A Type and Effect System for Deterministic Parallel Java

Robert Bocchino, Vikram Adve, Danny Dig, Sarita Adve, Stephen Heumann, Rakesh Komuravelli, Jeffrey Overbey, Patrick Simmons, Hyojin Sung, Mohsen Vakilian

http://dpj.cs.uiuc.edu/

This work is supported by Intel, Microsoft, and NSF

UPCRC Illinois
Universal Parallel Computing Research Center

Deterministic Execution

Many parallel programs are (intended to be) deterministic

- Non-interactive computation
- Accept input, compute, produce output
- Parallelism for *performance*, not part of *specification*

Same input always produces same output

Deterministic Execution

Many parallel programs are (intended to be) deterministic

- Non-interactive computation
- Accept input, compute, produce output
- Parallelism for *performance*, not part of *specification*

Same input always produces same output

It would help if our programming languages could guarantee deterministic execution

UPCRC Illinois
Universal Parallel Computing Research Center

Benefits of Guaranteed Determinism

Can reason "almost sequentially"

- Sequential semantics, parallel performance model

No subtle parallelism bugs

- No data races or deadlocks
- No complex memory models

Simpler testing

- Only one output to check per input
- Sequential to parallel port is easier

Simpler bug reproduction and debugging

UPCRC Illinois
Universal Parallel Computing Research Center

The Bad News

Some languages do guarantee determinism

- Functional, SIMD, explicit dataflow

But mainstream, general-purpose languages do not

- Imperative, OO languages (Java, C++, C#)

Expressive features obscure data flow

- Pointers/references to mutable objects
- Reference aliasing
- Inheritance and polymorphism

Deterministic Parallel Java (DPJ)

Modern object-oriented language

- We used Java because it's clean and safe
- Ideas apply to other OO languages (C++, C#) as well

Fork-join style of parallelism

- `foreach` loop, `cobegin` block
- Additional control constructs are future work

Type and effect system ensures noninterference

- Disallow conflicting operations between parallel tasks

UPCRC Illinois
Universal Parallel Computing Research Center

Deterministic Parallel Java (DPJ)

Modern object-oriented language

- We used Java because it's clean and safe
- Ideas apply to other OO languages (C++, C#) as well

Fork-join style of parallelism

- `foreach` loop, `cobegin` block
- Additional control constructs are future work

Type and effect system ensures noninterference

- Disallow conflicting operations between parallel tasks

Guarantees deterministic execution at compile time

UPCRC Illinois
Universal Parallel Computing Research Center

Contributions

Type system features for deterministic parallelism

- Region path lists for effect shapes and subtyping
- Novel array typing
  - Index parameterized arrays for parallel updates through references
  - Owner regions + library classes for divide and conquer recursion
- Support for commutative operations on concurrent data structures

Language description

Formal definition and soundess proof

Empirical evaluation

UPCRC Illinois
Universal Parallel Computing Research Center

Outline

Regions and Effects

New Type System Features

Evaluation

Conclusion

UPCRC Illinois
Universal Parallel Computing Research Center

Outline

Regions and Effects

New Type System Features

Evaluation

Conclusion

See our paper for more details

UPCRC Illinois
Universal Parallel Computing Research Center

Outline

Regions and Effects

New Type System Features

Evaluation

Conclusion

See our paper for more details

See our technical report for full definitions and proofs

UPCRC Illinois
Universal Parallel Computing Research Center

Regions and Effects

Regions

- A *region* is a name for a set of memory locations
- Programmer assigns regions to fields and array cells

Effects

- An *effect* is a read or write operation on a set of regions
- Programmer summarizes effects of method bodies

Compiler checks that

- Effect summaries are correct
- Parallel sections are noninterfering

Regions and Effects

Regions

- A *region* is a name for a set of memory locations
- Programmer assigns regions to fields and array cells

Effects

- An *effect* is a read or write operation on a set of regions
- Programmer summarizes effects of method bodies

Compiler checks that

- Effect summaries are correct
- Parallel sections are noninterfering

Documents side effects

Regions and Effects

Regions

- A *region* is a name for a set of memory locations
- Programmer assigns regions to fields and array cells

Effects

- An *effect* is a read or write operation on a set of regions
- Programmer summarizes effects of method bodies

Compiler checks that

- Effect summaries are correct
- Parallel sections are noninterfering

Documents side effects
Supports modular checking of noninterference

Example: A Pair Class

```
class Pair {
  region Fst, Snd;
  int fst in Fst;
  int snd in Snd;
  void setFst(int fst) writes Fst {
    this.fst = fst;
  }
  void setSnd(int snd) writes Snd {
    this.snd = snd;
  }
  void setBoth(int fst, int snd) {
    cobegin {
      setFst(fst); /* writes Fst */
      setSnd(snd); /* writes Snd */
    }
  }
}
```

Pair		
Pair.Fst	fst	3
Pair.Snd	snd	42

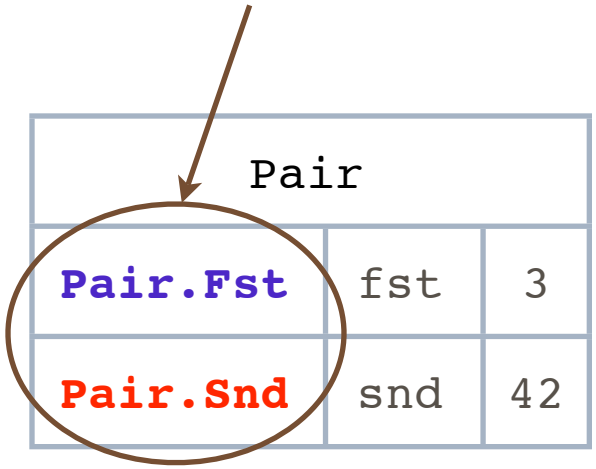
Declaring and using region names

UPCRC Illinois
Universal Parallel Computing Research Center

Example: A Pair Class

```
class Pair {
  region Fst, Snd;
  int fst in Fst;
  int snd in Snd;
  void setFst(int fst) writes Fst {
    this.fst = fst;
  }
  void setSnd(int snd) writes Snd {
    this.snd = snd;
  }
  void setBoth(int fst, int snd) {
    cobegin {
      setFst(fst); /* writes Fst */
      setSnd(snd); /* writes Snd */
    }
  }
}
```

