Correctness Tools in the DOE Ecosystem

- **Endangered species that require Federal protection.**

- **Overall as a community, we are not very sophisticated when using testing and correctness tools.**
  - How many of you have a “Test Engineer” or a “QA Engineer” position posted?
  - How many of you know of Coverity or SilkTest?

- **There are very good reasons for the status quo**
  - Sociological – we like hero programmers
  - Practical – hero programmers can find bugs
    - Serial code with side-effects separated by MPI_...

- **Things are changing**

Enter GASynchrony…

Enter GASynchrony…

- **A place with :**
  - Global Address Spaces which obfuscates and breeds bugs
  - Asynchronous Execution which obfuscates and breeds bugs
  - Heterogeneous Hardware which obfuscates and breeds bugs

Finding and Debugging Concurrency Bugs at Scale

Chang-Seo Park, Paul Hargrove, Costin Iancu, **Koushik Sen**

also joint work with

Jacob Burnim, Tayfun Elmas, David Gay, Nicholas Jalbert, Pallavi Joshi, Mayur Naik, Chang-Seo Park, Christos Stergiou

Automatically Testing Sequential Programs

- Combine static and dynamic analysis for test generation
- Automated testing of sequential programs
  - DART: Directed Automated Random Testing
  - CUTE: Concolic Testing

```
void foo (input)  {
        ….  semicolon
        ….  semicolon
        ….  semicolon
        ….  semicolon
        while (p) {
                ….  semicolon
                ….  semicolon
                ASSERT(good);
                ….  semicolon
        }
        ….  semicolon
        ….  semicolon
        ….  semicolon
}
```

Testing Concurrent Programs

- Concurrent Programming is hard
  - Bugs happen non-deterministically
  - Data races, deadlocks, atomicity violations, etc.
- Goals: build a tool to test and debug concurrent and parallel programs
  - More Practical: works for large programs
  - Efficient
  - No false alarms
  - Finds many bugs quickly
  - Reproducible
- **Active random testing**.

Active Testing

- Phase 1: Static or dynamic analysis to find potential concurrency bug patterns
  - such as data races, deadlocks, atomicity violations
- Phase 2: “Direct” testing (or model checking) based on the bug patterns obtained from phase 1
  - Confirm bugs

Example Data Race in UPC

- Simple matrix vector multiply and apply F

C = foo (A × B)

Simple Example in UPC

```
void matvec(shared [N] int A[N][N],
            shared int B[N],
            shared int C[N]) {
  upc_forall(int i = 0; i < N; i++; &C[i]) {
    int sum = 0;
    for(int j = 0; j < N; j++)
       sum += A[i][j] * B[j];
    sum = foo(sum);
    C[i] = sum;
  }
}

assert(C == foo(A*B));
```

foo is an expensive function

Simple Example in UPC

```
void matvec(shared [N] int A[N][N],
            shared int B[N],
            shared int C[N]) {
  upc_forall(int i = 0; i < N; i++; &C[i]) {
    int sum = 0;
    for(int j = 0; j < N; j++)
      sum += A[i][j] * B[j];
    sum = foo(sum);
    C[i] = sum;
  }
}
```

```
assert(C == foo(A*B));
```

foo is an expensive function

$foo(x) = x$

$$\underset{C}{\begin{pmatrix} ? \\ ? \end{pmatrix}} = \underset{A}{\begin{pmatrix} 1 & 1 \\ 1 & 1 \end{pmatrix}} \underset{B}{\begin{pmatrix} 1 \\ 1 \end{pmatrix}}$$

Simple Example in UPC

```
void matvec(shared [N] int A[N][N],
            shared int B[N],
            shared int C[N]) {
   upc_forall(int i = 0; i < N; i++; &C[i]) {
      int sum = 0;
      for(int j = 0; j < N; j++)
          sum += A[i][j] * B[j];
      sum = foo(sum);
      C[i] = sum;
   }
}
```

```
assert(C == foo(A*B));
```

foo is an expensive function

$$foo(x) = x$$

$$\underset{C}{\begin{pmatrix}2\\2\end{pmatrix}} = \underset{A}{\begin{pmatrix}1 & 1\\1 & 1\end{pmatrix}}\underset{B}{\begin{pmatrix}1\\1\end{pmatrix}}$$

Simple Example in UPC: Problem?

```
void matvec(shared [N] int A[N][N],
            shared int B[N],
            shared int C[N]) {
  upc_forall(int i = 0; i < N; i++; &C[i]) {
    int sum = 0;
    for(int j = 0; j < N; j++)
      sum += A[i][j] * B[j];
    sum = foo(sum);
    C[i] = sum;
  }
}

assert(C == foo(A*B));
```

Do you see any problem is this code?

foo is an expensive function

Simple Example in UPC: Data Race

```
void matvec(shared [N] int A[N][N],
            shared int B[N],
            shared int C[N]) {
   upc_forall(int i = 0; i < N; i++; &C[i]) {
      int sum = 0;
      for(int j = 0; j < N; j++)
         sum += A[i][j] * B[j];
      sum = foo(sum);
      C[i] = sum;
   }
}
```

Data Race!

```
assert(C == foo(A*B));
```

foo is an expensive function

Do you see any problem is this code?

Yes, if we call matvec(A,B,B)

Simple Example in UPC: Data Race

foo(x) = x

```
void matvec(shared [N] int A[N][N],
                        shared int B[N],
                        shared int C[N]) {
   upc_forall(int i = 0; i < N; i++; &C[i]) {
      int sum = 0;
      for(int j = 0; j < N; j++)
         sum += A[i][j] * B[j];
      sum = foo(sum);
      C[i] = sum;
   }
}
```

Data Race!

```
assert(C == foo(A*B));
```

$$\underset{B}{\begin{pmatrix} ? \\ ? \end{pmatrix}} = \underset{A}{\begin{pmatrix} 1 & 1 \\ 1 & 1 \end{pmatrix}} \underset{B}{\begin{pmatrix} 1 \\ 1 \end{pmatrix}}$$

Do you see any problem is this code?

Yes, if we call matvec(A,B,B)

foo is an expensive function

Simple Example in UPC: Data Race

foo(x) = x

```
void matvec(shared [N] int A[N][N],
                   shared int B[N],
                   shared int C[N]) {
   upc_forall(int i = 0; i < N; i++; &C[i]) {
      int sum = 0;
      for(int j = 0; j < N; j++)
         sum += A[i][j] * B[j];
      sum = foo(sum);
      C[i] = sum;
   }
}
```

$$\underset{B}{\begin{pmatrix}2\\?\end{pmatrix}} = \underset{A}{\begin{pmatrix}1 & 1\\1 & 1\end{pmatrix}}\underset{B}{\begin{pmatrix}2\\1\end{pmatrix}}$$

Data Race!

Do you see any problem is this code?

Yes, if we call matvec(A,B,B)

```
assert(C == foo(A*B));
```

foo is an expensive function

Simple Example in UPC: Data Race

foo(x) = x

```
void matvec(shared [N] int A[N][N],
                          shared int B[N],
                          shared int C[N]) {
   upc_forall(int i = 0; i < N; i++; &C[i]) {
      int sum = 0;
      for(int j = 0; j < N; j++)
         sum += A[i][j] * B[j];
      sum = foo(sum);
      C[i] = sum;
   }
}
```

$$\underset{B}{\begin{pmatrix}2\\3\end{pmatrix}} = \underset{A}{\begin{pmatrix}1 & 1\\1 & 1\end{pmatrix}}\underset{B}{\begin{pmatrix}2\\3\end{pmatrix}}$$

Data Race!

```
assert(C == foo(A*B));
```

foo is an expensive function

Do you see any problem is this code?

Yes, if we call matvec(A,B,B)

Simple Example in UPC: Trace

```
1: void matvec(shared [N] int A[N][N],
               shared int B[N],
               shared int C[N]) {
2:    upc_forall(int i = 0; i < N; i++; &C[i]) {
3:       int sum = 0;
4:       for(int j = 0; j < N; j++)
5:          sum += A[i][j] * B[j];
6:       sum = foo(sum);
7:       C[i] = sum;
8:    }
9:}

assert(C == foo(A*B));
```

foo is an expensive function

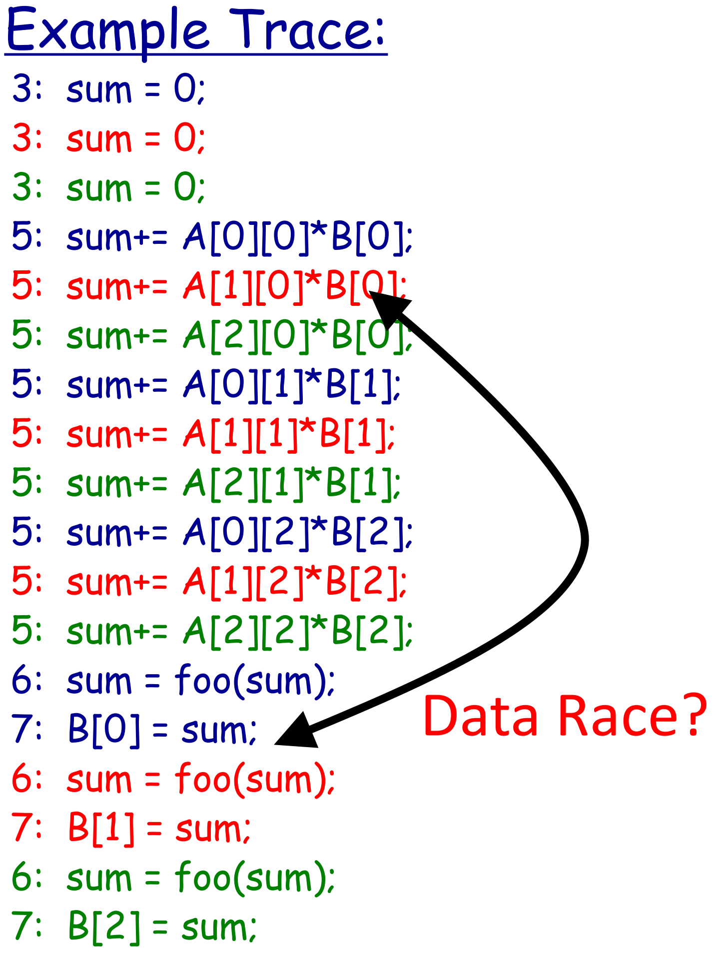
Example Trace:

```
3:  sum = 0;
3:  sum = 0;
3:  sum = 0;
5:  sum+= A[0][0]*B[0];
5:  sum+= A[1][0]*B[0];
5:  sum+= A[2][0]*B[0];
5:  sum+= A[0][1]*B[1];
5:  sum+= A[1][1]*B[1];
5:  sum+= A[2][1]*B[1];
5:  sum+= A[0][2]*B[2];
5:  sum+= A[1][2]*B[2];
5:  sum+= A[2][2]*B[2];
6:  sum = foo(sum);
7:  B[0] = sum;
6:  sum = foo(sum);
7:  B[1] = sum;
6:  sum = foo(sum);
7:  B[2] = sum;
```

Simple Example in UPC: Trace

```
1: void matvec(shared [N] int A[N][N],
                       shared int B[N],
                       shared int C[N]) {
2:    upc_forall(int i = 0; i < N; i++; &C[i]) {
3:       int sum = 0;
4:       for(int j = 0; j < N; j++)
5:          sum += A[i][j] * B[j];
6:       sum = foo(sum);
7:       C[i] = sum;
8:    }
9:}

assert(C == foo(A*B));
```

foo is an expensive function



Example Trace:

```
3:  sum = 0;
3:  sum = 0;
3:  sum = 0;
5:  sum+= A[0][0]*B[0];
5:  sum+= A[1][0]*B[0];
5:  sum+= A[2][0]*B[0];
5:  sum+= A[0][1]*B[1];
5:  sum+= A[1][1]*B[1];
5:  sum+= A[2][1]*B[1];
5:  sum+= A[0][2]*B[2];
5:  sum+= A[1][2]*B[2];
5:  sum+= A[2][2]*B[2];
6:  sum = foo(sum);
7:  B[0] = sum;
6:  sum = foo(sum);
7:  B[1] = sum;
6:  sum = foo(sum);
7:  B[2] = sum;
```

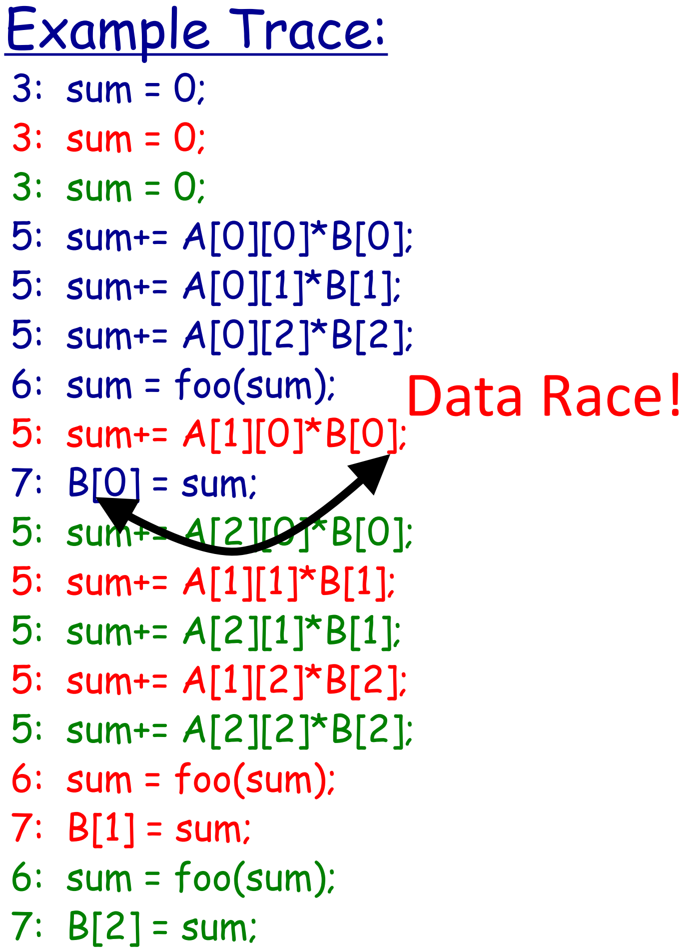
Data Race?

Simple Example in UPC: Trace

Goal 1. Nice to have a trace exhibiting the data race

```
1: void matvec(shared [N] int A[N][N],
               shared int B[N],
               shared int C[N]) {
2:    upc_forall(int i = 0; i < N; i++; &C[i]) {
3:       int sum = 0;
4:       for(int j = 0; j < N; j++)
5:          sum += A[i][j] * B[j];
6:       sum = foo(sum);
7:       C[i] = sum;
8:    }
9:}

assert(C == foo(A*B));
```

foo is an expensive function



Example Trace:

```
3:  sum = 0;
3:  sum = 0;
3:  sum = 0;
5:  sum+= A[0][0]*B[0];
5:  sum+= A[0][1]*B[1];
5:  sum+= A[0][2]*B[2];
6:  sum = foo(sum);
5:  sum+= A[1][0]*B[0];
7:  B[0] = sum;
5:  sum+= A[2][0]*B[0];
5:  sum+= A[1][1]*B[1];
5:  sum+= A[2][1]*B[1];
5:  sum+= A[1][2]*B[2];
5:  sum+= A[2][2]*B[2];
6:  sum = foo(sum);
7:  B[1] = sum;
6:  sum = foo(sum);
7:  B[2] = sum;
```

Data Race!

Simple Example in UPC: Trace

Goal 2. Nice to have a trace exhibiting the assertion failure

```
1: void matvec(shared [N] int A[N][N],
                          shared int B[N],
                          shared int C[N]) {
2:    upc_forall(int i = 0; i < N; i++; &C[i]) {
3:       int sum = 0;
4:       for(int j = 0; j < N; j++)
5:           sum += A[i][j] * B[j];
6:       sum = foo(sum);
7:       C[i] = sum;
8:    }
9:}

assert(C == foo(A*B));
```

foo is an expensive function

Example Trace:

```
3:  sum = 0;
3:  sum = 0;
3:  sum = 0;
5:  sum+= A[0][0]*B[0];
5:  sum+= A[0][1]*B[1];
5:  sum+= A[0][2]*B[2];
6:  sum = foo(sum);
7:  B[0] = sum;
5:  sum+= A[1][0]*B[0];
5:  sum+= A[2][0]*B[0];
5:  sum+= A[1][1]*B[1];
5:  sum+= A[2][1]*B[1];
5:  sum+= A[1][2]*B[2];
5:  sum+= A[2][2]*B[2];
6:  sum = foo(sum);
7:  B[1] = sum;
6:  sum = foo(sum);
7:  B[2] = sum;
```

Data Race!

