Parallel Programming Must Be Deterministic by Default

Robert Bocchino, Vikram Adve,
Sarita Adve, and Marc Snir

University of Illinois at Urbana-Champaign

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

UPCRC Illinois
Universal Parallel Computing Research Center

Parallel Programming Is Too Hard

Too many *nondeterministic* interleavings

Hard to reason about correctness

- Data races
- Deadlock
- Memory models

Hard to get testing coverage

- Must test multiple outputs per input
- Easy to miss corner cases

We Don't Need All That Nondeterminism

Many 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 visible output

We Don't Need All That Nondeterminism

Many 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 visible output

Parallel languages should *be deterministic by default*

We Don't Need All That Nondeterminism

Many 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 visible output

Parallel languages should *be deterministic by default*

Determinism should be *guaranteed*
unless nondeterminism is *explicitly requested*

Why Don't We Already Do This?

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

Our Proposed Research Goal

Bring *determinism by default* to mainstream languages

Benefits of achieving this goal:

- Enable “almost sequential” reasoning
- Avoid subtle parallelism bugs
  - No data races or deadlocks
  - No complex memory models
- Simplify testing of parallel programs
  - Test one output per input and you are done
- Simplify sequential to parallel porting
- Simplify bug reproduction and debugging

Our Proposed Research Agenda

1. How to guarantee determinism by default?
2. How to encapsulate nondeterministic behavior?
3. How to support explicit, controlled nondeterminism?
4. How to simplify development and porting?

Guaranteeing Determinism: Approaches

Language (type system)

- *Strengths*: Programmer control and documentation, modularity
- *Weaknesses*: Programmer effort (*perceived*), coarse granularity

Compiler (auto parallelization)

- *Strengths*: Less programmer effort
- *Weaknesses*: Limited effectiveness, brittle, opaque performance

Runtime (software and/or hardware)

- *Strengths*: Exploit runtime information
- *Weaknesses*: Overhead, complexity, opaque performance, weak guarantee

Guaranteeing Determinism: Approaches

Language (type system)

- *Strengths*: Programmer control and documentation, modularity
- *Weaknesses*: Programmer effort (*perceived*), coarse granularity

Strong language mechanisms are essential ←

Compiler (auto parallelization)

- *Strengths*: Less programmer effort
- *Weaknesses*: Limited effectiveness, brittle, opaque performance

Runtime (software and/or hardware)

- *Strengths*: Exploit runtime information
- *Weaknesses*: Overhead, complexity, opaque performance, weak guarantee

Guaranteeing Determinism: Approaches

Language (type system)

- *Strengths*: Programmer control and documentation, modularity
- *Weaknesses*: Programmer effort (*perceived*), coarse granularity

Strong language mechanisms are essential

Compiler (auto parallelization)

- *Strengths*: Less programmer effort
- *Weaknesses*: Limited effectiveness, brittle, opaque performance

Supplement with compiler and runtime techniques for greater expressivity

Runtime (software and/or hardware)

- *Strengths*: Exploit runtime information
- *Weaknesses*: Overhead, complexity, opaque performance, weak guarantee

Effect Systems

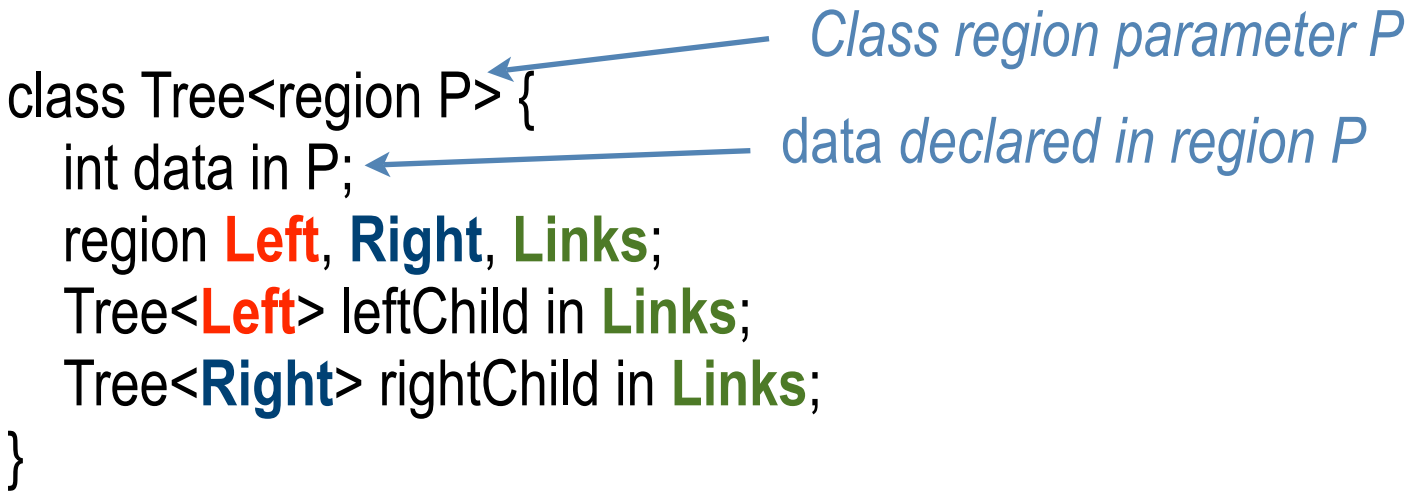
```
class Tree<region P> {
    int data in P;
    region Left, Right, Links;
    Tree<Left> leftChild in Links;
    Tree<Right> rightChild in Links;
}
```

