PARLab Parallel Boot Camp

Testing and Debugging Parallel Programs

Jacob Burnim
Electrical Engineering and Computer Sciences
University of California, Berkeley

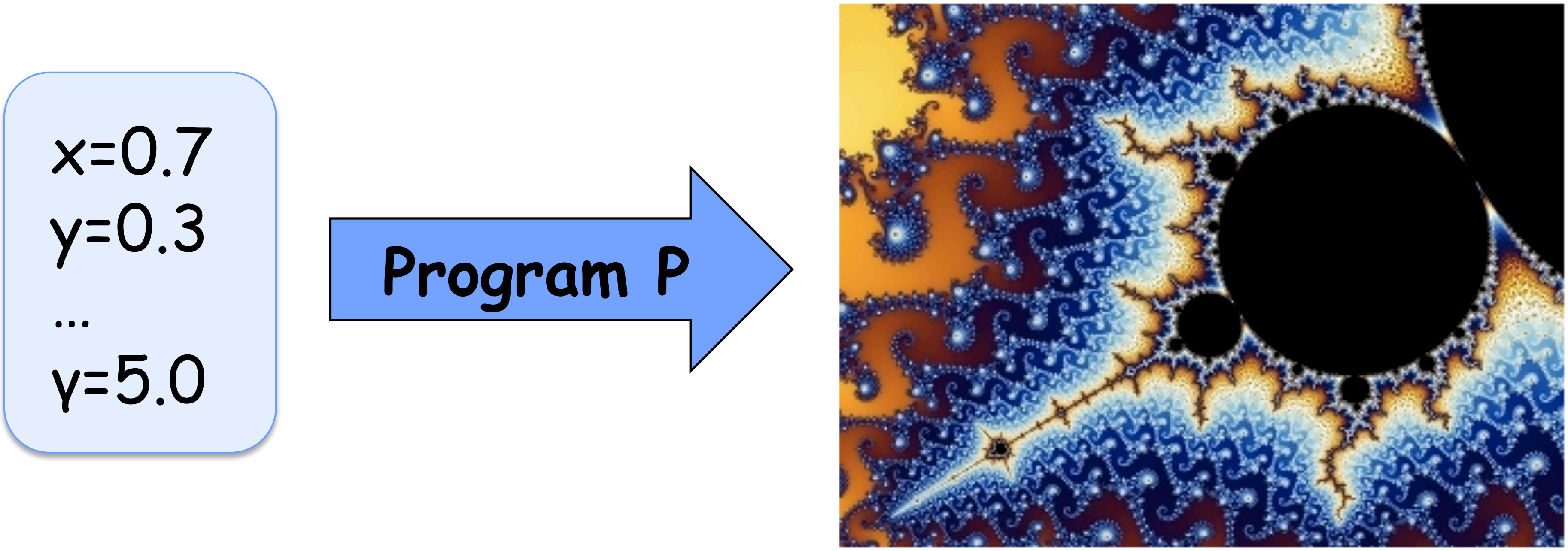
Parallel Correctness Challenges

- Parallel programming presents a number of new challenges to writing correct software.
  - New kinds of bugs: data races, deadlocks, etc.
  - More difficult to test programs and find bugs.
  - More difficult to reproduce errors.

- **Key Difficulty:** Potential non-determinism.
  - Order in which threads execute can change from run to run.
  - Some runs are correct while others hit bugs.

Parallel Correctness Challenges

- For **sequential** programs, we typically expect that same input ==> same output:

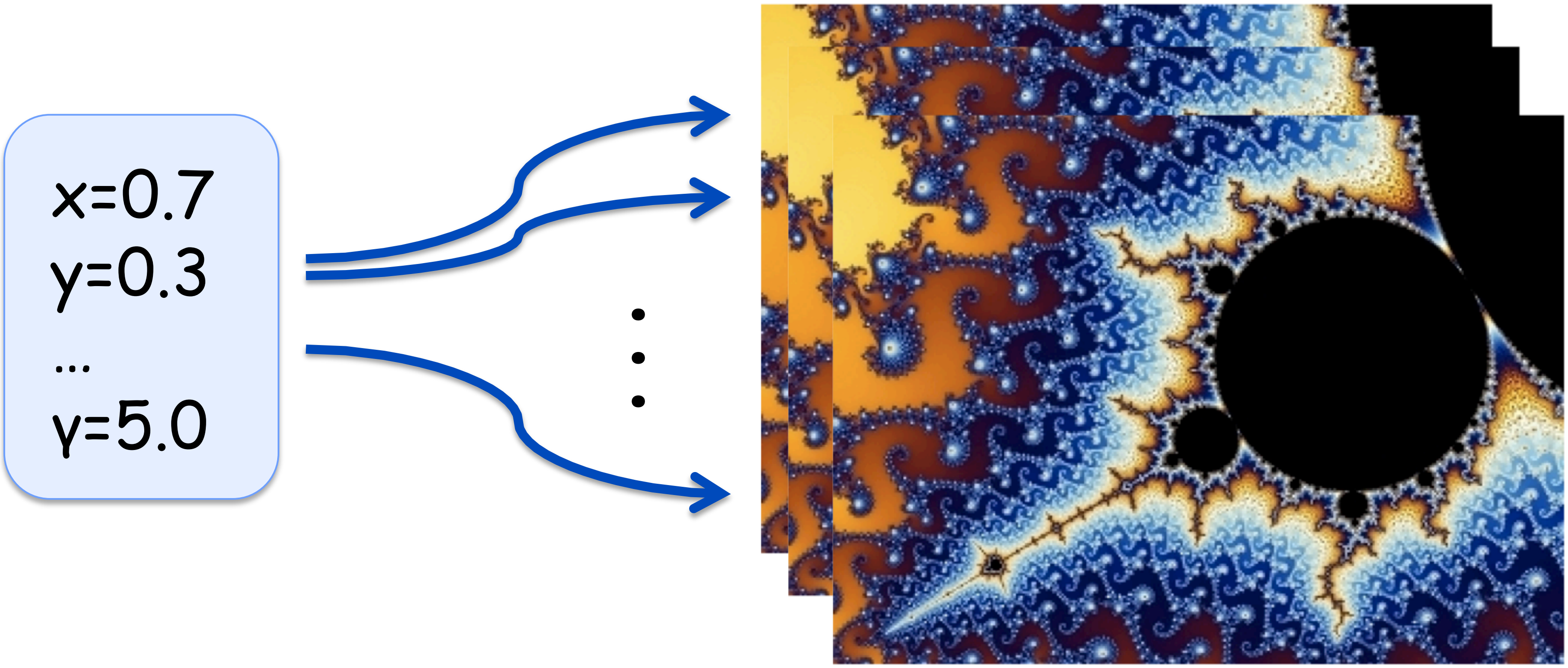


x=0.7
y=0.3
…
y=5.0

Program P

Parallel Correctness Challenges

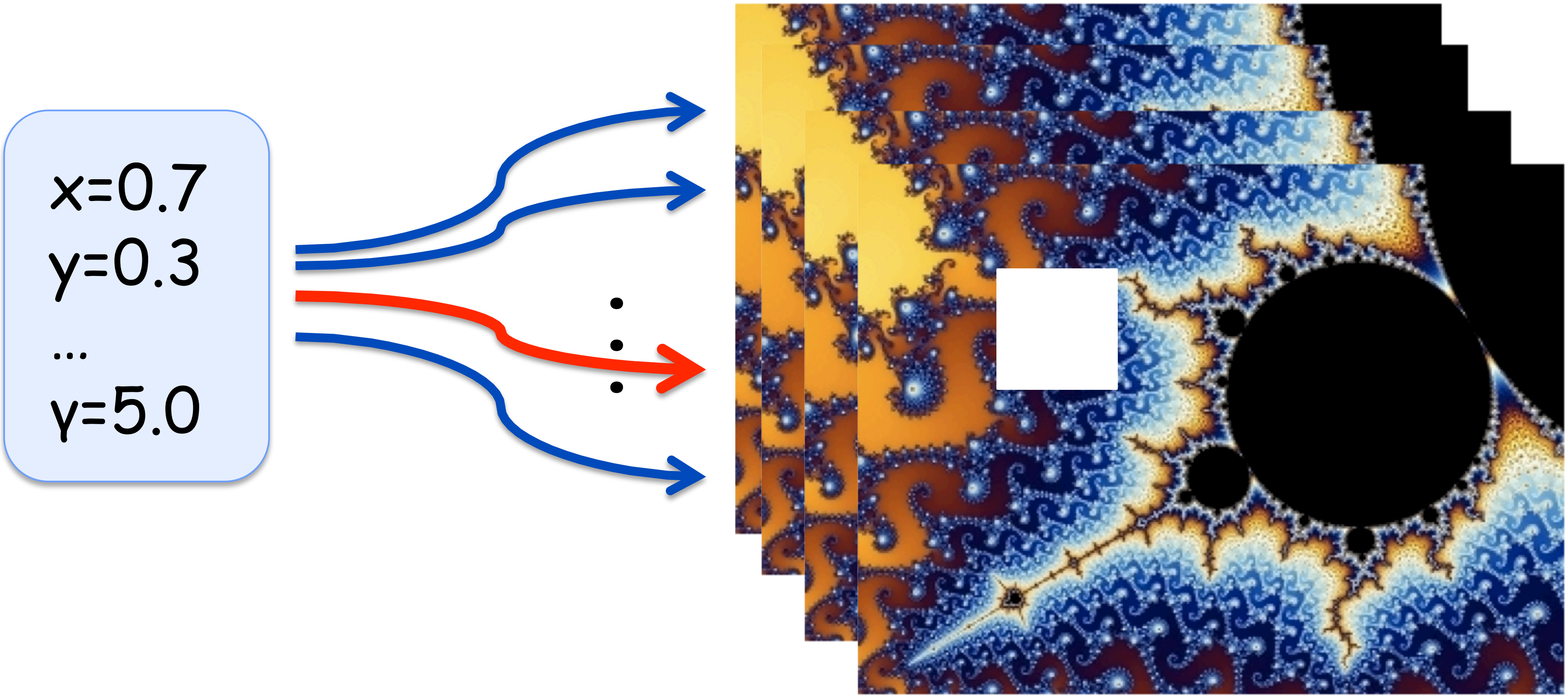
- But for **parallel** programs, threads can be scheduled differently each run:



x=0.7
y=0.3
…
y=5.0

Parallel Correctness Challenges

- But for **parallel** programs, threads can be scheduled differently each run:



x=0.7
y=0.3
…
y=5.0

Parallel Correctness Challenges

- But for **parallel** programs, threads can be scheduled differently each run.

