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
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*
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
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
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
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 soundness proof

Empirical evaluation
Outline

Regions and Effects
New Type System Features
Evaluation
Conclusion
Outline

Regions and Effects
New Type System Features
Evaluation
Conclusion

See our paper for more details
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
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 */
        }
    }
}

<table>
<thead>
<tr>
<th></th>
<th>Pair</th>
<th>Type</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pair.Fst</td>
<td>fst</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>Pair.Snd</td>
<td>snd</td>
<td>42</td>
<td></td>
</tr>
</tbody>
</table>

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 */
        }
    }
}

Region names have static scope (one per class)

<table>
<thead>
<tr>
<th>Pair</th>
<th>fst</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pair.Fst</td>
<td>snd</td>
<td>42</td>
</tr>
</tbody>
</table>

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 */
        }
    }
}

Writing method effect summaries

<table>
<thead>
<tr>
<th>Pair</th>
<th>fst</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pair.Fst</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pair.Snd</td>
<td>snd</td>
<td>42</td>
</tr>
</tbody>
</table>
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 */
        }
    }
}

Expressing parallelism

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Pair</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pair.Fst</td>
<td>fst</td>
<td>3</td>
</tr>
<tr>
<td>Pair.Snd</td>
<td>snd</td>
<td>42</td>
</tr>
</tbody>
</table>
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 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 */
        }
    }
}
Region Parameters

```java
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 */
    }
}
```
Region Parameters

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

Class is parameterized by region \texttt{P}

Field \texttt{data} resides in \texttt{P}

Types provide actual regions for computing effects
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 */
    }
}

Region Parameters

Class is parameterized by region P
Field data resides in P
Types provide actual regions for computing effects

Distinguish different object instances
Outline

Regions and Effects
New Type System Features
Evaluation
Conclusion
## Important Parallel Patterns

<table>
<thead>
<tr>
<th>Pattern</th>
<th>DPJ Support</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Important Parallel Patterns

<table>
<thead>
<tr>
<th>Pattern</th>
<th>DPJ Support</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Field-granularity updates to nested data structures</td>
<td>Region path lists (RPLs)</td>
</tr>
</tbody>
</table>


### Important Parallel Patterns

<table>
<thead>
<tr>
<th>Pattern</th>
<th>DPJ Support</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Field-granularity updates to nested data structures</td>
<td>Region path lists (RPLs)</td>
</tr>
<tr>
<td>2. Parallel updates through an array of object references</td>
<td>RPLs + Index-parameterized arrays</td>
</tr>
</tbody>
</table>
## Important Parallel Patterns

<table>
<thead>
<tr>
<th>Pattern</th>
<th>DPJ Support</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Field-granularity updates to nested data structures</td>
<td>Region path lists (RPLs)</td>
</tr>
<tr>
<td>2. Parallel updates through an array of object references</td>
<td>RPLs + Index-parameterized arrays</td>
</tr>
<tr>
<td>3. Parallel divide-and-conquer updates on arrays</td>
<td>RPLs + Owner regions + Library classes</td>
</tr>
</tbody>
</table>
## Important Parallel Patterns

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

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

*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
1. Region Path Lists (RPLs)

Root:P:R is a Region Path List, or RPL

RPLs naturally form a tree

Root:P:R is a child of Root:P:L
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;
    ...
}
Using RPLs to Write Types

In the source code, RPLs can begin with parameters.