Simple Example in UPC: Trace

Goal 3. Nice to have a trace
with fewer threads

```
1: void matvec(shared [N] int A[N][N],
                         shared int B[N],
                         shared int C[N]) {
2:    upc_forall(int i = 0; i < N; i++; &C[i]) {
3:       int sum = 0;
4:       for(int j = 0; j < N; j++)
5:          sum += A[i][j] * B[j];
6:       sum = foo(sum);
7:       C[i] = sum;
8:    }
9:}

assert(C == foo(A*B));
```

foo is an expensive function

Example Trace:

```
3:  sum = 0;
3:  sum = 0;
5:  sum+= A[0][0]*B[0];
5:  sum+= A[0][1]*B[1];
6:  sum = foo(sum);
7:  B[0] = sum;
5:  sum+= A[1][0]*B[0];
5:  sum+= A[1][1]*B[1];
6:  sum = foo(sum);
7:  B[1] = sum;
```

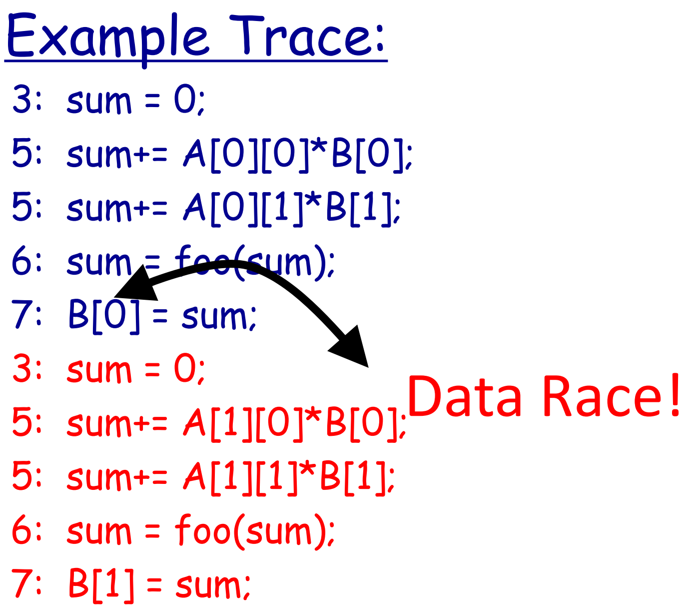
Data Race!

Simple Example in UPC: Trace

Goal 4. Nice to have a trace
with fewer context switches

```
1: void matvec(shared [N] int A[N][N],
                          shared int B[N],
                          shared int C[N]) {
2:    upc_forall(int i = 0; i < N; i++; &C[i]) {
3:       int sum = 0;
4:       for(int j = 0; j < N; j++)
5:          sum += A[i][j] * B[j];
6:       sum = foo(sum);
7:       C[i] = sum;
8:    }
9:}

assert(C == foo(A*B));
```

foo is an expensive function



Example Trace:

```
3:  sum = 0;
5:  sum+= A[0][0]*B[0];
5:  sum+= A[0][1]*B[1];
6:  sum = foo(sum);
7:  B[0] = sum;
3:  sum = 0;
5:  sum+= A[1][0]*B[0];
5:  sum+= A[1][1]*B[1];
6:  sum = foo(sum);
7:  B[1] = sum;
```

Data Race!

Simple Example in UPC: Assertion

Goal 5. Nice if the assertion is simpler

```
1: void matvec(shared [N] int A[N][N],
                           shared int B[N],
                           shared int C[N]) {
2:    upc_forall(int i = 0; i < N; i++; &C[i]) {
3:        int sum = 0;
4:        for(int j = 0; j < N; j++)
5:            sum += A[i][j] * B[j];
6:        sum = foo(sum);
7:        C[i] = sum;
8:    }
9:}
```

assert(C == foo(A*B));

foo is an expensive function

Example Trace:

```
3:  sum = 0;
5:  sum+= A[0][0]*B[0];
5:  sum+= A[0][1]*B[1];
6:  sum = foo(sum);
7:  B[0] = sum;
3:  sum = 0;
5:  sum+= A[1][0]*B[0];
5:  sum+= A[1][1]*B[1];
6:  sum = foo(sum);
7:  B[1] = sum;
```

Simple Example in UPC: Assertion

Goal 5. Nice if the assertion is simpler

```
1: void matvec(shared [N] int A[N][N],
                          shared int B[N],
                          shared int C[N]) {
2:    upc_forall(int i = 0; i < N; i++; &C[i]) {
3:        int sum = 0;
4:        for(int j = 0; j < N; j++)
5:            sum += A[i][j] * B[j];
6:        sum = foo(sum);
7:        C[i] = sum;
8:    }
9:}
```

assert(C == C’);

foo is an expensive function

Example Trace:

```
3:  sum = 0;
5:  sum+= A[0][0]*B[0];
5:  sum+= A[0][1]*B[1];
6:  sum = foo(sum);
7:  B[0] = sum;
3:  sum = 0;
5:  sum+= A[1][0]*B[0];
5:  sum+= A[1][1]*B[1];
6:  sum = foo(sum);
7:  B[1] = sum;
```

Goals: Summary

- Would be nice to have a trace
  - showing a data race (or some other concurrency bug)
  - showing an assertion violation due to a data race
  - with fewer threads
  - with fewer context switches
  - Simpler assertions [see our work on specification]

Goals: Summary

- Would be nice to have a trace
  - showing a data race (or some other concurrency bug)
  - showing an assertion violation due to a data race
  - with fewer threads
  - with fewer context switches
  - Simpler assertions [see our work on specification]

Active Testing: Phase I

```
1: void matvec(shared [N] int A[N][N],
                            shared int B[N],
                            shared int C[N]) {
2:    upc_forall(int i = 0; i < N; i++; &C[i]) {
3:       int sum = 0;
4:       for(int j = 0; j < N; j++)
5:           sum += A[i][j] * B[j];
6:       sum = foo(sum);
7:       C[i] = sum;
8:    }
9:}

assert(C == foo(A*B));
```

foo is an expensive function

Example Trace:

```
3:  sum = 0;
3:  sum = 0;
5:  sum+= A[0][0]*B[0];
5:  sum+= A[1][0]*B[0];
5:  sum+= A[0][1]*B[1];
5:  sum+= A[1][1]*B[1];
6:  sum = foo(sum);
7:  B[0] = sum;
6:  sum = foo(sum);
7:  B[1] = sum;
```

Active Testing: Phase I

1. Insert Instrumentations at compile time

```
                    shared int B[N],
                    shared int C[N]) {
     upc_forall(int i = 0; i < N; i++; &C[i]) {
        int sum = 0;
        for(int j = 0; j < N; j++)
            sum += A[i][j] * B[j];
        sum = foo(sum);
        C[i] = sum;
     }
}

assert(C == foo(A*B));
```

foo is an expensive function

Active Testing: Phase I

1. Insert Instrumentations at compile time

2. Run instrumented program normally -> Trace

```
                shared int B[N],
                shared int C[N]) {
2:   upc_forall(int i = 0; i < N; i++; &C[i]) {
6:      sum = foo(sum);
7:      C[i] = sum;
8:   }
9:}

assert(C == foo(A*B));
```

foo is an expensive function

Example Trace:

```
3:  sum = 0;
3:  sum = 0;
5:  sum+= A[0][0]*B[0];
5:  sum+= A[1][0]*B[0];
5:  sum+= A[0][1]*B[1];
5:  sum+= A[1][1]*B[1];
6:  sum = foo(sum);
6:  sum = foo(sum);
7:  B[0] = sum;
7:  B[1] = sum;
```

Active Testing: Phase I

1. Insert Instrumentations at compile time

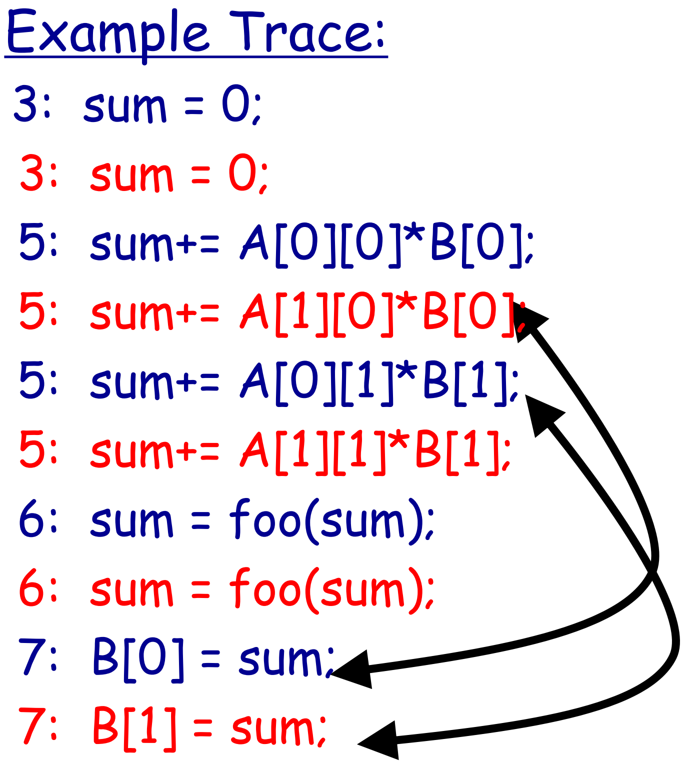
```
                 shared int B[N];
                 shared int C[N]) {
2:  upc_forall(int i = 0; i < N; i++; &C[i]) {
```

2. Run instrumented program normally -> Trace

```
6:      sum = foo(sum);
7:      C[i] = sum;
8:   }
```

3. Find potential data races

foo is an expensive function



Example Trace:

3: sum = 0;
3: sum = 0;
5: sum+= A[0][0]*B[0];
5: sum+= A[1][0]*B[0];
5: sum+= A[0][1]*B[1];
5: sum+= A[1][1]*B[1];
6: sum = foo(sum);
6: sum = foo(sum);
7: B[0] = sum;
7: B[1] = sum;

Active Testing: Phase I

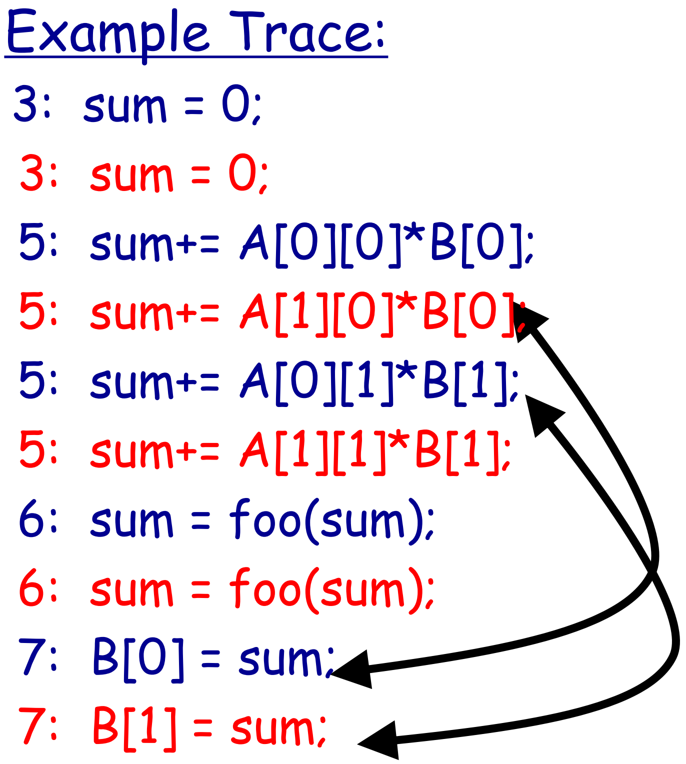
1. Insert Instrumentations at compile time

2. Run instrumented program normally -> Trace

3. Potential race between statements 5 and 7

```
                    shared int B[N],
                    shared int C[N]) {
2:   upc_forall(int i = 0; i < N; i++; &C[i]) {
6:       sum = foo(sum);
7:       C[i] = sum;
8:   }
```

foo is an expensive function



Example Trace:

3: sum = 0;
3: sum = 0;
5: sum+= A[0][0]*B[0];
5: sum+= A[1][0]*B[0];
5: sum+= A[0][1]*B[1];
5: sum+= A[1][1]*B[1];
6: sum = foo(sum);
6: sum = foo(sum);
7: B[0] = sum;
7: B[1] = sum;

Active Tes

Goals. 1. Confirm races
2. Check Assertion Failure

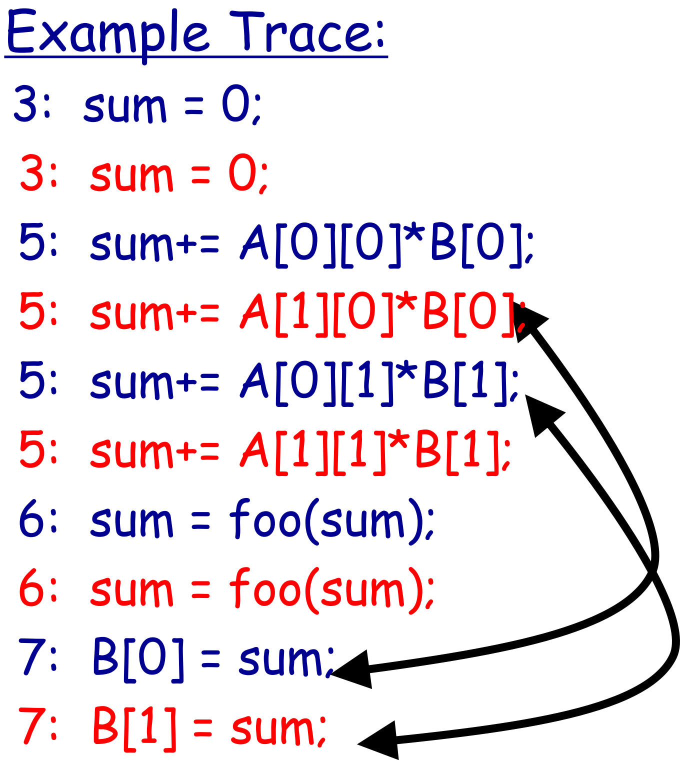
1. Insert Instrumentations at compile time

```
                   shared int B[N],
                   shared int C[N]) {
2:   upc_forall(int i = 0; i < N; i++; &C[i]) {
6:       sum = foo(sum);
7:       C[i] = sum;
8:   }
```

2. Run instrumented program normally -> Trace

3. Potential race between statements 5 and 7

foo is an expensive function



Example Trace:

3: sum = 0;
3: sum = 0;
5: sum+= A[0][0]*B[0];
5: sum+= A[1][0]*B[0];
5: sum+= A[0][1]*B[1];
5: sum+= A[1][1]*B[1];
6: sum = foo(sum);
6: sum = foo(sum);
7: B[0] = sum;
7: B[1] = sum;

Active Testing: Phase II

Control Scheduler using knowledge that (5,7) could race

```
1: void matvec(shared [N] int A[N][N],
                         shared int B[N],
                         shared int C[N]) {
2:    upc_forall(int i = 0; i < N; i++; &C[i]) {
3:       int sum = 0;
4:       for(int j = 0; j < N; j++)
5:          sum += A[i][j] * B[j];
6:       sum = foo(sum);
7:       C[i] = sum;
8:    }
9:}

assert(C == foo(A*B));
```

foo is an expensive function

<u>Generate Trace:</u>

```
3:  sum = 0;
3:  sum = 0;
5:  sum+= A[0][0]*B[0];
5:  sum+= A[0][1]*B[1];
6:  sum = foo(sum);
7:  B[0] = sum;
5:  sum+= A[1][0]*B[0];
5:  sum+= A[1][1]*B[1];
6:  sum = foo(sum);
7:  B[1] = sum;
```

Data Race!