- A bug may occur under only rare schedules.
  - In 1 run in 1000 or 10,000 or …

"Heisenbugs"

- May occur only under some configurations:
  - Particular OS scheduler.
  - When machine is under heavy load.
  - Only when debugging/logging is turned off!

Testing Parallel Programs

- For **sequential** programs:
  - Create several test inputs with known answers.
  - Run the code on each test input.
  - If all tests give correct input, have some confidence in the program.
  - Have intuition about which "edge cases" to test.
- But for **parallel** programs:
  - Each run tests only a single schedule.
  - How can we test many different schedules?
  - How confident can we be when our tests pass?

Outline

- Challenges for parallel testing.
- **Random testing of parallel programs.**
- Detecting and predicting parallel bugs.
- Active Random Testing of parallel programs.
- Conclusions.

Testing Parallel Programs

- **Possible Idea:** Can we just run each test thousands of times?
- **Problem:** Often not much randomness in OS scheduling.
  - May waste much effort, but test few different schedules.
  - **Recall:** Some schedules tend to occur only under certain configurations – hardware, OS, etc.
  - One easy parameter to change: load on machine.

Stress Testing

- **Idea:** Test parallel program while oversubscribing the machine.
  - On a 4-core system, run with 8 or 16 threads.
  - Run several instances of the program at a time.
  - Increase size to overflow cache/memory.
  - **Effect:** Timing of threads will change, giving different thread schedules.

- **Pro:** Very simple idea, easy to implement.
  - And often works!

Noise Making / Random Scheduling

- **Idea:** Run with random thread schedules.
  - E.g., insert code like:

```
if (rand() < 0.01) usleep(100);
if (rand() < 0.01) yield();
```

  - Can add to only "suspicious" or "tricky" code.
  - Or use tool to seize control of thread scheduling.

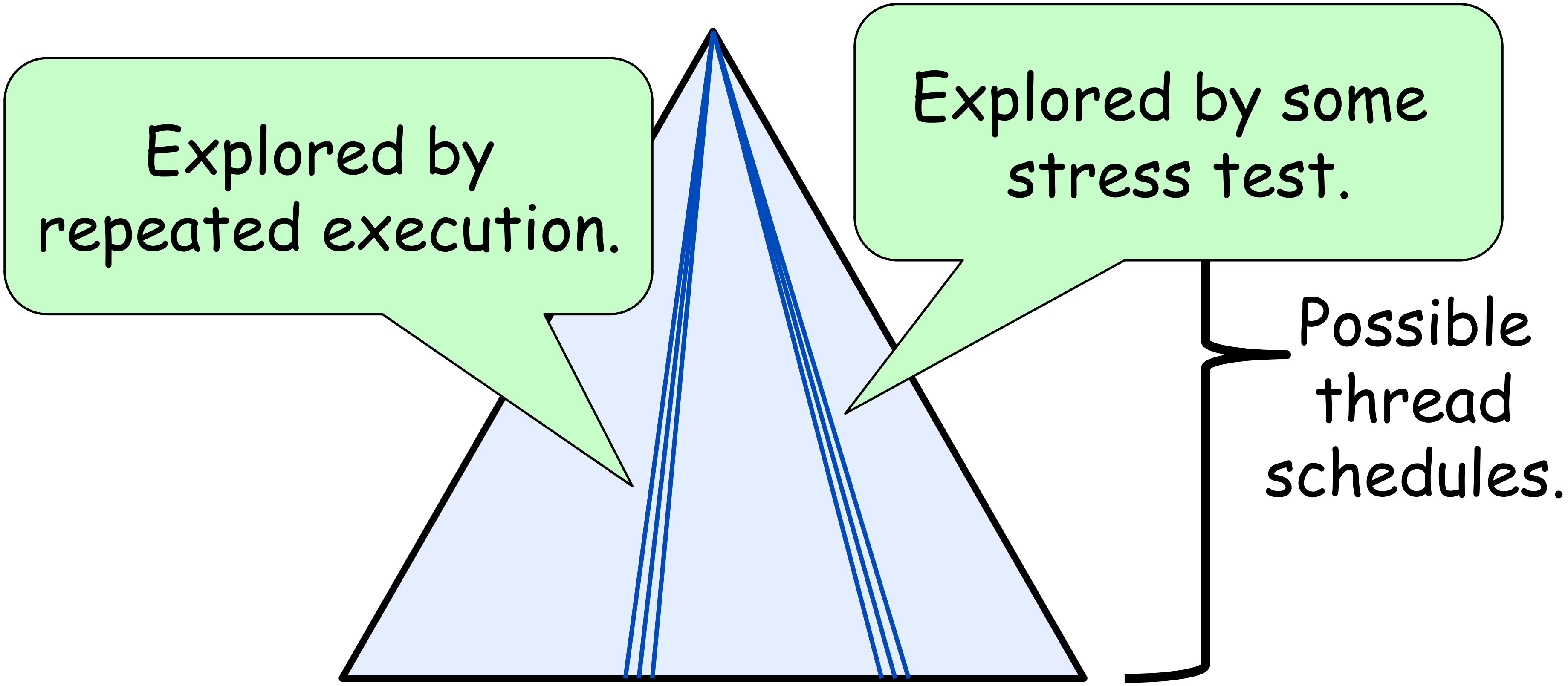
- **Pros:** Still fairly simple and often effective.
  - Explores different schedules than stress testing.
  - Many tools can perform this automatically.

Noise Making / Random Scheduling

- IBM's ConTest: Noise-making for Java.
  - Clever heuristics about where to insert delays.
- Berkeley's Thrille (C + pthreads) and CalFuzzer (Java) do simple random scheduling.
  - Extensible: Write testing scheduler for your app.
- Microsoft Research's Cuzz (for .NET).
  - New random scheduling algorithm with probabilistic guarantees for finding bugs.
  - Available soon.
- Many of these tools provide **replay** – same random number seed ==> same schedule.

Limitations of Random Scheduling

- Parallel programs have **huge** number of schedules – **exponential** in length of a run.



Explored by repeated execution.

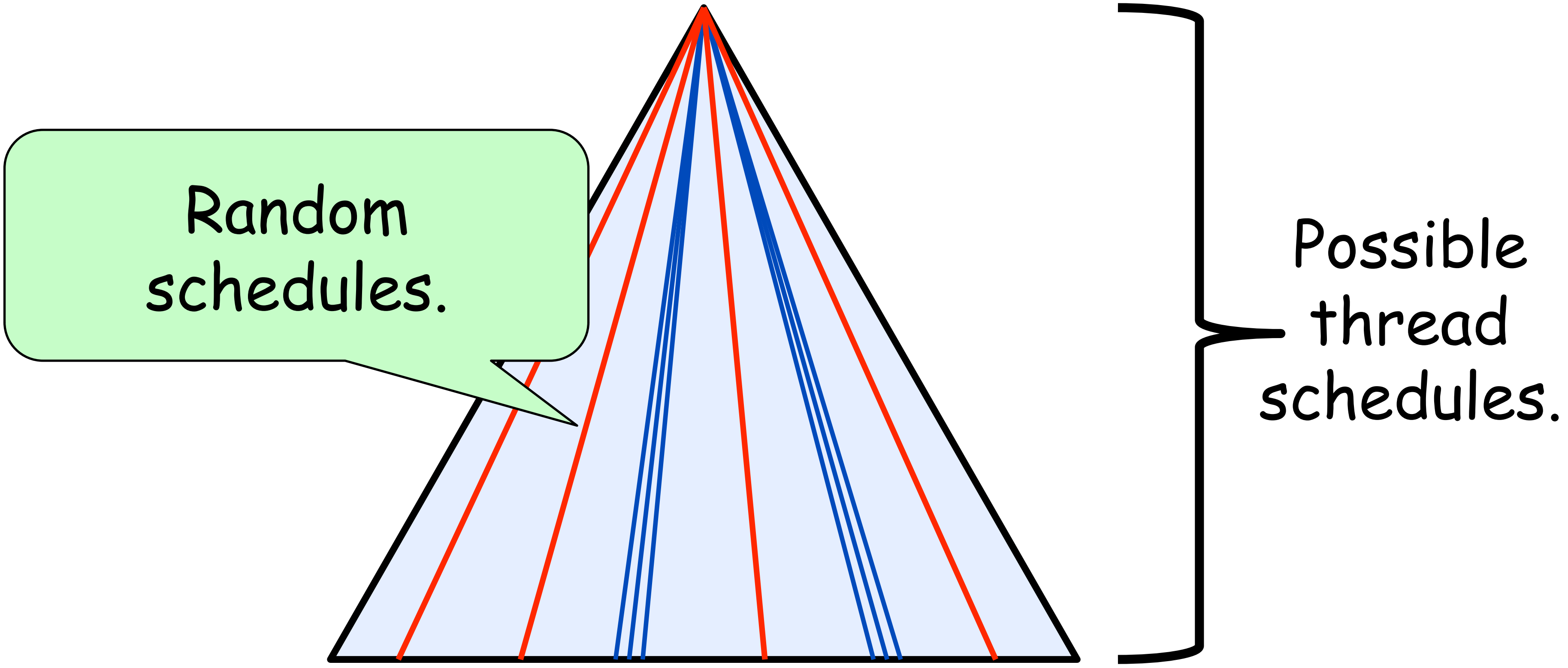
Explored by some stress test.

Possible thread schedules.

Limitations of Random Scheduling

- Parallel programs have **huge** number of schedules – **exponential** in length of a run.

