reasons explained in Reference Manual, § 8.2, and they are disjoint for the two parallel tasks (see Reference Manual, § 3.6.4).

Further examples: For a more realistic example of this pattern, see the implementation of parallel merge sort in file `MergeSort4.java` in directory `Benchmarks/Kernels/dpj` of the DPJ release.

4 Computations on Trees

This section explains how to write two common parallel patterns for tree processing in DPJ:

- § 4.1 shows how to build a tree data structure in parallel, using a divide-and-conquer strategy.
- § 4.2 shows how to recurse over a tree in parallel and update its nodes.

```
1  import DPJRuntime.*;
2
3  class DivideAndConquerArrayUpdate {
4      static <region R>void recursiveInit(ArraySliceInt<R> arr,
5                                          int val,
6                                          int sequentialSize)
7        writes R:* {
8          if (arr.length <= sequentialSize) {
9              // Base case: sequential
10             for (int i = 0; i < arr.length; ++i)
11                 arr.put(i, val);
12         }
13         // Recursive case: parallel
14         int mid = arr.length / 2;
15         final PartitionInt<R> segs =
16             new PartitionInt<R>(arr, mid);
17         cobegin {
18             // Effect is 'writes segs:[0]:*'
19             recursiveInit(segs.get(0), val, sequentialSize);
20             // Effect is 'writes segs:[1]:*'
21             recursiveInit(segs.get(1), val, sequentialSize);
22         }
23     }
24 }
```

Figure 4: Divide-And-Conquer Array Update.

4.1 Recursive Tree Build

This section shows how to use DPJ to implement a divide-and-conquer build of a tree data structure. We show how to build the tree so it can be used for a parallel algorithm that traverses and updates its nodes. This requires that every node have a different region in its type, and the regions have a tree structure. If read-only access to the tree is desired, then the DPJ types can be simplified. We leave that simplification to the reader as an exercise.

How to write the pattern: Figure 5 illustrates the Recursive Tree Build pattern, for an algorithm that builds a binary region tree for partitioning bodies in one-dimensional space. In this kind of tree, the data (the bodies) are in the leaf nodes; the inner nodes are there just to facilitate searching the data. For example, computing whether a ray intersects a body can be done in $\log(n)$ time, where n is the number of points.

Lines 6–20 of Figure 5 define the classes that the algorithm uses:

- Class `Body` (lines 6–9) defines a body with a mass and a position. The fields are declared `final`, so reading them has no effect.
- Class `Node` (line 10) defines an abstract node of the tree. It contains a `Body` representing its center of mass. For a leaf node, this is an actual physical body; and for an inner node, this is the average of all the bodies in the leaves under the node. There is one region parameter `R`, so that different instances of `Node` can have different regions, and the `centerOfMass` field is in region `R`.
- Class `InnerNode` (lines 11–17) defines an inner node of the tree. It inherits the `centerOfMass` field from `Node`. Additionally, it has a left bound and a right bound, defining a region of 1-D space; and it has a left child and a right child, defining the tree. Here we have used the RPLs `R:LeftRgn` and `R:RightRgn` to give the region of the tree node a region structure that mirrors the tree structure. For example, at runtime, the left child of the right child of the root node would have the type `Node<Root:RightRgn:LeftRgn>`.

```
1  package Tree;
2
3  import DPJRuntime.*;
4
5  class RecursiveTreeBuild {
6      static class Body {
7          final double mass, pos;
8          Body(double m, double p) { this.mass = m; this.pos = p; }
9      }
10     static abstract class Node<region R> { Body centerOfMass in R; }
11     static class InnerNode<region R> extends Node<R> {
12         final double leftBound, rightBound;
13         region LeftRgn, RightRgn;
14         Node<R:LeftRgn> leftChild in R:LeftRgn;
15         Node<R:RightRgn> rightChild in R:RightRgn;
16         InnerNode(double lb, double rb) pure { leftBound = lb; rightBound = rb; }
17     }
18     static class LeafNode<region R> extends Node<R> {
19         LeafNode(Body b) pure { centerOfMass = b; }
20     }
21     private static <region R>
22         int computeSplitPoint(ArraySlice<Body,R> arr,
23                               double midpoint)
24         reads R
25     {
26         int result = 0;
27         // Set result to the first index position in arr whose
28         // position is to the right of midpoint
29         return result;
30     }
31     public static <region RN,RA | RN:* # RA:*>
32         Node<RN>makeTree(ArraySlice<Body,RA:*> arr, double leftBound,
33                          double rightBound)
34         reads RA:* writes RN:*
35     {
36         if (arr.length == 0) return null;
37         if (arr.length == 1) return new LeafNode<RN>(arr.get(0));
38         double midpoint = (leftBound + rightBound) / 2;
39         int splitPoint = computeSplitPoint(arr, midpoint);
40         Partition<Body,RA:*> segs = new Partition<Body,RA:*>(arr, splitPoint);
41         InnerNode<RN> node = new InnerNode<RN>(leftBound, rightBound);
42         cobegin {
43             node.leftChild = RecursiveTreeBuild.<region RN:InnerNode.LeftRgn,RA>
44                 makeTree(segs.get(0), leftBound, midpoint);
45             node.rightChild = RecursiveTreeBuild.<region RN:InnerNode.RightRgn,RA>
46                 makeTree(segs.get(1), midpoint, rightBound);
47         }
48         return node;
49     }
50 }
```

