Dynamic Determinism Checking for Structured Parallelism

Edwin Westbrook[1], Raghavan Raman[2], Jisheng Zhao[3], Zoran Budimlić[3], Vivek Sarkar[3]

[1]Kestrel Institute [2]Oracle Labs [3]Rice University

Deterministic Parallelism for the 99%

- Parallelism is crucial for performance
- Hard to understand for most programmers
- Deterministic parallelism is much easier
- In many cases, non-determinism is a bug

How to catch / prevent non-determinism bugs?

Two Sorts of Code

1. High-performance parallel libraries
   - Uses complex and subtle parallel constructs
   - Written by concurrency experts: the 1%
2. Deterministic application code
   - Uses parallel libraries in a deterministic way
   - Parallelism behavior is straightforward
   - Written by everybody else: the 99%

We focus on application code

Determinism via Commutativity

1. Identify pairs of operations which commute
   – Operations = parallel library primitives (the 1%)
   – Verified independently of this work
2. Ensure operations in parallel tasks commute
   – I.e., verify the application code (the 99%)

Our Approach: HJd

- HJd = Habanero Java with determinism
  - Builds on our prior race-freedom work [RV’11,ECOOP’12]
- Determinism is checked dynamically
  - For application code, not parallel libraries
- Determinism failures throw exceptions
  - Because non-determinism is a bug!
- Checking itself uses a deterministic structure
- Leads to low overhead: 1.26x slowdown!

Example: Counting Factors in Parallel

```
class CountFactors {
  int countFactors (int n) {
    AtomicInteger cnt
      = new AtomicInteger();
    finish {
      for (int i = 2; i < n; ++i)
        async {
          if (n % i == 0)
            cnt.increment();
        }}
    return cnt.get ();
}}
```

Join child tasks

Fork task

Increment cnt in parallel

Get result after finish

Specifying Commutativity for Libraries

- Methods annotated with “commutativity sets”
  - Each pair of methods in set commute
- Syntax:

```
@CommSets{S_1, …,S_n} <method sig>
```

  - States method is in sets S_1 through S_n
  - Commutes with all other methods in these sets

Commutativity Sets for AtomicInteger

```
final class AtomicInteger {
  @CommSets{"read"} int get () { ... }
  @CommSets{"modify"} void increment()
                              { ... }
  @CommSets{"modify"} void decrement()
                              { ... }
  @CommSets{"read","modify"} int initValue()
                              { ... }
  int incrementAndGet () { ... }
```

get commutes with itself

inc/dec commute with themselves and each other

Commutes with anything

Commutes with nothing (not even itself)

Irreflexive Commutativity

- Some methods do not commute with themselves, but commute with other methods
- Specified with `@Irreflexive`

Example #2: Blocking Atomic Queues

```
final class AtomicQueue {
  @CommSets{"modify"} @Irreflexive
  void add (Object o) { ... }

  @CommSets{"modify"} @Irreflexive
  Object remove () { ... }
}
```

Add and remove commute: queue order unchanged

Self-commuting add (or remove) changes the queue!

Commutativity means Commutativity

- Queue `add` might self-commute for some uses
  - E.g. worklist-based algorithms: each queue item is consumed in the same way
- Still cannot mark `add` as self-commuting
- Instead, change library to capture use case

Example #3: Atomic Worklists

```
final class AtomicWorklist {
  interface Handler () {
      void handle (Object o);   }

  void setHandler (Handler h) { ... }

  @CommSets{"modify"}
  void add (Object o) { ... }
}
```