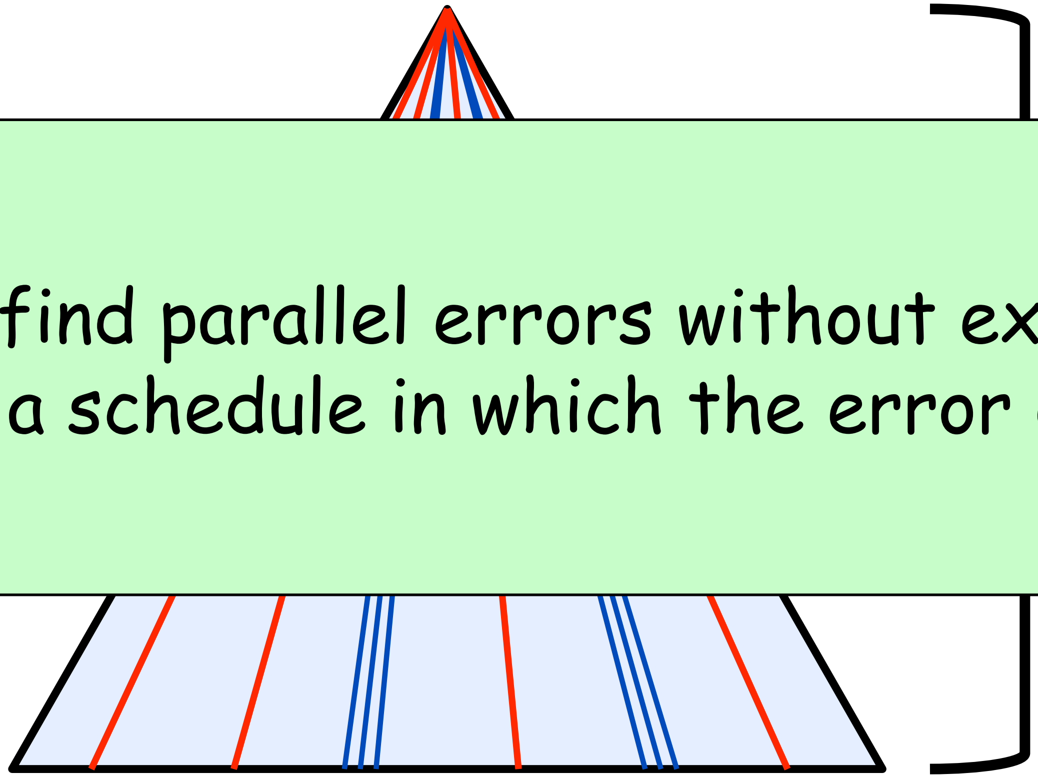
Vast majority of schedules will never be tested.



Limitations of Random Scheduling

- Parallel programs have **huge** number of schedules – **exponential** in length of a run.

Vast majority of schedules will never be tested.



Can we find parallel errors without explicitly testing a schedule in which the error occurs?

Outline

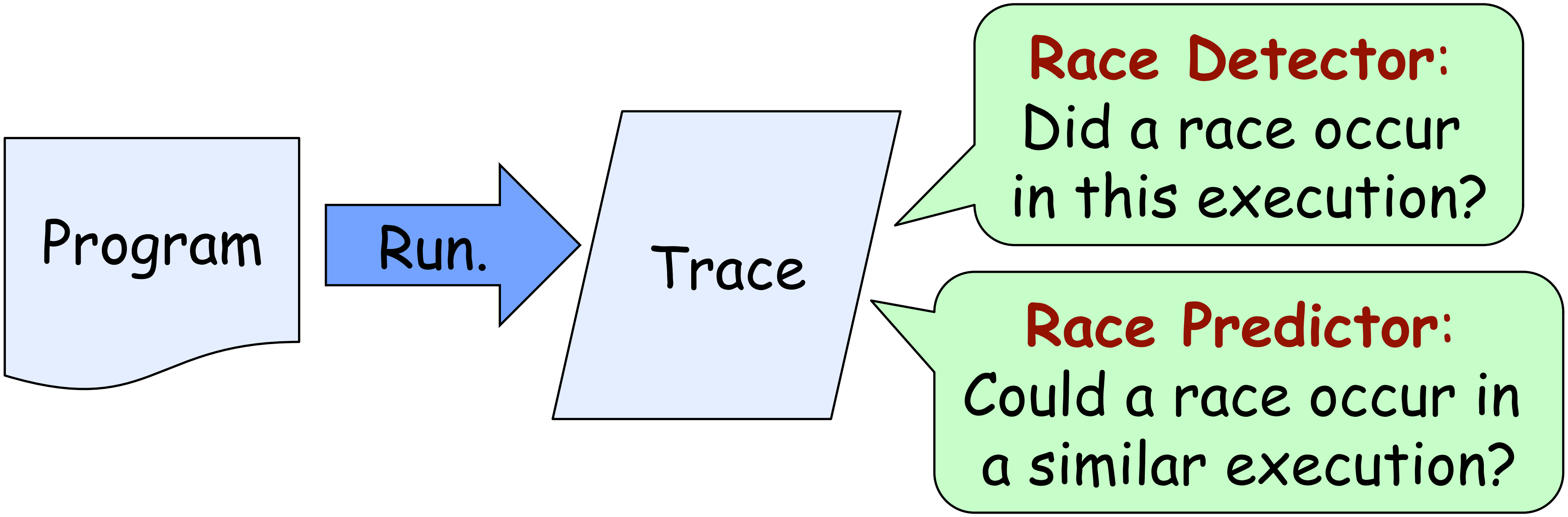
- Challenges for parallel testing.
- **Random testing of parallel programs.**
- Detecting and predicting parallel bugs.
- Active Random Testing of parallel programs.
- Conclusions.

Detecting/Predicting Parallel Bugs

- Say we observe a test run of a parallel program that doesn't obviously fail.
- **Key Question:** Can we find possible parallel bugs by examining the execution?

Detecting/Predicting Parallel Bugs

- Say we observe a test run of a parallel program that doesn't obviously fail.
- **Key Question:** Can we find possible parallel bugs by examining the execution?



Program

Run.

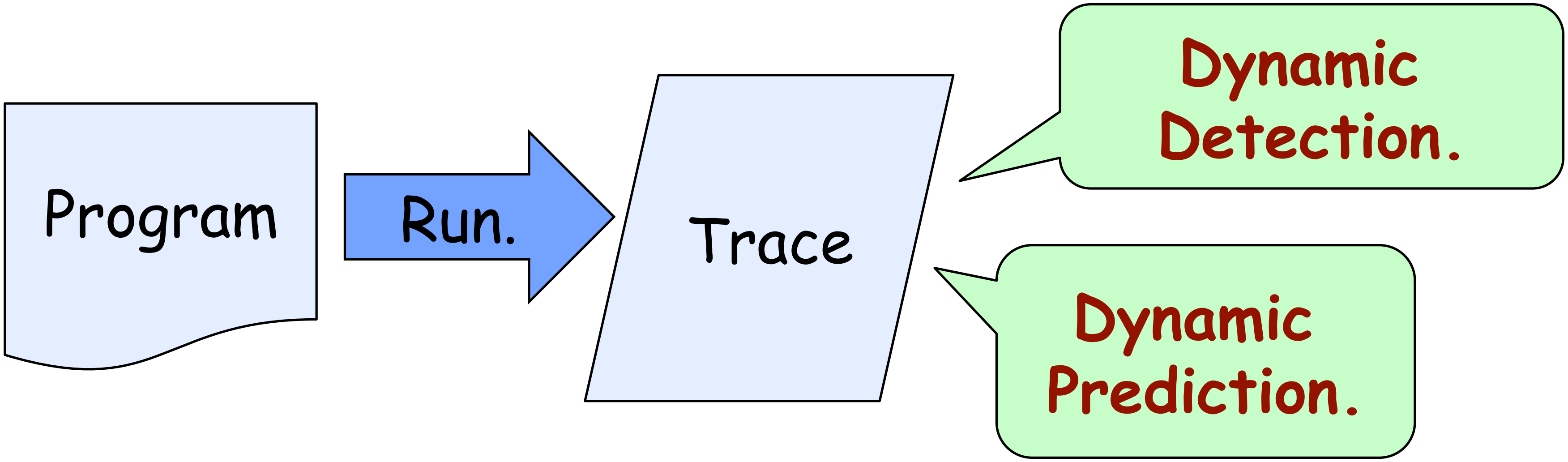
Trace

Race Detector:
Did a race occur in this execution?

Race Predictor:
Could a race occur in a similar execution?

Detecting/Predicting Parallel Bugs

- Techniques/tools exist for:
  - Data races.
  - Atomicity violations.
  - Deadlocks.
  - Memory consistency errors.



Program

Run.

Trace

Dynamic Detection.

Dynamic Prediction.

Data Race Detection/Prediction

- **Recall:** *A* **data race** occurs when two threads **concurrently** access the same memory, and a least one is a write.

```
        int x = 0;

Thread 1:           Thread 2:
 t1 = x;             t2 = x;
 x = t1 + 1;         x = t2 + 1;
```

Data race between two writes causes lost update – x can incorrectly be 1 instead of 2.

