Safe Nondeterminism in a Deterministic-by-Default Parallel Language

Robert L. Bocchino Jr., Stephen Heumann, Nima Honarmand, Sarita Adve, Vikram Adve, *University of Illinois*

Adam Welc, Tatiana Shpeisman, Yang Ni, *Intel*

Deterministic Parallel Java (DPJ)

Object-oriented parallel language
- Java is a nice research target
- Ideas apply to other OO languages as well

Guarantees determinism at compile time
- For given input, output is schedule-independent
- Programmer writes annotations (regions and effects)
- Compiler uses annotations to prove determinism
  - Simple analysis despite complex aliasing and data flow
  - Strong guarantee with no runtime overhead
Benefits and Costs of DPJ

Benefits

• Easier to reason about parallel code (like sequential code)
• Easier to test parallel code (one output per input)
• No subtle parallelism bugs (races, deadlocks)
• Simpler bug detection and debugging

Costs

• Programmer annotation burden
  - Inferring annotations can help [M. Vakilian et al., ASE 2009]
• Excludes nondeterministic algorithms by design
  - We can add nondeterminism with some simple language extensions
• Expressivity limitations even for deterministic codes
  - Frameworks etc. can help
Benefits and Costs of DPJ

Benefits

• Easier to reason about parallel code (like sequential code)
• Easier to test parallel code (one output per input)
• No subtle parallelism bugs (races, deadlocks)
• Simpler bug detection and debugging

Costs

• Programmer annotation burden
  - Inferring annotations can help [M. Vakilian et al., ASE 2009]
• Excludes nondeterministic algorithms by design
  - We can add nondeterminism with some simple language extensions
• Expressivity limitations even for deterministic codes
  - Frameworks etc. can help

Focus of today’s talk
Why Add Non-determinism to DPJ?

Non-deterministic algorithms are important

- Several answers are acceptable for given input
- Requiring determinism is unnecessary
- Deterministic schedule may hurt performance

Examples

- Database transactions
- Branch and bound search
- Graph algorithms (clustering, mesh refinement)
Outline

Basic DPJ Language (Review)
Controlling Nondeterminism
New Language Features
Optimizing the Implementation
Evaluation
Conclusion
Overview of DPJ

Programmer
• Partitions object fields into regions
• Writes effect summaries on methods
• Uses `cobegin` and `foreach` to specify fork-join parallelism
  - `cobegin`: Parallel statements
  - `foreach`: Parallel loop iterations

Compiler
• Checks correctness of effects summaries
• Checks noninterference of parallel tasks
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>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>

Declaring and using region names
Example: A Pair Class

```java
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 */
        }
    }
}
```

**Declaring and using region names**

<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>
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 */
            }
    }
}
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></th>
<th></th>
</tr>
</thead>
<tbody>
<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>
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
Example: A Pair Class

```java
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 */
        }
    }
}
```