Goal. Generate this execution

Active Testing: Phase II

Control Scheduler using knowledge that (5,7) could race

Generate Trace:

```
1: void matvec(shared [N] int A[N][N],
                        shared int B[N],
                        shared int C[N]) {
2:    upc_forall(int i = 0; i < N; i++; &C[i]) {
3:       int sum = 0;
4:       for(int j = 0; j < N; j++)
5:           sum += A[i][j] * B[j];
6:       sum = foo(sum);
7:       C[i] = sum;
8:    }
9:}

assert(C == foo(A*B));
```

foo is an expensive function

Active Testing: Phase II

Control Scheduler using knowledge that (5,7) could race

<u>Generate Trace:</u>

3: sum = 0;

```
1: void matvec(shared [N] int A[N][N],
                           shared int B[N],
                           shared int C[N]) {
2:    upc_forall(int i = 0; i < N; i++; &C[i]) {
3:       int sum = 0;
4:       for(int j = 0; j < N; j++)
5:           sum += A[i][j] * B[j];
6:       sum = foo(sum);
7:       C[i] = sum;
8:    }
9:}

assert(C == foo(A*B));
```

foo is an expensive function

Active Testing: Phase II

Control Scheduler using knowledge that (5,7) could race

```
1: void matvec(shared [N] int A[N][N],
               shared int B[N],
               shared int C[N]) {
2:   upc_forall(int i = 0; i < N; i++; &C[i]) {
3:     int sum = 0;
4:     for(int j = 0; j < N; j++)
5:       sum += A[i][j] * B[j];
6:     sum = foo(sum);
7:     C[i] = sum;
8:   }
9:}

assert(C == foo(A*B));
```

foo is an expensive function

Generate Trace:

3: sum = 0;

Active Testing: Phase II

Control Scheduler using knowledge that (5,7) could race

```
1: void matvec(shared [N] int A[N][N],
                          shared int B[N],
                          shared int C[N]) {
2:    upc_forall(int i = 0; i < N; i++; &C[i]) {
3:        int sum = 0;
4:        for(int j = 0; j < N; j++)
5:            sum += A[i][j] * B[j];
6:        sum = foo(sum);
7:        C[i] = sum;
8:    }
9:}

assert(C == foo(A*B));
```

foo is an expensive function

Generate Trace:

3: sum = 0;

3: sum = 0;

Active Testing: Phase II

Control Scheduler using knowledge that (5,7) could race

```
1: void matvec(shared [N] int A[N][N],
               shared int B[N],
               shared int C[N]) {
2:    upc_forall(int i = 0; i < N; i++; &C[i]) {
3:       int sum = 0;
4:       for(int j = 0; j < N; j++)
5:          sum += A[i][j] * B[j];
6:       sum = foo(sum);
7:       C[i] = sum;
8:    }
9:}

assert(C == foo(A*B));
```

foo is an expensive function

Generate Trace:

```
3:  sum = 0;
3:  sum = 0;
```

Active Testing: Phase II

Control Scheduler using knowledge that (5,7) could race

```
1: void matvec(shared [N] int A[N][N],
                      shared int B[N],
                      shared int C[N]) {
2:    upc_forall(int i = 0; i < N; i++; &C[i]) {
3:       int sum = 0;
4:       for(int j = 0; j < N; j++)
5:           sum += A[i][j] * B[j];
6:       sum = foo(sum);
7:       C[i] = sum;
8:    }
9:}

assert(C == foo(A*B));
```

foo is an expensive function

Generate Trace:

```
3:  sum = 0;
3:  sum = 0;
5:  sum+= A[0][0]*B[0];
```

Active Testing: Phase II

Control Scheduler using knowledge that (5,7) could race

```
1: void matvec(shared [N] int A[N][N],
                          shared int B[N],
                          shared int C[N]) {
2:    upc_forall(int i = 0; i < N; i++; &C[i]) {
3:       int sum = 0;
4:       for(int j = 0; j < N; j++)
5:          sum += A[i][j] * B[j];
6:       sum = foo(sum);
7:       C[i] = sum;
8:    }
9:}

assert(C == foo(A*B));
```

foo is an expensive function

Generate Trace:

3: sum = 0;

3: sum = 0;

5: sum+= A[0][0]*B[0];

Do not postpone
if there is a deadlock

Postponed = {5: sum+= A[0][0]*B[0];}

Active Testing: Phase II

Control Scheduler using knowledge that (5,7) could race

```
1: void matvec(shared [N] int A[N][N],
                              shared int B[N],
                              shared int C[N]) {
2:    upc_forall(int i = 0; i < N; i++; &C[i]) {
3:       int sum = 0;
4:       for(int j = 0; j < N; j++)
5:           sum += A[i][j] * B[j];
6:       sum = foo(sum);
7:       C[i] = sum;
8:    }
9:}

assert(C == foo(A*B));
```

foo is an expensive function

Generate Trace:

3: sum = 0;
3: sum = 0;
5: sum+= A[0][0]*B[0];
5: sum+= A[0][1]*B[1];

Do not postpone
if there is a deadlock

Postponed = {5: sum+= A[0][0]*B[0];}

Active Testing: Phase II

Control Scheduler using knowledge that (5,7) could race

```
1: void matvec(shared [N] int A[N][N],
                          shared int B[N],
                          shared int C[N]) {
2:    upc_forall(int i = 0; i < N; i++; &C[i]) {
3:       int sum = 0;
4:       for(int j = 0; j < N; j++)
5:          sum += A[i][j] * B[j];
6:       sum = foo(sum);
7:       C[i] = sum;
8:    }
9:}

assert(C == foo(A*B));
```

foo is an expensive function

Generate Trace:

3: sum = 0;
3: sum = 0;
5: sum+= A[0][0]*B[0];
5: sum+= A[0][1]*B[1];
6: sum = foo(sum);
7: B[0] = sum;

Postponed = {5: sum+= A[0][0]*B[0];}

Active Testing: Phase II

Control Scheduler using knowledge that (5,7) could race

```
1: void matvec(shared [N] int A[N][N],
                          shared int B[N],
                          shared int C[N]) {
2:    upc_forall(int i = 0; i < N; i++; &C[i]) {
3:       int sum = 0;
4:       for(int j = 0; j < N; j++)
5:          sum += A[i][j] * B[j];
6:       sum = foo(sum);
7:       C[i] = sum;
8:    }
9:}

assert(C == foo(A*B));
```

foo is an expensive function

Generate Trace:

```
3:  sum = 0;
3:  sum = 0;
5:  sum+= A[0][0]*B[0];
5:  sum+= A[0][1]*B[1];
6:  sum = foo(sum);
7:  B[0] = sum;
```

Postponed = {5: sum+= A[0][0]*B[0];}

Active Testing: Phase II

Control Scheduler using knowledge that (5,7) could race

```
1: void matvec(shared [N] int A[N][N],
                              shared int B[N],
                              shared int C[N]) {
2:    upc_forall(int i = 0; i < N; i++; &C[i]) {
3:       int sum = 0;
4:       for(int j = 0; j < N; j++)
5:           sum += A[i][j] * B[j];
6:       sum = foo(sum);
7:       C[i] = sum;
8:    }
9:}

assert(C == foo(A*B));
```

foo is an expensive function

Generate Trace:

```
3:  sum = 0;
3:  sum = 0;
5:  sum+= A[0][0]*B[0];
5:  sum+= A[0][1]*B[1];
6:  sum = foo(sum);
7:  B[0] = sum;
```

Race? yes

Postponed = {5: sum+= A[0][0]*B[0];}

Active Testing: Phase II

Control Scheduler using knowledge that (5,7) could race

```
1: void matvec(shared [N] int A[N][N],
                      shared int B[N],
                      shared int C[N]) {
2:    upc_forall(int i = 0; i < N; i++; &C[i]) {
3:       int sum = 0;
4:       for(int j = 0; j < N; j++)
5:          sum += A[i][j] * B[j];
6:       sum = foo(sum);
7:       C[i] = sum;
8:    }
9:}

assert(C == foo(A*B));
```

foo is an expensive function

Generate Trace:

3: sum = 0;

3: sum = 0;

5: sum+= A[0][0]*B[0];

5: sum+= A[0][1]*B[1];

6: sum = foo(sum);

5: sum+= A[0][0]*B[0]; 7: B[0] = sum;

Postponed = { }

Active Testing: Phase II

Control Scheduler using knowledge that (5,7) could race

```
1: void matvec(shared [N] int A[N][N],
                          shared int B[N],
                          shared int C[N]) {
2:    upc_forall(int i = 0;          C[i]) {
3:       int sum = 0;
4:       for(int j = 0
5:          sum += A[
6:       sum = foo(sum);
7:       C[i] = sum;
8:    }
9:}

assert(C == foo(A*B));
```

foo is an expensive function

Racing Statements Temporally Adjacent

Generate Trace:

```
3:  sum = 0;
3:  sum = 0;
5:  sum+= A[0][0]*B[0];
5:  sum+= A[0][1]*B[1];
6:  sum = foo(sum);
7:  B[0] = sum;
5:  sum+= A[1][0]*B[0];
```

Achieved Goal 1.
Confirmed race

Active Testing: Phase II

Control Scheduler using knowledge that (5,7) could race

```
1: void matvec(shared [N] int A[N][N],
                          shared int B[N],
                          shared int C[N]) {
2:    upc_forall(int i = 0; i < N; i++; &C[i]) {
3:       int sum = 0;
4:       for(int j = 0; j < N; j++)
5:           sum += A[i][j] * B[j];
6:       sum = foo(sum);
7:       C[i] = sum;
8:    }
9:}

assert(C == foo(A*B));
```

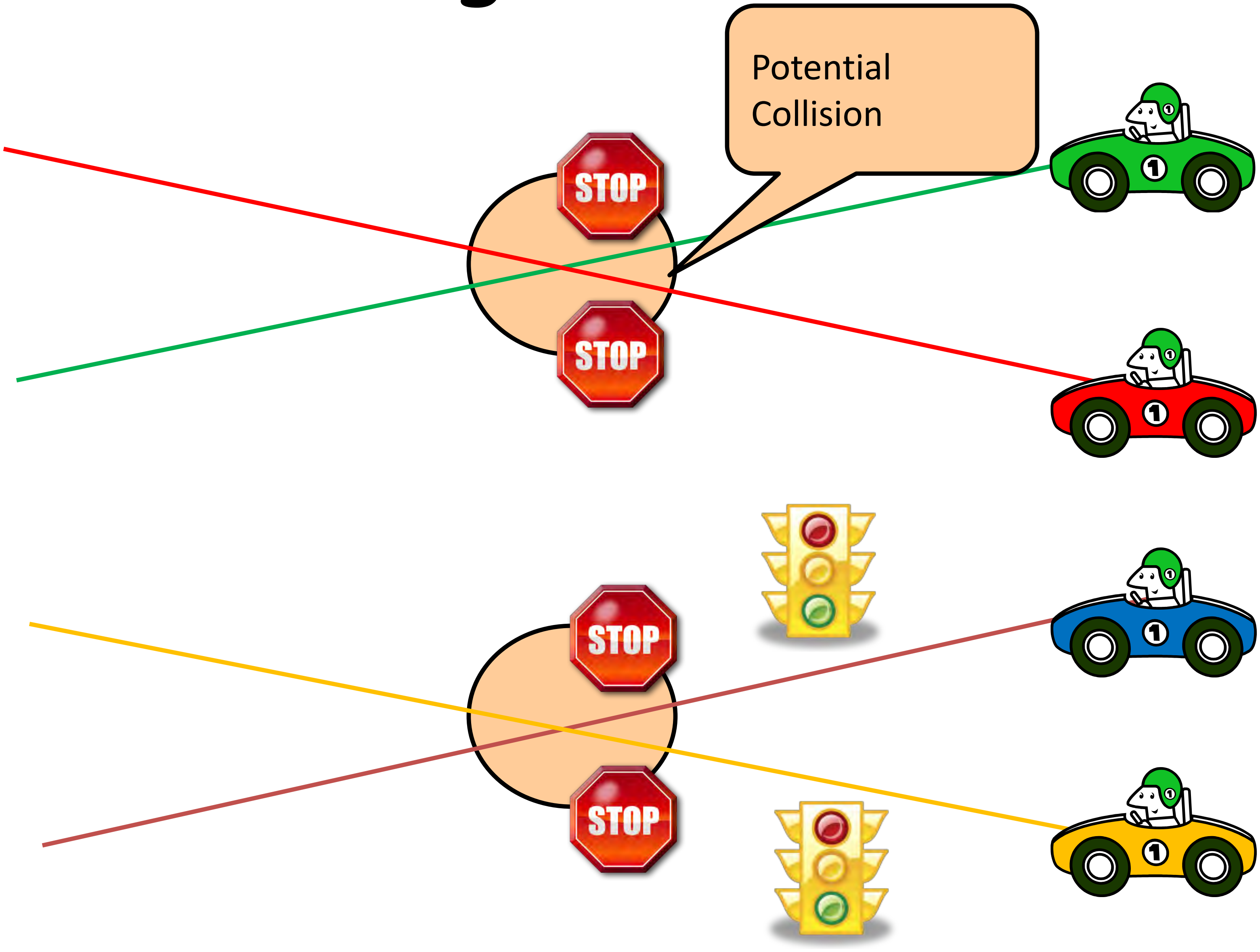
foo is an expensive function

Generate Trace:

```
3:  sum = 0;
3:  sum = 0;
5:  sum+= A[0][0]*B[0];
5:  sum+= A[0][1]*B[1];
6:  sum = foo(sum);
7:  B[0] = sum;
5:  sum+= A[1][0]*B[0];
5:  sum+= A[1][1]*B[1];
6:  sum = foo(sum);
7:  B[1] = sum;
```

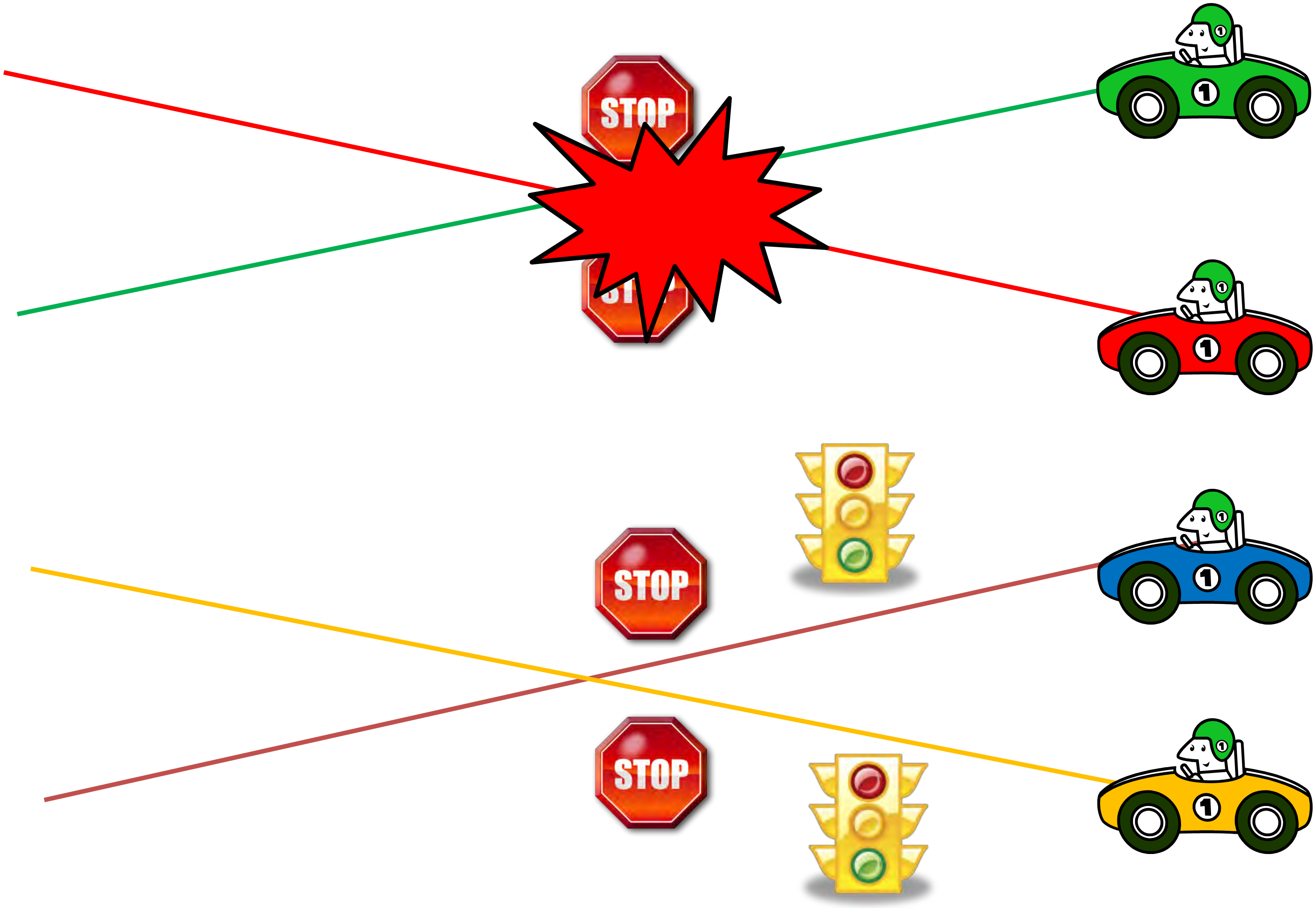
Achieved Goal 2.
Assertion Failure

Active Testing Cartoon: Phase I



Potential Collision

Active Testing Cartoon: Phase II



Active Testing:
Predict and Confirm Potential Bugs

- Phase I: Predict potential bug patterns:
  - Data races: Eraser or lockset based [PLDI'08]
  - Atomicity violations: cycle in transactions and happens-before relation [FSE'08]
  - Deadlocks: cycle in resource acquisition graph [PLDI'09]
  - Publicly available tool for Java/Pthreads/UPC [CAV'09]
  - Memory model bugs: cycle in happens-before graph [ISSTA'11]
  - For UPC programs running on thousands of cores [SC'11]
- Phase II: Direct testing using those patterns to confirm real bugs

Active Testing Advantages

- Practical and efficient
- Finds many bugs quickly
- Finds rare bugs with high probability
- Creates an actual execution showing a bug
- Reproducible

Challenges for UPC

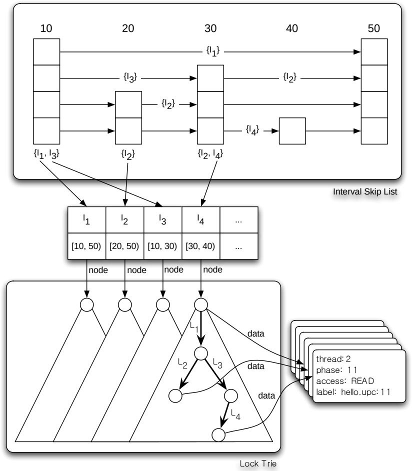
- Java and pthreads programs
  - Synchronization with locks and condition variables
  - Single node
- UPC has different programming model (SPMD)
  - Large scale
  - Bulk communication
  - Collective operations with data movement
  - Memory consistency
- Store shared memory access information locally
  - Using efficient data structures (Interval Skip List and Lock Trie)
  - Keep only the weakest accesses
- At barrier boundary, send access info to “owner” thread

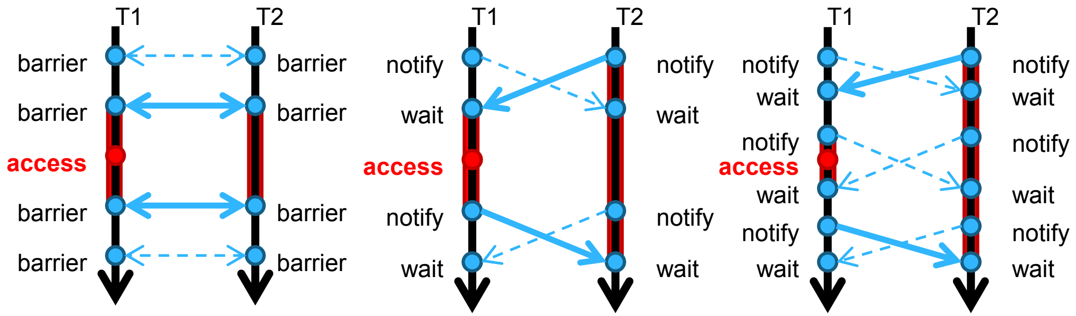
Results

Benchmark	LoC	Runtime	ThrilleRacer		ThrilleTester	
			Overhead	Pot. race	Overhead	Conf. race
guppie	227	2.094s	12%	2	1.7%	2
knapsack	191	2.099s	14.9%	2	1.8%	2
lapalce	123	2.101s	16.3%	0	-	-
mcop	358	2.183s	0.7%	0	-	-
psearch	777	2.982s	1.8%	3	3.8%	2
FT 2.3	2306	8.711s	6.1%	2	4.8%	2
CG 2.4	1939	3.812s	0.5%	0	-	-
EP 2.4	763	10.02s	0.9%	0	-	-
FT 2.4	2374	7.036s	0.1%	1	4.2%	1
IS 2.4	1449	3.073s	1.1%	0	-	-
MG 2.4	2314	4.895s	3.1%	2	1.2%	2
BT 3.3	9626	48.78s	0.5%	8	0.8%	0
LU 3.3	6311	37.05s	0.5%	0	-	-
SP 3.3	5691	59.56s	0.2%	8	3.0%	0

How Well Does it Scale?

