# A Language for Deterministic-by-Default Parallel Programming *

Robert L. Bocchino Jr., Stephen Heumann, Nima Honarmand,
Rakesh Komuravelli, Jeffrey Overbey, Patrick Simmons,
Hyojin Sung, Mohsen Vakilian, Sarita V. Adve,
Vikram S. Adve, Danny Dig, and Marc Snir

University of Illinois at Urbana-Champaign
dpj@cs.uiuc.edu

**Abstract.** When using today's common shared-memory parallel programming models, subtle errors can lead to unintended nondeterministic behavior and bugs that appear only under certain thread interleavings. In contrast, we believe that a programming model should *guarantee* deterministic behavior unless the programmer specifically calls for nondeterminism. We describe our implementation of such a *deterministic-by-default* parallel programming language.

Deterministic Parallel Java (DPJ) is an extension to Java that uses a *region-based type and effect system* to guarantee deterministic parallel semantics through static checking. Data in the heap is partitioned into *regions*, so the compiler can calculate the *read and write effects* of each variable access in terms of regions. Methods are annotated with *effect summaries*, so the effects of a full program can be checked with a modular analysis. Using this system, the compiler can verify that the effects of the different operations within each parallel region are noninterfering. The DPJ type system includes several novel features to enable expressive support for widely used parallel idioms.

We describe an experimental evaluation of DPJ that shows it can express a wide range of realistic parallel programs with good performance. We also describe a method for *inferring* method effect summaries, which can ease the burden of writing annotations. In addition, we briefly discuss several areas of ongoing and future work in the DPJ project.

## 1 Introduction

Commodity processors have reached the limit of single-core scaling, and we are now seeing more and more cores on a chip. As a result, desktop computers have become parallel machines. To harness that power, mainstream programmers — most of whom are used to writing sequential code — must now become *parallel* programmers.

* This work is funded by Microsoft and Intel through the Universal Parallel Computing Research Center (UPCRC) at the University of Illinois and by NSF grants 07-02724, 07-20772, 08-33128 and 08-33188.

Unfortunately, parallel programming is not very easy, given the state of the art in languages and tools today. The dominant model is multithreaded shared memory programming, using locks and other low-level mechanisms for synchronization. This model is complex to program and difficult to get right. There are many possible thread interleavings, causing *nondeterministic* program behaviors (i.e., different visible results for different executions on the same input). All that nondeterminism can lead to subtle bugs such as data races, atomicity violations, and deadlocks — bugs that are extremely challenging to diagnose and eliminate. Testing is also much more difficult than in the sequential case, because the state space is so much larger.

The irony is that a vast number of computational algorithms (though not all) are *deterministic* — a given input produces the same externally visible output in every execution — if implemented correctly. Examples of such algorithms include scientific computing, encryption/decryption, sorting, compiler and program analysis, and processor simulation. For these algorithms, nondeterminism is not wanted; if it exists, it is a bug. However, today's models force programmers to implement such algorithms in a nondeterministic style and then convince themselves that the program will behave deterministically.

By contrast, a *deterministic-by-default programming model* [7] can *guarantee* deterministic behavior *unless* nondeterministic behavior is explicitly requested by the programmer in disciplined ways. Such a model can ease the development and maintenance of parallel programs in several ways. Programmers are freed from reasoning about difficult issues such as data races, deadlocks, and memory models. They can start with a sequential implementation and incrementally add parallelism, while retaining the same program behavior. They can use familiar sequential tools for debugging and testing. And they can test an application only once for each input, thus avoiding the state space explosion caused by nondeterministic models.

However, while guaranteed determinism is available for some restricted styles of parallel programming (e.g., data parallel [13], or pure functional [11]), it remains a challenging research problem to guarantee determinism for imperative, object-oriented languages such as Java, C++, and C#. In such languages, object references, aliasing, and updates to mutable state obscure the data dependences between parts of a program, making it hard to prove that those dependences are respected by the program's synchronization. This is a very important problem as many applications that need to be ported to multicore platforms are written in these languages.

In this paper, we give an overview of our research on *Deterministic Parallel Java*, or DPJ. DPJ is a parallel language designed to address the problems stated above. It has the following key features:

1. **Core deterministic language:** DPJ allows the creation of explicitly parallel tasks that are *guaranteed* to run deterministically. Determinism is guaranteed via a *type and effect system* [12] that tracks accesses by the parallel tasks to the shared heap. The programmer partitions the heap into *regions* and writes *method effect summaries* stating which regions are accessed by

each method in the program. The compiler uses the region and effect annotations to check that there are no conflicting accesses to the same memory location by any pair of parallel tasks. As a result, the parallel program not only is deterministic, but also *produces identical results to an obvious sequential program* (i.e., the program that results when the parallel constructs are replaced by sequential composition). Thus, the programming model is very intuitive. The DPJ type and effect system incorporates novel features that make it more expressive than previous systems for common parallel patterns [9].

