

Deterministic Parallel Java: Towards Deterministic-by-Default Parallel Programming

Robert Bocchino, Vikram Adve

with Sarita Adve, Danny Dig, Stephen Heumann, Nima Honarmand,
Rakesh Komuravelli, Patrick Simmons, Marc Snir, Hyojin Sung, Mohsen
Vakilian (*University of Illinois*)

and Adam Welc, Tatiana Shpeisman, Yang Ni, Ali-Reza Adl Tabatabai
(*Intel Research*)



UPCRC Illinois
Universal Parallel Computing
Research Center

Proposal

Parallel languages should be deterministic by default

I.e., Determinism should be *guaranteed* unless non-determinism is requested explicitly

Deterministic semantics:

1. Fixed input gives unique output (up to acceptable precision)
2. Obvious sequential equivalent

Deterministic Parallel Java: Project Overview

Explicitly parallel, deterministic-by-default, language

- Novel region-based type and effect system
- Today: No run-time checks; may add them in future

Enforces safe *use* of parallel frameworks

- Enforce safety requirements on client code

Disciplined support for non-deterministic code

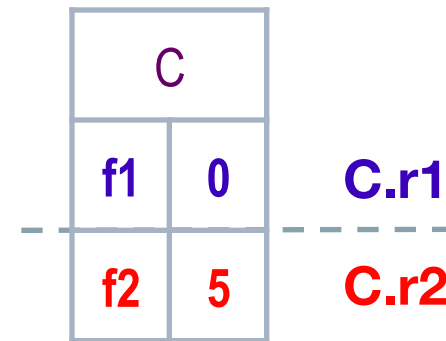
- Explicit; data race free; isolated

DPJizer: Interactive porting environment

- Eclipse plug-in to infer DPJ annotations

Example: Regions and Effects

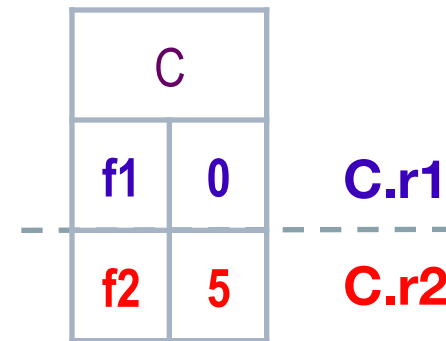
```
class C {  
  region r1, r2;  
  int f1 in r1;  
  int f2 in r2;  
  void m1(int x) writes r1 { f1 = x; }  
  void m2(int y) writes r2 { f2 = y; }  
  void m3(int x, int y) {  
    cobegin {  
      m1(x);  
      m2(y);  
    }  
  }  
}
```



Partitioning the heap

Example: Regions and Effects

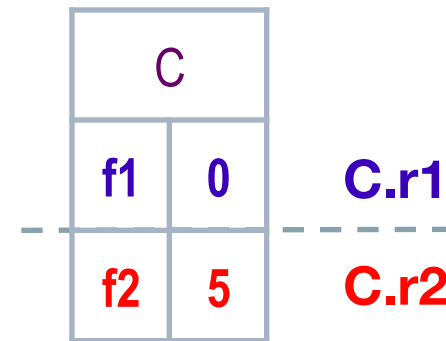
```
class C {  
  region r1, r2;  
  int f1 in r1;  
  int f2 in r2;  
  void m1(int x) writes r1 { f1 = x; }  
  void m2(int y) writes r2 { f2 = y; }  
  void m3(int x, int y) {  
    cobegin {  
      m1(x);  
      m2(y);  
    }  
  }  
}
```



Summarizing method effects

Example: Regions and Effects

```
class C {  
  region r1, r2;  
  int f1 in r1;  
  int f2 in r2;  
  void m1(int x) writes r1 { f1 = x; }  
  void m2(int y) writes r2 { f2 = y; }  
  void m3(int x, int y) {  
    cobegin {  
      m1(x); // Inferred effect = writes r1  
      m2(y); // Inferred effect = writes r2  
    }  
  }  
}
```