Data Race Detection/Prediction

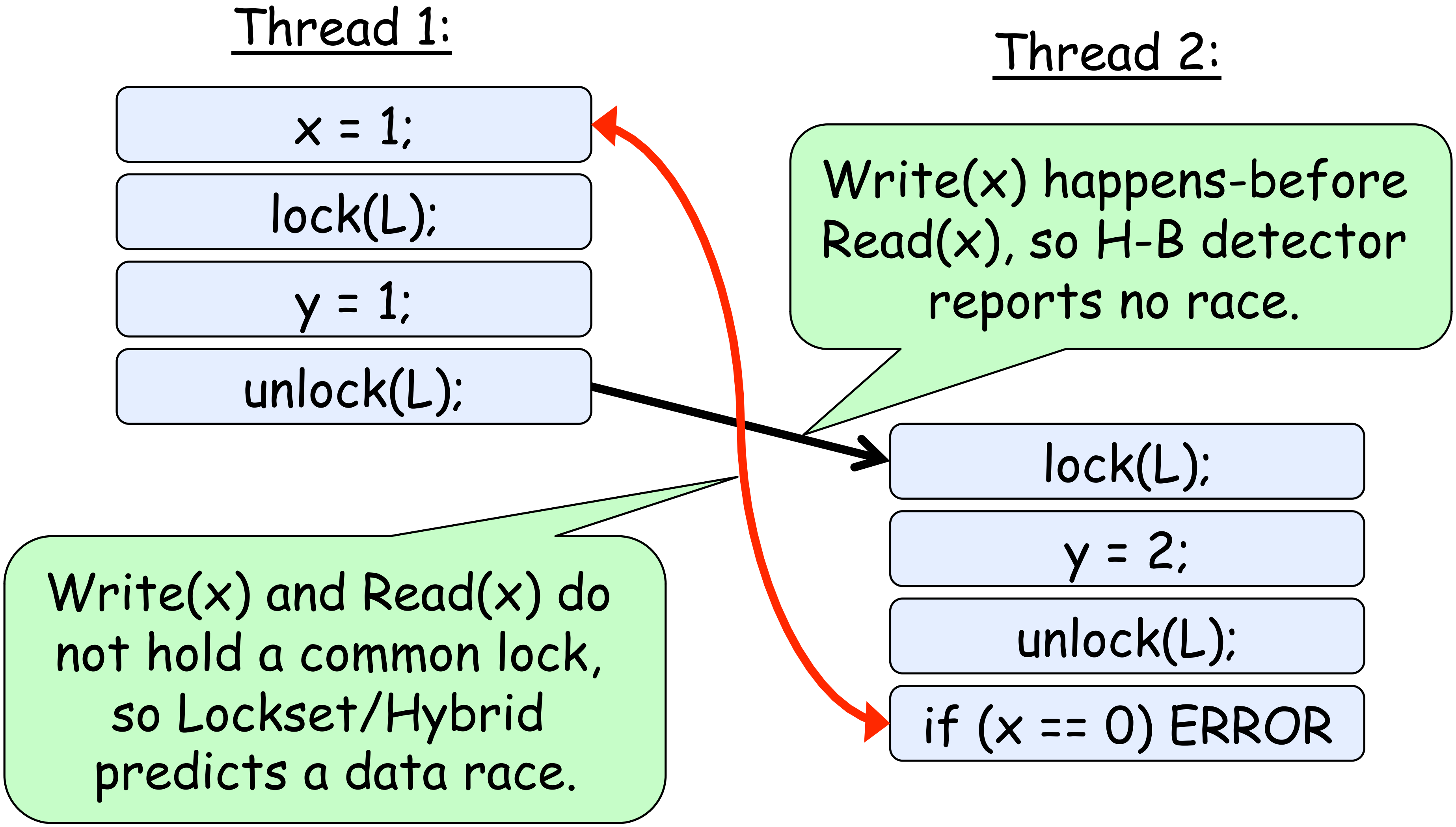
- 20+ years of research on race detection.
- Happens-Before Race Detection [Schonberg '89]:
  - Do two accesses to a variable occur, at least one a write, with no intervening synchronization?
  - No false warnings.
- Lockset Race Prediction [Savage, et al., '97]:
  - Does every access to a variable hold a common lock?
  - Efficient, but many **false warnings**.
- Hybrid Race Prediction [O'Callahan, Choi, 03]:
  - Combines Lockset with Happens-Before for better performance and fewer false warnings vs. Lockset.

Coverage vs. False Warnings

- **False Warning:** Tool reports a data race, but the race cannot happen in a real run.
- **Coverage:** How many of the real data races does a tool report?

- Hybrid race prediction:
  - Better coverage but more false warnings.
- Happens-Before race detection:
  - Fewer false warnings (still some, in practice) and less coverage.

Data Race Example I



Thread 1:

x = 1;

lock(L);

y = 1;

unlock(L);

Thread 2:

lock(L);

y = 2;

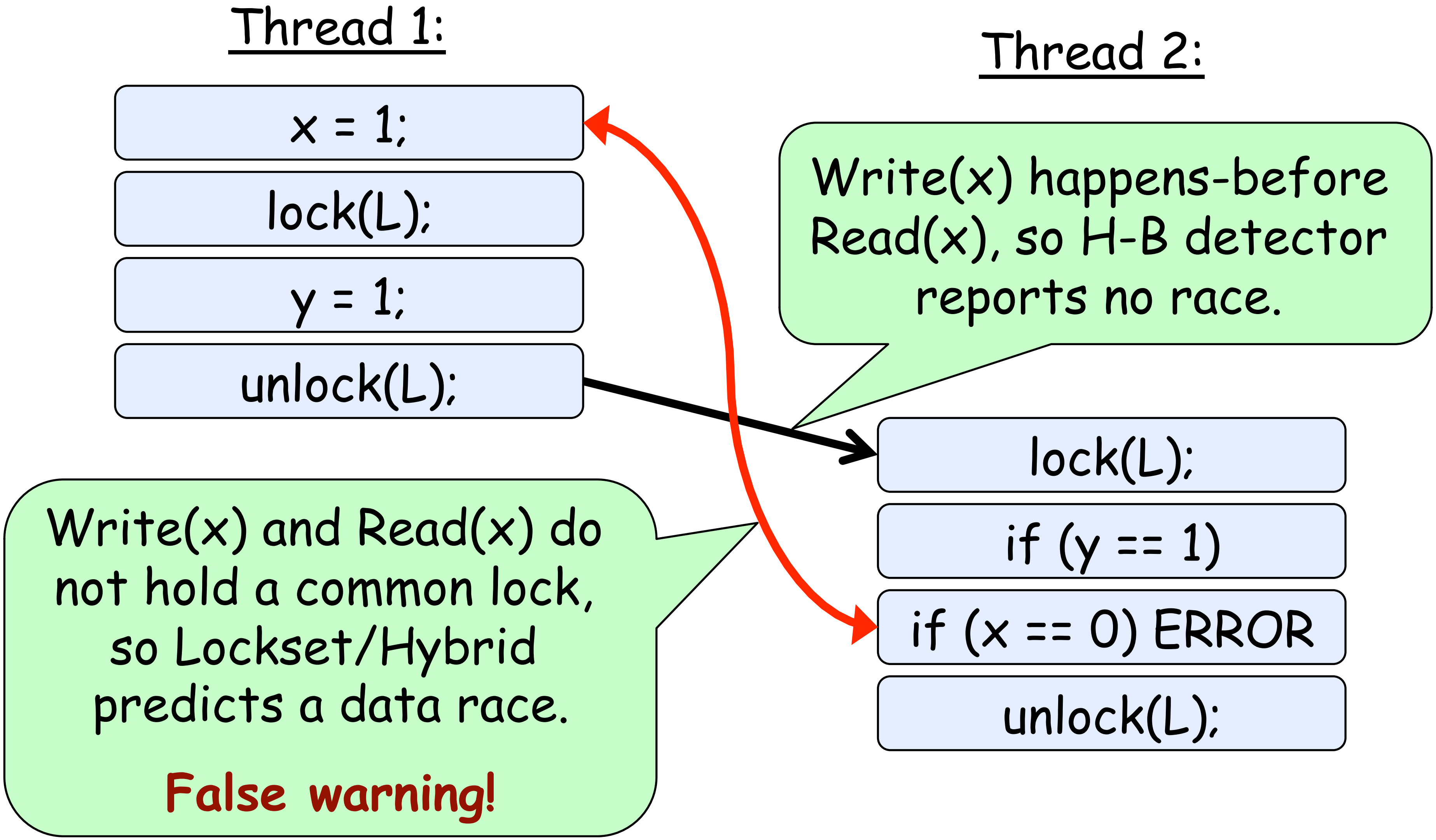
unlock(L);

if (x == 0) ERROR

Write(x) happens-before Read(x), so H-B detector reports no race.

Write(x) and Read(x) do not hold a common lock, so Lockset/Hybrid predicts a data race.

Data Race Example II



Thread 1:

```
x = 1;
lock(L);
y = 1;
unlock(L);
```

Thread 2:

```
lock(L);
if (y == 1)
if (x == 0) ERROR
unlock(L);
```

Write(x) happens-before Read(x), so H-B detector reports no race.

Write(x) and Read(x) do not hold a common lock, so Lockset/Hybrid predicts a data race.

False warning!

Dynamic Data Race Tools

- Intel Thread Checker for C + pthreads.
  - Happens-Before race detection.
- Valgrind-based tools for C + pthreads.
  - **Helgrind** and **DRD** (Happens-Before).
  - **ThreadSanitizer** (Hybrid).
- CHESS performs race detection for .NET
- CalFuzzer and Thrille: hybrid race detection for Java and C + pthreads.

Atomicity Detection/Prediction

- Dynamic detection and prediction tools exist for **atomicity** bugs, too.

```
int balance = 0;
lock L;

@atomic
void deposit(int a) {
  lock(L);
  int t = balance;
  unlock(L);
  lock(L);
  balance = t + a;
  unlock(L);
}
```

- Parallel calls to deposit intended to happen all-at-once (**atomically**).
- No data races because of lock L.
- But deposit can be wrongly interrupted.

Atomicity Detection/Prediction

- CalFuzzer predicts **atomicity** bugs for Java. (Not yet implemented in Thrille.)
  - User must specify which methods or other blocks of code are intended to be atomic.
  - Or CalFuzzer can guess – e.g. synchronized methods, bodies of parallel loops, etc.
- Large body of research on detecting/ predicting atomicity violations, but few publicly available tools.

Deadlock Prediction

- CalFuzzer also predicts **deadlocks** for Java. (Not yet implemented in Thrille.)

```
           lock L1, L2;

Thread 1:                Thread 2:
 ...                       ...
 lock(L1);                 lock(L2);
 ...                       ...
 lock(L2);                 lock(L1);
```

Aside: Static Analysis

- Have only discussed dynamic analyses.
  - Examine a real run/trace of a program.
- Static analyses predict data races, deadlocks, etc., without running a program.
  - Only examine the source code.
  - Area of active research for ~20 years.
  - Potentially much better coverage than dynamic analysis – examines all possible runs.
  - But typically also more false warnings.
- CHORD: static race and deadlock prediction for Java.