2. **Object-oriented frameworks:** DPJ supports the use of *object-oriented frameworks*, a common programming technique in real-world code. Frameworks allow an expert parallel programmer to build up parallel abstractions (for instance, parallel operations on collections) that can be easily used by non-experts (usually by writing some sequential code to adapt the framework to a specific use). DPJ incorporates novel features for writing frameworks that are generic enough to support highly extensible frameworks, yet permit the framework writer to constrain the side effects of user code so that they cannot make parallelism within the framework unsafe [5].
3. **Controlled nondeterminism:** DPJ supports the use of *controlled nondeterminism*, for algorithms such as branch-and-bound search, where several answers are acceptable [8]. Nondeterminism is introduced through explicitly nondeterministic parallel constructs that allow conflicting memory accesses between pairs of parallel tasks. The effect system tracks effects done in statements marked `atomic`, and it prohibits conflicts between unguarded accesses. The language implementation uses software transactional memory (STM) [10] to enforce *isolation* for `atomic` statements. The language guarantees *determinism by default*: i.e., code is deterministic unless it contains *explicit* nondeterminism. Because it prohibits interference between unguarded accesses, the language also guarantees *strong isolation* for atomic statements (STM alone provides only weak isolation) and race freedom.

We presented the core type and effect system together with an experimental evaluation of the core deterministic language in OOPSLA 2009 [9], while the support for frameworks and nondeterminism are the subject of ongoing research [5, 8].

The rest of this paper proceeds as follows. In Section 2, we describe the key features of DPJ's core type and effect system for determinism. In Section 3, we describe an experimental evaluation of the core language showing (1) DPJ can express a range of parallel algorithms with good performance; (2) there are some limitations in expressivity as the price of the determinism guarantee; and (3) the annotation burden is nontrivial, but still very small compared with the effort spent writing, testing and debugging parallel programs, and is reasonable given the safety and documentation benefits. In Section 4, we describe an algorithm and an interactive Eclipse-based tool for inferring DPJ's method effect summaries, which can substantially ease the annotation effort when writing DPJ programs. In Section 5, we summarize ongoing and future work on object-oriented framework support, controlled nondeterminism, inferring region

```
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  }
```

**Fig. 1.** Regions and effects in DPJ

information, and supplementing DPJ's static checks with runtime checks. In Section 6 we wrap up the discussion.

## 2 Type and Effect System for Determinism

### 2.1 Regions and Effects

DPJ allows the programmer to partition the heap into *regions*. The DPJ compiler uses these to determine the *effects* of each statement in terms of read and write accesses to regions. In order to avoid the need for interprocedural analysis, programmers must provide *method effect summaries* that describe the effects of methods. DPJ's `cobegin` and `foreach` constructs are used to specify parallel regions of the code: `cobegin` specifies two or more statements to be executed in parallel, and `foreach` calls for the iterations of a loop to be run in parallel. The DPJ compiler uses a simple intraprocedural analysis to check that the effect summaries are correct, and that any two memory accesses from tasks that may run in parallel with each other have noninterfering effects.

Figure 1 gives a simple example of regions and effects in DPJ. Line 2 declares two region names `X` and `Y`. Lines 3 and 4 declare the instance fields `x` and `y`, and place them in the regions `X` and `Y`, respectively. These region names have static scope, so all instances of the `Point` class will have their `x` field in the same region `Point.X`. Programmers can also declare *local regions* (not shown) within a method; these are used to express effects on objects that do not escape the method.

Lines 5 and 6 show the use of method effect summaries. The `setX` method writes to the field `this.x`, which generates a `writes X` effect (since `x` is in the region `X`). This effect is recorded in the effect summary for the method. Similarly, `setY` has an effect `writes Y`. Methods can also have `reads` effects on a region if they only read it, so the general form of a method's effect summary is `reads` *region-list* `writes` *region-list*. If a region is both read and written, only the `writes` effect needs to be specified. Also, methods with no effects may be declared as `pure`. If the effect summary for a method is omitted, the compiler conservatively assumes that it writes the entire heap. This allows standard Java code to be compiled as DPJ code, but methods called inside parallel tasks need to have effect summaries.

The DPJ compiler enforces a few simple rules for effect summaries. They must cover all the actual heap effects of the method. For example, the compiler would give an error if the `setX` method were given the effect summary `pure`, since

this does not cover its actual `writes X` effect. However, the effect summary may be conservative, covering effects that the method doesn't actually have. The effects of an overridden method must include the effects of any overriding method. By enforcing these two rules, the compiler ensures that wherever there is a method call in the code, the effect summary of the called method always fully covers the actual effects that the call will have at run time, even in the presence of subclassing. These effect summaries, plus the types of references and the regions of variables, allow the compiler to infer conservatively the effects of each expression, statement, task, and method body in the program.