Compiler uses effects to check noninterference

### Expressing parallelism

```plaintext
<table>
<thead>
<tr>
<th>Pair</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<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 */
    }  
  }  
}
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 */
        }
}

Class is parameterized by region P
Field data resides in P
Classes are instantiated to types with regions
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 \texttt{data} resides in \( P \)

Classes are instantiated to types with regions

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 */
        }
}
More Realistic Patterns

Support for several important patterns
- Parallel updates through arrays of disjoint references
- Divide and conquer updates
  - Linked trees of arbitrary depth
  - Recursively partitioned arrays
- Commutative operations on concurrent data structures

New type and effect mechanisms
- See OOPSLA 2009 paper for details

Good performance on several realistic benchmarks
- Zero runtime overhead (checking done by compiler)
- But some limitations in expressivity
**Performance Results**

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

**Diagram:**
- Barnes-Hut (200,000)
- Merge Sort (100 million)
- IDEA Encryption (35 million)
- K-Means (300,000)
- Collision Tree (360,000)
- Monte Carlo (60,000)

**Legend:**
- Blue line: Barnes-Hut (200,000)
- Red line: Merge Sort (100 million)
- Green line: IDEA Encryption (35 million)
- Pink line: K-Means (300,000)
- Orange line: Collision Tree (360,000)
- Black line: Monte Carlo (60,000)

**Graph:**
- **Number of cores** on the x-axis
- **Speedup** on the y-axis

**System Specifications:**
- 4 x 6 core x86 (Dell R900), 2GB main memory per core
Outline

Basic DPJ Language (Review)

Controlling Nondeterminism

New Language Features

Optimizing the Implementation

Evaluation

Conclusion
Controlling Nondeterminism

Adding nondeterminism doesn’t mean all bets are off!

• Don’t revert to low-level synchronization
  - E.g., locks or CAS
  - Brittle, not composable, hard to reason about

• Don’t revert to wild shared memory, races, etc.

Still want strong compile-time guarantees
What Guarantees Should Exist?

Determinism by default
• Program is deterministic unless nondeterminism explicitly requested

Strong isolation (atomicity)
• Programmer can identify sections of code to run as if in isolation
• Isolation is strong, i.e., no conflicts with any other code

Race freedom
• I.e., no unsynchronized conflicting accesses (volatile OK)
• These kill the semantics under the Java memory model
What Guarantees Should Exist?

Determinism by default

- Program is deterministic unless nondeterminism explicitly requested

Strong isolation (atomicity)

- Programmer can identify sections of code to run as if in isolation
- Isolation is strong, i.e., no conflicts with any other code

Race freedom

- I.e., no unsynchronized conflicting accesses (volatile OK)
- These kill the semantics under the Java memory model

Ordinary threads programming provides none of these guarantees
Our Approach

Use software transactional memory (STM)

- Obvious choice for providing isolation guarantee
- Not an essential choice, though a robust one
  - Could use automated locking strategies for some patterns
  - Research is not as mature
- Does carry scalar performance penalty

Our contribution: Leverage and extend the type system

- Get much stronger guarantees than with STM alone
  - Determinism by default
  - Strong isolation
  - Race freedom
- Eliminate unnecessary STM synchronization
Our Approach

Use software transactional memory (STM)

- Obvious choice for providing isolation guarantee
- Not an *essential* choice, though a robust one
  - Could use automated locking strategies for some patterns
  - Research is not as mature
- Does carry scalar performance penalty

*Our contribution*: Leverage and extend the type system

- Get *much stronger guarantees* than with STM alone
  - Determinism by default
  - Strong isolation
  - Race freedom
- Eliminate unnecessary STM synchronization

STM does not provide at all
Our Approach

Use software transactional memory (STM)

• Obvious choice for providing isolation guarantee
• Not an essential choice, though a robust one
  - Could use automated locking strategies for some patterns
  - Research is not as mature
• Does carry scalar performance penalty

Our contribution: Leverage and extend the type system

• Get much stronger guarantees than with STM alone
  - Determinism by default
  - Strong isolation
  - Race freedom
• Eliminate unnecessary STM synchronization
Our Approach

Use software transactional memory (STM)
• Obvious choice for providing isolation guarantee
• Not an essential choice, though a robust one
  - Could use automated locking strategies for some patterns
  - Research is not as mature
• Does carry scalar performance penalty

Our contribution: Leverage and extend the type system
• Get much stronger guarantees than with STM alone
  - Determinism by default
  - Strong isolation
  - Race freedom
• Eliminate unnecessary STM synchronization
Outline

Basic DPJ Language (Review)
Controlling Nondeterminism
New Language Features
Optimizing the Implementation
Evaluation
Conclusion
Summary of Language Features

Expressing nondeterministic parallelism

- `foreach_nd (int i in 0, n)`: Parallel loop
- `cobegin_nd { S1; ...; Sn}`: Parallel statements

Expressing isolation

- Atomic statement `atomic s` executes statement `s` in isolation

```plaintext
cobegin_nd {
    atomic { x = 0; y = x; };
    atomic x = 1;
}
y == 1
```

Effect system

- **Atomic effects** keep track of effects done in atomic statements
- Compiler uses effects to enforce guarantees
Summary of Language Features

Expressing nondeterministic parallelism

- `foreach_nd (int i in 0, n)`: Parallel loop
- `cobegin_nd { S1; ...; Sn}`: Parallel statements

Expressing isolation

- Atomic statement (`atomic s`) executes statement `s` in isolation

```
cobegin_nd {
    atomic { x = 0; y = x; };
    atomic x = 1;
}
```

Effect system

- *Atomic effects* keep track of effects done in atomic statements
- Compiler uses effects to enforce guarantees