Outline

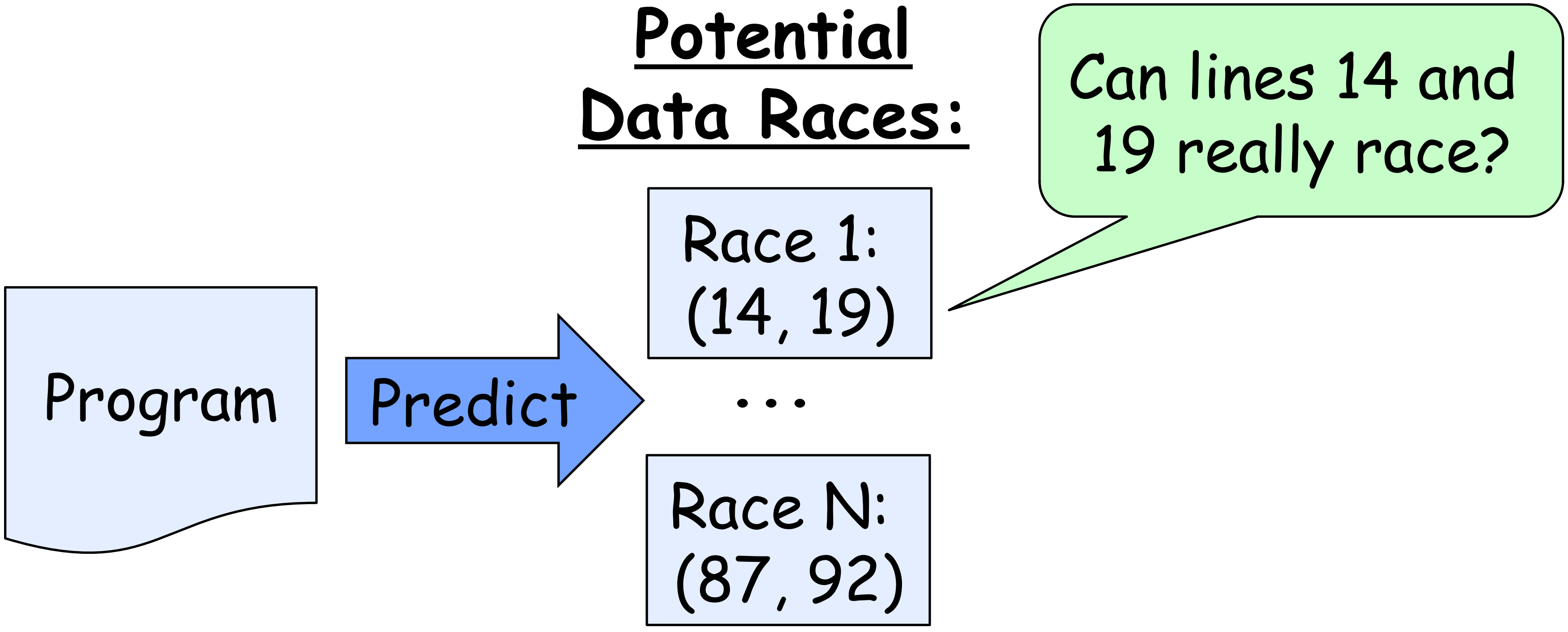
- Challenges for parallel testing.
- Random testing of parallel programs.
- Detecting and predicting parallel bugs.
- **Active Random Testing of parallel programs.**
- Conclusions.

Active Random Testing Overview

- **Problem:** Random testing can be very effective for parallel programs, but can miss many potential bugs.
- **Problem:** Predictive analyses find many bugs, but can have false warnings.
  - Time consuming and difficult to examine reported bugs and determine whether or not they are real.
- **Key Idea:** Combine them – use predictive analysis to find potential bugs, then **biased** random testing to actually create each bug.

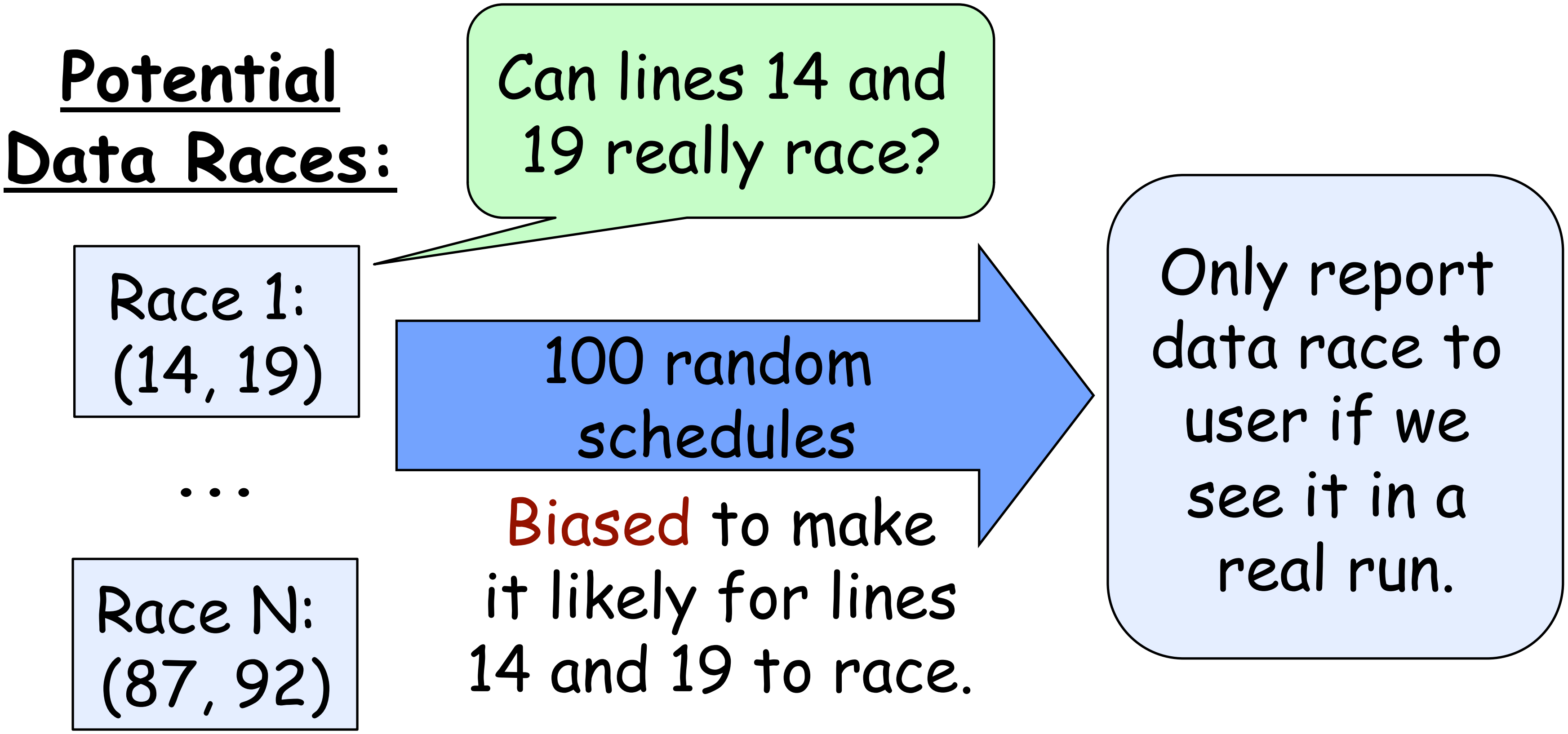
Active Random Testing Overview

- **Key Idea:** Use predictive analysis to find potential bugs, then **biased** random testing to try to actually create each bug.



Active Random Testing Overview

- **Key Idea:** Use predictive analysis to find potential bugs, then **biased** random testing to try to actually create each bug.



Potential Data Races:

Race 1: (14, 19)

…

Race N: (87, 92)

Can lines 14 and 19 really race?

100 random schedules

Biased to make it likely for lines 14 and 19 to race.

Only report data race to user if we see it in a real run.

Active Random Testing

- CalFuzzer is our extensible, open-source tool for active testing of Java programs.
  - For data races, atomicity bugs, and deadlocks.
  - RaceFuzzer is the active testing algorithm for data races – will show by example.
- Thrille for C + pthreads.
  - For data races.
- And UPC-Thrille for Unified Parallel C.
  - Part of the Berkeley UPC system by year's end.