- Maximum 8% slowdown at 8K cores
  - Franklin Cray XT4 Supercomputer at NERSC
  - Quad-core 2.x3GHz CPU and 8GB RAM per node
  - Portals interconnect
- Optimizations for scalability
  - Efficient Data Structures
  - Minimize Communication
  - Sampling with Exponential Backoff





Further Challenges!

- Targeted a simple programming paradigm
  - Threads and shared memory
- Similar techniques are available for MPI and CUDA
  - ISP, DAMPI, MARMOT, Umpire, MessageChecker
  - TASS uses symbolic execution
  - PUG for CUDA
- Analyze programs that mix different paradigms
  - OpenMP, MPI, CUDA
  - Need to correlate non-determinism across paradigms

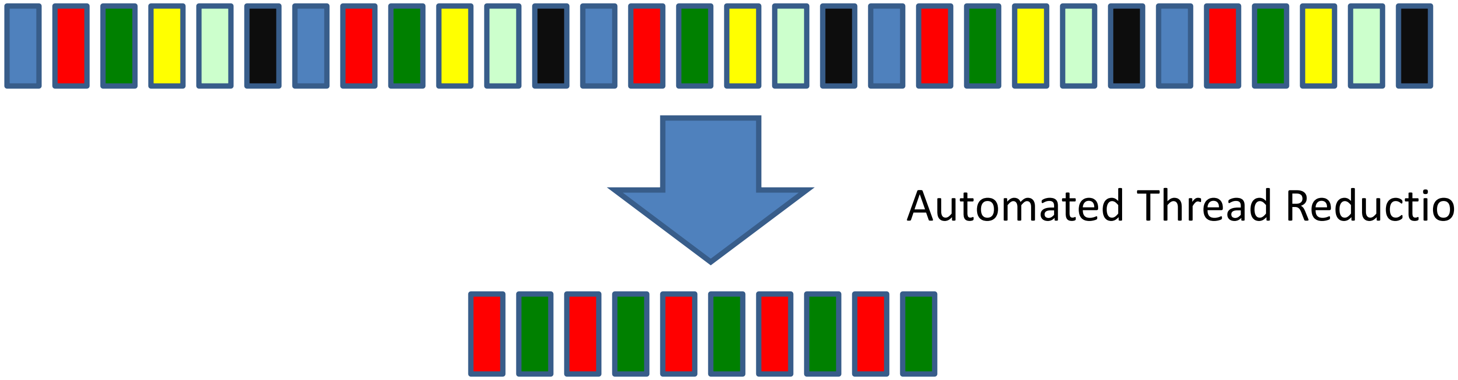
Found a Bug. Now what?

Found a Bug. Now what?

Help Programmers to debug!

Found a Bug. Now what?

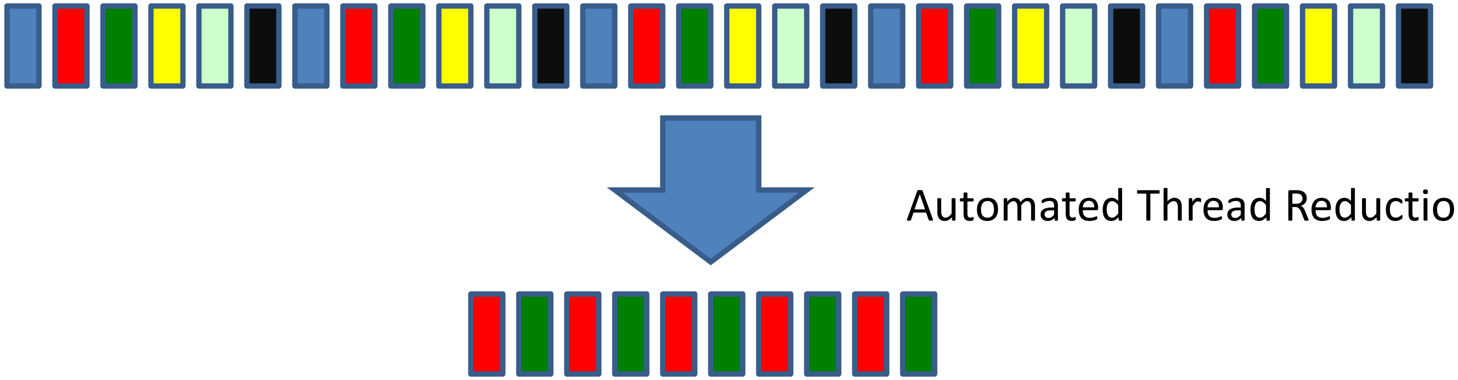
Goal 3: Show a buggy trace having fewer threads



Automated Thread Reduction

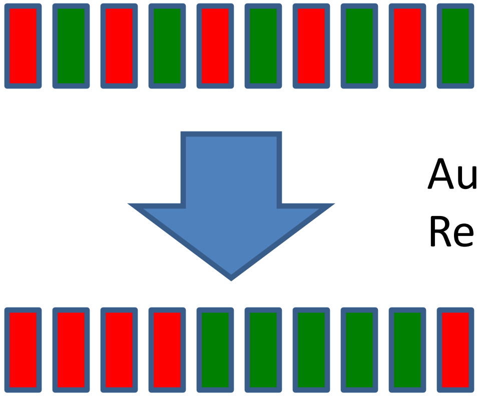
Found a Bug. Now what?

Goal 3: Show a buggy trace having fewer threads



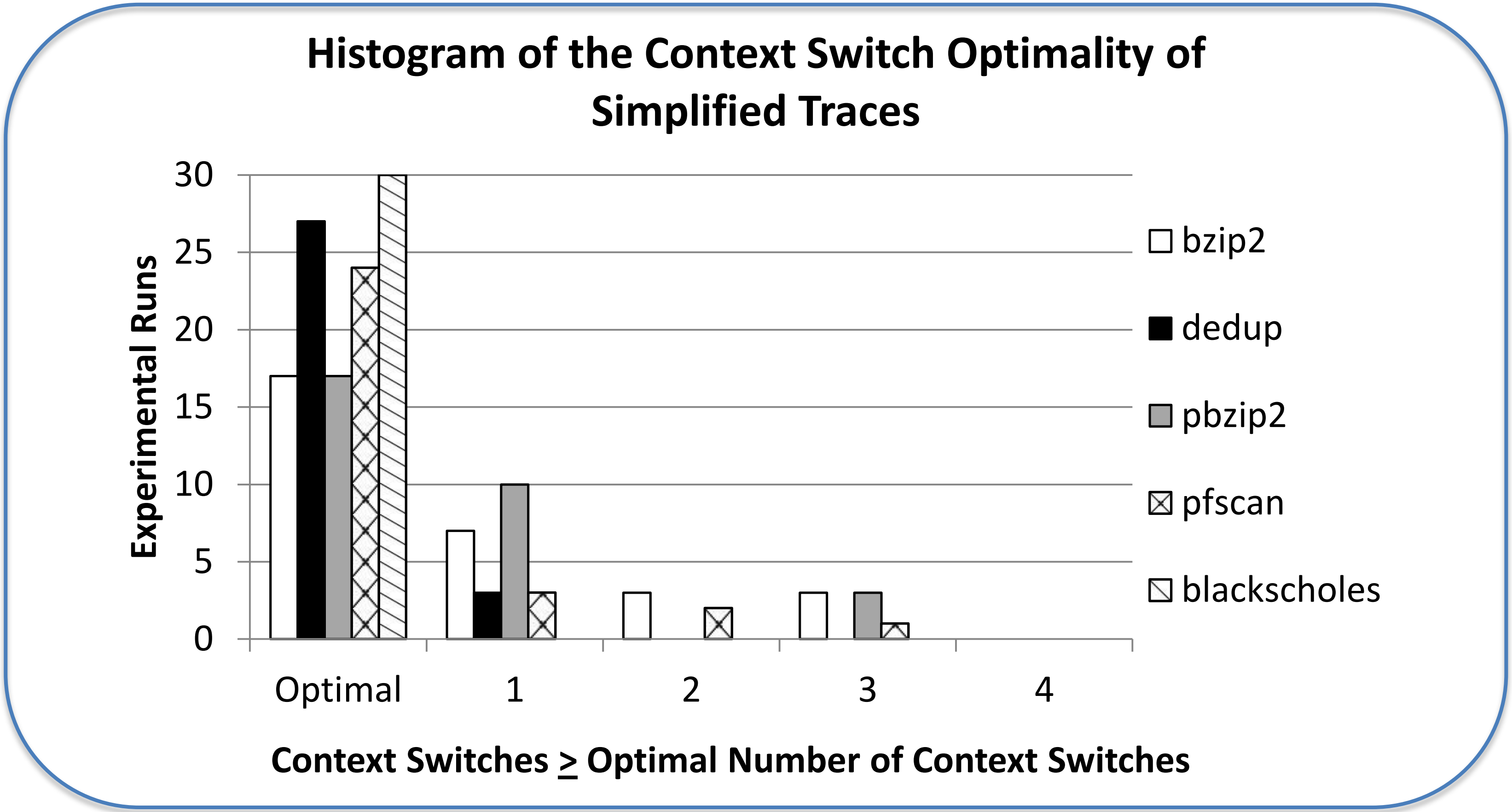
Automated Thread Reduction

Goal 4: Show a buggy trace having fewer context switches



Automated Context Switch Reduction

Our Experience with C/PThreads



- Over 90% of simplified traces were within 2 context switches of optimal.

Small model hypothesis

- Small model hypothesis for Parallel Programs
  - 1. Most bugs can be found with few threads
    - 2-3 threads
    - No need to run on thousands of nodes
  - 2. Most bugs can be found with fewer context switches [Musuvathi and Qadeer, PLDI 07]
    - Helps in sequential debugging

So many tools!

So many tools!

Can I use

printf?

Give me printf!!!

```
void matvec(shared [N] int A[N][N],
            shared int B[N],
            shared int C[N]) {
  upc_forall(int i = 0; i < N; i++; &C[i]) {
    int sum = 0;
    for(int j = 0; j < N; j++)
      printf(“B[%d] = %d\n”,j,B[j]);
      sum += A[i][j] * B[j];
    sum = foo(sum);
    printf(“C[%d] = %d\n”,i,sum);
    C[i] = sum;
  }
}

assert(C == foo(A*B));
```

Problem with printf!!!

```
void matvec(shared [N] int A[N][N],
            shared int B[N],
            shared int C[N]) {
  upc_forall(int i = 0; i < N; i++; &C[i]) {
    int sum = 0;
    for(int j = 0; j < N; j++)
      printf(“B[%d] = %d\n”,j,B[j]);
      sum += A[i][j] * B[j];
    sum = foo(sum);
    printf(“C[%d] = %d\n”,i,sum);
    C[i] = sum;
  }
}

assert(C == foo(A*B));
```

- Prints info only when a single thread reaches an interesting state

Problem with printf!!!

```
void matvec(shared [N] int A[N][N],
            shared int B[N],
            shared int C[N]) {
  upc_forall(int i = 0; i < N; i++; &C[i]) {
    int sum = 0;
    for(int j = 0; j < N; j++)
      printf("B[%d] = %d\n",j,B[j]);
      sum += A[i][j] * B[j];
    sum = foo(sum);
    printf("C[%d] = %d\n",i,sum);
    C[i] = sum;
  }
}

assert(C == foo(A*B));
```

- Prints info only when a single thread reaches an interesting state
- Need to print when a set of threads reach an interesting concurrent state

Concurrent printf for debugging

```
void matvec(shared [N] int A[N][N],
                      shared int B[N],
                      shared int C[N]) {
   upc_forall(int i = 0; i < N; i++; &C[i]) {
      int sum = 0;
      for(int j = 0; j < N; j++)
 (P1,B==C && j==i):
            printf(“B[%d]=%d\n”,j,B[j]);
         sum += A[i][j] * B[j];
      sum = foo(sum);
(P1,B==C && j==i):
         printf(“C[%d] = %d\n”,i,sum);
      C[i] = sum;
   }
}

assert(C == foo(A*B));
```

- Need to print when a set of threads reach an interesting concurrent state
- Split a printf
- Print if there is a data race or conflict

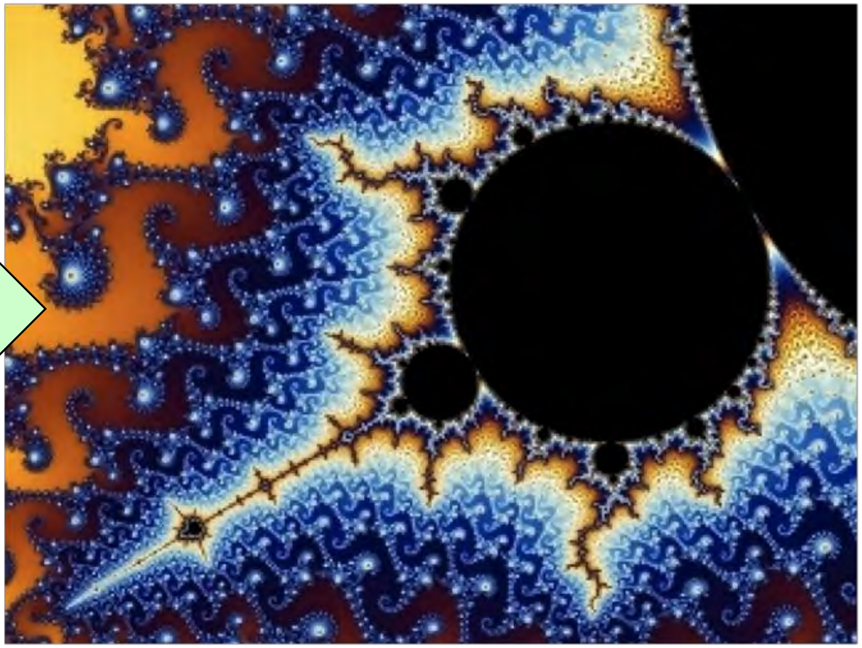
How do I Assert Correctness?

```
void matvec(shared [N] int A[N][N],
            shared int B[N],
            shared int C[N]) {
   upc_forall(int i = 0; i < N; i++; &C[i]) {
      int sum = 0;
      for(int j = 0; j < N; j++)
         sum += A[i][j] * B[j];
      sum = foo(sum);
      C[i] = sum;
   }
}

assert(C == foo(A*B));
```

Asserting Correctness?

```
x=0.7
y=0.3
…
y=5.0
```

Run with active testing



Assertions can be quite COMPLEX!

- Traditional functional correctness specs.
  - Relate program’s output to its input.
  - Generally complex and difficult to write:

$$"_{0 \text{£} x < width} \cdot \; "_{0 \text{£} y < height} \cdot$$

$$\left(\left| f_{iter}^{\,maxiter}(0) \right| < 2 \;\; \text{Ù} \;\; img[x][y] = 0 \right)$$

$$\text{Ú} \;\; \$_{1 \text{£} i < maxiter} \cdot \;\; \left| f_{iter}^{\,i}(0) \right| \; \text{³} \; 2 \;\; \text{Ù} \;\; "_{1 \text{£} j < i} \cdot \left| f_{iter}^{\,j}(0) \right| < 2$$

$$\text{Ù} \;\; img[x][y] = \mathrm{HSB}\left((i/maxiter)^{g}, 1, 1 \right)$$

where $f_{iter}(c) \;=\; c^{2} + \left(xcenter + (xoff + x)/res \right) + i\left(ycenter + (yoff - y)/res \right)$

Parallel Specifications?