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;
    ...
}

<table>
<thead>
<tr>
<th>P=Root</th>
</tr>
</thead>
<tbody>
<tr>
<td>data</td>
</tr>
<tr>
<td>Root:Root</td>
</tr>
<tr>
<td>left</td>
</tr>
<tr>
<td>Root:L:Root</td>
</tr>
<tr>
<td>right</td>
</tr>
<tr>
<td>Root:R:Root</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>P=Root:L</th>
</tr>
</thead>
<tbody>
<tr>
<td>data</td>
</tr>
<tr>
<td>Root:L:Root</td>
</tr>
<tr>
<td>left</td>
</tr>
<tr>
<td>Root:L:R:Root</td>
</tr>
<tr>
<td>right</td>
</tr>
<tr>
<td>Root:L:R:R</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>P=Root:R</th>
</tr>
</thead>
<tbody>
<tr>
<td>data</td>
</tr>
<tr>
<td>Root:R:Root</td>
</tr>
<tr>
<td>left</td>
</tr>
<tr>
<td>Root:R:L:Root</td>
</tr>
<tr>
<td>right</td>
</tr>
<tr>
<td>Root:R:R:R</td>
</tr>
</tbody>
</table>
Using RPLs to Write Types

```java
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.

<table>
<thead>
<tr>
<th>P=Root</th>
<th>data</th>
<th>Root</th>
</tr>
</thead>
<tbody>
<tr>
<td>left</td>
<td>Root:L</td>
<td></td>
</tr>
<tr>
<td>right</td>
<td>Root:R</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>P=Root:L</th>
<th>data</th>
<th>Root:L</th>
</tr>
</thead>
<tbody>
<tr>
<td>left</td>
<td>Root:L:L</td>
<td></td>
</tr>
<tr>
<td>right</td>
<td>Root:L:R</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>P=Root:R</th>
<th>data</th>
<th>Root:R</th>
</tr>
</thead>
<tbody>
<tr>
<td>left</td>
<td>Root:R:L</td>
<td></td>
</tr>
<tr>
<td>right</td>
<td>Root:R:R</td>
<td></td>
</tr>
</tbody>
</table>
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();
        }
    }
}
class Tree<\texttt{region} \texttt{P}> \{ 
    \texttt{region} L, R;
    \texttt{int} \texttt{data} \texttt{in} \texttt{P};
    \texttt{Tree<\texttt{P}:L>} \texttt{left} \texttt{in} \texttt{P}:L;
    \texttt{Tree<\texttt{P}:R>} \texttt{right} \texttt{in} \texttt{P}:R;
    \texttt{int} \texttt{increment()} \texttt{writes} \texttt{P:*} \{ 
        ++data; // \texttt{writes P */}
        \texttt{cobegin} \{ 
            /* \texttt{writes P:L:* */}
            \texttt{if} (\texttt{left} \neq \texttt{null}) \texttt{left.increment()};
            /* \texttt{writes P:R:* */}
            \texttt{if} (\texttt{right} \neq \texttt{null}) \texttt{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();  
        }  
    }  
}
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<region P:L> left in P:L;
    Tree<region P:R> right in P:R;
    int increment() \textbf{writes} P:* {
        ++data; /* \textbf{writes} P */
        cobegin {
            /* \textbf{writes} P:L:* */
            if (left != null) left.increment();
            /* \textbf{writes} P:R:* */
            if (right != null) right.increment();
        }
    }
}

\[ P:L:* \subseteq P:* \ (\text{inclusion}) \]
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() \texttt{writes P:*} {
    ++data; /* \texttt{writes P} */
    cobegin {
      /* \texttt{writes P:L:*} */
      if (left != null) left.increment();
      /* \texttt{writes P:R:*} */
      if (right != null) right.increment();
    }
  }
}

\( P:L:* \subseteq P:* \) (inclusion)
\( P:R:* \subseteq P:* \) (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();  
        }  
    }  
}

P:L:* ⊆ P:* (inclusion)
P:R:* ⊆ P:* (inclusion)
P:L:* ∩ P:R:* = ∅ (disjointness)
2. Updating Objects through an Array

Key property is disjointness of references

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

DPJ provides a novel array type $C<[i]>[i] \neq i$

Disjoint

Non-disjoint
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 {
        ...
    }
    ...
}


C<[0]>
int x in [0]

C<[1]>
int x in [1]
Index-Parameterized Arrays

Class is parameterized by region \( P \)

```java
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 \( P \)

```java
class C<\text{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

UPCRC Illinois
Universal Parallel Computing Research Center
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
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
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

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]:*> */ );
  }
}

Partition

Subarrays

Disjoint subarrays
Array Partitioning Example

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

Partition

Disjoint subarrays

Subarrays

RPLs with owner regions
4. Commutativity Annotations

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

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

```java
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
Outline

Regions and Effects
New Type System Features
Evaluation
Conclusion
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
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
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/