When a `cobegin` or `foreach` construct is used, the DPJ compiler will check that the effects of the concurrent tasks are pairwise non-interfering. That is, two pieces of code that can run concurrently may not have effects on the same region, unless both those effects are read effects (or the effects are covered by a commutative clause, described below). Thus, a call to `setX` could be run in parallel with a call to `setY`, since their effects are on different regions, but two calls to `setX` could not be run in parallel (even if they are on different objects and thus don't actually access the same memory: DPJ's checks are based only on regions).

**Commutativity Annotations** There are cases where a parallel computation is deterministic even though it has interfering reads and writes. For example, if we have a concurrent set implementation that uses locking internally, two inserts into such a set can go in either order and preserve determinism, even though they involve interfering writes to the set object. DPJ supports this kind of pattern by allowing method declarations to contain a `commutative` annotation. The annotation indicates that (1) two invocations of the method are *atomic* with respect to each other, i.e. the result will be the same as if one occurred and then the other; and (2) either order of invocations produces the same result. These properties are not checked by the compiler; the `commutative` annotation is a trusted assertion from the programmmer that they hold for a method. In the compiler, calls to methods marked `commutative` generate *invocation effects*. These special effects keep track of the method called as well as its regular effects, so that the compiler can ignore interference of effect between two calls to the same commutative method, while still detecting any interference of effect between those calls and other operations.

### 2.2 Region Parameters

In DPJ, classes and methods can be written with *region parameters*, which become bound to actual regions when the method is invoked, or the class is instantiated into a type. Figure 2 shows a version of the `Point` class the takes `X` and `Y` as region parameters, specified in line 1. The keyword `region` distinguishes them from Java generic type parameters. As before, `x` is placed in the region `X`, and `y` in `Y`. When `Point` is instantiated as a type, and objects of that type are created, a region argument is given to specify the actual regions that will be

```
1  class Point<region X, Y> {
2    double x in X;
3    double y in Y;
4
5    region X1, Y1, X2, Y2;
6    static Point<X1, Y1> pointOne = new Point<X1, Y1>();
7    static Point<X2, Y2> pointTwo = new Point<X2, Y2>();
8
9    static void translate(double deltaX, double deltaY) {
10     cobegin {
11       pointOne.x += deltaX;  /* writes X1 */
12       pointOne.y += deltaY;  /* writes Y1 */
13       pointTwo.x += deltaX;  /* writes X2 */
14       pointTwo.y += deltaY;  /* writes Y2 */
15     }
16   }
17 }
```

**Fig. 2.** Class region parameters

bound to `X` and `Y`, and thus will contain the object's `x` and `y` fields. Lines 6 and 7 illustrate this, creating objects of type `Point<X1, Y1>` and `Point<X2, Y2>`.

When computing the effects of accessing fields, the compiler substitutes the actual regions for the formal region parameters. So the write to `pointOne.x` in line 11 writes to the region `X1`, and the write to `pointTwo.y` in line 14 writes to the region `Y2`. All four operations in the `cobegin` block write to different regions, so they are non-interfering and can be run in parallel.

The DPJ type system ensures that this type of reasoning is sound by preventing assignments between variables with incompatible region typing. For example, a reference of type `Point<X1, Y1>` cannot be assigned to a variable of type `Point<X2, Y2>`. DPJ also supports *disjointness constraints* that stipulate that region parameters are disjoint from each other, or from other regions. The compiler will check that these constraints are satisfied when classes are instantiated or methods are invoked, thereby ensuring soundness.

### 2.3 Region Path Lists (RPLs) and Nested Effects

It is often useful to express *region nesting*, which allows the heap to be partitioned hierarchically. For example, in a parallel update traversal of a binary tree, it is important to distinguish "the left subtree" from "the right subtree," and it is natural to do that through hierarchical regions. DPJ allows this kind of nesting structure through *region path lists*, or RPLs. An RPL is a colon-separated list of names, starting with `Root`, such as `Root:L:R`. RPLs naturally form a tree rooted at `Root`. In DPJ, every region is actually specified by an RPL; a bare region name like `L` is equivalent to `Root:L`. RPLs may be *partially specifed* by using a `*` element to stand in for zero or more names at any position in the RPL, thereby specifying sets of regions.