- Traditional functional correctness specs.
  - Relate program's output to its input.
  - Generally complex and difficult to write:

Is there an easier way to specify just the **parallel correctness**?

where $f_{iter}(c) = c^2 + \left(xcenter + (xoff + x)/res\right)$
$+ i\left(ycenter + (yoff - y)/res\right)$

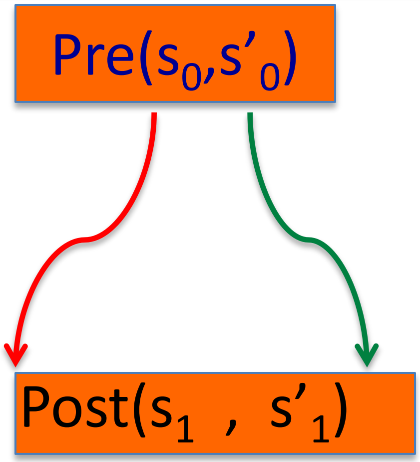
Assert that Parallelism is Correct

```
void matvec(shared [N] int A[N][N],
            shared int B[N],
            shared int C[N]) {
  upc_forall(int i = 0; i < N; i++; &C[i]) {
    int sum = 0;
    for(int j = 0; j < N; j++)
      sum += A[i][j] * B[j];
    sum = foo(sum);
    C[i] = sum;
  }
}
```

```
assert(C == C’);
```

Assertion for Parallelism Correctness

```
deterministic assume Pre(s_0,s_0'){
    P
} assert Post(s_1,s_1')
```



Pre(s_0,s'_0)

Post(s_1 , s'_1)

final state

final state

CACM’10, FSE’09 [ACM SIGSOFT Distinguished paper], ICSE’10[Best Paper Award]

Conclusion

- Active testing has been successfully used to find and reproduce real bugs in Java and C/C++ programs
    - combine static/dynamic analysis and testing
- Bugs can be detected using fewer threads
- Need concurrent extensions to printfs and breakpoints
- New mechanisms for specification

Bugs Found

- In NPB 2.3 FT,
  - Wrong lock allocation function causes real races in validation code
  - Spurious validation failure errors

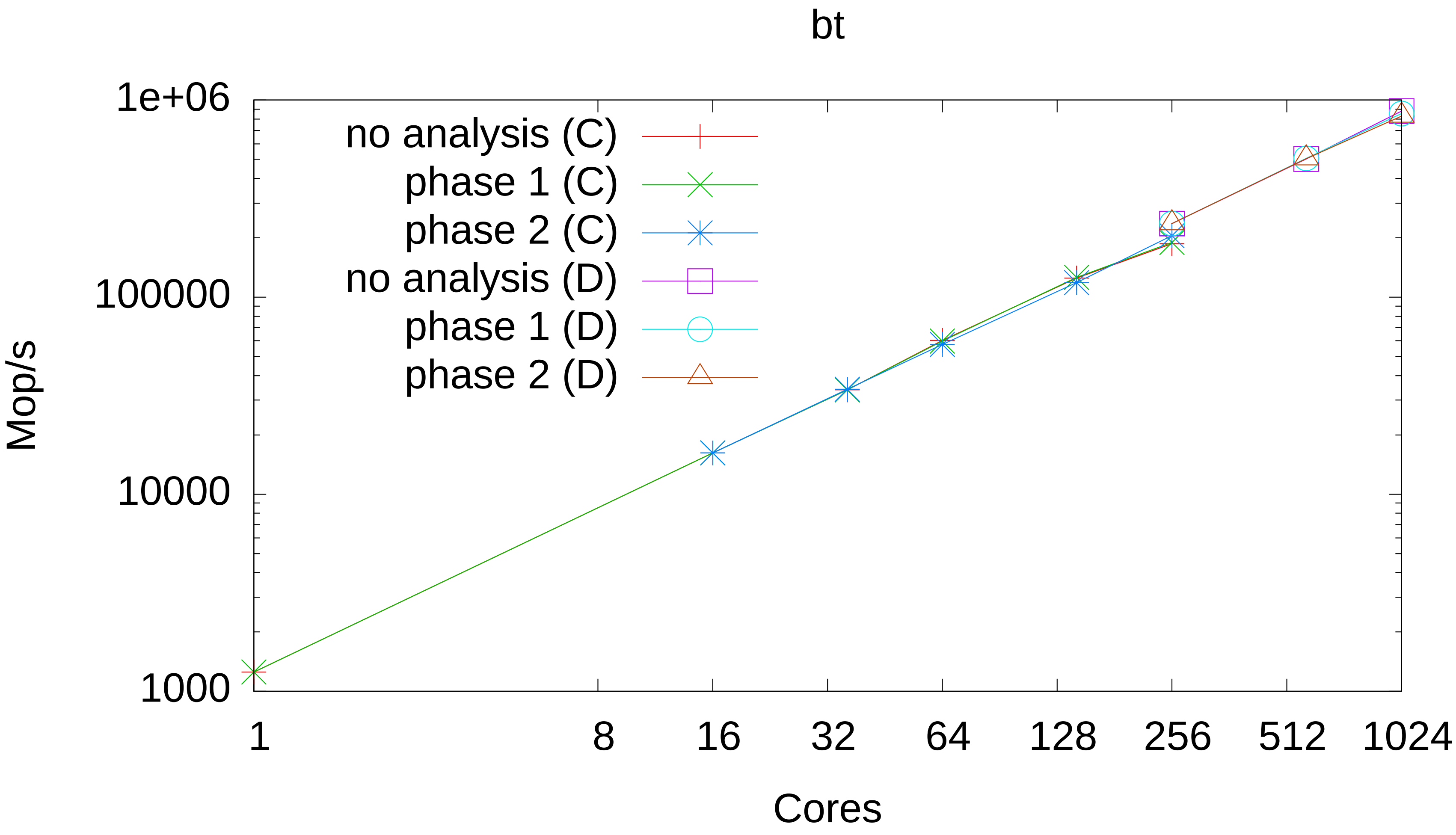
```
shared   dcomplex *dbg_sum;
static upc_lock_t *sum_write;

sum_write = upc_global_lock_alloc(); // wrong function

upc_lock (sum_write);
{
   dbg_sum->real = dbg_sum->real + chk.real;
   dbg_sum->imag = dbg_sum->imag + chk.imag;
}
upc_unlock (sum_write);
```

Bugs Found

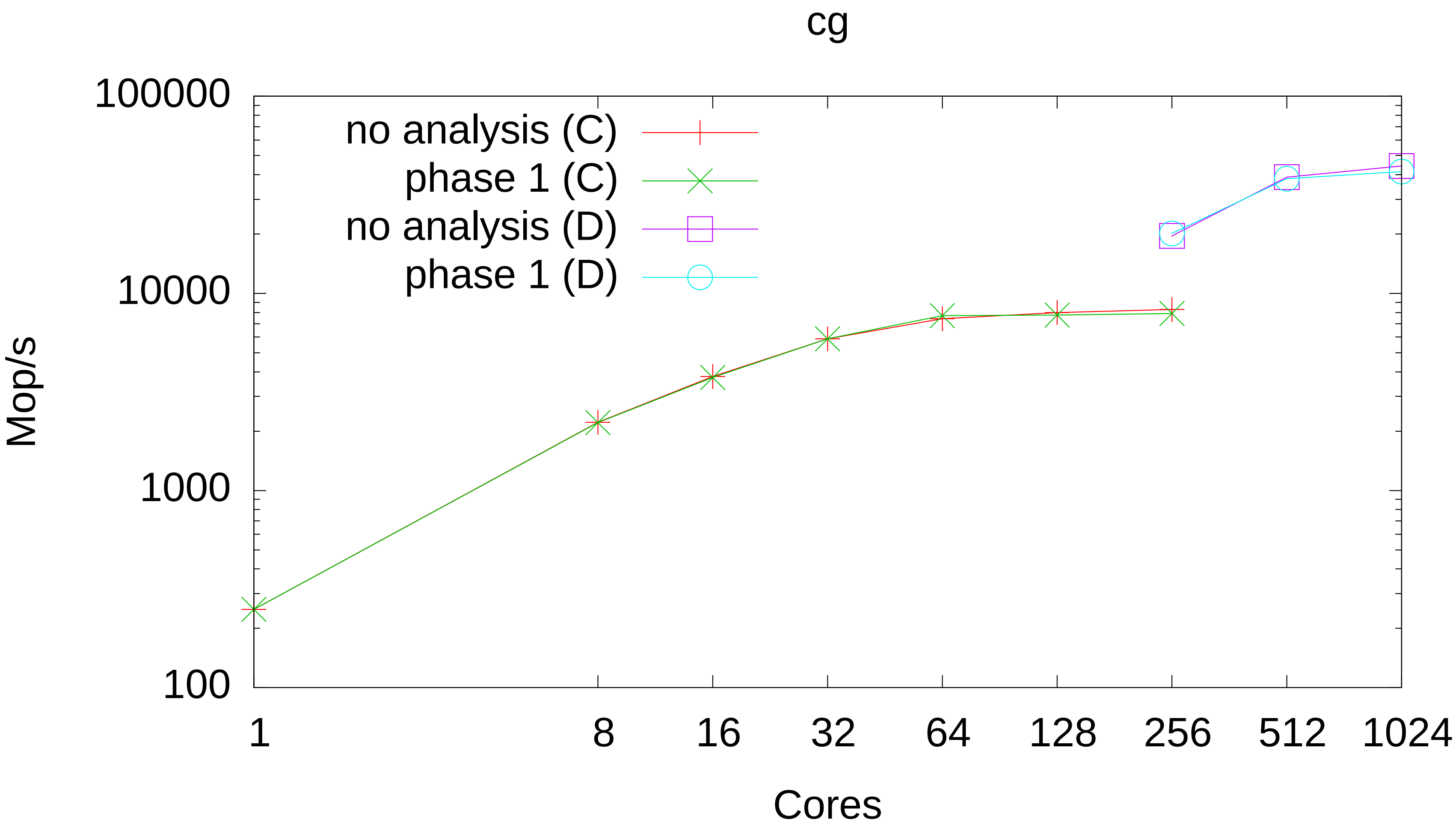
- In SPLASH2 lu,
  - Multiple initialization of vector without locks
  - Benign but performance bug

```
void InitA()
{
    …
    for (j=0; j<n; j++) {
        for (i=0; i<n; i++) {
            rhs[i] += a[i+j*n]; // executed by all threads
        }
    }
}
```

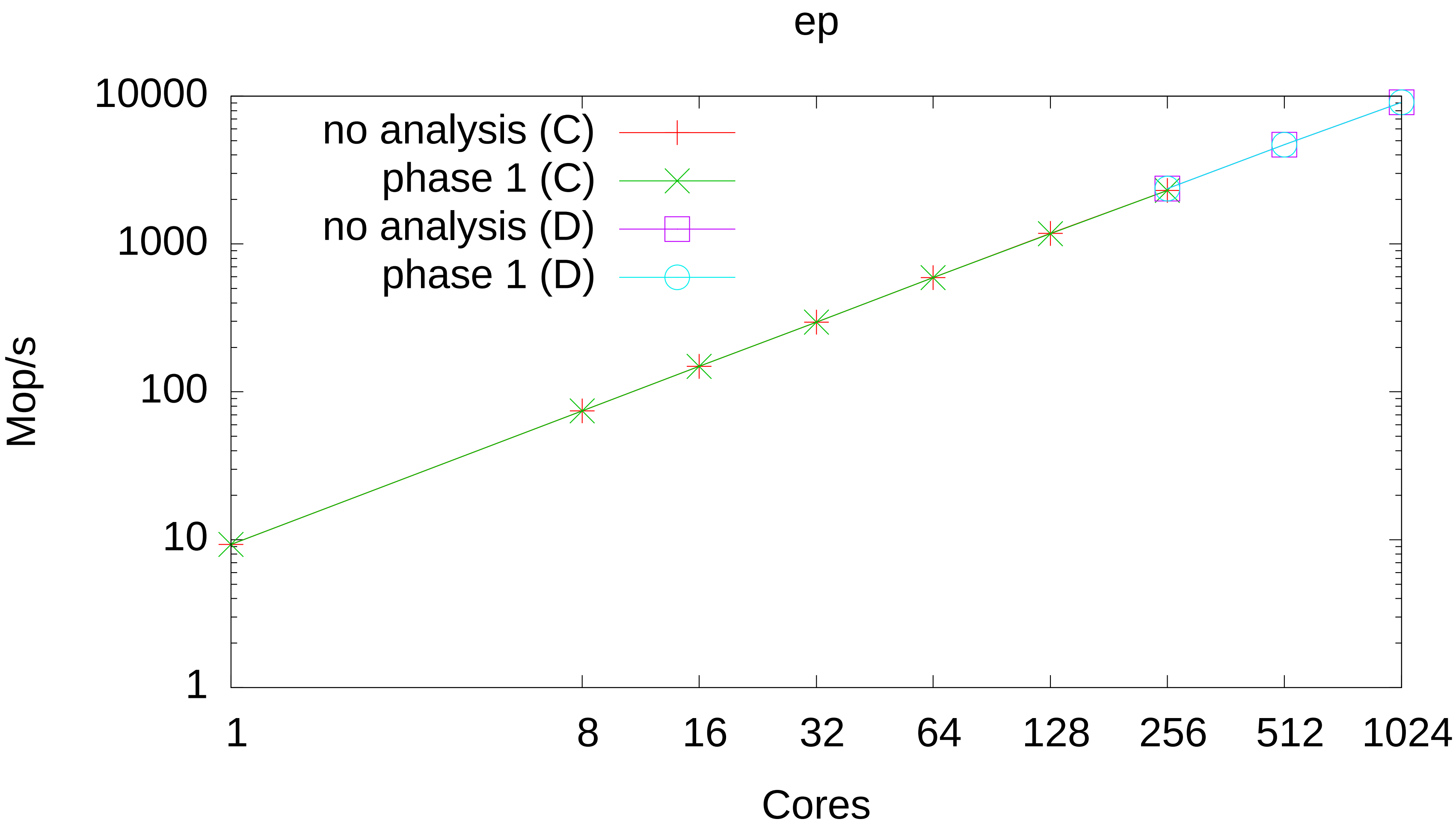
How Well Does it Scale?



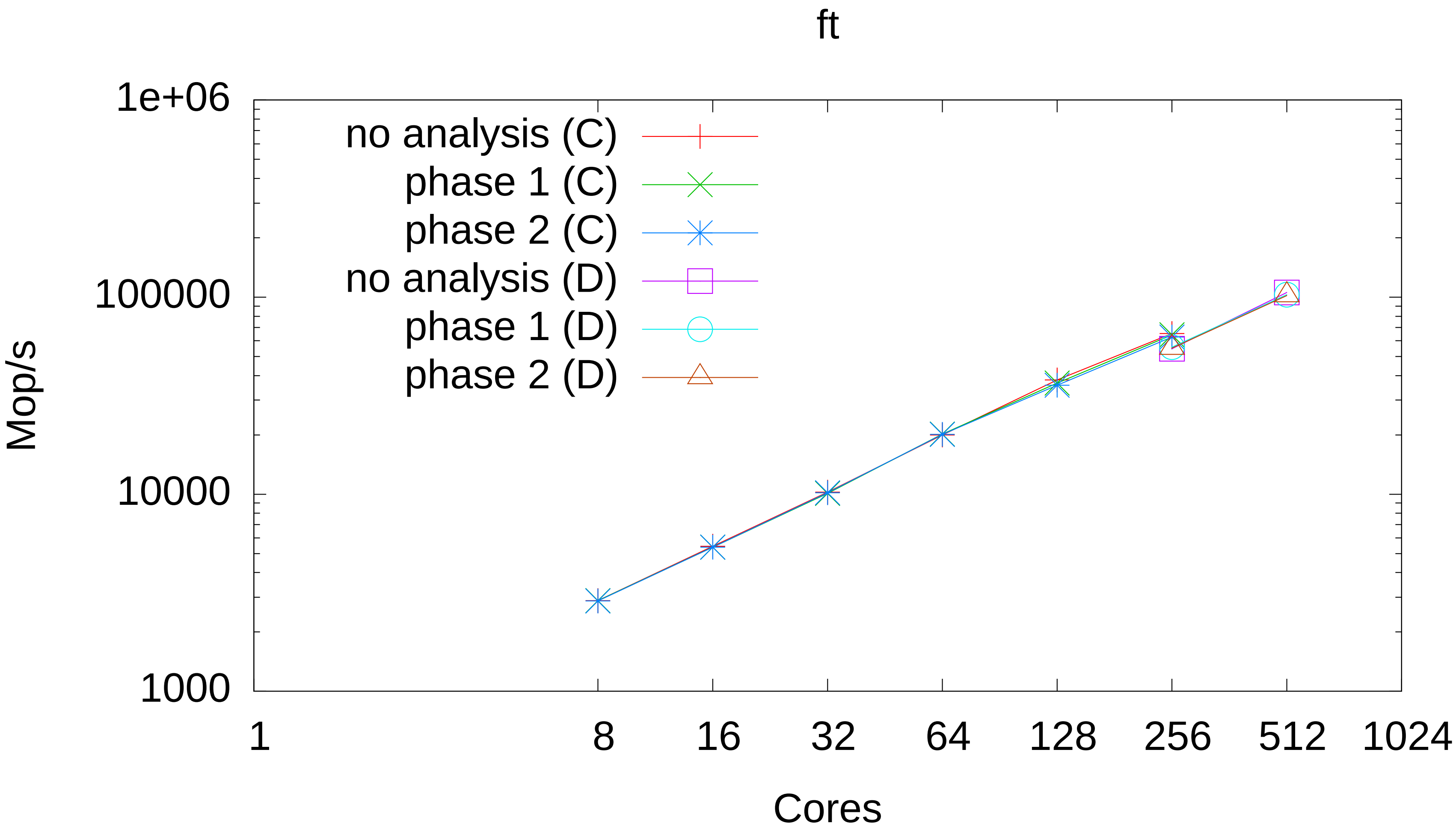
How Well Does it Scale?