RACEFUZZER using an example

```
Thread1              Thread2              Thread3
foo(o1);             bar(o1);             foo(o2);

sync foo(C x) {      bar(C y) {
  s1:  g1();           s6:  if (y.f==1)
  s2:  g2();           s7:     ERROR;
  s3:  g3();         }
  s4:  g4();
  s5:  x.f = 1;
}
```

Run Predictive Analysis: Statement pair (s5,s6) are in race

RACEFUZZER using an example

```
Thread1              Thread2              Thread3
foo(o1);             bar(o1);             foo(o2);

sync foo(C x) {      bar(C y) {
  s1:  g1();           s6:  if (y.f==1)
  s2:  g2();           s7:      ERROR;
  s3:  g3();         }
  s4:  g4();
  s5:  x.f = 1;
}
```

Run Predictive Analysis: Statement pair (s5,s6) are in race

RACEFUZZER using an example

(s5,s6) in race

```
Thread1          Thread2          Thread3
foo(o1);         bar(o1);         foo(o2);

sync foo(C x) {  bar(C y) {
  s1:  g1();       s6:  if (y.f==1)
  s2:  g2();       s7:      ERROR;
  s3:  g3();     }
  s4:  g4();
  s5:  x.f = 1;
}
```

Goal: Create a trace exhibiting the race

RACEFUZZER using an example

(s5,s6) in race

```
Thread1               Thread2              Thread3
foo(o1);              bar(o1);             foo(o2);

sync foo(C x) {       bar(C y) {
  s1:  g1();            s6:  if (y.f==1)
  s2:  g2();            s7:      ERROR;
  s3:  g3();          }
  s4:  g4();
  s5:  x.f = 1;
}
```

Example Trace:

```
s1:  g1();
s2:  g2();
s3:  g3();
s1:  g1();
s2:  g2();
s3:  g3();
s4:  g4();
s5:  o1.f = 1;
s6:  if (o1.f==1)
s7:      ERROR;
s4:  g4();
s5:  o2.f = 1;
```

Racing Statements Temporally Adjacent

Goal: Create a trace exhibiting the race

RACEFUZZER using an example

(s5,s6) in race

Thread1

```
foo(o1);

sync foo(C x) {
  s1:  g1();
  s2:  g2();
  s3:  g3();
  s4:  g4();
  s5:  x.f = 1;
}
```

Thread2

```
bar(o1);

bar(C y) {
  s6:  if (y.f==1)
  s7:       ERROR;
}
```

Thread3

```
foo(o2);
```

Execution:

RACEFUZZER using an example

(s5,s6) in race

Thread1

```
foo(o1);

sync foo(C x) {
  s1:  g1();
  s2:  g2();
  s3:  g3();
  s4:  g4();
  s5:  x.f = 1;
}
```

Thread2

```
bar(o1);

bar(C y) {
  s6:  if (y.f==1)
  s7:      ERROR;
}
```

Thread3

```
foo(o2);
```

Execution:

```
  s1:  g1();
```

RACEFUZZER using an example

(s5,s6) in race

Thread1

```
foo(o1);

sync foo(C x) {
  s1:  g1();
  s2:  g2();
  s3:  g3();
  s4:  g4();
  s5:  x.f = 1;
}
```

Thread2

```
bar(o1);

bar(C y) {
  s6:  if (y.f==1)
  s7:       ERROR;
}
```

Thread3

```
foo(o2);
```

Execution:

```
  s1:  g1();
```

RACEFUZZER using an example

(s5,s6) in race

Thread1

```
foo(o1);

sync foo(C x) {
  s1:  g1()
  s2:  g2();
  s3:  g3();
  s4:  g4();
  s5:  x.f = 1;
}
```

Thread2

```
bar(o1);

bar(C y) {
  s6:  if (y.f==1)
  s7:      ERROR;
}
```

Thread3

```
foo(o2);
```

Execution:

```
  s1:  g1();
  s1:  g1();
```

RACEFUZZER using an example

(s5,s6) in race

Thread1

```
foo(o1);

sync foo(C x) {
  s1:  g1()
  s2:  g2();
  s3:  g3();
  s4:  g4();
  s5:  x.f = 1;
}
```

Thread2

```
bar(o1);

bar(C y) {
  s6:  if (y.f==1)
  s7:      ERROR;
}
```

Thread3

```
foo(o2);
```

Execution:

```
  s1:  g1();
  s1:  g1();
```

RACEFUZZER using an example

(s5,s6) in race

Thread1

```
foo(o1);

sync foo(C x) {
  s1:  g1()
  s2:  g2();
  s3:  g3();
  s4:  g4();
  s5:  x.f = 1;
}
```

Thread2

```
bar(o1);

bar(C y) {
  s6:  if (y.f==1)
  s7:      ERROR;
}
```

Thread3

```
foo(o2);
```

Execution:

```
  s1:  g1();
  s1:  g1();
  s6:  if (o1.f==1)
```

RACEFUZZER using an example

(s5,s6) in race

Thread1

```
foo(o1);

sync foo(C x) {
  s1:  g1()
  s2:  g2();
  s3:  g3();
  s4:  g4();
  s5:  x.f = 1;
}
```

Thread2

```
bar(o1);

bar(C y) {
  s6:  if (y.f==1)
  s7:      ERROR;
}
```

Thread3

```
foo(o2);
```

Execution:

```
  s1:  g1();
  s1:  g1();
  s6:  if (o1.f==1)
```

RACEFUZZER using an example

(s5,s6) in race

Thread1

```
foo(o1);

sync foo(C x) {
  s1:  g1()
  s2:  g2();
  s3:  g3();
  s4:  g4();
  s5:  x.f = 1;
}
```

Thread2

```
bar(o1);

bar(C y) {
  s6:  if (y.f==1)
  s7:       ERROR;
}
```

Thread3

```
foo(o2);
```

Execution:

```
  s1:  g1();
  s1:  g1();
  s6:  if (o1.f==1)
```

Postponed = { }

RACEFUZZER using an example

(s5,s6) in race

```
Thread1               Thread2               Thread3          Execution:
foo(o1);              bar(o1);              foo(o2);           s1:  g1();
                                                               s1:  g1();
sync foo(C x) {       bar(C y) {                               s6:  if (o1.f==1)
  s1:  g1()             s6:  if (y.f==1)
  s2:  g2();            s7:      ERROR;
  s3:  g3();          }
  s4:  g4();
  s5:  x.f = 1;
}
```

Do not postpone
if there is a deadlock

Postponed = { s6: if (o1.f==1)}

RACEFUZZER using an example

(s5,s6) in race

Thread1

```
foo(o1);

sync foo(C x) {
  s1:  g1()
  s2:  g2();
  s3:  g3();
  s4:  g4();
  s5:  x.f = 1;
}
```

Thread2

```
bar(o1);

bar(C y) {
  s6:  if (y.f==1)
  s7:       ERROR;
}
```

Thread3

```
foo(o2);
```

Execution:

```
  s1:  g1();
  s1:  g1();
```

Postponed = {s6: if (o1.f==1) }

RACEFUZZER using an example

(s5,s6) in race

Thread1

```
foo(o1);

sync foo(C x) {
  s1:  g1()
  s2:  g2();
  s3:  g3();
  s4:  g4();
  s5:  x.f = 1;
}
```

Thread2

```
bar(o1);

bar(C y) {
  s6:  if (y.f==1)
  s7:      ERROR;
}
```

Thread3

```
foo(o2);
```

Execution:

```
  s1:  g1();
  s1:  g1();
  s2:  g2();
```

Postponed = {s6: if (o1.f==1) }

RACEFUZZER using an example

(s5,s6) in race

```
Thread1              Thread2               Thread3        Execution:
foo(o1);             bar(o1);              foo(o2);         s1:  g1();
                                                            s1:  g1();
sync foo(C x) {      bar(C y) {                             s2:  g2();
  s1:  g1()            s6:  if (y.f==1)
  s2:  g2();           s7:      ERROR;
  s3:  g3();         }
  s4:  g4();
  s5:  x.f = 1;
}
```

Postponed = {s6: if (o1.f==1) }

RACEFUZZER using an example

(s5,s6) in race

```
Thread1              Thread2              Thread3        Execution:
foo(o1);             bar(o1);             foo(o2);         s1:  g1();
                                                           s1:  g1();
sync foo(C x) {      bar(C y) {                            s2:  g2();
  s1:  g1()            s6:  if (y.f==1)                    s2:  g2();
  s2:  g2();           s7:     ERROR;                      s3:  g3();
  s3:  g3();         }                                     s3:  g3();
  s4:  g4();                                               s4:  g4();
  s5:  x.f = 1;                                            s5:  o2.f = 1;
}
```

Postponed = {s6: if (o1.f==1) }

RACEFUZZER using an example

(s5,s6) in race

Thread1

```
foo(o1);

sync foo(C x) {
  s1:  g1()
  s2:  g2();
  s3:  g3();
  s4:  g4();
  s5:  x.f = 1;
}
```

Thread2

```
bar(o1);

bar(C y) {
  s6:  if (y.f==1)
  s7:      ERROR;
}
```

Thread3

```
foo(o2);
```

Execution:

```
  s1:  g1();
  s1:  g1();
  s2:  g2();
  s2:  g2();
  s3:  g3();
  s3:  g3();
  s4:  g4();
  s5:  o2.f = 1;
```

Postponed = {s6: if (o1.f==1) }

RACEFUZZER using an example

(s5,s6) in race

Thread1

```
foo(o1);

sync foo(C x) {
  s1:  g1()
  s2:  g2();
  s3:  g3();
  s4:  g4();
  s5:  x.f = 1;
}
```

Thread2

```
bar(o1);

bar(C y) {
  s6:  if (y.f==1)
  s7:      ERROR;
}
```

Thread3

```
foo(o2);
```

Execution:

```
  s1:  g1();
  s1:  g1();
  s2:  g2();
  s2:  g2();
  s3:  g3();
  s3:  g3();
  s4:  g4();
  s5:  o2.f = 1;
```

Race?