Figure 3 shows a tree class that uses RPLs so that it can be traversed in parallel, updating its elements. The key feature used to allow this is a *parameterized RPL*: RPLs can begin with a region parameter, as with the RPLs `P:L` and `P:R` in lines 4-5. Thus, a `Tree<Root>` object would have its `value` field in `Root`, while

```
1  class Tree<region P> {
2    region L, R;
3    double value in P;
4    Tree<P:L> left in P:L;
5    Tree<P:R> right in P:R;
6
7    int scaleValues(double multiple) writes P:* {
8      value *= multiple; /* writes P */
9      cobegin {
10       if (left != null) left.scaleValues(multiple);    /* writes P:L:* */
11       if (right != null) right.scaleValues(multiple);  /* writes P:R:* */
12     }
13   }
14 }
```

**Fig. 3.** A tree class using region path lists for hierarchical regions

its left child would be of type `Tree<Root:L>`, and so would have its `value` field in `Root:L`, and the right child would have its `value` field in `Root:R`. Each node of the tree has its `value` field in a distinct region, with an RPL reflecting its position in the tree. The DPJ type system enforces this structure by preventing assignments with incompatible region typing; for example, the assignment `left = right` would be illegal.

The `scaleValues` method in lines 7-13 shows how to write a parallel recursive traversal that updates the `value` field of each node. The parallelism here is safe, because line 10 accesses only the left subtree, while line 11 accesses only the right subtree, so they are non-interfering. The distinction between the two subtrees is reflected in the regions: the left subtree is entirely in regions described by `P:L:*`, and the right subtree is in regions described by `P:R:*`. These two RPLs describe disjoint sets of memory locations, so the effects of the two statements inside the cobegin are noninterfering, and thus the parallelism is safe.

### 2.4 Arrays

DPJ has two major features for parallel computations on arrays: *index-parameterized arrays* and *subarrays*. Index-parameterized arrays allow for safe parallel updates of objects through an array of references, while subarrays allow an array to be dynamically partitioned into disjoint pieces that can be operated on in parallel.

**Index-parameterized arrays** The index-parameterized array mechanism involves two components. The first is an RPL element `[`$e$`]`, where $e$ is an integer expression, corresponding to cell $e$ of an array. This is called an *array RPL element*. Second, DPJ supports an *index-parameterized array type* that allows the region and type of the array cell $e$ to be written using the RPL element `[`$e$`]`. For example, we can specify that cell $e$ resides in region `Root:[`$e$`]` and has type `C<Root:[`$e$`]>`. Thus, each cell in an array can be in a distinct region, and each cell can also have a distinct region-parameterized type, which can be used to determine what region its fields are in. Thus, it is possible to ensure that updates to objects referenced by different elements of the array will be non-interfering, based on their region typing.

```
1  static void translateArray(Point<[_], [_]>[]<[_]> points, double dX, double dY) {
2    foreach (int i in 0, points.length) {
3      points[i].x += dX;  /* writes Root:[i] */
4      points[i].y += dY;  /* writes Root:[i] */
5    }
6  }
```

**Fig. 4.** Parallel updates using an index-parameterized array

Figure 4 illustrates this. The `translateArray` method takes an array of `Point` objects (from Figure 2) and iterates over it in parallel with a `foreach` loop, updating the coordinates of each one. The type of `points`, shown in line 1, is an index-parameterized array type. The element type `Point<[_], [_]>` says that the array element at index $n$ is a reference to an object of type `Point<[`$n$`], [`$n$`]>`. The type system will ensure that this is actually the case; in particular, this means that each reference in the array must be to a distinct object, with its `x` and `y` fields in a distinct region `Root:[`$n$`]`. The final `<[_]>` in the type of `points` indicates that cell $n$ of the array itself is also in region `Root:[`$n$`]`. Thus, it is safe to update the `x` and `y` fields of each point in parallel, as the parallel loop in the `translateArray` method does.

**Subarrays** DPJ allows *dynamic array partitioning*: an array may be divided *in place* (i.e., without copies) into two or more disjoint parts (*subarrays*), which can be updated in parallel. DPJ provides a class `DPJArray` that wraps an ordinary Java array and provides a view into a contiguous segment of it, parameterized by a start position and length. It also provides a class `DPJPartition`, which represents an indexed set of `DPJArray` objects, all of which point into mutually disjoint segments of the same underlying array. The `DPJPartition` constructor takes a `DPJArray` object, together with some parameters indicating how to split it up. Once a `DPJPartition` object $p$ is created, the programmer can call $p$.get($i$) to access the $i$th segment of the partition.

Since the `DPJPartition` constructor ensures that all the segments are mutually disjoint, they can be operated on in parallel. To support this, the `DPJArray` object returned by calling $p$.get($i$) will have the *owner RPL* $p$`:[`$i$`]:*` in its type. An owner RPL is like an RPL, except that it begins with a `final` local variable instead of with `Root`. This allows different partitions of the same array to be represented. Because of the tree structure of DPJ regions, $p$`:[0]:*` and $p$`:[1]:*` are disjoint region sets, so they can be operated on in parallel. Array partitioning can be applied recursively, making it useful for divide-and-conquer algorithms on arrays, such as quicksort.