Expressing parallelism

Supporting Parallel Patterns

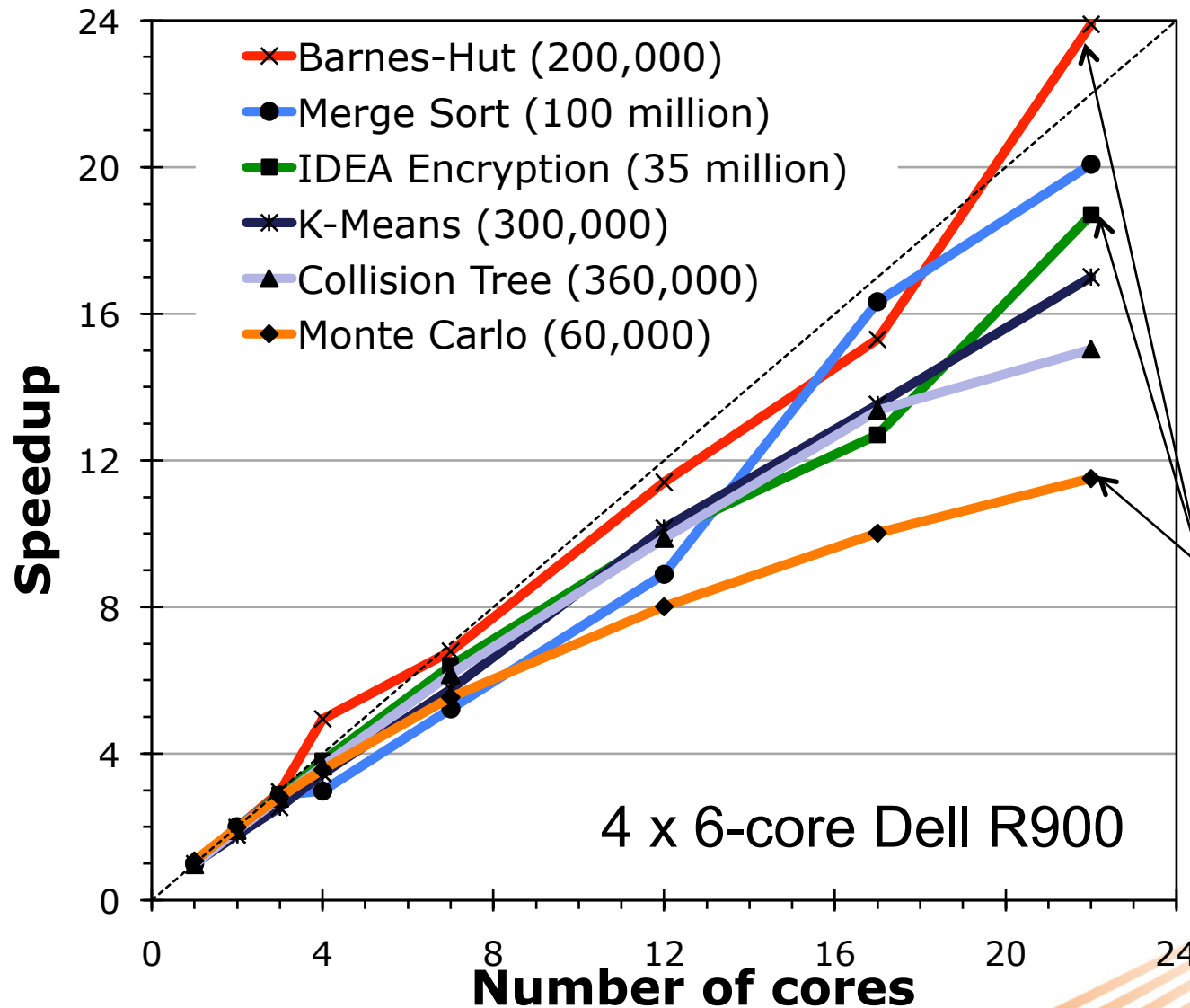
DPJ uses novel features to support parallel codes

- Parallel operations on arrays of references
- Divide and conquer on nested structures
- Divide and conquer operations on arrays
- Commutative operations