Now add self-commutes: all elems get same handler

Dynamically Verifying Determinism

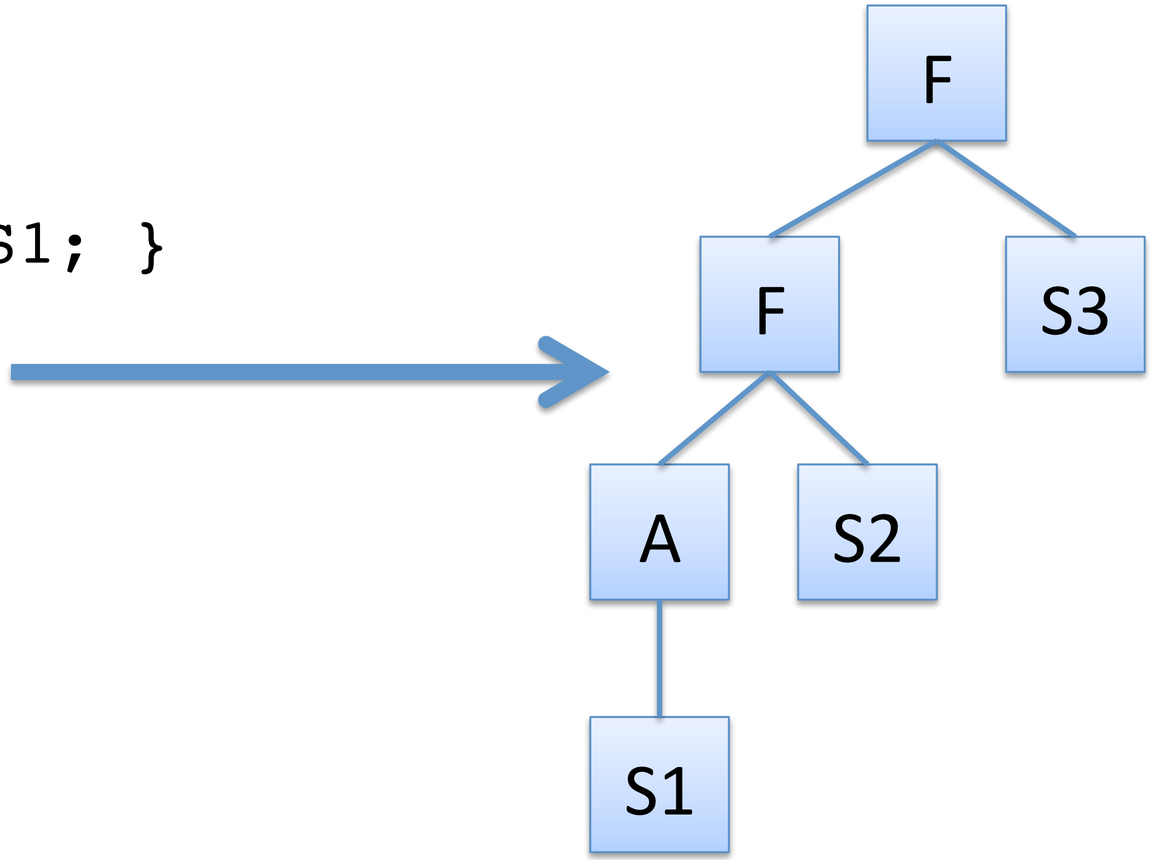
- Each parallel library call → a dynamic check
  - Ensures no non-commuting methods *could possibly* run in parallel
- HJd exposes these checks to the user
  - Construct called *permission region* [RV '11]
  - Many calls can be grouped into one check
  - See our paper for more details

The Key to Dynamic Checks: DPST

- DPST = Dynamic Program Structure Tree
- Deterministic representation of parallelism
- May-happen-in-parallel check with no synch
  - Low overhead!

DPST by Example

```
finish {
  finish {
    async { S1; }
    S2;
  }
  S3;
}
```