Figure 5: Recursive Tree Build.

- Class `LeafNode` (lines 18–20) just adds a constructor to `Node`, so that a leaf node representing a physical body may be created.

The rest of the code defines the methods that do the tree building:

- Method `computeSplitPoint` (lines 21–30) is a helper method that takes an array of bodies and a point in space. It computes and returns the index into the array such that the elements to the left of the index are all and only the elements to the left of the point in space. The array is assumed to be sorted by position. We don't show this code. It can easily be implemented in parallel using a divide-and-conquer strategy.
- Method `makeTree` (lines 31 and following) does the actual tree building and is discussed below.

Method `makeTree` uses divide-and-conquer recursion to build the tree. It takes an array of bodies to build into a tree (assumed to be pre-sorted by position), and a left and right bound representing the region of space that the tree is partitioning. It returns a node representing the root of the tree. The method has two region parameters, `RN` and `RA`. `RN` is the region associated with the type of the node returned, while `RA` is the region associated with the array of bodies coming in. The parameters are constrained so that writes under `RN` and under `RA` are noninterfering. (For example two different names `A` and `B` would satisfy this constraint, but `A` and `A:B` would not.)

The body of `makeTree` does the following:

- If the input array is empty, return `null` (line 36).
- If the input array has just one element, then create a fresh leaf node for it (line 37).
- Otherwise, handle the recursive case.

The recursive case works as follows:

- Compute the index into the array corresponding to the midpoint of the current region of space, and split the array there (lines 38–40).
- Make a new `InnerNode` corresponding to the current region of space (line 41).
- In parallel, populate the left and right children of the new inner node, by calling `makeTree` recursively (lines 42–47).
- Return the created node (line 48).

The interesting types and effects are inside the `cobegin`. Notice that lines 43–44 call `makeTree` with `RN = RN:InnerNode.LeftRgn`, which creates the right type for assigning into `node.leftChild`. Notice also that the effects of the two branches of the `cobegin` are `reads RA:* writes RN:InnerNode.LeftRgn:*` and `reads RA:* writes RN:InnerNode.RightRgn:*`, and these two effects are disjoint.

Further examples: The file `Quadtree.java` in directory `Benchmarks/Kernels/dpj` of the DPJ uses this pattern to build a quadtree (i.e., a tree in which each node has up to four children) for partitioning two-dimensional space. The main difference is that index-parameterized arrays (Reference Manual § 4.3) are used instead of the regions `LeftRgn` and `RightRgn` to distinguish the different branches of the tree. The partition step is implemented sequentially, but it could easily be parallelized.

The Barnes-Hut benchmark in the DPJ release (`Benchmarks/Applications/Barnes-Hut/dpj`) builds an octree for partitioning three-dimensional space. The tree build phase is not parallelized in that benchmark, but it could be using this pattern.

```
1  package Tree;
2
3  import DPJRuntime.*;
4  import Tree.RecursiveTreeBuild.*;
5
6  class RecursiveTreeUpdate {
7      public static <region R>Body updateCenterOfMass(Node<R> node)
8          writes R:*
9      {
10         if (node == null) return null;
11         if (node instanceof LeafNode<R>)
12             return node.centerOfMass;
13         Body leftCoM, rightCoM;
14         InnerNode<R> innerNode = (InnerNode<R>) node;
15         cobegin {
16             // Effect is 'writes R:InnerNode.LeftRgn:*'
17             leftCoM = updateCenterOfMass(innerNode.leftChild);
18             // Effect is 'writes R:InnerNode.RightRgn:*'
19             rightCoM = updateCenterOfMass(innerNode.rightChild);
20         }
21         Body result = null;
22         if (leftCoM != null && rightCoM != null)
23             result = new Body((leftCoM.mass+rightCoM.mass)/2,
24                               (leftCoM.pos+rightCoM.pos)/2);
25         else if (leftCoM != null)
26             result = new Body(leftCoM.mass, leftCoM.pos);
27         else if (rightCoM != null)
28             result = new Body(rightCoM.mass, rightCoM.pos);
29         innerNode.centerOfMass = result;
30         return result;
31     }
32 }
```

Figure 6: Recursive Tree Update.