Region names have static scope (one per class)



Declaring and using region names

Example: A Pair Class

```
class Pair {
  region Fst, Snd;
  int fst in Fst;
  int snd in Snd;
  void setFst(int fst) writes Fst {
    this.fst = fst;
  }
  void setSnd(int snd) writes Snd {
    this.snd = snd;
  }
  void setBoth(int fst, int snd) {
    cobegin {
      setFst(fst); /* writes Fst */
      setSnd(snd); /* writes Snd */
    }
  }
}
```

Pair		
Pair.Fst	fst	3
Pair.Snd	snd	42

Writing method effect summaries

UPCRC Illinois
Universal Parallel Computing Research Center

Example: A Pair Class

```
class Pair {
  region Fst, Snd;
  int fst in Fst;
  int snd in Snd;
  void setFst(int fst) writes Fst {
    this.fst = fst;
  }
  void setSnd(int snd) writes Snd {
    this.snd = snd;
  }
  void setBoth(int fst, int snd) {
    cobegin {
      setFst(fst); /* writes Fst */
      setSnd(snd); /* writes Snd */
    }
  }
}
```

Pair		
Pair.Fst	fst	3
Pair.Snd	snd	42

Expressing parallelism

UPCRC Illinois
Universal Parallel Computing Research Center

Region Parameters

```
class SimpleTree<region P> {
  region L, R;
  int data in P;
  SimpleTree<L> left = new SimpleTree<L>();
  SimpleTree<R> right = new SimpleTree<R>();
  void updateChildren()
    cobegin {
      left.data = 0; /* writes L */
      right.data = 1; /* writes R */
    }
  }
}
```

UPCRC Illinois
Universal Parallel Computing Research Center

Region Parameters

Class is parameterized by region P

```
class SimpleTree<region P> {
  region L, R;
  int data in P;
  SimpleTree<L> left = new SimpleTree<L>();
  SimpleTree<R> right = new SimpleTree<R>();
  void updateChildren()
    cobegin {
      left.data = 0; /* writes L */
      right.data = 1; /* writes R */
    }
  }
}
```

UPCRC Illinois
Universal Parallel Computing Research Center

Region Parameters

Class is parameterized by region P

Field data resides in P

```
class SimpleTree<region P> {
  region L, R;
  int data in P;
  SimpleTree<L> left = new SimpleTree<L>();
  SimpleTree<R> right = new SimpleTree<R>();
  void updateChildren()
    cobegin {
      left.data = 0; /* writes L */
      right.data = 1; /* writes R */
    }
  }
}
```

UPCRC Illinois
Universal Parallel Computing Research Center

Region Parameters

```
class SimpleTree<region P> {
  region L, R;
  int data in P;
  SimpleTree<L> left = new SimpleTree<L>();
  SimpleTree<R> right = new SimpleTree<R>();
  void updateChildren()
    cobegin {
      left.data = 0; /* writes L */
      right.data = 1; /* writes R */
    }
  }
}
```

Class is parameterized by region `P`

Field `data` resides in `P`

Types provide actual regions for computing effects

UPCRC Illinois
Universal Parallel Computing Research Center

Region Parameters

```
class SimpleTree<region P> {
  region L, R;
  int data in P;
  SimpleTree<L> left = new SimpleTree<L>();
  SimpleTree<R> right = new SimpleTree<R>();
  void updateChildren()
    cobegin {
      left.data = 0; /* writes L */
      right.data = 1; /* writes R */
    }
  }
}
```

Class is parameterized by region `P`

Field `data` resides in `P`

Types provide actual regions for computing effects

Distinguish different object instances

UPCRC Illinois
Universal Parallel Computing Research Center

Outline

Regions and Effects

New Type System Features

Evaluation

Conclusion

UPCRC Illinois
Universal Parallel Computing Research Center

Important Parallel Patterns

Pattern	DPJ Support

Important Parallel Patterns

Pattern	DPJ Support
1. Field-granularity updates to nested data structures	Region path lists (RPLs)

Important Parallel Patterns

Pattern	DPJ Support
1. Field-granularity updates to nested data structures	Region path lists (RPLs)
2. Parallel updates through an array of object references	RPLs + Index-parameterized arrays

Important Parallel Patterns

Pattern	DPJ Support
1. Field-granularity updates to nested data structures	Region path lists (RPLs)
2. Parallel updates through an array of object references	RPLs + Index-parameterized arrays
3. Parallel divide-and-conquer updates on arrays	RPLs + Owner regions + Library classes