Postponed = {s6: if (o1.f==1) }

RACEFUZZER using an example

(s5,s6) in race

Thread1

```
foo(o1);

sync foo(C x) {
  s1:  g1()
  s2:  g2();
  s3:  g3();
  s4:  g4();
  s5:  x.f = 1;
}
```

Thread2

```
bar(o1);

bar(C y) {
  s6:  if (y.f==1)
  s7:      ERROR;
}
```

Thread3

```
foo(o2);
```

Execution:

```
  s1:  g1();
  s1:  g1();
  s2:  g2();
  s2:  g2();
  s3:  g3();
  s3:  g3();
  s4:  g4();
  s5:  o2.f = 1;
```

Race?
NO
o1.f ≠ o2.f

Postponed = {s6: if (o1.f==1) }

RACEFUZZER using an example

(s5,s6) in race

Thread1

```
foo(o1);

sync foo(C x) {
  s1:  g1()
  s2:  g2();
  s3:  g3();
  s4:  g4();
  s5:  x.f = 1;
}
```

Thread2

```
bar(o1);

bar(C y) {
  s6:  if (y.f==1)
  s7:      ERROR;
}
```

Thread3

```
foo(o2);
```

Execution:

```
s1:  g1();
s1:  g1();
s2:  g2();
s2:  g2();
s3:  g3();
s3:  g3();
s4:  g4();

s5:  o2.f = 1;
```

Postponed = {s6: if (o1.f==1),   s5: o2.f = 1; }

RACEFUZZER using an example

(s5,s6) in race

Thread1	Thread2	Thread3	Execution:
foo(o1);	bar(o1);	foo(o2);	s1: g1();
			s1: g1();
sync foo(C x) {	bar(C y) {		s2: g2();
s1: g1()	s6: if (y.f==1)		s2: g2();
s2: g2();	s7: ERROR;		s3: g3();
s3: g3();	}		s3: g3();
s4: g4();			s4: g4();
s5: x.f = 1;			
}			

Postponed = {s6: if (o1.f==1), s5: o2.f = 1; }

RACEFUZZER using an example

(s5,s6) in race

```
Thread1            Thread2              Thread3         Execution:
foo(o1);           bar(o1);             foo(o2);          s1:  g1();
                                                          s1:  g1();
sync foo(C x) {    bar(C y) {                             s2:  g2();
  s1:  g1()          s6:  if (y.f==1)                     s2:  g2();
  s2:  g2();         s7:      ERROR;                      s3:  g3();
  s3:  g3();       }                                      s3:  g3();
  s4:  g4();                                              s4:  g4();
  s5:  x.f = 1;                                           s4:  g4();
}
```

Postponed = {s6: if (o1.f==1), s5: o2.f = 1; }

RACEFUZZER using an example

(s5,s6) in race

```
Thread1              Thread2              Thread3         Execution:
foo(o1);             bar(o1);             foo(o2);          s1:  g1();
                                                            s1:  g1();
sync foo(C x) {      bar(C y) {                             s2:  g2();
  s1:  g1()            s6:  if (y.f==1)                     s2:  g2();
  s2:  g2();           s7:      ERROR;                      s3:  g3();
  s3:  g3();         }                                      s3:  g3();
  s4:  g4();                                                s4:  g4();
  s5:  x.f = 1;                                             s4:  g4();
}                                                           s5:  o1.f = 1;
```

Postponed = {s6: if (o1.f==1), s5: o2.f = 1; }

RACEFUZZER using an example

(s5,s6) in race

Thread1	Thread2	Thread3	Execution:
foo(o1);	bar(o1);	foo(o2);	s1: g1();
			s1: g1();
sync foo(C x) {	bar(C y) {		s2: g2();
s1: g1()	s6: if (y.f==1)		s2: g2();
s2: g2();	s7: ERROR;		s3: g3();
s3: g3();	}		s3: g3();
s4: g4();			s4: g4();
s5: x.f = 1;			s4: g4();
}			s5: o1.f = 1;

Postponed = {s6: if (o1.f==1), s5: o2.f = 1; }

RACEFUZZER using an example

(s5,s6) in race

Thread1

```
foo(o1);

sync foo(C x) {
  s1:  g1()
  s2:  g2();
  s3:  g3();
  s4:  g4();
  s5:  x.f = 1;
}
```

Thread2

```
bar(o1);

bar(C y) {
  s6:  if (y.f==1)
  s7:      ERROR;
}
```

Thread3

```
foo(o2);
```

Execution:

```
s1:  g1();
s1:  g1();
s2:  g2();
s2:  g2();
s3:  g3();
s3:  g3();
s4:  g4();
s4:  g4();
s5:  o1.f = 1;
```

Race?
YES
o1.f = o1.f

Postponed = {s6: if (o1.f==1), s5: o2.f = 1; }

RACEFUZZER using an example

(s5,s6) in race

Thread1
```
foo(o1);

sync foo(C x) {
  s1:  g1()
  s2:  g2();
  s3:  g3();
  s4:  g4();
  s5:  x.f = 1;
}
```

Thread2
```
bar(o1);

bar(C y) {
  s6:  if (y.f==1)
  s7:      ERROR;
}
```

Thread3
```
foo(o2);
```

Execution:
```
  s1:  g1();
  s1:  g1();
  s2:  g2();
  s2:  g2();
  s3:  g3();
  s3:  g3();
  s4:  g4();
  s4:  g4();
```

```
s6:  if (o1.f==1)    s5:  o1.f = 1;
```

Postponed = {s5: o2.f = 1; }

RACEFUZZER using an example

(s5,s6) in race

```
Thread1            Thread2              Thread3       Execution:
foo(o1);           bar(o1);             foo(o2);        s1:  g1();
                                                        s1:  g1();
sync foo(C x) {    bar(C y) {                           s2:  g2();
  s1:  g1()          s6:  if (y.f==1)                   s2:  g2();
  s2:  g2();         s7:      ERROR;                    s3:  g3();
  s3:  g3();       }                                    s3:  g3();
  s4:  g4();                                            s4:  g4();
  s5:  x.f = 1;                                         s4:  g4();
}                                                       s5:  o1.f = 1;
                                                        s6:  if (o1.f==1)
```

Postponed = {s5: o2.f = 1; }

RACEFUZZER using an example

(s5,s6) in race

Thread1

```
foo(o1);

sync foo(C x) {
  s1:  g1()
  s2:  g2();
  s3:  g3();
  s4:  g4();
  s5:  x.f = 1;
}
```

Thread2

```
bar(o1);

bar(C y) {
  s6:  if (y.f==1)
  s7:      ERROR;
}
```

Thread3

```
foo(o2);
```

Execution:

```
  s1:  g1();
  s1:  g1();
  s2:  g2();
  s2:  g2();
  s3:  g3();
  s3:  g3();
  s4:  g4();
  s4:  g4();
  s5:  o1.f = 1;
  s6:  if (o1.f==1)
```

Racing Statements Temporally Adjacent

Postponed = {s5: o2.f = 1; }

RACEFUZZER using an example

(s5,s6) in race

Thread1

```
foo(o1);

sync foo(C x) {
  s1:  g1()
  s2:  g2();
  s3:  g3();
  s4:  g4();
  s5:  x.f = 1;
}
```

Thread2

```
bar(o1);

bar(C y) {
  s6:  if (y.f==1)
  s7:      ERROR;
}
```

Thread3

```
foo(o2);
```

Execution:

```
s1:  g1();
s1:  g1();
s2:  g2();
s2:  g2();
s3:  g3();
s3:  g3();
s4:  g4();
s4:  g4();
s5:  o1.f = 1;
s6:  if (o1.f==1)
s7:      ERROR;
```

Racing Statements Temporally Adjacent

Postponed = {s5: o2.f = 1; }

RACEFUZZER using an example

(s5,s6) in race

Thread1

```
foo(o1);

sync foo(C x) {
  s1:  g1()
  s2:  g2();
  s3:  g3();
  s4:  g4();
  s5:  x.f = 1;
}
```

Thread2

```
bar(o1);

bar(C y) {
  s6:  if (y.f==1)
  s7:      ERROR;
}
```

Thread3

```
foo(o2);
```

Execution:

```
s1:  g1();
s1:  g1();
s2:  g2();
s2:  g2();
s3:  g3();
s3:  g3();
s4:  g4();
s4:  g4();
s5:  o1.f = 1;
s6:  if (o1.f==1)
s7:      ERROR;
s5:  o2.f = 1;
```

Racing Statements Temporally Adjacent

Postponed = { }

Another RACEFUZZER Example

```
Thread1{
1: lock(L);
2:f1();
3:f2();
4:f3();
5:f4();
6:f5();
7:unlock(L);
8:if (x==0)
9:      ERROR;
}
```

```
Thread2{
10:     x = 1;
11:     lock(L);
12:     f6();
13:     unlock(L);
}
```

Another RACEFUZZER Example

```
Thread1{
1: lock(L);
2:f1();
3:f2();
4:f3();
5:f4();
6:f5();
7:unlock(L);
8:if (x==0)
9:      ERROR;
}
```

```
Thread2{
10:     x = 1;
11:     lock(L);
12:     f6();
13:     unlock(L);
}
```

Race

This race would occur rarely under a normal or naively-random execution.

RaceFuzzer creates the race with high probability.

Racing Pair: (8,10)

Another RACEFUZZER Example

```
Thread1{
1: lock(L);
2:f1();
3:f2();
4:f3();
5:f4();
6:f5();
7:unlock(L);
8:if (x==0)
9:      ERROR;
}
```

```
Thread2{
10:     x = 1;
11:     lock(L);
12:     f6();
13:     unlock(L);
}
```

Racing Pair: (8,10)   Postponed Set = {Thread2}

Another RACEFUZZER Example

```
Thread1{                Thread2{
1: lock(L);             10:     x = 1;
2:f1();                 11:     lock(L);
3:f2();                 12:     f6();
4:f3();                 13:     unlock(L);
5:f4();                 }
6:f5();
7:unlock(L);
8:if (x==0)
9:      ERROR;
}
```

Another RACEFUZZER Example

```
Thread1{
1: lock(L);
2:f1();
3:f2();
4:f3();
5:f4();
6:f5();
7:unlock(L);
8:if (x==0)
9:      ERROR;
}
```

```
Thread2{
10:     x = 1;
11:     lock(L);
12:     f6();
13:     unlock(L);
}
```