```
cobegin_nd {
    atomic { x = 0; y = x; };
    atomic x = 1;
}
```

\( y \neq 1 \)
Semantics of `cobegin_nd`

Atomic statements generate atomic effects

- `atomic { x = 5; y = 2; } // writes atomic Rx, atomic Ry`

Only atomic effects may interfere

```
cobegin_nd {
    atomic { x = 0; y = 0; };
    atomic { x = 1; y = 1; };
}
```

```
cobegin_nd {
    { x = 0; y = 0; };
    { x = 1; y = 1; };
}
```

OK  Error
Semantics of `cobegin_nd`

Atomic statements generate atomic effects

- `atomic { x = 5; y = 2; } // writes atomic Rx, atomic Ry`

Only atomic effects may interfere

```
cobegin_nd {
    atomic { x = 0; y = 0; };
    atomic { x = 1; y = 1; };
}
```

```
cobegin_nd {
    { x = 0; y = 0; };
    { x = 1; y = 1; };
}
```

OK  Error

These rules guarantee strong isolation and race freedom
Ordinary effects cover atomic effects...

- `void f() writes Rx { atomic x = 5; }` // OK

...But not vice versa...

- `void f() writes atomic Rx { x = 5; }` // Error

...So we can do sound analysis of method invocations

```c

cobegin
{
    f();
    g();
}
```
Semantics of Effect Summaries

Ordinary effects cover atomic effects...

- void f() writes Rx { atomic x = 5; } // OK

...But not vice versa...

- void f() writes atomic Rx { x = 5; } // Error

...So we can do sound analysis of method invocations

```java
cobegin_nd {
    f();
    g();
}
```

Atomic effects here mean operations occurred in transactions
Semantics of `cobegin`

Atomic effects “disappear” in branches of `cobegin`

```plaintext
cobegin {
    atomic { x = 0 }; // writes Rx, not writes atomic Rx
    atomic { y = 1 }; // writes Ry, not writes atomic Ry
}
```

So `cobegin` still has deterministic semantics:

```plaintext
cobegin {
    atomic x = 0;
    atomic x = 1;
}                   cobegin {
    atomic x = 0;    cobegin {
    atomic y = 0 }
    atomic y = 1 }
}                   Error
Error
```
Semantics of cobegin

Atomic effects “disappear” in branches of cobegin

```plaintext
cobegin {
    atomic { x = 0 }; // writes Rx, not writes atomic Rx
    atomic { y = 1 }; // writes Ry, not writes atomic Ry
}
```

So cobegin still has deterministic semantics:

```plaintext
cobegin {
    atomic x = 0;
    atomic x = 1;
}
cobegin Nd {
    cobegin { atomic x = 0; atomic y = 0 };
    cobegin { atomic x = 1; atomic y = 1 };
}
```

Error  Error

These rules guarantee determinism by default
Reasoning About Programs

Strong isolation and race freedom

- Programmer can think of execution as set of isolated code chunks
  - Sequential code sections
  - `cobegin`, `foreach` branches
  - `atomic` statements inside an `_nd`
  - Sequences between `atomic` statements inside an `_nd`
- Chunks occur in program order

Determinism by default

- If an isolated section contains no `_nd`, it is input-output deterministic
- This is true even inside an `_nd`
Outline

Basic DPJ Language (Review)
Controlling Nondeterminism
New Language Features
Optimizing the Implementation
Evaluation
Conclusion
class C {
    region rx, ry;
    int x in rx, y in ry;
    void m1() reads atomic rx writes atomic ry {
        atomic { int z = x; y = 0; }
    }
    void m2() reads atomic rx writes atomic ry {
        atomic { int z = x; y = 1; }
    }
    void m3() {
        cobegin_nd { m1(); m2; }
    }
}
class C {
    region rx, ry;
    int x in rx, y in ry;
    void m1() reads atomic rx writes atomic ry {
        atomic {
            int z = x; y = 0;
        }
    }
    void m2() reads atomic rx writes atomic ry {
        atomic {
            int z = x; y = 1;
        }
    }
    void m3() {
        cobegin_nd {
            m1(); m2;
        }
    }
}
class C {
    region rx, ry;
    int x in rx, y in ry;
    void m1() reads atomic rx writes atomic ry {
        atomic { int z = x; y = 0; }
    }
    void m2() reads atomic rx writes atomic ry {
        atomic { int z = x; y = 1; }
    }
    void m3() {
        cobegin nd { m1(); m2; }
    }
}
class C {
    region rx, ry;
    int x in rx, y in ry;

    void m2() reads atomic rx writes atomic ry {
        atomic { int z = x; y = 1; }
    }
    void m3() {
        cobegin Nd { m1(); m2; }
    }
}
class C {
    region rx, ry;
    int x in rx, y in ry;
    void m1() writes atomic rx writes atomic ry {
        atomic { x = 0; y = 0; }
    }
    void m2() reads atomic rx writes atomic ry {
        atomic { int z = x; y = 1; }
    }
    void m3() {
        cobegin_nd { m1(); m2; }
    }
}
class C {
    region rx, ry;
    int x in rx, y in ry;
    void m1() writes atomic rx writes atomic ry {
        atomic { x = 0; y = 0; }
    }
    void m2() reads atomic rx writes atomic ry {
        atomic { int z = x; y = 1; }
    }
    void m3() {
        cobegin_nd { m1(); m2; }
    }
}
Type System Extensions

class C {
    region rx, atomic ry;
    int x in rx, y in ry;
    void m1() reads rx writes atomic ry {
        atomic { int z = x; y = 0; }
    }
    void m2() reads rx writes atomic ry {
        atomic { int z = x; y = 1; }
    }
    void m3() {
        cobegin_nd { m1(); m2; }
    }
}
class C {
    region rx, atomic ry;
    int x in rx, y in ry;
    void m1() reads rx writes atomic ry {
        atomic { int z = x; y = 0; }
    }
    void m2() reads rx writes atomic ry {
        atomic { int z = x; y = 1; }
    }
    void m3() {
        cobegin_nd { m1(); m2; }
    }
}
class C {
    region rx, atomic ry;
    int x in rx, y in ry;
    void m1() reads rx writes atomic ry {
        atomic { int z = x; y = 0; }
    }
    void m2() reads rx writes atomic ry {
        atomic { int z = x; y = 1; }
    }
    void m3() {
        cobegin_nd { m1(); m2; }
    }
}

Region ry is declared atomic
Region rx is not declared atomic; effect is reads rx
class C {
    region rx, atomic ry;
    int x in rx, y in ry;
    void m1() reads rx writes atomic ry {
        atomic { int z = x; y = 0; }
    }
    void m2() reads rx writes atomic ry {
        atomic { int z = x; y = 1; }
    }
    void m3() {
        cobegin_nd { m1(); m2; }
    }
}
Implementation

Read of non-atomic region gets no barrier (normal read)

Write of non-atomic region gets a *log-only* barrier
  - Don’t need synchronization (no concurrent access)
  - But enclosing transaction may still be aborted
    - Have to record old value in undo log, for restore on abort

*Future work*
  - No log-only barrier if object thrown away on abort
  - Common pattern for objects created inside transactions
Outline

Basic DPJ Language (Review)
Controlling Nondeterminism
New Language Features
Optimizing the Implementation
Evaluation
Conclusion
Formal Language and Correctness

Syntax and semantics are done

- Syntax and static semantics for formal core language
- Small-step operational semantics captures interleavings

Soundness proofs are in development

- Assume serializability of atomic sections (STM gives us this)
- Show that type system gives the three properties
  - Input-output determinism for code that contains no `nd`
  - Strong isolation
  - Race freedom
Experiments

Benchmarks
• Traveling Salesman Problem (branch and bound search)
• Delaunay mesh refinement (graph algorithm)
• OO7 (synthesized database queries)

Expressivity
• We can express all three codes in a natural way

Performance
• No extra overhead from standard STM implementation
• Benefits from barrier elimination
Barrier Elimination Results

![Graph showing barrier elimination results for TSP, Delaunay, and OO7 across different thread counts.](image)
Barrier Elimination Results

In Delaunay and OO7, most computation occurs in transactions.
Barrier Elimination Results

![Graph showing barrier elimination results for TSP, Delaunay, and OO7 with optimized and unoptimized time comparisons for 1, 2, 3, 4, 7, 12, 17, and 22 threads.]
Conclusion

Basic DPJ [*OOPSLA 2009*]

- Supports deterministic codes with noninterfering parallelism
- Gives good performance, strong guarantee for realistic codes
- Some limitations in expressivity, programmer burden

Support for nondeterministic computations

- New language constructs: `atomic`, `_nd`, atomic effects
- Strong safety guarantees
- Improved performance from barrier elimination