4.2 Recursive Tree Update

This section shows how to traverse the tree created in § 4.1 in parallel and update its nodes. The tree structure of the regions in the types of the nodes allow this traversal and update to be done with provable noninterference.

How to write the pattern: Figure 6 illustrates the Recursive Tree Update pattern, for an algorithm that traverses the tree constructed in Figure 5 and updates the center of mass of each node. The work is done by a single method, `updateCenterOfMass`, which accepts a tree node, updates the center of mass fields of that node and all its descendants, and returns the center of mass of that node.

The method body works as follows. If the input node is a leaf node, then the method just returns its body as the center of mass (lines 10–11). Otherwise, the method recursively and in parallel calls `updateCenterOfMass` on the left and right nodes (lines 13–20) and constructs a new `Body` representing the center of mass (lines 21–28). It then stores the `Body` representing the center of mass into the input node and returns the `Body`.

In lines 15–20, the effects of each of the two branches of the `cobegin` statement are as shown. Because of the tree structure of the regions of the tree nodes, the effects on the left and write subtrees are disjoint.

Further examples: The Barnes-Hut benchmark in the DPJ release (`Benchmarks/Applications/Barnes-Hut/dpj`) does a similar center-of-mass computation on an octtree. The center-of-mass phase is not parallelized in that benchmark, but it could be using this pattern.

5 Computing with Local Objects

This section discusses a common parallel pattern that we call Local Objects. Often an object is created by some task, and not seen by any other task running in parallel with that one. Either it is used only by the creating task, and its lifetime ends with that task; or it is stored into global memory, but no reference to it is seen until after all the parallel tasks have completed. We call this kind of object a *local object*. Local objects are useful, because effects on them by the creating task are hidden from other tasks running in parallel, so the effects cannot cause interference.

This section discusses two kinds of local objects:

1. § 5.1 discusses *iteration-local objects*. These objects are created inside a `foreach` iteration, and are local to that iteration.

2. § 5.2 discusses *method-local objects*. These objects are created inside a method, and are local to that method scope and its callees.

5.1 Iteration-Local Objects

There are two ways to write the Iteration-Local Objects pattern in DPJ: using local regions, and using array regions.

How to write the pattern with local regions: Method `usingLocalRegions` (line 12 of Figure 7) illustrates the Iteration-Local Objects pattern using local regions. Class `LocalObject` (lines 4–10) has an integer field `value` and a method `produceValue`. The `produceValue` method accepts an integer value, stores it to the object's `value` field, and returns it. The effect of `produceValue` is to write to `R`, the region of the `LocalObject` type. The rest of the program (lines 11 and following) creates an array and writes to it in parallel. Line 11 creates an array of integers called `results`. The array is index-parameterized (Reference Manual § 4.2.2), so that every cell is in its own region.

Line 18 produces two effects, both disjoint for distinct iterations of the `foreach` loop at line 13. First, because `results` is index-parameterized, the effect of the assignment to `results[i]` is `writes [i]`; the write is to a different region for each value of `i` in each distinct iteration. Second, the effect of invoking `produceValue` is `writes R` with `R` = `LocalRegion`, that is, `writes LocalRegion`. Since the scope of `LocalRegion` is local to an iteration of the `foreach`, this effect does not cause any interference across `foreach` iterations.

```
1  import DPJRuntime.*;
2
3  class IterationLocalObjects {
4      class LocalObject<region R> {
5          int value in R;
6          int produceValue(int val) writes R {
7              value = val;
8              return value;
9          }
10     }
11     IPArrayInt results = new IPArrayInt(10);
12     void usingLocalRegions() {
13         foreach (int i in 0, 10) {
14             region LocalRegion;
15             LocalObject<LocalRegion> localObject =
16                 new LocalObject<LocalRegion>();
17             // Effect 'writes LocalRegion' is local to an iteration
18             results[i] = localObject.produceValue(i);
19         }
20     }
21     void usingArrayRegions() {
22         foreach (int i in 0, 10) {
23             LocalObject<[i]> localObject =
24                 new LocalObject<[i]>();
25             // Effect 'writes [i]' is local to an iteration
26             results[i] = localObject.produceValue(i);
27         }
28     }
29 }
```