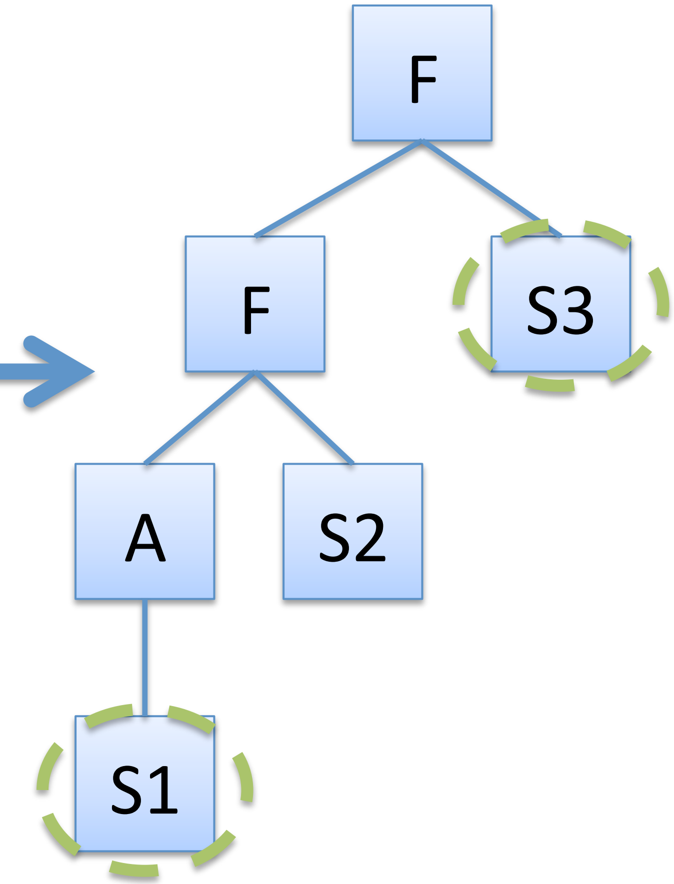


May-Happen-In-Parallel with DPST

1. Find least common ancestor (LCA) of 2 nodes
2. Check if leftmost child of LCA on LCA → node path is an async
3. If so, return true, otherwise return false

MHIP with DPST

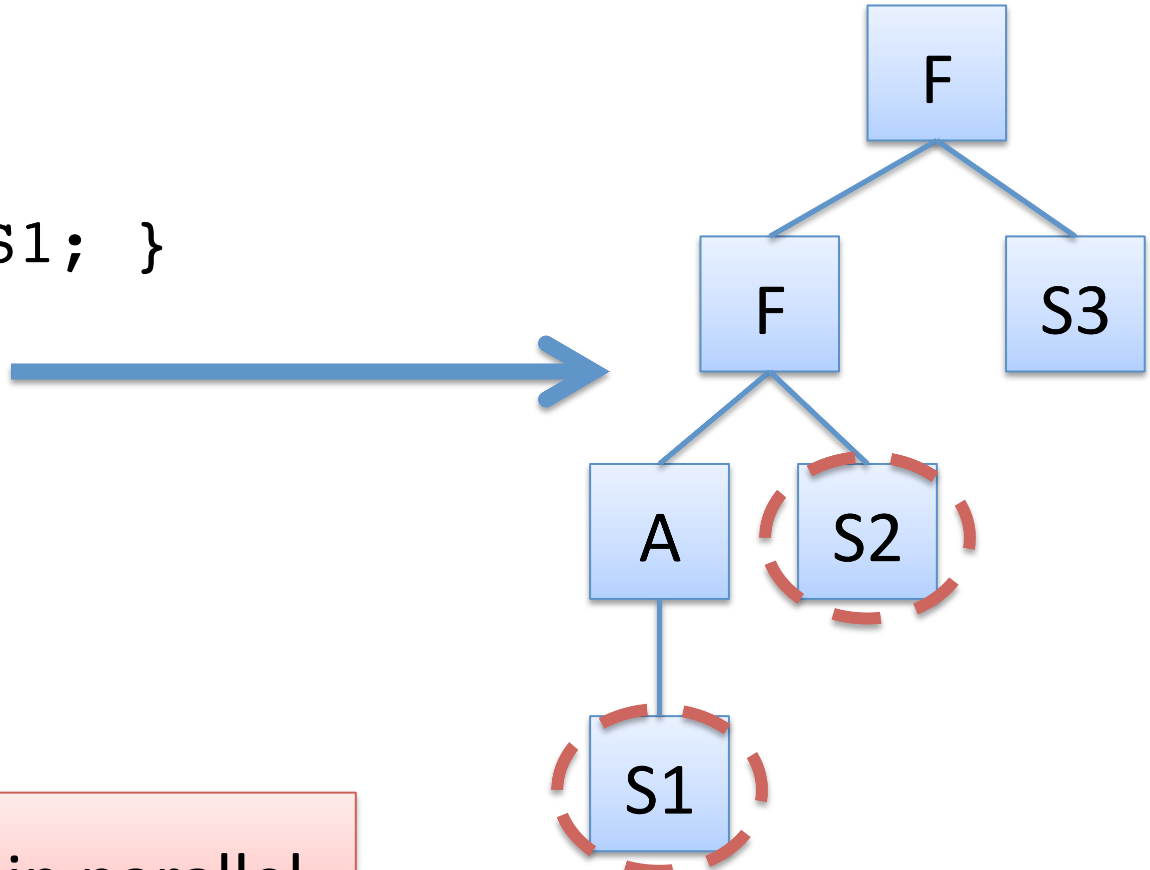
```
finish {
  finish {
    async { S1; }
    S2;
  }
  S3;
}
```