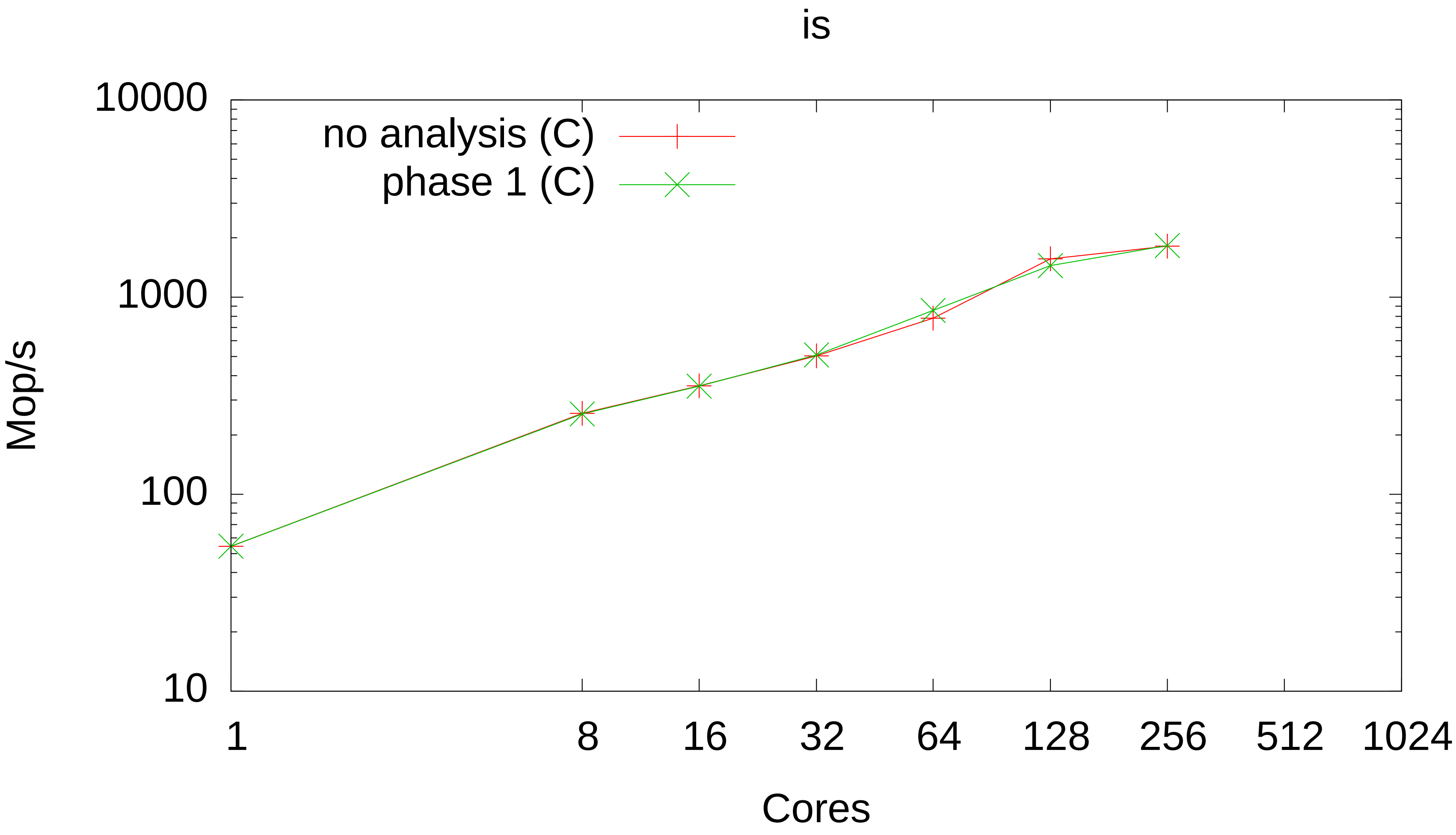
How Well Does it Scale?



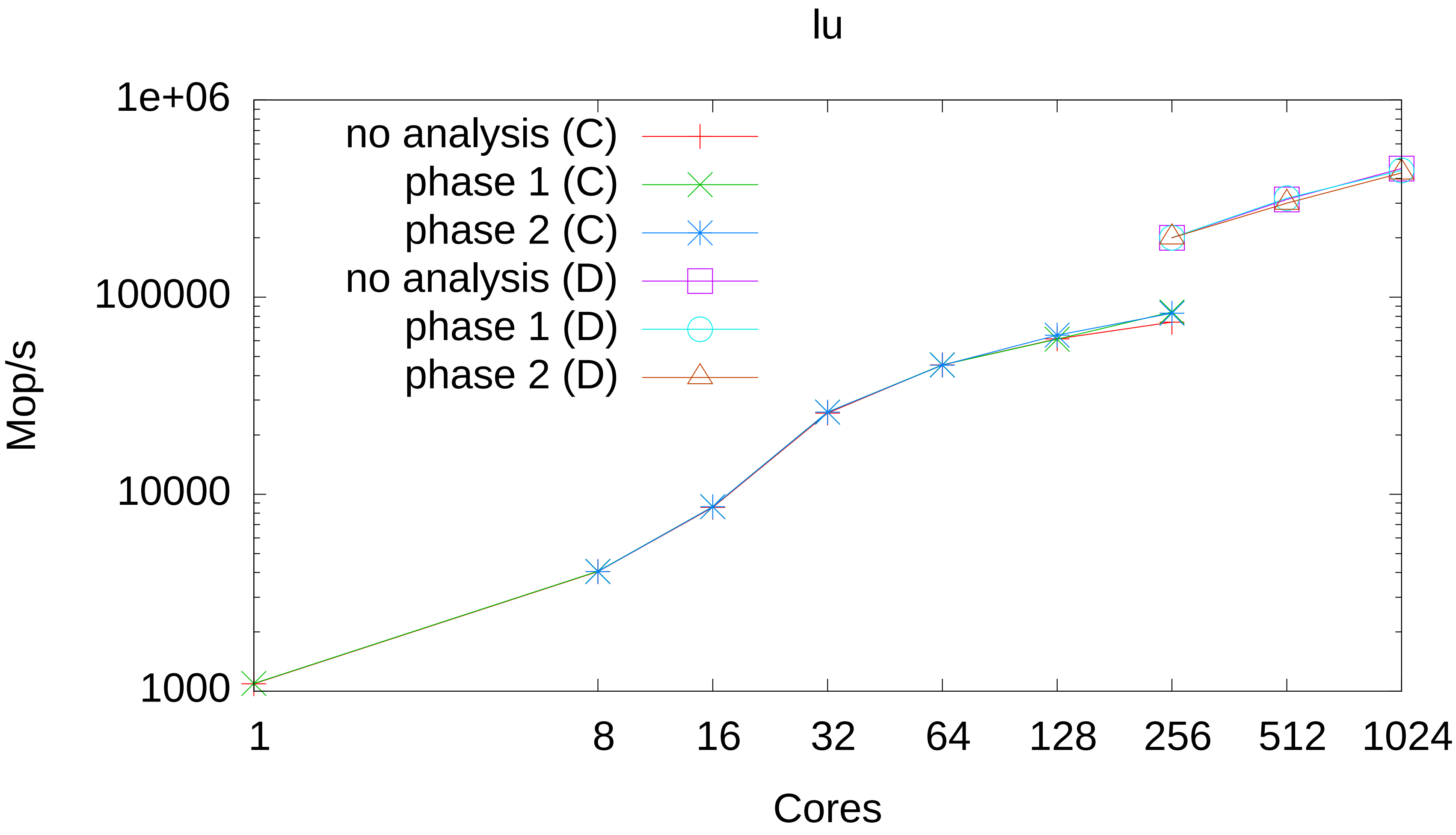
How Well Does it Scale?



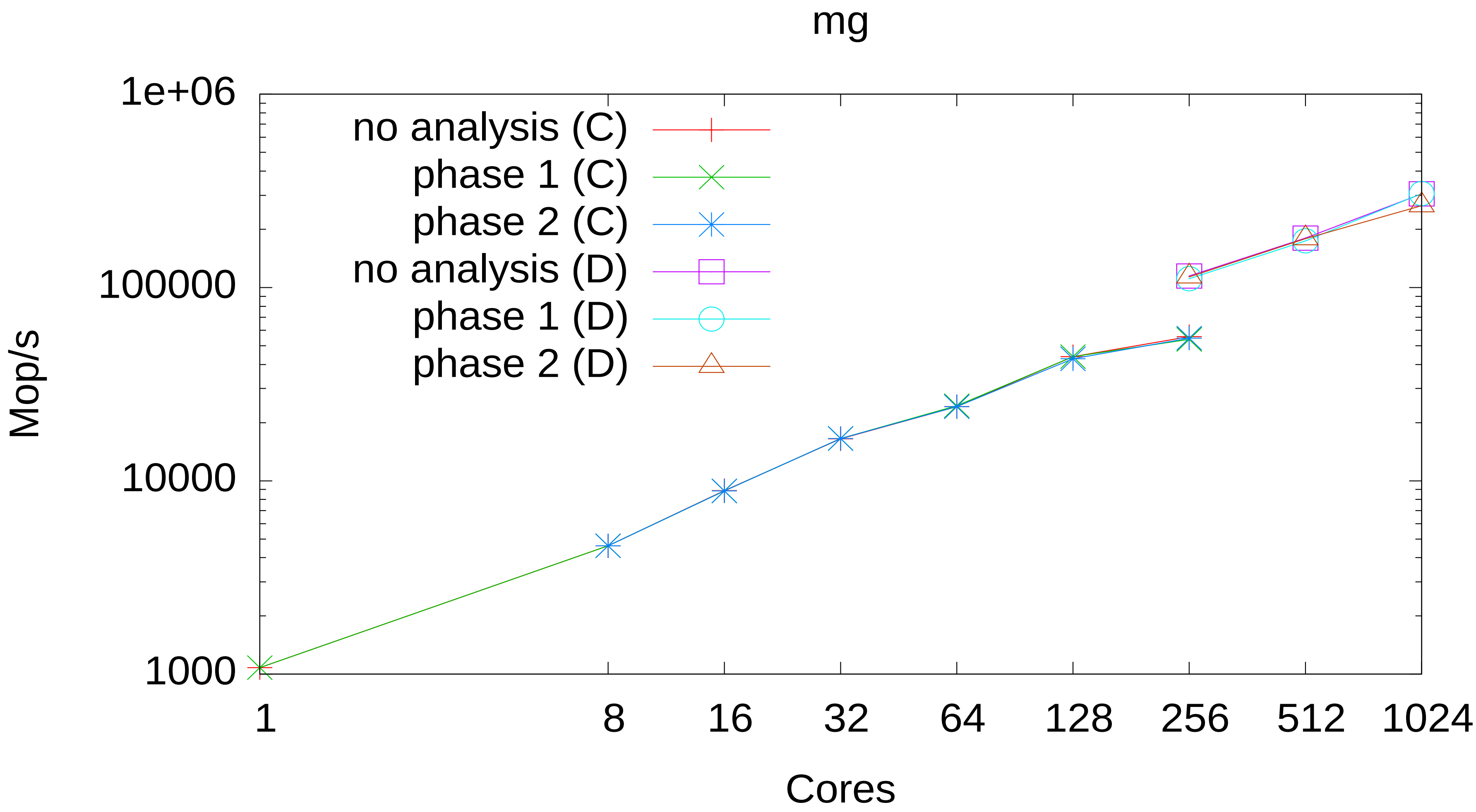
How Well Does it Scale?



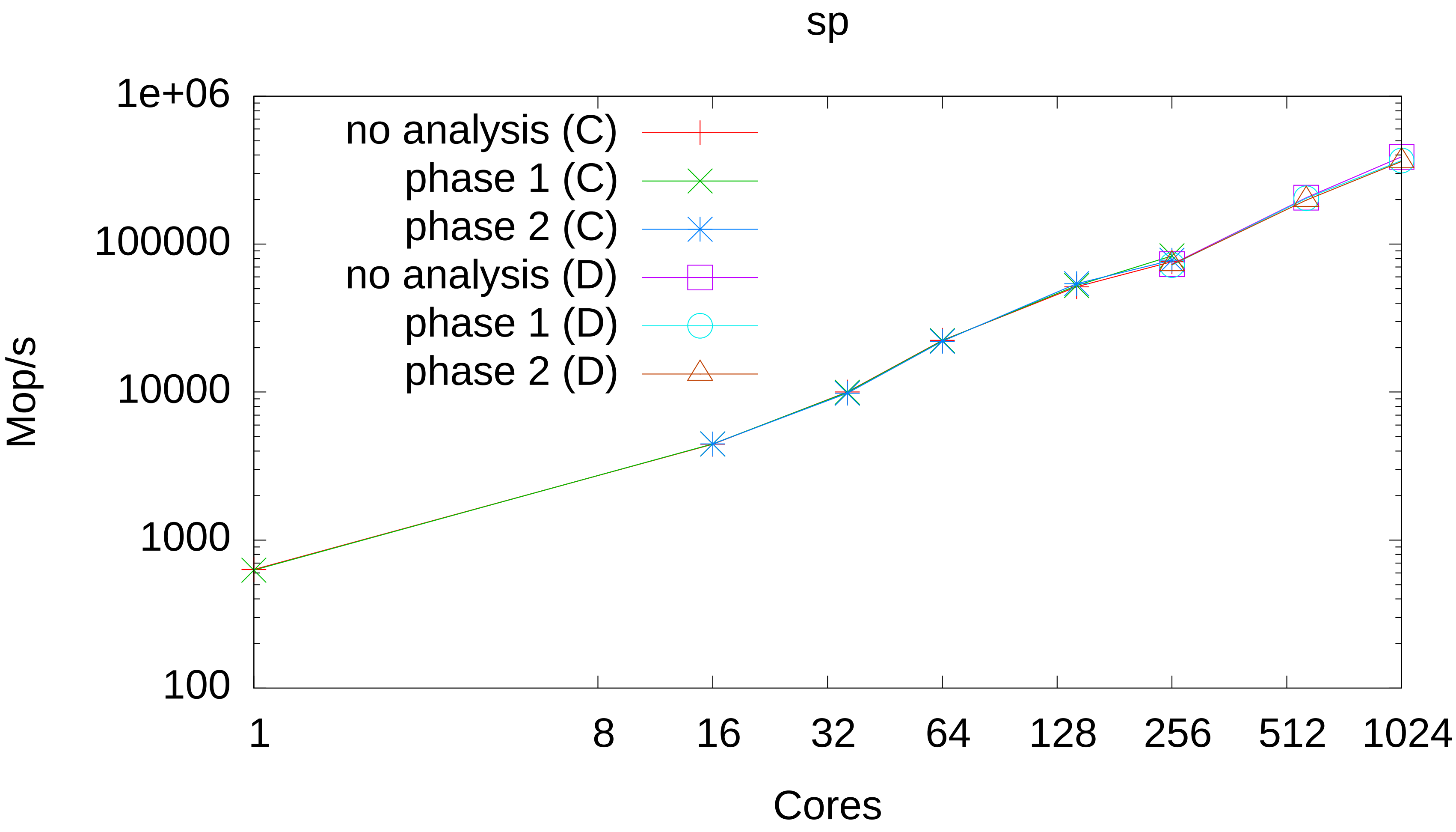
How Well Does it Scale?



How Well Does it Scale?



How Well Does it Scale?

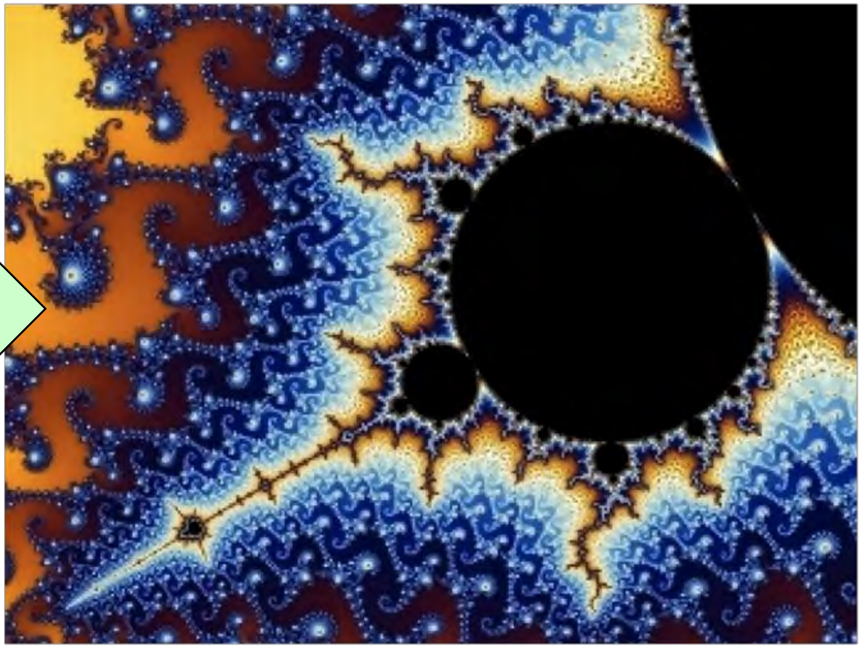


Active Testing Limitations

- Run active testing on application + input.
- What if bug pattern occurs, but no crash?
  - Can’t ask the user
  - Need **specification** from programmer.

```
x=0.7
y=0.3
…
y=5.0
```

Run with
active testing



Parallel Specifications?