Important Parallel Patterns

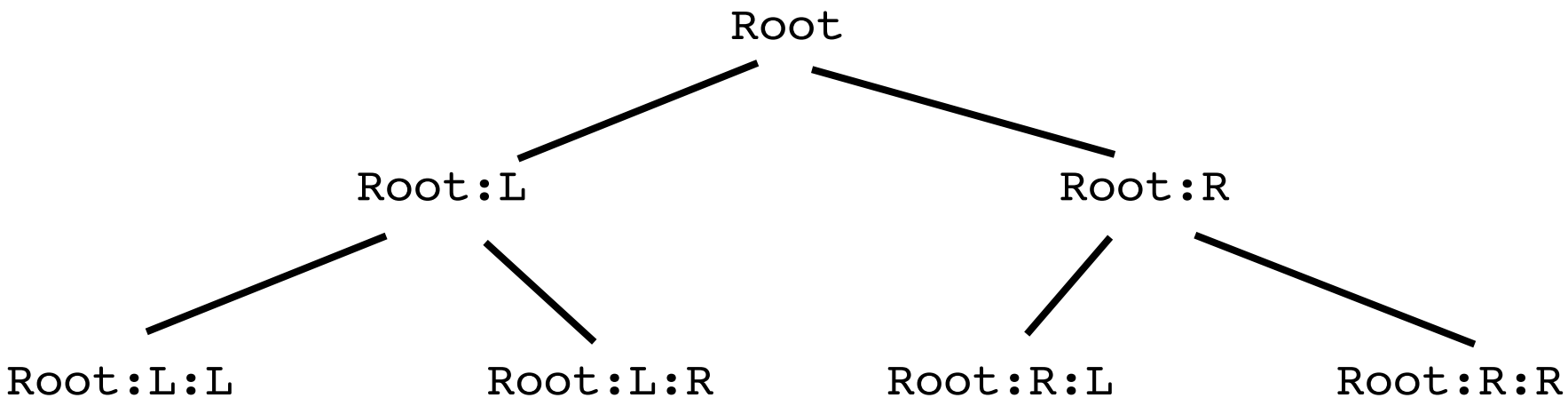
Pattern	DPJ Support
1. Field-granularity updates to nested data structures	Region path lists (RPLs)
2. Parallel updates through an array of object references	RPLs + Index-parameterized arrays
3. Parallel divide-and-conquer updates on arrays	RPLs + Owner regions + Library classes
4. Commutative operations on concurrent data structures	Commutativity annotation + invocation effect

Important Parallel Patterns

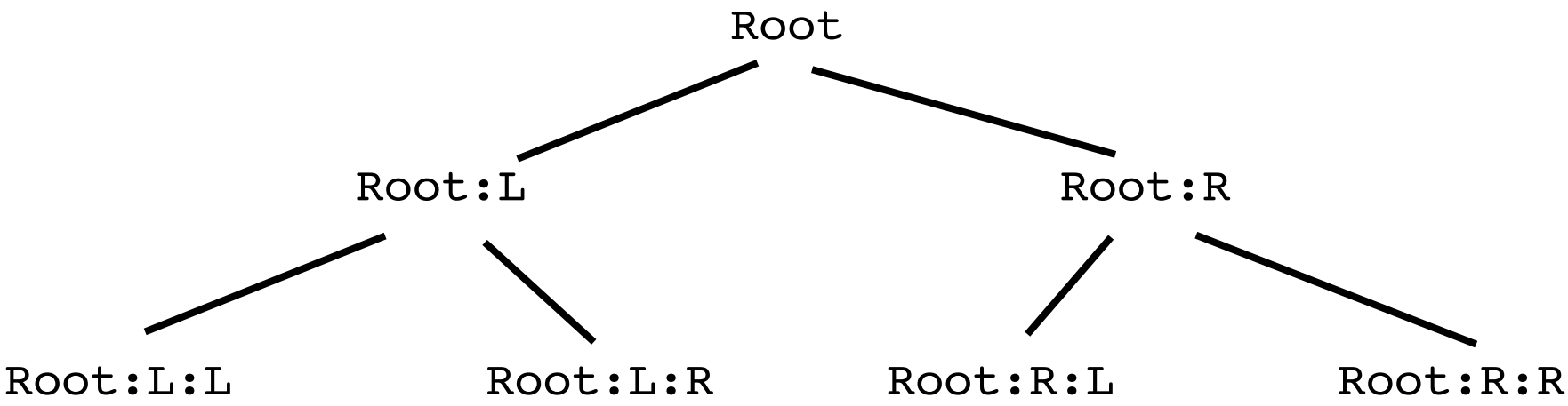
Pattern	**DPJ Support**
1. Field-granularity updates to nested data structures	Region path lists (RPLs)
2. Parallel updates through an array of object references	RPLs + Index-parameterized arrays
3. Parallel divide-and-conquer updates on arrays	RPLs + Owner regions + Library classes
4. Commutative operations on concurrent data structures	Commutativity annotation + invocation effect