## 3 Experimental Results

To evaluate DPJ, we built an initial implementation of it and used DPJ to write several benchmark codes [9]. We used a modified version of Sun's *javac* compiler that translates from DPJ code into ordinary Java source code. As a runtime system, we use the *ForkJoinTask* framework that will be included as part of the standard library in Java 7 [1]. It uses a work-stealing scheduler similar to the

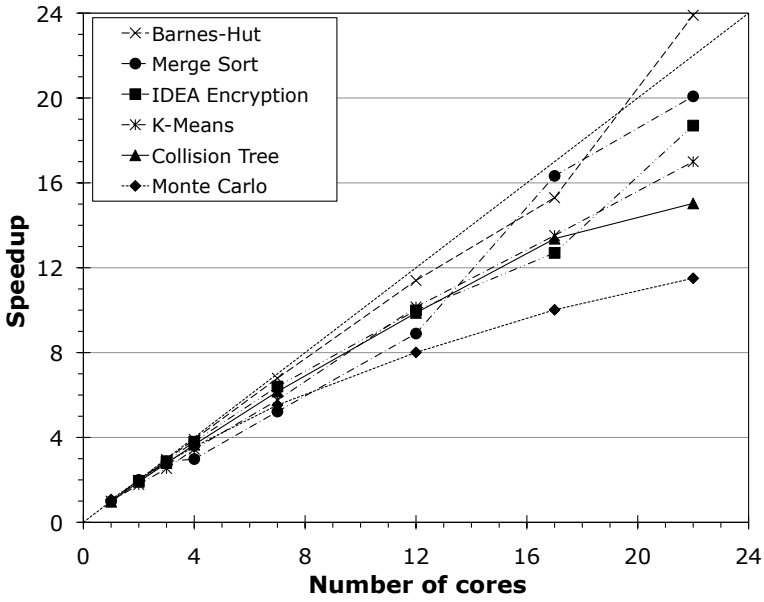


**Fig. 5.** Speedups of DPJ parallel codes compared to sequential versions

one in Cilk [4] to support a lightweight parallel tasking model. Our compiler translates DPJ's `cobegin` and `foreach` constructs into calls to this framework.

We implemented versions of the following programs with DPJ for our evaluation: A Monte Carlo financial simulation and an IDEA encryption code from the Java Grande benchmark suite, a parallel merge sort algorithm, the force computation from the Barnes-Hut n-body simulation [16], k-means clustering from the STAMP benchmarks [14], and a tree-based collision detection algorithm from the open-source JMonkey game engine. In each case, we started with a sequential version and then added DPJ annotations.

### 3.1 Performance

One of the metrics on which we evaluated DPJ was performance: what are the speedups achieved in practice for each algorithm? This gives a quantitative measure of how much parallelism can be expressed when using the DPJ type system. For this evaluation, we ran each algorithm on a 24-core multiprocessor system running Linux (a Dell R900 with four six-core Xeon E7450 processors). The speedups achieved are shown in Figure 5. These speedups are relative to a sequential version of each code, with no overhead from DPJ or other multithreaded runtimes. The results shown are for the inputs and algorithmic parameters that gave the best performance, since we wanted to evaluate how much parallelism DPJ can express, and the sensitivity of the algorithms to these factors is not a consequence of using DPJ.

All the algorithms achieved moderate to good scalability in our evaluation, with Barnes-Hut and Merge Sort showing near-ideal performance scaling. For

three of the codes, we had a version parallelized with Java threads to compare against: the Java Grande codes for IDEA and Monte Carlo, and a version of Barnes-Hut that we wrote and tuned. In each of these cases, the DPJ version achieved speedups similar to or better than the Java version on the same machine. Thus, the parallelism in these codes could be expressed as efficiently with DPJ as with a lower-level parallel programming model that does not provide DPJ's guarantees of race-freedom and determinism. The basic version of DPJ described here requires no speculation or runtime checks (except for very small runtime costs for the array partitioning mechanism), so it adds negligible runtime overhead for achieving determinism.

### 3.2 Expressivity

DPJ achieved good performance in our evaluation and imposed essentially no runtime overhead to achieve its guarantee of determinism, However, the type system does impose some constraints on algorithmic design choices. We evaluated whether DPJ was able to express all the available parallelism in our benchmarks. For Merge Sort, Monte Carlo, IDEA, K-Means, and Collision Tree, we we able to express *all* of the available parallelism (apart from vector parallelism, which we did not consider here). The Barnes-Hut algorithm has four major phases in each time step: tree building, center-of-mass computation, force calculations, and position calculations. The last three phases can be straightforwardly expressed with DPJ, but we only studied the force calculation here, since it is the dominant part of the overall computation. The tree building phase could also be expressed, but we would have to use a divide-and-conquer approach, rather than inserting bodies from the root using locking as in [16].