- Traditional functional correctness specs.
  - Relate program’s output to its input.
  - Generally complex and difficult to write:

$$
\begin{aligned}
&\forall_{0 \le x < width} \,.\, \forall_{0 \le y < height} \,.\\
&\quad \left(\left| f_{iter}^{\,maxiter}(0) \right| < 2 \;\wedge\; img[x][y] = 0 \right)\\
&\quad \vee\; \exists_{1 \le i < maxiter} \,.\, \left| f_{iter}^{\,i}(0) \right| \ge 2 \;\wedge\; \forall_{1 \le j < i} \,.\, \left| f_{iter}^{\,j}(0) \right| < 2\\
&\qquad\qquad \wedge\; img[x][y] = \mathrm{HSB}\left((i/maxiter)^{\gamma}, 1, 1 \right)
\end{aligned}
$$

$$
\begin{aligned}
\text{where}\quad f_{iter}(c) \;=\; & c^2 + \left(xcenter + (xoff + x)/res \right)\\
& + i\left(ycenter + (yoff - y)/res \right)
\end{aligned}
$$

Parallel Specifications?

- Traditional functional correctness specs.
  - Relate program's output to its input.
  - Generally complex and difficult to write:

Is there an easier way to specify just the **parallel correctness**?

where $$f_{iter}(c) = c^2 + \left(xcenter + (xoff + x)/res\right) + i\left(ycenter + (yoff - y)/res\right)$$

Lightweight Parallel Specs

- **Goal: Lightweight** specifications for **parallel correctness**.
  - Easy for programmers to write.
  - With testing, effective in finding real bugs.
- Semantic determinism [CACM’10, FSE ’09, ICSE ’10].
- Semantic atomicity [ASPLOS ‘11].
- Nondeterministic sequential specifications for parallel correctness [HotPar’10, PLDI’11]

Deterministic Specification

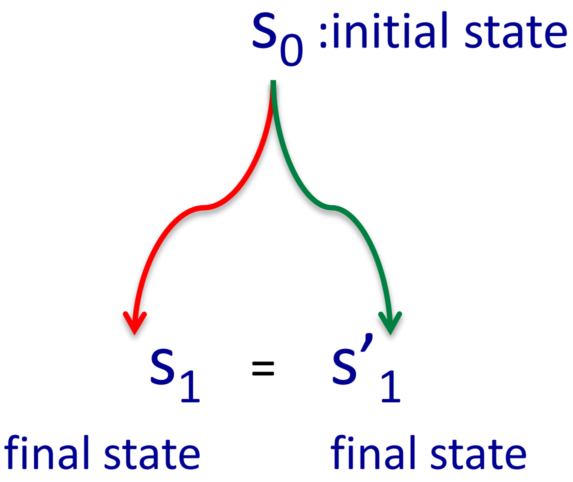
```
// Parallel fractal render
mandelbrot(params, img);
```

- **Goal:** Specify deterministic behavior.
  - Same initial parameters => same image.
  - Non-determinism is internal.

Deterministic Specification

```
deterministic {
   // Parallel fractal render
   mandelbrot(params, img);
}
```

- **Specifies**: Two runs from same initial program state have same result state for any pair of schedules



S_0 :initial state

$S_1 = S'_1$

final state final state

Deterministic Specification

```
double A[][], b[], x[];
...
deterministic {
   // Solve A*x = b in parallel
   lufact_solve(A, b, x);
}
```

- Too restrictive – different schedules may give slightly different floating-point results.

Deterministic Specification

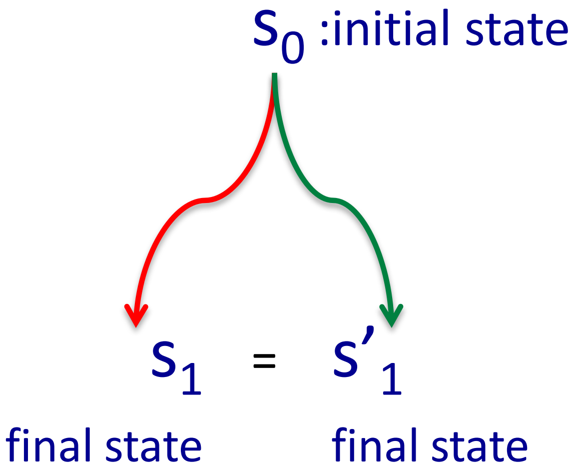
```
set t = new RedBlackTreeSet();
deterministic {
   t.add(3)  || t.add(5);
}
```

- Too restrictive – internal structure of set may differ depending on order of adds.

Semantic Determinism

- Too strict to require every interleaving to give exact same program state:

```
deterministic {
    P
}
```



s_0 :initial state

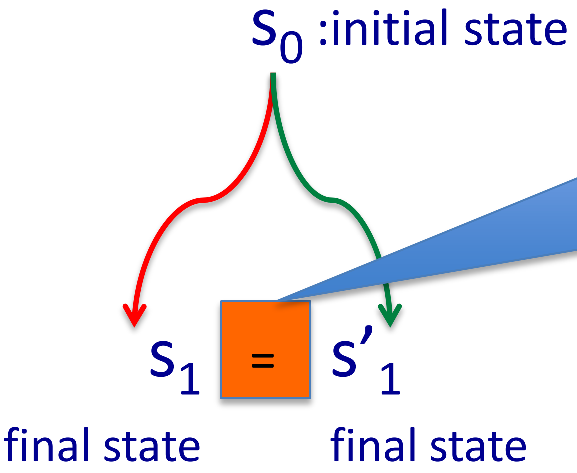
$s_1 = s'_1$

final state final state

Semantic Determinism

- Too strict to require every interleaving to give exact same program state:

```
deterministic {
    P
}
```



S_0 :initial state

S_1 = S'_1

final state

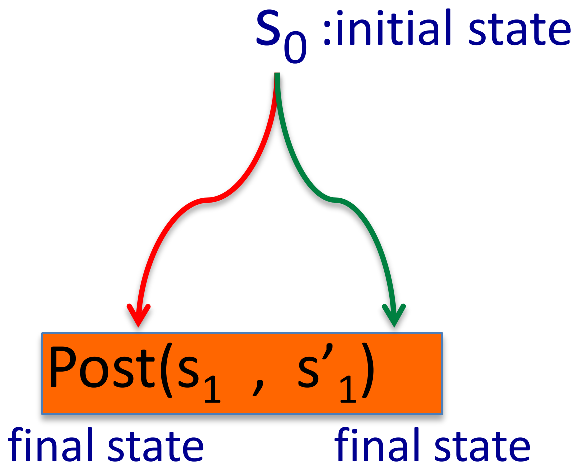
final state

Predicate!
Should be
user-defined.

Semantic Determinism

- Too strict to require every interleaving to give exact same program state:

```
deterministic {
    P
} assert Post(s_1,s'_1)
```



S_0 :initial state

Post(s_1 , s'_1)

final state

final state

Semantic Determinism

```
double A[][], b[], x[];
...
deterministic {
   // Solve A*x = b in parallel
   lufact_solve(A, b, x);
} assert (|x – x’| < ε)
```

“Bridge” predicate

Semantic Determinism

```
set t = new RedBlackTreeSet();
deterministic {
   t.add(3)  || t.add(5);
} assert (t.equals(t’))
```

- Resulting sets are *semantically* equal.

Preconditions for Determinism

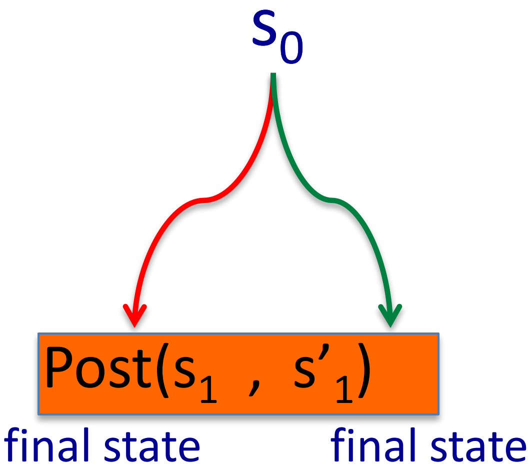
```
set t = …
deterministic {
   t.add(3) || t.add(5);
} assert (t.equals(t’))
…
deterministic {
   t.add(4) || t.add(6);
} assert (t.equals(t’))
```

- Too strict – initial states must be identical
  - Not compositional.

Preconditions for Determinism

- Too strict to require identical initial states:

```
deterministic {
    P
} assert Post(s1,s1')
```



s_0

Post(s_1 , s'_1)

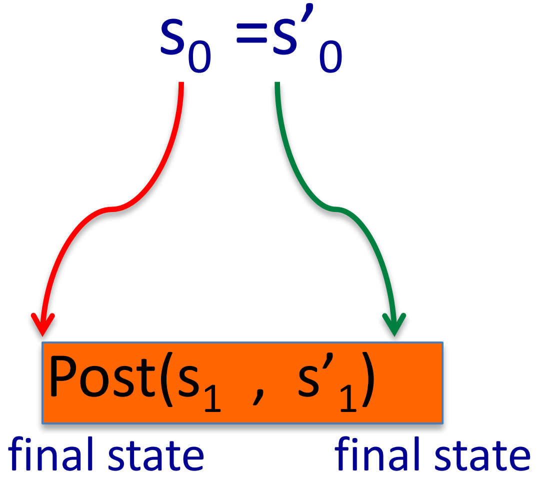
final state

final state

Preconditions for Determinism

- Too strict to require identical initial states:

```
deterministic {
   P
} assert Post(s₁,s₁')
```

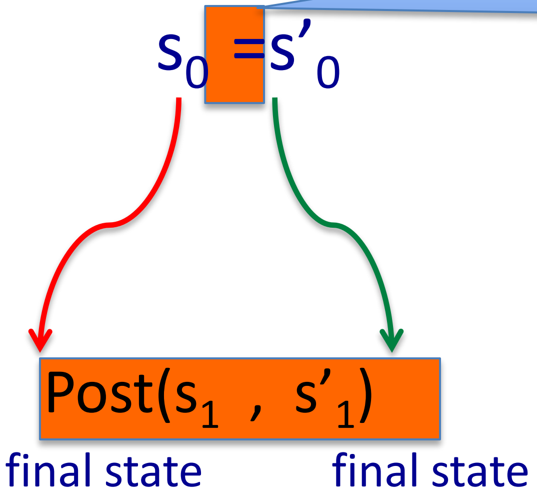


Preconditions for Determinism

- Too strict to require identical initial states:

```
deterministic assume ($s_0$ = $s_0'$){
    P
} assert Post($s_1$,$s_1'$)
```

Predicate! Should be user-defined.



$s_0 = s'_0$

Post(s_1 , s'_1)

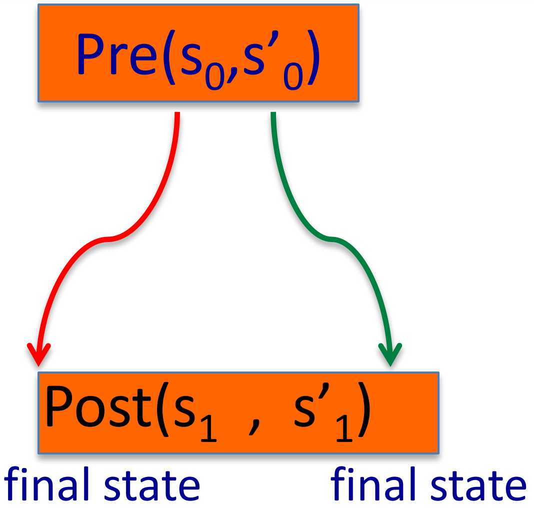
final state

final state

Preconditions for Determinism

- Too strict to require identical initial states:

```
deterministic assume Pre(s0,s0'){
    P
} assert Post(s1,s1')
```



$\mathrm{Pre}(s_0, s'_0)$

$\mathrm{Post}(s_1 \;,\; s'_1)$

final state

final state

Bridge predicates/assertions

$$\texttt{deterministic assume } \underbrace{\texttt{Pre}(s_0,s_0')}_{\text{“Bridge” predicate}} \ \{$$

$$\quad \texttt{P}$$

$$\}\ \underbrace{\texttt{assert Post}(s_1,s_1')}_{\text{“Bridge” assertion}}$$

Preconditions for Determinism

```
set t = ...
deterministic assume (t.equals(t’) {
   t.add(4) || t.add(6);
} assert (t.equals(t’))
```

- **Specifies**: Semantically equal sets yield semantically equal sets.

Deterministic Specs

- Can effectively test deterministic specs.
  - Added assertions to 13 benchmarks.
  - Used Active Testing to test if concurrency issues (data races, atomicity violations, etc.) could lead to violations of deterministic spec.
- Developed techniques for automatically inferring these specs
- See our CACM 10, FSE 09, ICSE 09, ASPLOS 11 papers for details

Checking correctness of parallel programs

Parallel program

Satisfies?

Complicated, end-to-end reasoning

Functional specification

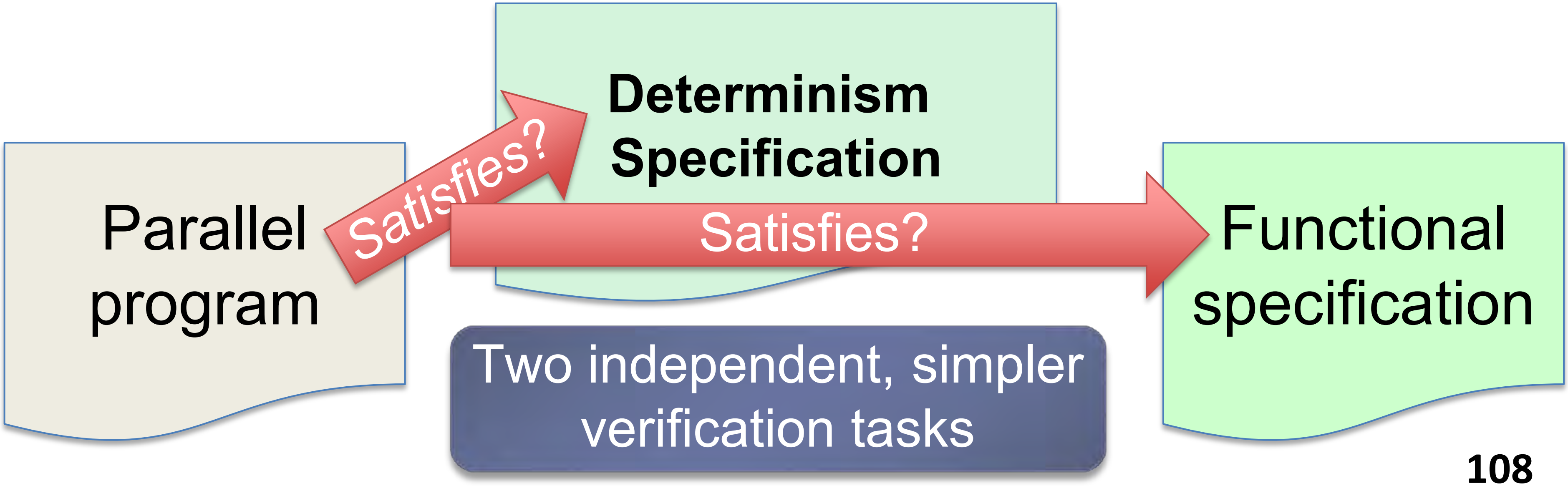
Our proposal: Separate reasoning about functional correctness and thread schedules

Parallel program

Satisfies?

Functional specification

Complicated,
end-to-end reasoning



Satisfies?

Determinism Specification

Parallel program

Satisfies?