Figure 7: Iteration-Local Objects.

```
1  import DPJRuntime.*;
2
3  class MethodLocalObjects {
4      class LocalObject<region R> {
5          int value in R;
6          int produceValue(int val) writes R {
7              value = val;
8              return value;
9          }
10     }
11     <region R>int sumReduce(ArraySliceInt<R> arr) reads R {
12         if (arr.length == 0) return 0;
13         if (arr.length == 1) return arr.get(0);
14         int mid = arr.length/2;
15         int left, right;
16         cobegin {
17             left = sumReduce(arr.subslice(0,mid));
18             right = sumReduce(arr.subslice(mid+1,arr.length-mid));
19         }
20         region LocalRegion;
21         LocalObject<LocalRegion> localObject =
22             new LocalObject<LocalRegion>();
23         // Effect 'writes LocalRegion' is local to method
24         int result = localObject.produceValue(left + right);
25         return result;
26     }
27 }
```

Figure 8: Method-Local Objects.

How to write the pattern with array regions: Method `usingArrayRegions` (line 21 of Figure 7 illustrates the Iteration-Local Objects pattern using array regions. The code is the same as in method `usingLocalRegions`, except that instead of an iteration-local region, it binds the array region `[i]` to the RPL parameter of the `LocalObject` class in iteration `i` of the `foreach` loop. The effect of invoking `produceValue` in line 14 is `writes [i]`, which is a distinct region for each iteration of the loop.
Further examples: The Monte Carlo benchmark in the DPJ release uses the array region version of this pattern. Monte Carlo operates in parallel on a set of independent tasks, and uses a local object called `PriceStock`, parameterized by an iteration index region, to store temporary data for processing the task. See the file `AppDemo.java` in the directory `Benchmarks/Applications/MonteCarlo/dpj`.

5.2 Method-Local Objects

How to write the pattern: Figure 8 illustrates the Method-Local Objects pattern with a simple sum reduction that also performs an incidental computation on local data. Class `LocalObject` (lines 4–10) is the same as in § 5.1. Method `sumReduce` (lines 11 and following) takes as input a `ArraySliceInt` (Reference Manual § 8.2). In the base cases it either does nothing (empty array) or returns the only value (array with one element). Otherwise, it divides the array in half and processes each half recursively and in parallel. This part of the computation is similar to the Divide-and-Conquer Array Update pattern (§ 3.3), except that the array is only read, not written.

The rest of the computation (lines 20 and following) declares a local region (Reference Manual § 2.3.3) called `LocalRegion`, and creates a `LocalObject` with the local region as its RPL argument. The effect of invoking `localObject.produceValue` in line 24 is `writes LocalRegion`, as shown in line 23. This effect is a local effect

(Reference Manual § 5.3), so it doesn't have to be reported in the declared effect of `sumReduce` (Reference Manual § 2.3.1). The write effect on the `LocalObject` instance in each invocation of `sumReduce` is therefore hidden, or *masked*, from the callee, and doesn't cause any interference at the `cobegin` in line 16.

Further examples: The Collision Tree benchmark in the DPJ release uses method-local objects. Collision Tree computes whether two trees representing objects in space have any intersection, by recursively traversing the trees in parallel. The computation is read-only on the trees, but the computation on each tree node passes results (a list of intersecting triangles) up to its parent. Before recursively calling itself in parallel on two subtrees, the `intersect` method creates fresh objects to hold the results from one of the subtrees, placing their data in a local region. The computation on the other subtree reuses the data structures used for the parent node, so the computations on the two subtrees write their results to different regions and can be safely run in parallel. See the file `CollisionTree.java` in the directory `Benchmarks/Applications/CollisionTree/src/com/jme/bounding`.

Method-local regions are also useful for algorithms that explore a tree-shaped search space: for example, a recursive solver (such as a tic-tac-toe or n queens solver). In such algorithms, typically some local state (like a board position) is copied and sent to all possible next moves. Each next move does its computation on its private copy of the state, then returns its result to the parent. The private copy of the state is a method-local object.

6 Associative Reductions

This section discusses how to implement associative reductions in DPJ. A reduction is a common parallel pattern for computing a single datum from a collection of data. The transformation from the collection of data to the single datum is called "reducing" the collection. For example, a sum reduction of a collection of integers sums all the elements in the collection to produce a single integer representing the sum.