All the major novel features of DPJ were used in at least one of the benchmarks, including distinctions between RPLs from both the left and right, index-parameterized arrays, array partitioning, and commutativity annotations. Our evaluation did show some limitations of the language design. In Barnes-Hut, the elements of an array must be reordered on each time step. In the DPJ version, the array is index-parameterized, and this requires us to copy the data objects rather than simply re-ordering the references to them in the array, because their region typing must be consistent with their new position in the array. Using frameworks or run-time checks to relax this type of restriction is one direction of our ongoing work, described in Section 5.

### 3.3 Annotation burden

To assess how much effort is required to port a program to DPJ, we examined the annotations that were required in each of the benchmark programs. We found that the fraction of lines of code changed varied from 1.0% to 22.6%, averaging only 10.7% across the six programs. Most of the changed lines involved writing RPL arguments when instantiating types, followed by writing method effect summaries. We think this annotation burden is reasonable when considered in light of the advantages provided by DPJ.

We believe that when developing a program using *any* parallel programming model, the time and effort spent writing, testing, and debugging it will be dominated by the time needed to understand the parallelism and sharing patterns, and to debug the parallel code. DPJ's regions and effects provide a concrete model for the programmer to use when reasoning about parallelism and sharing, and they allow the programmer to write down his or her understanding of these patterns in a way that will be checked by the compiler. This catches errors, and it provides documentation for future programmers seeking to understand the code. DPJ's determinism guarantee also greatly simplifies debugging, since there will be no nondeterministic variation between different runs of a program for a single input, and moreover, the programmer can reason about a parallel program in terms of an equivalent sequential version. Furthermore, programming tools can help the programmer to write annotations, as described next.

## 4 Inferring Method Effect Summaries

We have been working on a development tool called DPJizer that helps developers write DPJ programs by *inferring* some of the annotations. In this section we discuss an algorithm we have developed for inferring method effect summaries [17]. In Section 5, we briefly discuss other forms of annotation inference that we are investigating.

The algorithm for inferring method effect summaries takes as input a DPJ program with all annotations except the summaries (i.e., region declarations and uses, and parallel control constructs), and it infers an effect summary for every method in the program. The inferred summaries must satisfy two properties:

1. The inferred effect summary of a method must cover all the actual effects of that method, including the effects of callees and the effects of overriding methods.
2. The inferred summaries must be precise enough to express the parallelism in the program (assuming the parallelism can be expressed in DPJ).

We have implemented the summary inference algorithm in a refactoring tool called DPJizer based on the Eclipse infrastructure for performing textual changes. As a result, the user interface follows the standard user interface for refactorings in Eclipse.

**Constraint Collection** The summary inference algorithm is an interprocedural analysis that proceeds in two phases. The first phase is *constraint collection*. It captures all the conditions necessary to satisfy property (1) above. There are four kinds of constraints: reads, writes, invokes, and overrides.

*Reads and writes constraints:* A reads or writes constraint on method $M$ means that $M$ performed the corresponding read or write directly in its body. For example, the statement `x=5` occurring in $M$, where `x` is in region $R$, produces the constraint `writes` $R$ for $M$.

*Invokes and overrides constraints:* The invokes constraints propagate effects from callee methods to caller methods. For example, if method $M_1$ calls method

$M_2$, and $M_2$ writes region $R$, then $M_1$ must also report the effect `writes` $R$. Similarly, overrides constraints propagate effects from overriding methods to overridden methods.

Internally, the constraints are represented by a directed graph with methods as nodes and constraints as edges. (For invokes constraints, this graph is a call graph.) When a method or its enclosing class has region parameters, the edge is labeled with the substitution of actual for formal region arguments. For example, if method $M_1$ has region parameter $P$, and $M_1$ is invoked by $M_2$ with region $R$ bound to $P$, then the algorithm labels the edge $M_2 \rightarrow M_1$ with the substitution $P \mapsto R$. That means that $M_2$ must cover the effects of $M_1$ under the substitution $P \mapsto R$.

**Constraint Solving** The second phase of the algorithm solves the constraints collected by the first phase, using a heuristic to generate effects that satisfy property (2) in practice. For each edge $M_1 \rightarrow M_2$ with label $\theta$ in the constraint graph, the constraint solver applies the substitution $\theta$ on the effects of method $M_2$ and adds the resulting effects to the set of effects of method $M_1$.