Formalism

- Type system
- Proof of *non-interference* for all legal programs

Performance Evaluation



- B-H, Collision Tree are **highly irregular**
- DPJ **expresses full parallelism** (except SIMD) in all but B-H
- Close to or better than hand-tuned **Java threads** versions

Analysis

Strengths

- + Captures *all* non-SIMD parallelism in all but Barnes Hut
- + Introduces *no* inherent run-time overheads
- + Allows incremental porting (e.g., JMonkey), tuning

Weaknesses

- Cannot express some idioms
 - E.g., array reshuffling (Barnes Hut), tree rebalancing
- Some DPJ features can be complex or constraining
 - *Complex syntax*: Index-parameterized arrays
 - *Constraining*: Superclass method's effects must be a superset of any subclass method's effects

Parallel Frameworks

Valuable For Parallel Computing ...

- Division of labor: parallelism experts vs. users
- Easy for user (write sequential code)
- Many real world (parallel) examples exist
 - MapReduce; ParallelArray; Algorithm templates (TBB)

... And Address a Key DPJ Limitation

- Idioms that cannot be checked by type system

... But Challenging

- User must follow many **unchecked** safety rules
- Must be easily extensible

Support for Frameworks in DPJ

Idea 1: Enable design by contract for framework APIs

- Not been done before for shared memory parallelism

Idea 2: Check framework internals via other means but hook into the type system

- Testing, program verification, etc.

We show how to ...

- Use DPJ “off the shelf” to write safe container APIs
 - Constrain aliasing and effects
- Allow greater flexibility via generic types and effects
 - Effect variables, type region parameters
- Make different forms of verification interoperate
 - Type system uses two predicates: *disjoint-rgn*, *disjoint-ref*
 - These predicates must be discharged externally

Writing Realistic Frameworks

- **Array: safe wrapper around Java `ParallelArray`**
 - Operations: `create()`, `withMapping()`, `reduce()`
 - Example client: *Monte Carlo* (Java Grande)
- **Tree (from scratch, inspired by tree algorithms)**
 - Operations: `buildTree()`, `visitPO()`
 - Example client: *Barnes-Hut center of mass*
- **Experience**
 - Safe frameworks express algorithms well
 - Writing API is sometimes tricky but client code is simple
 - **pure or one or two *extra* read effects**
 - More flexible: e.g., reordering array; rebalancing tree

Non-deterministic Parallelism

Numerous non-deterministic algorithms, programs

- **Branch-and-bound optimization, e.g., for TSP**
- **Clustering algorithms**
- **Delaunay mesh refinement**
- **Servers with transactional parallelism**
- **...**

Common Feature

- **Non-commutative parallel updates**
- **Synchronized for atomicity (not ordering)**

Example: Writing TSP in DPJ

Non-determinism must be explicit

atomic statement synchronizes conflicting accesses

```
foreach_nd (int i in 0, Nworkers-1) {  
  atomic {  
    remove path-prefix pfx from pq;  
    if (pfx is long enough) return pfx;  
    extend pfx and insert in pq;  
  }  
  for (each Hamiltonian cycle with prefix pfx) {  
    atomic {  
      if (tour.length() < bestTour.length())  
        bestTour = tour;  
    }  
  }  
}
```

Conflicting operations must appear in **atomic**

Safety Guarantees for Non-determinism

- Program is data-race free
- Execution is serialization of
 - (a) `foreach`; (b) `cobegin`; (c) `atomic`; (d) reads/writes outside these
- `foreach`, `cobegin` retain most of their guarantees
 - Can reason about them in isolation
 - Retain sequential equivalence
 - Retain input-output determinism *if* they do not enclose `foreach_nd` or `cobegin_nd`

Summary

DPJ today: strong semantic guarantees

- Deterministic semantics *unless* explicitly requested otherwise
- Through simple compile-time type checking
- Safe use of parallel frameworks
- Non-deterministic code is (a) explicit; (b) data race free; (c) explicit

Future work: ease of adoption

- Extend to C++
- DPJizer: Interactive porting tool
- Experience with real world software

See dpj.cs.uiuc.edu for references.

Extra Slides