A reduction is *associative* if it can be represented as an associative binary operation on the elements. For example, a sum reduction is associative because it is represented by the binary operation +, which is associative; that means that for any integers a, b, and c, $a+(b+c)=(a+b)+c$.

How to write the pattern: Associative reductions can be written efficiently as parallel algorithms. There are at least two ways to do it, both supported in DPJ:

1. *Recursive reduction* Use divide-and-conquer recursion.

2. *Iterative reduction* Divide the input collection up into pieces, loop over all the pieces in parallel and reduce each piece to a partial result, then reduce the partial results to the final result, either sequentially or in parallel.

Below we illustrate method (2) (iterative reduction). § 5.2 shows an example of a divide-and-conquer sum reduction.

Figure 9 lists a simple DPJ program that implements an iterative integer sum reduction. Line 4 declares a private region `AccumRgn` for holding the accumulated results. Line 5 declares an accumulation variable `sum` in `AccumRgn`. Lines 6–8 give the signature for the public method `reduce`, which takes a `ArraySliceInt` and a tile size (i.e., the number of array elements to process in each task) and does the reduction. Method `reduce` has a region parameter `R`, so a `ArraySliceInt` with any RPL argument can be passed to its `arr` parameter. The region parameter is declared so that binding `R=r` at a call site must satisfy `r:* # AccumRgn` (Reference Manual § 2.4.3). The method is declared to read under `R` (because it is reading the array), and to write `AccumRgn` (because it uses `AccumRgn` to compute the sum)

The body of the `reduce` method sets `sum` to zero and uses a blocked array to do the partial sums. The array blocking is similar to Blocked Array Update (§ 3.2), except the array is only read, and not written. Inside the `foreach` loop (lines 14–20), there is a sequential partial sum computation, followed by a global accumulation. The accumulation invokes the private method `updateSum` (defined starting in line 23). That method is declared `commutative` (Reference Manual § 2.3.4), because it is properly synchronized to allow concurrent execution, and the operation it performs commutes with itself. This is a typical way to write an accumulation operation

```
1  import DPJRuntime.*;
2
3  public class IntegerSumReduction {
4      private region AccumRgn;
5      private static int sum in AccumRgn;
6      public static <region R | R:* # AccumRgn>
7        int reduce(ArraySliceInt<R> arr, int tileSize)
8          reads R:* writes AccumRgn
9      {
10         sum = 0;
11         PartitionInt<R> segs =
12             PartitionInt.<region R>
13             stridedPartition(arr, tileSize);
14         foreach(int i in 0, segs.length) {
15             int partialSum = 0;
16             ArraySliceInt<R:*> seg = segs.get(i);
17             for (int j = 0; j < seg.length; ++j)
18                 partialSum += seg.get(j);
19             updateSum(partialSum);
20         }
21         return sum;
22     }
23     private static commutative synchronized
24         void updateSum(int partialSum)
25             writes AccumRgn
26     {
27             sum += partialSum;
28     }
29     public static void main(String[] args) {
30         region MainRgn;
31         int SIZE = 1000000;
32         int TILESIZE = 1000;
33         ArraySliceInt<MainRgn> arr =
34             new ArraySliceInt<MainRgn>(SIZE);
35         arr.put(42, 42);
36         int sum = reduce(arr, TILESIZE);
37         System.out.println("sum="+sum);
38     }
39 }
```

Figure 9: Iterative sum reduction in DPJ.

in DPJ. If the number of parallel tasks is small, one could also have each task store its partial result into an array cell, and then sum the elements of that array in a separate sequential phase.

Lines 29 and following show a `main` method that creates a new integer array of size 1,000,000, initializes element 42 of the array with the value 42, and calls `reduce` with a tile size of 1000. Since Java initializes the other array cells to 0, hopefully the answer is 42.

Further examples: The following benchmarks in the DPJ release implement reductions:

- `SumReduce.java` in `Benchmarks/Kernels/dpj` is a simple sum reduction kernel, using the divide-and-conquer strategy.
- The Monte Carlo benchmark computes results for many tasks, then reduces the partial results to a single answer using an iterative reduction. The final accumulation step is sequential. See method `processResults` in file `AppDemo.java` in directory

  `Benchmarks/Applications/MonteCarlo/dpj`.
- The k-means clustering benchmark uses an iterative reduction to compute a histogram. See method `work` in file `Normal.java` in directory `Benchmarks/Applications/KMeans/dpj`.