Not in parallel

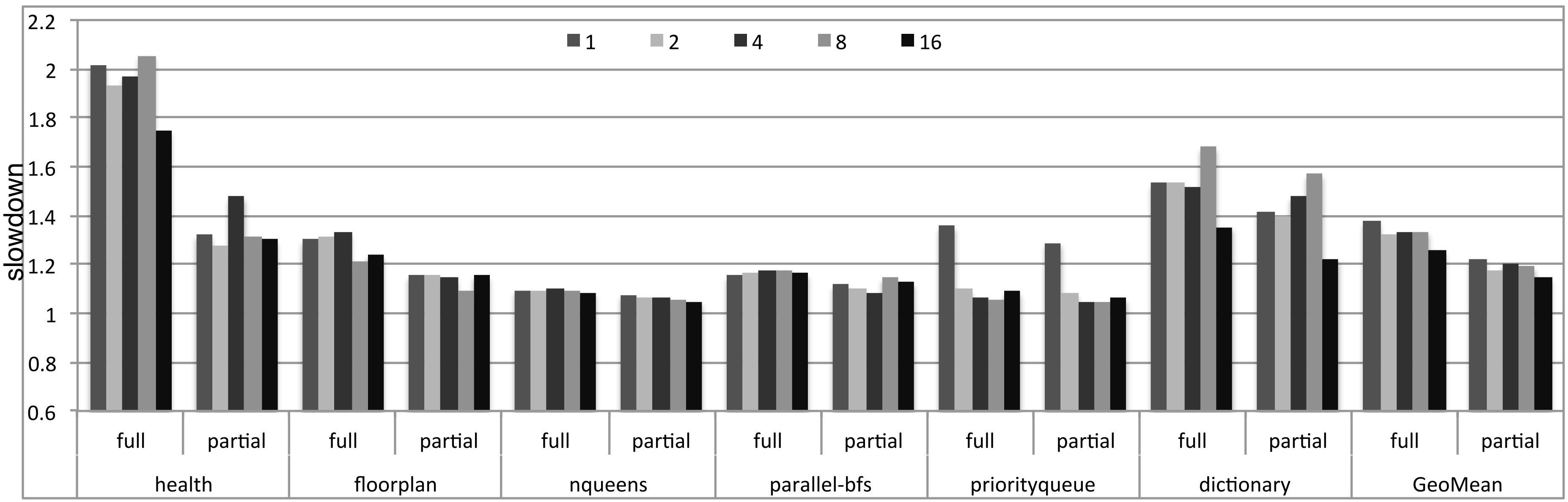
MHIP with DPST

```
finish {
  finish {
    async { S1; }
    S2;
  }
  S3;
}
```



Maybe in parallel

Results: Slowdown ≈ 1.26x



Conclusions: Determinism for the 99%

- Deterministic parallelism is easier
- Non-determinism is often a bug
  - For application code, the 99%
- HJd reports non-determinism bugs with low overhead: 1.26x

Backup Slides

Related Work on HJ

- Race freedom → determinism for HJ (PPPJ ’11)
- Finish accumulators (WoDet ‘13)
- Race-freedom for HJ (RV ‘11, ECOOP ‘12)

DPST for Permission Regions

```
finish {
  permit m1(x) {
    async { S1; }}
  async {
    permit m2(y) { S2; }
  }
  permit m3(x) {
    S3;
}}
```