If applied naively, this approach would never reach a fixed point, because left-recursive substitutions such as $P \mapsto P : R$ can lead to an infinite number of different RPLs. Therefore, the algorithm uses the `*` RPL element (Section 2.3) to *truncate* any RPL of longer than a fixed length, which is tunable by the programmer. For example, if the programmer sets the maximum length to three, then the RPL `A:B:C:D` is truncated to `A:B:*`. This technique generates effects that are precise enough to capture the parallelism in the examples we studied.

**Evaluation** We evaluated our refactoring tool on eleven programs, with a total of around 5000 lines of of DPJ code, and 406 read or write effects appearing in effect summaries. We removed the effect summaries and had our refactoring tool infer them. In all cases the tool inferred correct effects, i.e., the effects satisfied property (1) above. In addition, the inferred effects satisfied property (2) in all cases: they were precise enough that the compiler could prove noninterference for the parallel tasks. Moreover, 84 of the 406 handwritten effects were either redundant or less precise than the ones inferred by the tool. In all other cases, the inferred effects were as precise as the corresponding handwritten effects. Despite the extra precision, we found that the simplifications mentioned above kept the inferred effects relatively simple and comprehensible. Overall, this evaluation shows that our tools infers effect summaries *fully automatically*, and these summaries are both correct and fine-grained enough to permit the full parallelism identified by the programmer.

## 5 Ongoing and Future Work

### 5.1 Object-Oriented Parallel Frameworks

We are extending DPJ to support object-oriented parallel frameworks [5]. To date we have investigated two kinds of frameworks. First, we have written collection frameworks (similar to ParallelArray for Java [2]) that provide parallel

operations such as map, reduce, filter, and scan on their elements via user-defined methods. Second, we have implemented a pipeline algorithm template (similar to the Pipeline template in Intel's Threading Building Blocks [15]). For both kinds of frameworks, the key challenge is to prevent the user-defined methods from causing conflicts when invoked by the parallel framework. For example, a user-defined map function must not do an unsynchronized write to a global variable. The OO framework support incorporates several extensions to the DPJ type and effect system for expressing generic types and effects in framework APIs, with appropriate constraints on the effects of user-defined methods [5, 6].

Frameworks provide a way to make DPJ more expressive, by representing operations (such as reshuffling an array of references, then updating the elements in parallel, as discussed in Section 3.2) that the core type and effect system described in Section 2 cannot express. Such operations can be encapsulated in a framework (for example, a parallel array with a reshuffling operation). The type and effect system can then check that *uses* of the framework conform to the API requirements, while the framework *internals* are checked by more flexible (but more complex and/or weaker) methods such as program logic or testing. This approach separates concerns between framework designer and user, and it fosters modular checking. It also enables different forms of verification with different tradeoffs in complexity and power to work together.

Frameworks can also to extend the fork-join model of DPJ by adding other parallel control abstractions, including other forms of synchronization (e.g., producer-consumer) or domain-specific abstractions (e.g., kd-tree querying and sparse matrix computations). For very common or general patterns (such as pipelines), some of these framework abstractions could become first-class language features.

### 5.2 Controlled Nondeterminism

We are extending DPJ to support controlled nondeterminism [8], for algorithms such as branch-and-bound search, where several correct answers are equally acceptable. The key new language features for controlled nondeterminism are (1) `foreach_nd` and `cobegin_nd` statements that operate like `foreach` and `cobegin`, except they allow interference among their parallel tasks; (2) `atomic` statements supported by software transactional memory (STM); and (3) *atomic regions* and *atomic effects* for reasoning about which memory regions may have interfering effects, and where effects occur inside `atomic` statements.

Together, these features provide several strong guarantees for nondeterministic programs. First, the extended language is *deterministic by default*: the program is guaranteed to be deterministic *unless* the programmer explicitly introduces nondeterminism with `foreach_nd` or `cobegin_nd`. Second, the extended language provides both strong isolation (i.e., the program behaves as if it is a sequential interleaving of `atomic` statements and unguarded code sections) and data race freedom. This is true even if the underlying STM provides only weak isolation (i.e., it allows individual statements within `atomic` statements to be interleaved with other program statements outside any `atomic` statement). Third,

`foreach` and `cobegin` are guaranteed to behave as isolated statements (as if they are enclosed in an `atomic` statement). Finally, the extended type and effect system allows the compiler to boost the STM performance by removing unnecessary synchronization for memory accesses that can never cause interference.

### 5.3 Inferring Region Information

The next step after inferring method effect summaries (Section 4) is to infer region and type information, given a program annotated with parallel constructs. This is a more challenging goal, because there are many more degrees of freedom. Our strategy is to have the programmer put in partial information (e.g., partition the heap into regions and put in the parallel constructs), and then have the tool use domain knowledge about how the DPJ type and effect system works to infer the types and effects that can guarantee noninterference.