No previous type-based system supports any of these patterns

1. Region Path Lists (RPLs)

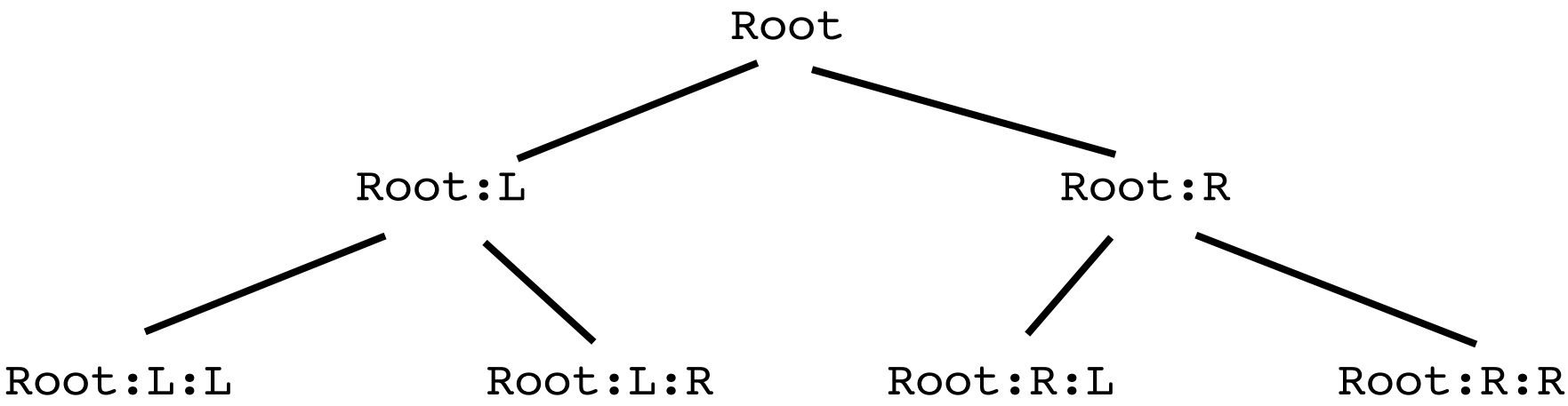


1. Region Path Lists (RPLs)



`Root:L:R` *is a Region Path List, or RPL*

1. Region Path Lists (RPLs)

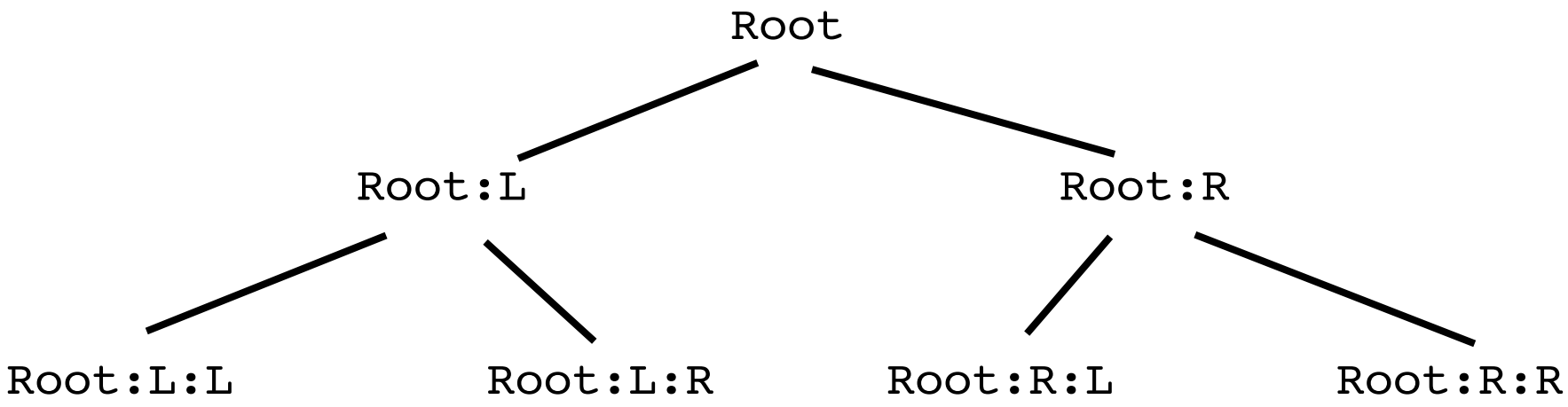


`Root:L:R` *is a Region Path List, or RPL*

RPLs naturally form a tree

UPCRC Illinois
Universal Parallel Computing Research Center

1. Region Path Lists (RPLs)



`Root:L:R` *is a Region Path List, or RPL*

RPLs naturally form a tree

`Root:L:R` *is a child of* **`Root:L`**

UPCRC Illinois
Universal Parallel Computing Research Center

Using RPLs to Write Types

```
class Tree<region P> {
  region L, R;
  int data in P
  Tree<P:L> left in P:L;
  Tree<P:R> right in P:R;
  ...
}
```

Using RPLs to Write Types

```
class Tree<region P> {
  region L, R;
  int data in P
  Tree<P:L> left in P:L;
  Tree<P:R> right in P:R;
  ...
}
```

In the source code, RPLs can begin with parameters

Using RPLs to Write Types

```
class Tree<region P> {
  region L, R;
  int data in P
  Tree<P:L> left in P:L;
  Tree<P:R> right in P:R;
  ...
}
```

In the source code, RPLs can begin with parameters

P=Root	
data	Root
left	Root:L
right	Root:R

P=Root:L	
data	Root:L
left	Root:L:L
right	Root:L:R

P=Root:R	
data	Root:R
left	Root:R:L
right	Root:R:R

Using RPLs to Write Types

```
class Tree<region P> {
  region L, R;
  int data in P
  Tree<P:L> left in P:L;
  Tree<P:R> right in P:R;
  ...
}
```

In the source code, RPLs can begin with parameters

In the dynamic semantics, parameters are erased via left-recursive substitution

P=Root	
data	Root
left	Root:L
right	Root:R

P=Root:L	
data	Root:L
left	Root:L:L
right	Root:L:R

P=Root:R	
data	Root:R
left	Root:R:L
right	Root:R:R

Using RPLs To Write Effects

```
class Tree<region P> {
  region L, R;
  int data in P
  Tree<P:L> left in P:L;
  Tree<P:R> right in P:R;
  int increment() writes P:* {
    ++data; /* writes P */
    cobegin {
      /* writes P:L:* */
      if (left != null) left.increment();
      /* writes P:R:* */
      if (right != null) right.increment();
  }
}
```

Using RPLs To Write Effects

```
class Tree<region P> {
  region L, R;
  int data in P;
  Tree<P:L> left in P:L;
  Tree<P:R> right in P:R;
  int increment() writes P:* {
    ++data; /* writes P */
    cobegin {
      /* writes P:L:* */
      if (left != null) left.increment();
      /* writes P:R:* */
      if (right != null) right.increment();
  }
}
```

Using RPLs To Write Effects

```
class Tree<region P> {
  region L, R;
  int data in P;
  Tree<P:L> left in P:L;
  Tree<P:R> right in P:R;
  int increment() writes P:* {
    ++data; /* writes P */
    cobegin {
      /* writes P:L:* */
      if (left != null) left.increment();
      /* writes P:R:* */
      if (right != null) right.increment();
  }
}
```

Method effect summary

Using RPLs To Write Effects

```
class Tree<region P> {
  region L, R;
  int data in P;
  Tree<P:L> left in P:L;
  Tree<P:R> right in P:R;
  int increment() writes P:* {
    ++data; /* writes P */
    cobegin {
      /* writes P:L:* */
      if (left != null) left.increment();
      /* writes P:R:* */
      if (right != null) right.increment();
  }
}
```