Effect Systems

Class region parameter P

```
class Tree<region P> {
    int data in P;
    region Left, Right, Links;
    Tree<Left> leftChild in Links;
    Tree<Right> rightChild in Links;
}
```

Effect Systems



Effect Systems

```
class Tree<region P> {
    int data in P;
    region Left, Right, Links;
    Tree<Left> leftChild in Links;
    Tree<Right> rightChild in Links;
}
```

Class region parameter P

data *declared in region P*

Region names Left, Right, Links

Effect Systems

```
class Tree<region P> {
    int data in P;
    region Left, Right, Links;
    Tree<Left> leftChild in Links;
    Tree<Right> rightChild in Links;
}
```

Class region parameter P

data *declared in region P*

Region names Left, Right, Links

Field in Links *points to* Tree<Left>

Effect Systems

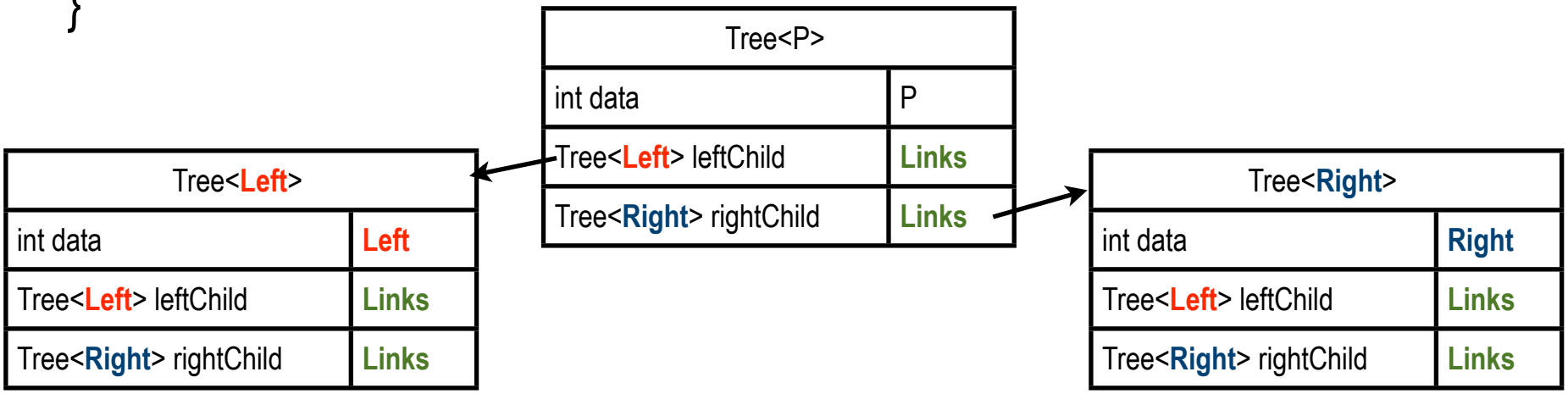
```
class Tree<region P> {
   int data in P;
   region Left, Right, Links;
   Tree<Left> leftChild in Links;
   Tree<Right> rightChild in Links;
}
```

Class region parameter P

data *declared in region P*

Region names Left, Right, Links

Field in Links *points to* Tree<Left>



Deterministic Parallel Java (DPJ)

Explicit type and effect system [see our Tech Reports]

- Recursive parallelism on linked data structures
- Array computations
  - Flat parallel traversals
  - Recursive partitioning (divide and conquer)
- Support for object-oriented frameworks

Runtime support [ongoing work]

- Fine-grain synchronization
- Fail-stop checks for greater expressivity

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

Hidden Nondeterminism

Programmer provides trusted annotation (e.g., library API)

- *class Set<E> {*
  ***commutative** void add(E e); // add commutes with itself...*
  ...
  }

Compiler uses annotation to prove determinism

- *foreach (int i in 0, n) {*
  set.add(A[i]); // ...so this code is safe
  }

Visible Nondeterminism

Sometimes necessary for high performance

- *Example:* Branch and bound, graph clustering

Carefully controlled

- Explicitly requested by programmer
- Atomic and race free
- Isolated: Nondeterministic and deterministic code do not interfere

```
foreach_nd (...) {
    // Potentially nondeterministic code
}
```

Will a Language Solution Be Usable?

Benefits outweigh the costs

- Effect annotations aid reasoning the programmer must do anyway
- Checkable contracts at interfaces enhance modularity

Technical solutions can reduce the costs

- Effect inference
- Runtime checks
- Integrated development environment

Summary

***Guaranteed determinism* can ease parallel programming**

For mainstream OO languages we need

- Strong language solutions (type and effect)
- Supplemented by runtime checks and tools

Deterministic Parallel Java project at Illinois

- Java-based
- Applicable to other OO languages (C++, C#)

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

UPCRC Illinois
Universal Parallel Computing Research Center