Another RACEFUZZER Example

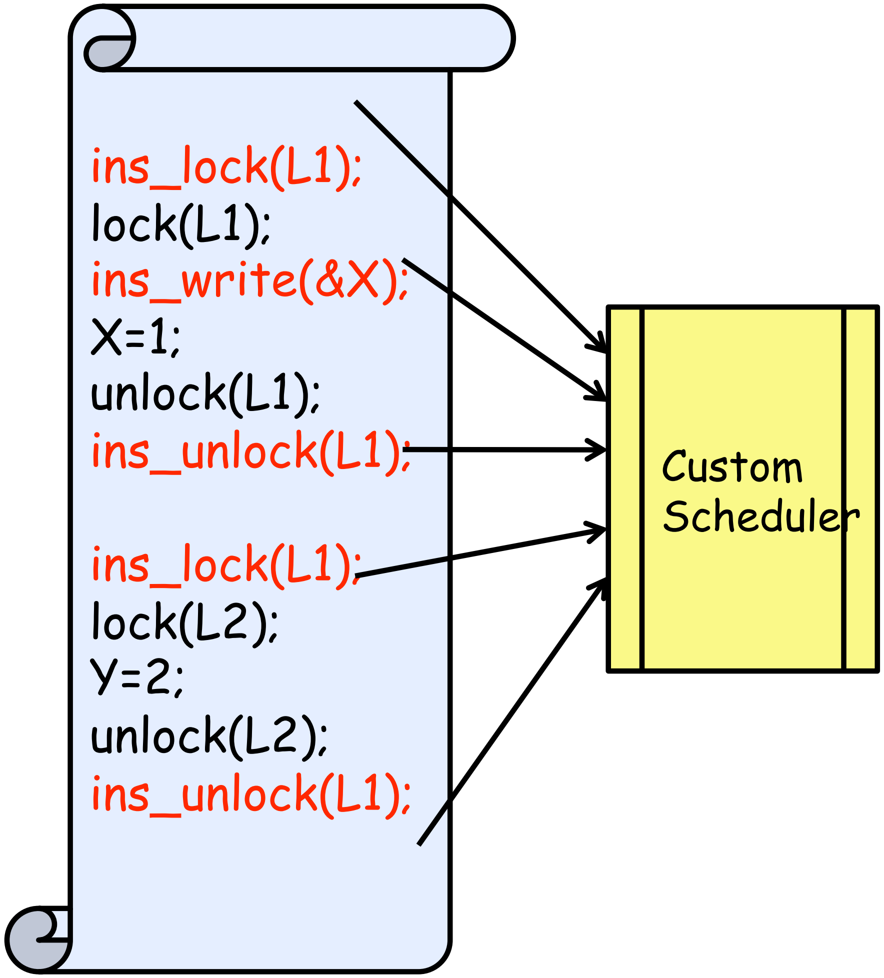
```
Thread1{
1: lock(L);
2:f1();
3:f2();
4:f3();
5:f4();
6:f5();
7:unlock(L);
8:if (x==0)
9:      ERROR;
}
```

```
Thread2{
10:     x = 1;
11:     lock(L);
12:     f6();
13:     unlock(L);
}
```

Hit error with 0.5 probability

Implementation

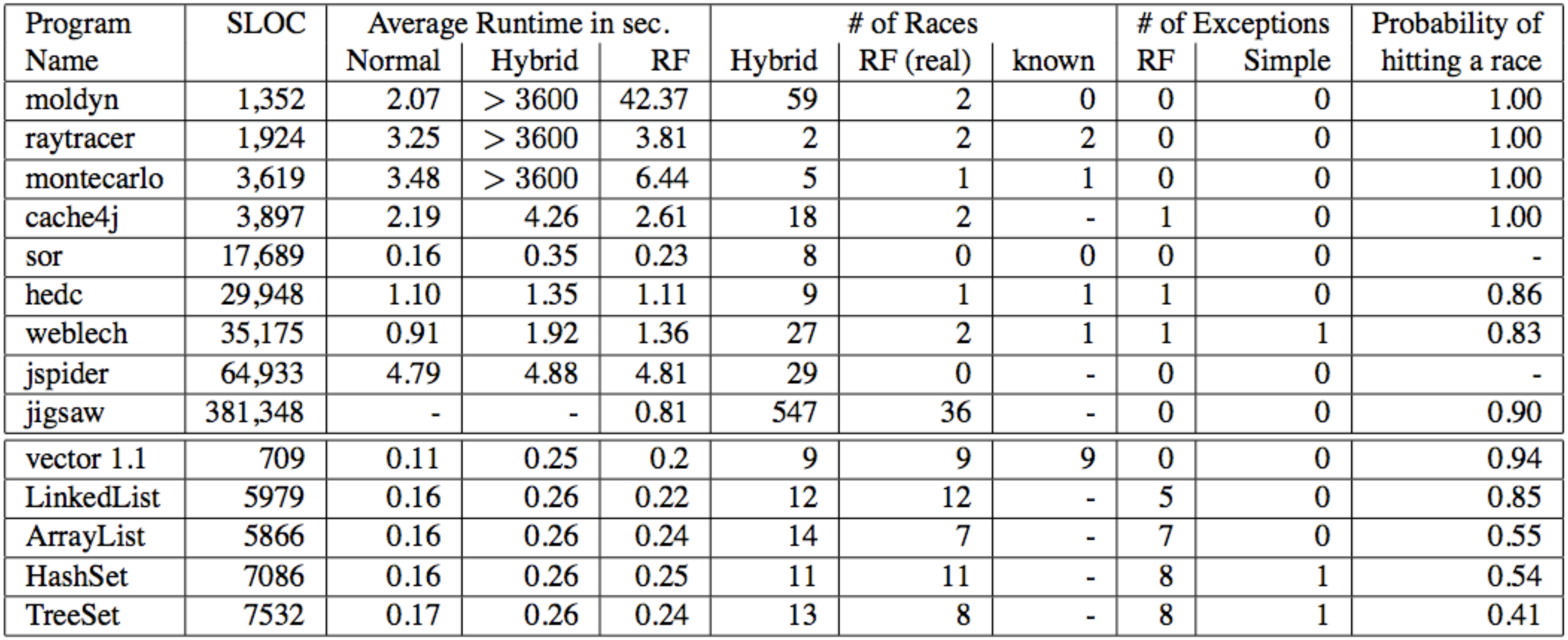
- RaceFuzzer: Part of CalFuzzer tool suite
- Instrument source using SOOT compiler framework
- Instrumentations are used to "hijack" the scheduler
  - Implement a custom scheduler
  - Run one thread at a time
  - Use semaphores to control threads
- Deadlock detector
  - Because we cannot instrument native method calls



```
ins_lock(L1);
lock(L1);
ins_write(&X);
X=1;
unlock(L1);
ins_unlock(L1);

ins_lock(L1);
lock(L2);
Y=2;
unlock(L2);
ins_unlock(L1);
```

Experimental Results



Program Name	SLOC	Average Runtime in sec.			# of Races			# of Exceptions		Probability of hitting a race
		Normal	Hybrid	RF	Hybrid	RF (real)	known	RF	Simple	
moldyn	1,352	2.07	> 3600	42.37	59	2	0	0	0	1.00
raytracer	1,924	3.25	> 3600	3.81	2	2	2	0	0	1.00
montecarlo	3,619	3.48	> 3600	6.44	5	1	1	0	0	1.00
cache4j	3,897	2.19	4.26	2.61	18	2	-	1	0	1.00
sor	17,689	0.16	0.35	0.23	8	0	0	0	0	-
hedc	29,948	1.10	1.35	1.11	9	1	1	1	0	0.86
weblech	35,175	0.91	1.92	1.36	27	2	1	1	1	0.83
jspider	64,933	4.79	4.88	4.81	29	0	-	0	0	-
jigsaw	381,348	-	-	0.81	547	36	-	0	0	0.90
vector 1.1	709	0.11	0.25	0.2	9	9	9	0	0	0.94
LinkedList	5979	0.16	0.26	0.22	12	12	-	5	0	0.85
ArrayList	5866	0.16	0.26	0.24	14	7	-	7	0	0.55
HashSet	7086	0.16	0.26	0.25	11	11	-	8	1	0.54
TreeSet	7532	0.17	0.26	0.24	13	8	-	8	1	0.41

Active Testing: Useful Features

- Classify real races from false alarms.
  - No false warnings.
- Inexpensive **replay** of a concurrent execution exhibiting a real race or other parallel bug
- Separate some harmful data races from benign races – i.e. whether or not the race leads to a crash or wrong output.
- Embarrassingly parallel.
  - Test different potential races / other bugs at the same time.

Active Testing: Limitations

- Not complete: can miss a real race.
  - Can only detect races that happen on the given test suite on some schedule.
- May not be able to separate all real races from false warnings.
  - Random scheduling may fail to create real race.
- May not be able to separate harmful races from benign races.
  - If error behavior not seen in random runs.
- Program is run sequentially during testing.

Active Testing Summary

- Combines benefits of random testing and predictive analysis.
  - Random testing amazingly effective in practice.
  - Even more so when biased with information about predicted bugs.
  - Can replay executions for debugging.

- Available now for Java (CalFuzzer) and Thrille (C + pthreads).
- UPC-Thrille for Unified Parallel C.
  - Part of the Berkeley UPC system by year's end.

Outline

- Challenges for parallel testing.
- Random testing of parallel programs.
- Detecting and predicting parallel bugs.
- Active Random Testing of parallel programs.
- **Conclusions.**

Conclusions

- Many tools available right now to help find bugs in parallel software.
  - Data races, atomicity violations, deadlocks.
- But no silver bullet.
  - Have to carefully design how an application threads will coordinate and share/protect data.
  - Tools will help catch mistakes when the design is accidentally not followed.
  - Ad hoc parallelization likely to never be correct, even with these tools.

ANY QUESTIONS?

Pointers to Tools I

- IBM ConTest (Noise-Making for Java): https://www.research.ibm.com/haifa/projects/verification/contest/index.html
- Cuzz (Random scheduling for C++/.NET): http://research.microsoft.com/en-us/projects/cuzz/

Pointers to Tools II

- Intel Thread Checker and Parallel Inspector (C/C++):
  http://software.intel.com/en-us/intel-thread-checker/
  http://software.intel.com/en-us/intel-parallel-inspector/
- Helgrind, DRD, ThreadSanitizer
  (Dynamic Data Race Detection/Prediction for C/C++):
  http://valgrind.org/docs/manual/hg-manual.html
  http://valgrind.org/docs/manual/drd-manual.html
  http://code.google.com/p/data-race-test/
- CHORD (Static Race/Deadlock Detection for Java):
  http://code.google.com/p/jchord/

Pointers to Tools III

- CalFuzzer (Java):
  http://srl.cs.berkeley.edu/~ksen/calfuzzer/
- Thrille (C):
  http://github.com/nicholasjalbert/Thrille
- CHESS (C++/.NET Model Checking, Race Detection):
  http://research.microsoft.com/en-us/projects/chess/default.aspx
- Java Path Finder (Model Checking for Java):
  http://babelfish.arc.nasa.gov/trac/jpf