Method effect summary

Effect inferred from type + summary

Using RPLs To Write Effects

```
class Tree<region P> {
  region L, R;
  int data in P;
  Tree<P:L> left in P:L;
  Tree<P:R> right in P:R;
  int increment() writes P:* {
    ++data; /* writes P */
    cobegin {
      /* writes P:L:* */
      if (left != null) left.increment();
      /* writes P:R:* */
      if (right != null) right.increment();
  }
}
```

Method effect summary

Effect inferred from type + summary

P:L:* ⊆ P:* *(inclusion)*

Using RPLs To Write Effects

```
class Tree<region P> {
  region L, R;
  int data in P;
  Tree<P:L> left in P:L;
  Tree<P:R> right in P:R;
  int increment() writes P:* {
    ++data; /* writes P */
    cobegin {
      /* writes P:L:* */
      if (left != null) left.increment();
      /* writes P:R:* */
      if (right != null) right.increment();
  }
}
```

Method effect summary

Effect inferred from type + summary

P:L:* ⊆ P:* *(inclusion)*

P:R:* ⊆ P:* *(inclusion)*

Using RPLs To Write Effects

```
class Tree<region P> {
  region L, R;
  int data in P;
  Tree<P:L> left in P:L;
  Tree<P:R> right in P:R;
  int increment() writes P:* {
    ++data; /* writes P */
    cobegin {
      /* writes P:L:* */
      if (left != null) left.increment();
      /* writes P:R:* */
      if (right != null) right.increment();
  }
}
```

Method effect summary

Effect inferred from type + summary

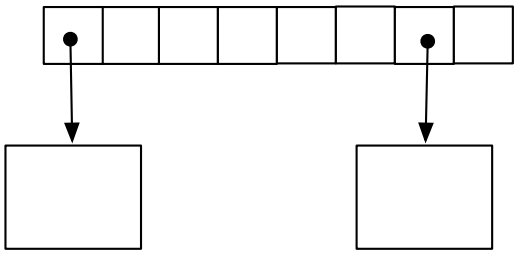
$\mathtt{P:L:*} \subseteq \mathtt{P:*}$ *(inclusion)*

$\mathtt{P:R:*} \subseteq \mathtt{P:*}$ *(inclusion)*

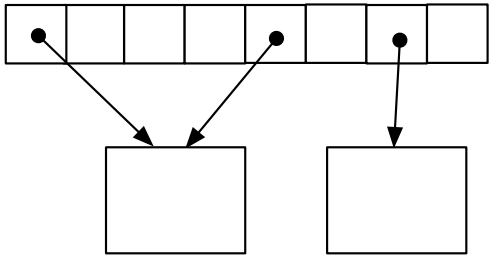
$\mathtt{P:L:*} \cap \mathtt{P:R:*} = \varnothing$ *(disjointness)*

2. Updating Objects through an Array

Key property is disjointness of references



Disjoint



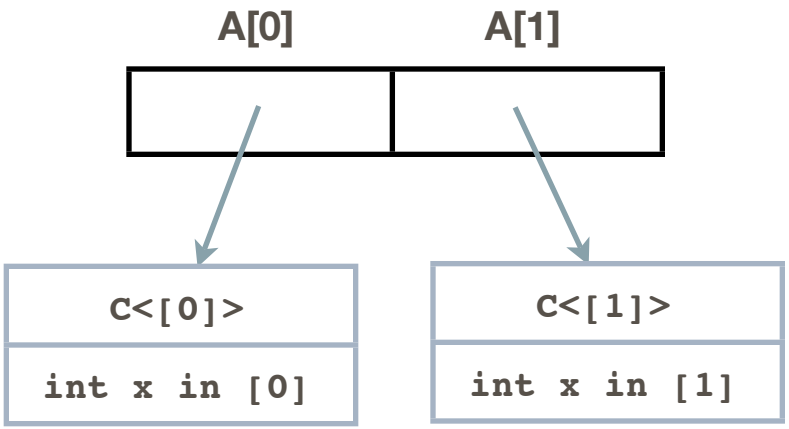
Non-disjoint

DPJ provides a novel array type `C<[i]>[]#i`

- Region `[i]` is parameterized by the index variable `i`
- Cell *i* of array `A` has type `C<[`*i*`]>`
- If $i \neq j$, then `A[`*i*`]` $\neq$ `A[`*j*`]`

Index-Parameterized Arrays

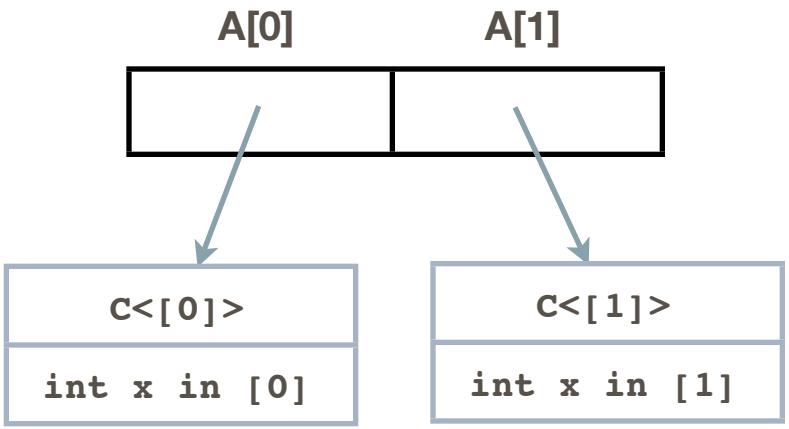
```
class C<region P> {
  int x in P
  void update() writes P {
   ...
  }
  ...
}
```



Index-Parameterized Arrays

Class is parameterized by region P

```
class C<region P> {
  int x in P
  void update() writes P {
   ...
  }
  ...
}
```