### 5.4 Runtime Checks

We are investigating the use of runtime checking of types and effects, as a complementary mode to compile time checking. The advantage of runtime checking is that it is can support precise checking in cases for which the type system is not expressive enough. The disadvantages are (1) it adds overhead, so it may not be useful except for testing and debugging; and (2) it provides only a fail-stop check, instead of a compile-time guarantee of correctness. One place where runtime checking could be particularly useful is in relaxing the prohibition on inconsistent type assignments (e.g., in array reshuffling, discussed above). In particular, if the runtime can guarantee that a reference is unique, then a safe cast to a different type can be performed at runtime [3]. Using both static and runtime checks, and providing good integration between the two, would provide a powerful set of options for managing the tradeoffs between expressivity, performance, and correctness guarantees of the different approaches.

## 6 Discussion

Disciplined deterministic and non-deterministic parallel programming models have strengths and challenges. On the positive side, they hold out the promise to greatly simplify parallel program understanding, debugging, testing, and maintenance. Annotation-based solutions like DPJ also provide useful documentation and modular reasoning, which have far broader value in software development. Furthermore, DPJ shows that these goals can be achieved largely through compile-time checking, which further reduces debugging, testing, and maintenance costs.

The challenges for popular acceptance of such models, however, are daunting. Perhaps most important, programmers and large software teams are reluctant at worst, and extremely slow at best, to adopt new language mechanisms that do not directly contribute new functionality (even if the net gain in productivity

could allow more time spent on functionality and performance). Interactive tools to *infer* the annotations as far as possible will be critical to overcome this reluctance. A second challenge is that the language technologies are still immature and there is insufficient experience with realistic applications to know whether the new mechanisms will be able to express the idioms in such applications, whether the annotation burden for such large applications will be manageable, and whether the inference tools will be scaleable enough to work on them. The DPJ group is working closely with Intel and an Intel client with a large application code base to study these questions in commercial application settings.

## References


1. http://gee.cs.oswego.edu/dl/concurrency-interest.
2. http://gee.cs.oswego.edu/dl/jsr166/dist/extra166ydocs/index.html?extra166y/package-tree.html.
3. Z. Anderson, D. Gay, R. Ennals, and E. Brewer. SharC: Checking data sharing strategies for multithreaded C. *PLDI*, 2008.
4. R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: An efficient multithreaded runtime system. *PPOPP*, 1995.
5. R. L. Bocchino and V. S. Adve. Types, regions, and effects for safe programming with object-oriented parallel frameworks. Technical report, Univ. of Illinois at Urbana-Champaign, http://dpj.cs.uiuc.edu, 2009 (Revised 2010).
6. R. L. Bocchino and V. S. Adve. Core DPJ with Object-Oriented Framework Support. Technical report, Univ. of Illinois at Urbana-Champaign, http://dpj.cs.uiuc.edu, 2010.
7. R. L. Bocchino, V. S. Adve, S. V. Adve, and M. Snir. Parallel programming must be deterministic by default. *HOTPAR*, 2009.
8. R. L. Bocchino, S. Heumann, N. Honarmand, S. Adve, V. Adve, A. Welc, T. Shpeisman, and Y. Ni. Safe nondeterminism in a deterministic-by-default parallel language. Technical report, Univ. of Illinois at Urbana-Champaign, http://dpj.cs.uiuc.edu, 2010.
9. R. L. Bocchino Jr., V. S. Adve, D. Dig, S. V. Adve, S. Heumann, R. Komuravelli, J. Overbey, P. Simmons, H. Sung, and M. Vakilian. A type and effect system for Deterministic Parallel Java. *OOPSLA*, 2009.
10. J. Larus and R. Rajwar. *Transactional Memory (Synthesis Lectures on Computer Architecture)*. Morgan & Claypool Publishers, 2007.
11. H.-W. Loidl et al. Comparing parallel functional languages: Programming and performance. *Higher Order Symbol. Comput.*, 2003.
12. J. M. Lucassen and D. K. Gifford. Polymorphic effect systems. *POPL*, 1988.
13. M. Metcalf and J. Reid. *Fortran 90 Explained*. Oxford University Press, 1992.
14. C. C. Minh, J. Chung, C. Kozyrakis, and K. Olukotun. STAMP: Stanford transactional applications for multi-processing. *IISWC*, 2008.
15. J. Reinders. *Intel Threading Building Blocks: Outfitting C++ for Multi-core Processor Parallelism*. O'Reilly Media, 2007.
16. J. P. Singh, W.-D. Weber, and A. Gupta. SPLASH: Stanford parallel applications for shared-memory. Technical report, Stanford University, 1992.
17. M. Vakilian, D. Dig, R. Bocchino, J. Overbey, V. Adve, and R. Johnson. Inferring method effect summaries for nested heap regions. *ASE*, 2009.