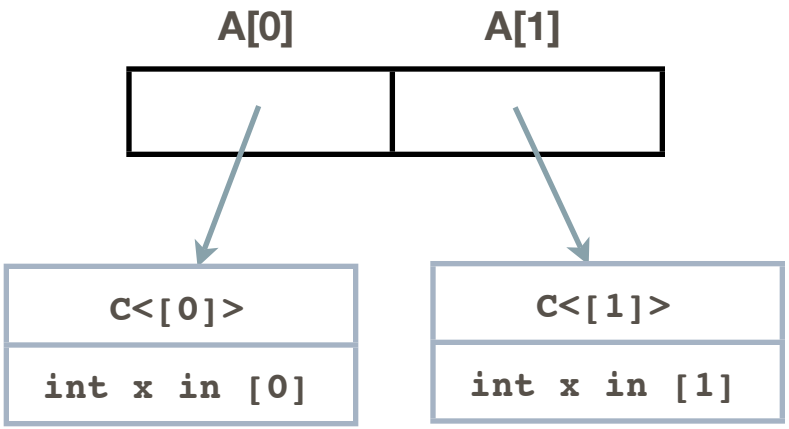


Index-Parameterized Arrays

Class is parameterized by region P

```
class C<region P> {
  int x in P
  void update() writes P {
   ...
  }
  ...
}
```

Field is placed in parameterized region

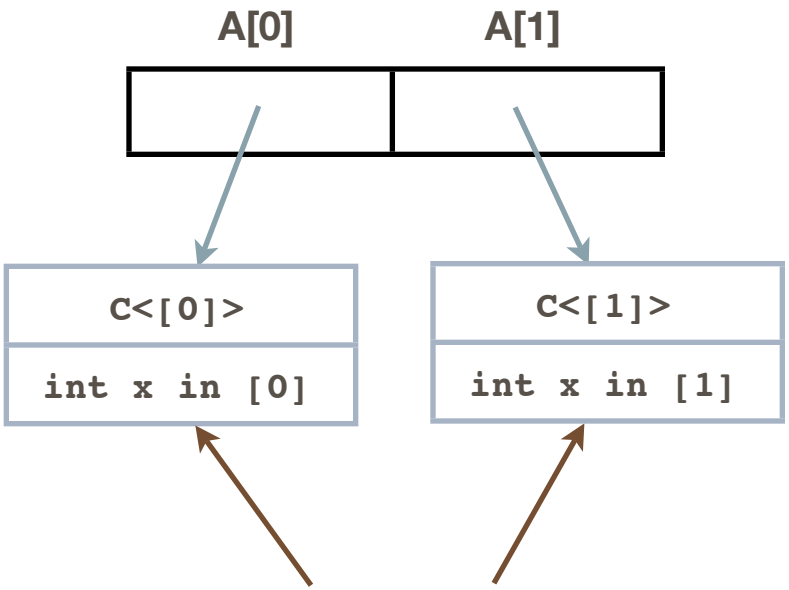


Index-Parameterized Arrays

Class is parameterized by region P

```
class C<region P> {
  int x in P
  void update() writes P {
   ...
  }
  ...
}
```

Field is placed in parameterized region



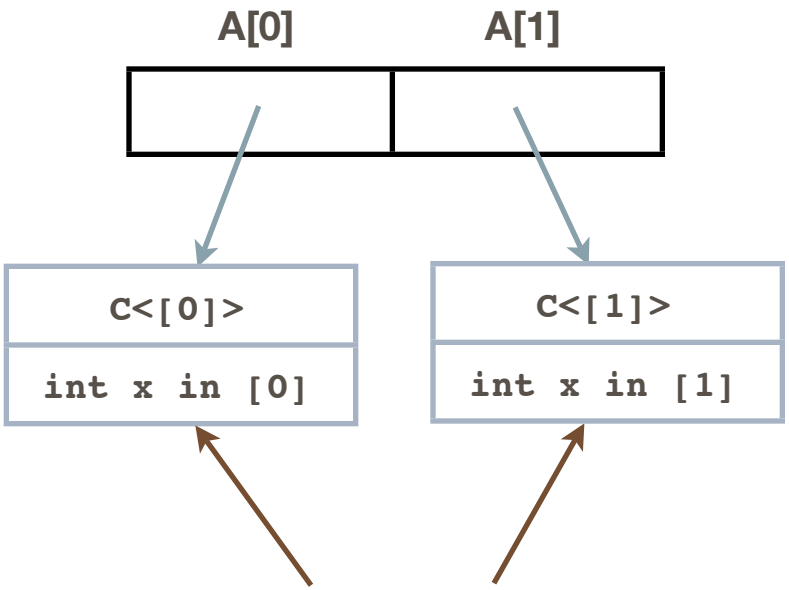
Objects in different array cells are created with different regions

Index-Parameterized Arrays

Class is parameterized by region P

```
class C<region P> {
  int x in P
  void update() writes P {
   ...
  }
  ...
}
```

Field is placed in parameterized region



Objects in different array cells are created with different regions

A[0].update() and A[1].update() are noninterfering

3. Parallel Divide and Conquer

Library classes express array partitioning

- `DPJArray`: Represents a subrange "view" of a Java array
- `DPJPartition`: Collection of disjoint subarrays

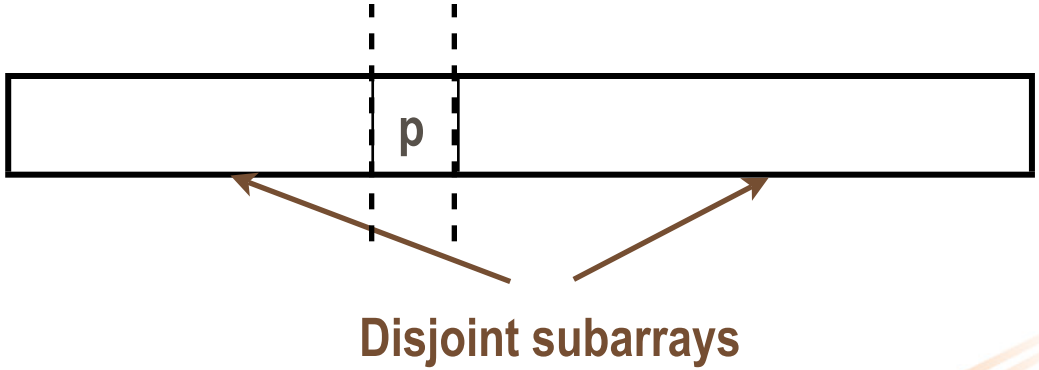
Type system enforces noninterference of effect

- *Owner region* (similar to object ownership) + RPLs
  - Associates partition with local variable
  - Supports multiple partitions of the same array
- Each subarray in a partition resides in a distinct region
- Recursion makes an array into a tree of regions

UPCRC Illinois
Universal Parallel Computing Research Center

Array Partitioning Example

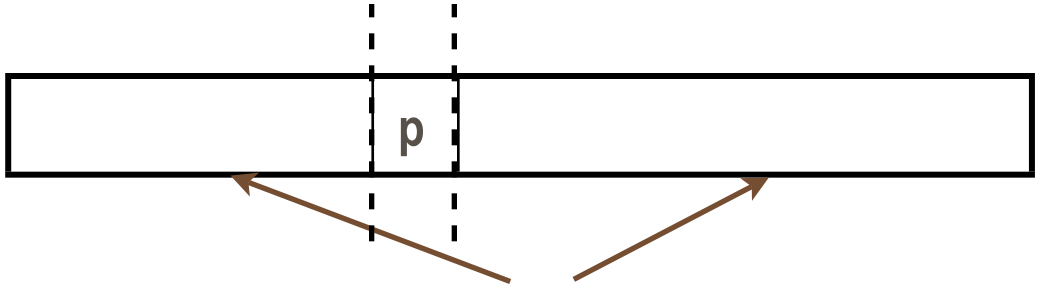
```
static <region R>void quicksort(DPJArray<R> A) writes R:* {
  int p = quicksortPartition(A);
  /* Chop array into two disjoint pieces */
  DPJPartition<R> segs = new DPJPartition<R>(A, p);
  cobegin {
    quicksort(segs.get(0) /* DPJArray<segs:[0]:*> */ );
    quicksort(segs.get(1) /* DPJArray<segs:[1]:*> */ );
  }
}
```



Array Partitioning Example

Partition

```
static <region R>void quicksort(DPJArray<R> A) writes R:* {
  int p = quicksortPartition(A);
  /* Chop array into two disjoint pieces */
  DPJPartition<R> segs = new DPJPartition<R>(A, p);
  cobegin {
    quicksort(segs.get(0) /* DPJArray<segs:[0]:*> */ );
    quicksort(segs.get(1) /* DPJArray<segs:[1]:*> */ );
  }
}
```



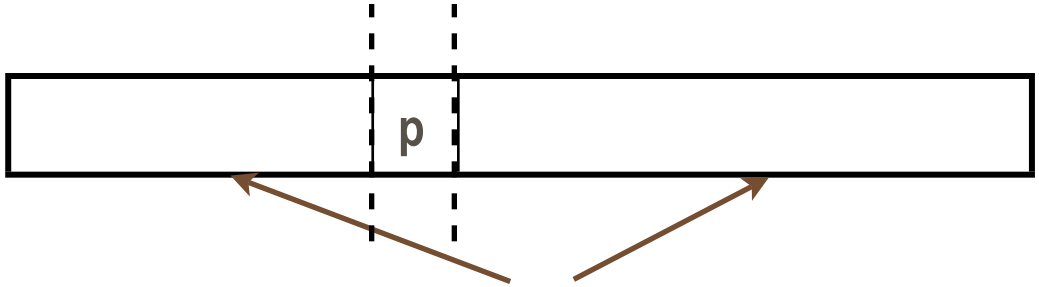
Disjoint subarrays

Array Partitioning Example

Partition

```
static <region R>void quicksort(DPJArray<R> A) writes R:* {
  int p = quicksortPartition(A);
  /* Chop array into two disjoint pieces */
  DPJPartition<R> segs = new DPJPartition<R>(A, p);
  cobegin {
    quicksort(segs.get(0) /* DPJArray<segs:[0]:*> */ );
    quicksort(segs.get(1) /* DPJArray<segs:[1]:*> */ );
  }
}
```

Subarrays



Disjoint subarrays

UPCRC Illinois
Universal Parallel Computing Research Center

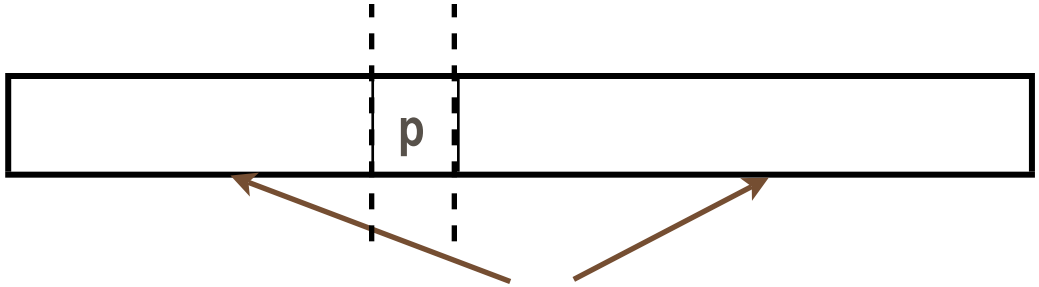
Array Partitioning Example

Partition

```
static <region R>void quicksort(DPJArray<R> A) writes R:* {
  int p = quicksortPartition(A);
  /* Chop array into two disjoint pieces */
  DPJPartition<R> segs = new DPJPartition<R>(A, p);
  cobegin {
    quicksort(segs.get(0) /* DPJArray<segs:[0]:*> */ );
    quicksort(segs.get(1) /* DPJArray<segs:[1]:*> */ );
  }
}
```

Subarrays

RPLs with owner regions



Disjoint subarrays

4. Commutativity Annotations

```
interface Set<type T, region R> {
    commutative void add(T e) writes R;
}
```

Annotation is

- Provided by library programmer
- Trusted by the compiler (not checked)

Annotation means

- `cobegin { add(e1); add(e2); }` is equivalent to one of
  - `add(e1); add(e2);`
  - `add(e2); add(e1);`
- Both sequences are equivalent

UPCRC Illinois
Universal Parallel Computing Research Center

Invocation Effect

```
foreach (int i in 0, n) {
    /* invokes Set.add with writes R */
    set.add(A[i]);
}
```

Noninterfering effects

- `invokes add with writes R`
- `invokes add with writes R`

Interfering effects

- `invokes add with writes R`
- `invokes size with reads R`

UPCRC Illinois
Universal Parallel Computing Research Center

Invocation Effect

```
foreach (int i in 0, n) {
    /* invokes Set.add with writes R */
    set.add(A[i]);
}
```

Noninterfering effects

- `invokes add with writes R`
- `invokes add with writes R`

Interfering effects

- `invokes add with writes R`
- `invokes size with reads R`

Invocation effect can also appear in method effect summary

UPCRC Illinois
Universal Parallel Computing Research Center

Outline

Regions and Effects

New Type System Features

Evaluation

Conclusion

UPCRC Illinois
Universal Parallel Computing Research Center

Implementation and Experiments

Implementation

- Extend Sun's javac compiler to do DPJ to Java translation
- Compile to ForkJoinTask Framework (Java 7)

Benchmarks

- IDEA encryption (Java Grande, 228 SLOC)
- Merge Sort (295 SLOC)
- K Means clustering (STAMP, 501 SLOC)
- Barnes-Hut force computation (SPLASH, 682 SLOC)
- JMonkey collision detection (1032 SLOC of large application)
- Monte Carlo financial simulation (Java Grande, 2877 SLOC)

Research Questions

1. Expressiveness

- Successful in expressing realistic parallel algorithms
- There are some limitations
  - Can't shuffle elements of an index-parameterized array

2. Usefulness of features

- All features described used in benchmarks

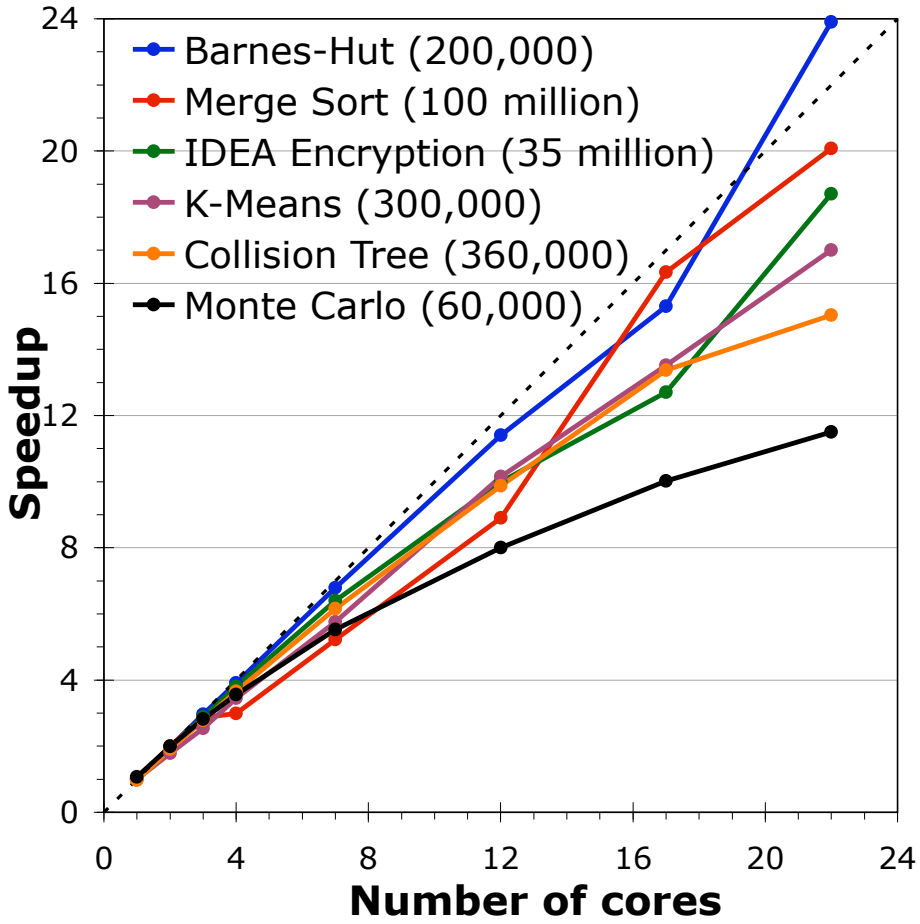
3. Porting effort

- Defaults allow incremental porting
  - JMonkey: Most of large application untouched
- Modified about 10% of total SLOC (most were types)
  - Effect inference [ASE 2009] improves this

4. Performance

UPCRC Illinois
Universal Parallel Computing Research Center

Performance Results



- DPJ enables good performance
- BH, Merge Sort showed near-ideal speedup on 16–22 cores
- IDEA, Monte Carlo, and BH nearly matched or beat handwritten threads

4 x 6 core x86 (Dell R900), 2GB main memory per core

UPCRC Illinois
Universal Parallel Computing Research Center

Conclusion

Deterministic parallel programming model

- Object-oriented language with references
- Compile-time determinism guarantee

Expressivity and performance

- Six realistic parallel codes
- Good performance
- Annotations useful and not too burdensome

Compiler will be available for download soon

http://dpj.cs.uiuc.edu/

UPCRC Illinois
Universal Parallel Computing Research Center