Functional specification

Two independent, simpler verification tasks

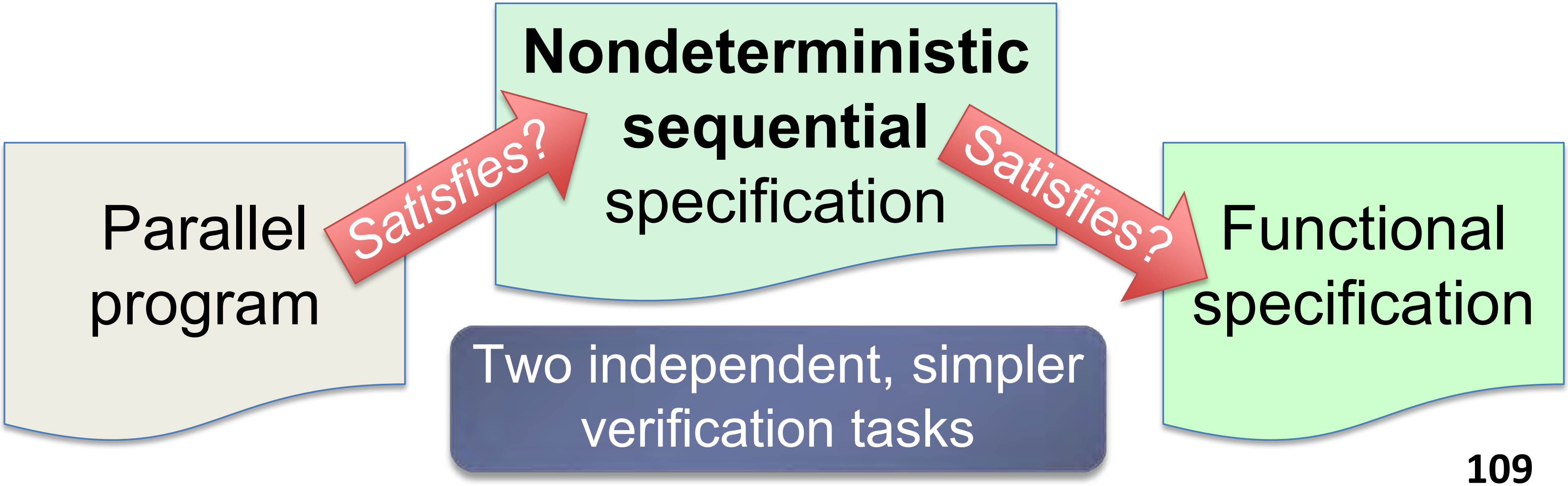
Our proposal: Separate reasoning about functional correctness and thread schedules

Parallel program

Satisfies?

Functional specification

Complicated, end-to-end reasoning



Parallel program

Satisfies?

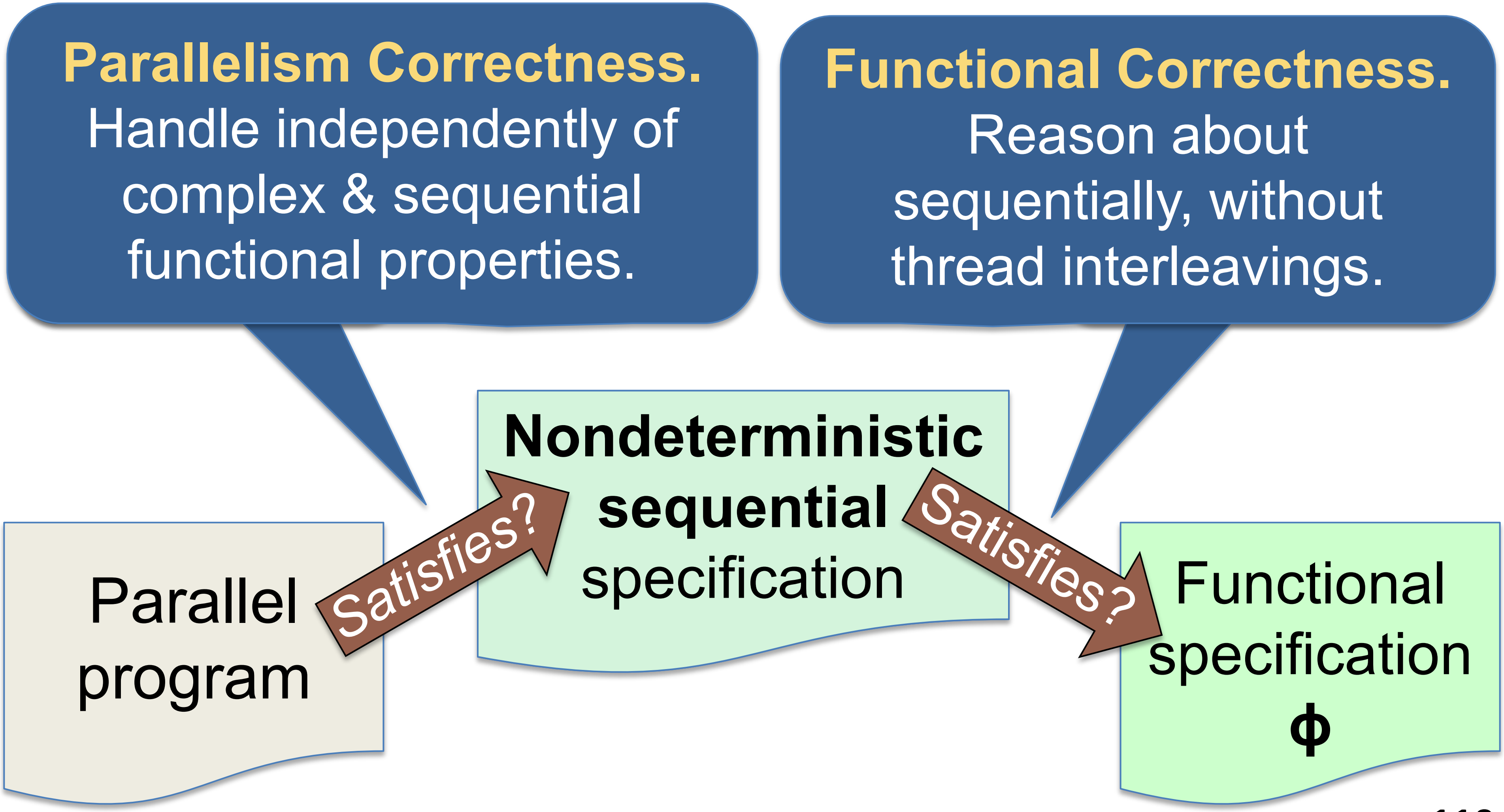
Nondeterministic sequential specification

Satisfies?

Functional specification

Two independent, simpler verification tasks

Goal: Decompose effort in addressing parallelism and functional correctness



Parallelism Correctness. Handle independently of complex & sequential functional properties.

Functional Correctness. Reason about sequentially, without thread interleavings.

Parallel program

Satisfies?

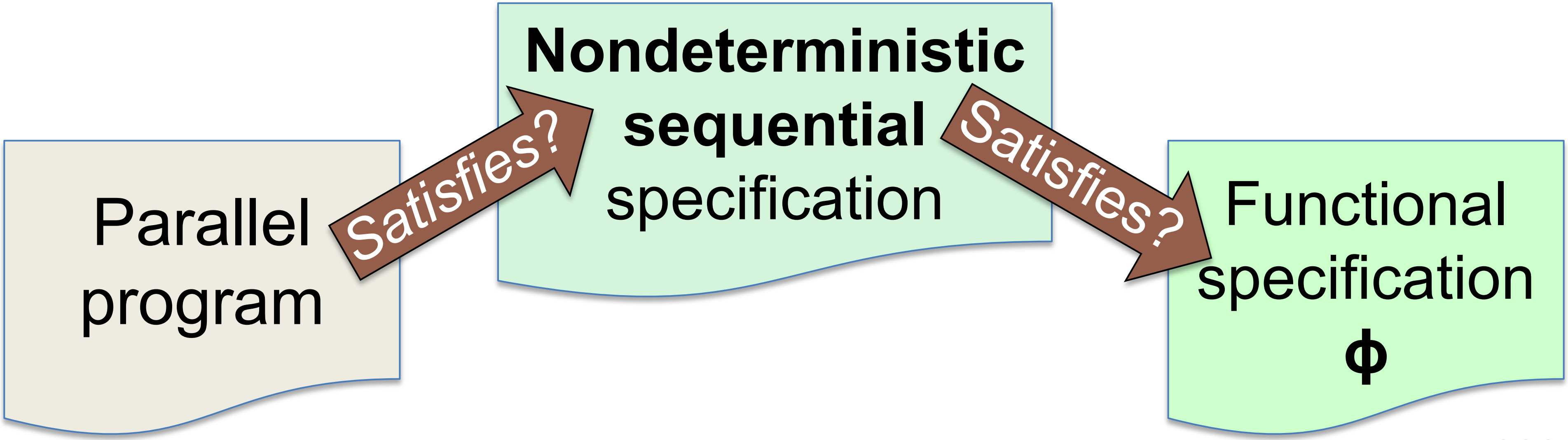
Nondeterministic sequential specification

Satisfies?

Functional specification **ϕ**

Goal: Decompose effort in addressing parallelism and functional correctness

1. **NDSeq: easy-to-write spec for parallelism.**
1. **Runtime checking of NDSeq specifications.**



Parallel program

Satisfies?

Nondeterministic sequential specification

Satisfies?

Functional specification **ϕ**

Motivating Example

- **Goal:** Find minimum-cost item in list.

```
for (i in [1..N]):
  c = min_cost
  b = lower_bound(i)
  if b >= c:
    continue
  cost = compute_cost(i)
  if cost < min_cost:
    min_cost = cost
    min_item = i
```

Input: N items.

Output: min_cost and min_item.

Motivating Example

- **Goal:** Find minimum-cost item in list.

```
for (i in [1..N]):
  c = min_cost
  b = lower_bound(i)
  if b >= c:
    continue
  cost = compute_cost(i)
  if cost < min_cost:
    min_cost = cost
    min_item = i
```

Computes **cheap** lower bound on cost of **i**.

Prune when **i** cannot have minimum-cost.

Computes cost of item **i**. **Expensive.**

Motivating Example

- **Goal:** Find minimum-cost item in list.

```
for (i in [1..N]):
  c = min_cost
  b = lower_bound(i)
  if b >= c:
    continue
  cost = compute_cost(i)
  if cost < min_cost:
    min_cost = cost
    min_item = i
```

How do we parallelize this code?

Parallel Motivating Example

- **Goal:** Find min-cost item in list, **in parallel**.

```
parallel-for (i in [1..N]):
  c = min_cost
  b = lower_bound(i)
  if b >= c:
    continue
  cost = compute_cost(i)
  synchronized (lock):
    if cost < min_cost:
      min_cost = cost
      min_item = i
```

Claim: Parallelization is clearly correct.

How can we specify this parallel correctness?

Specifying Parallel Correctness

- **Idea**: Use sequential program as spec.

```
parallel-for (i in [1..N]):
  c = min_cost
  b = lower_bound(i)
  if b >= c:
    continue
  cost = compute_cost(i)
  synchronized (lock):
    if cost < min_cost:
      min_cost = cost
      min_item = i
```

No.

Satisfies?

```
for (i in [1..N]):
  c = min_cost
  b = lower_bound(i)
  if b >= c:
    continue
  cost = compute_cost(i)

  if cost < min_cost:
    min_cost = cost
    min_item = i
```

Parallel-Sequential Equivalence?

items: **(1)** bound: 5 cost: 5 | (**2**) bound: 5 cost: 5

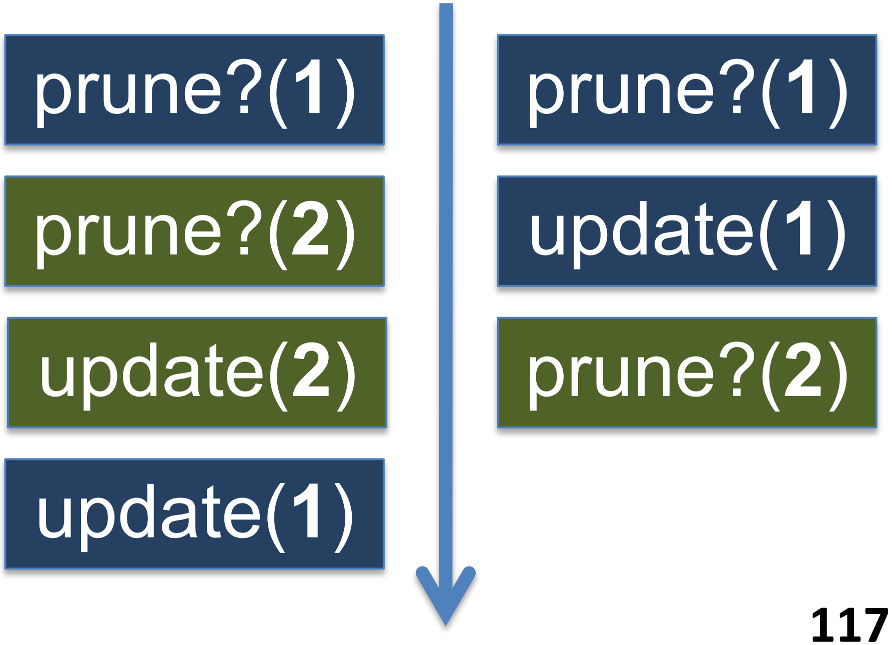
min_item: – **(2)**

min_cost: ∞ **5**

```
parallel-for (i in [1..N]):
  c = min_cost
  b = lower_bound(i)
  if b >= c:
    continue
  cost = compute_cost(i)
  synchronized (lock):
    if cost < min_cost:
      min_cost = cost
      min_item = i
```

But sequential program:

- Returns min_item = **(1)**.
- Prunes (2).



Specifying Parallel Correctness

Must give sequential spec this freedom.

```
parallel-for (i in [1..N]):
  c = min_cost
  b = lower_bound(i)
  if b >= c:
    continue
  cost = compute_cost(i)
  synchronized (lock):
    if cost < min_cost:
      min_cost = cost
      min_item = i
```

Process items in a **nondeterministic** order.

Avoid pruning by scheduling check before updates.

Nondeterministic Sequential Spec

Runs iterations **in any order.**

```
parallel-for (i in [1..N]):
  c = min_cost
  b = lower_bound(i)
  if b >= c:
    continue
      min_cost = cost
      min_item = i
```

Can **choose**
not to prune item.

```
nd-for (i in [1..N]):
  c = min_cost
  b = lower_bound(i)
  if * && b >= c:
    continue
  cost = compute_cost(i)

  if cost < min_cost:
    min_cost = cost
    min_item = i
```

Nondeterministic Sequential Spec

- Parallelism correct if no more nondeterminism:

```
parallel-for (i in [1..N]):
  c = min_cost
  b = lower_bound(i)
  if b >= c:
    continue
  cost = compute_cost(i)
  synchronized (lock):
    if cost < min_cost:
      min_cost = cost
      min_item = i
```

Satisfies?

Yes.

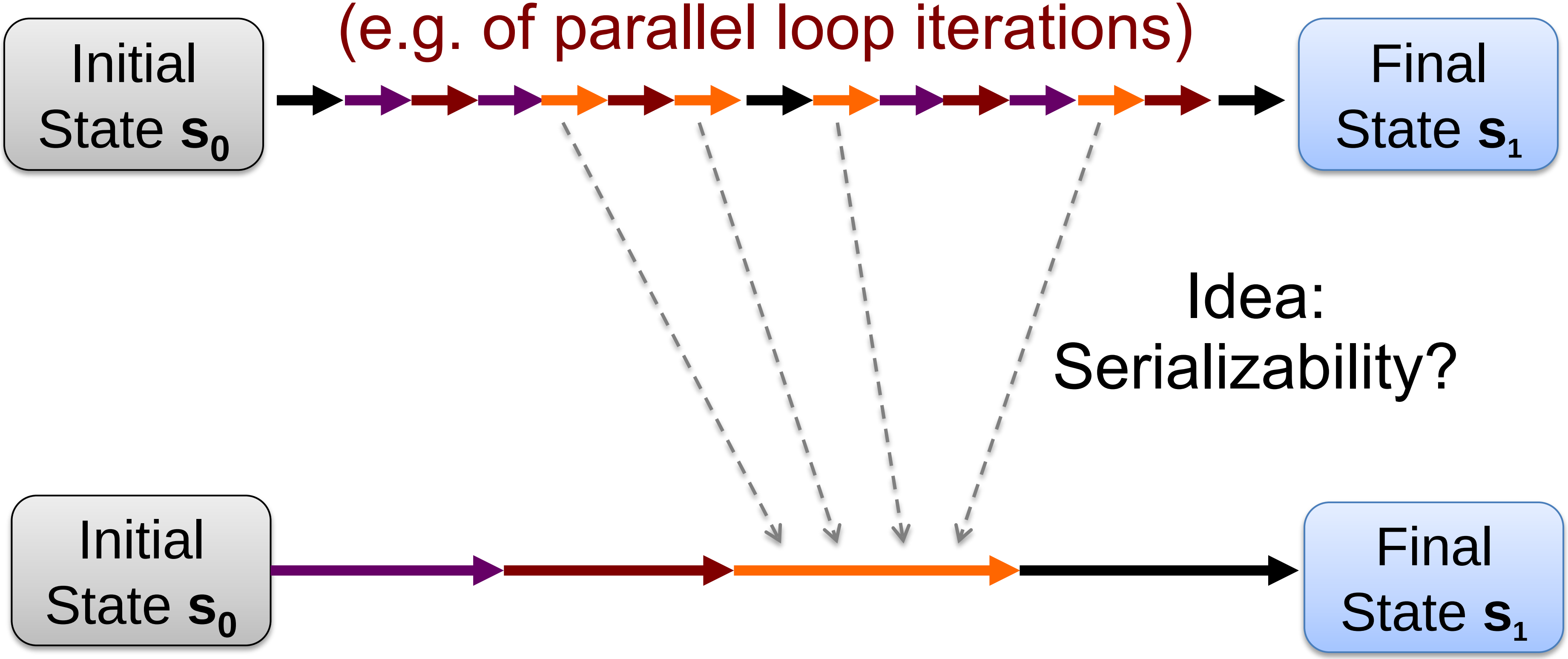
```
nd-for (i in [1..N]):
  c = min_cost
  b = lower_bound(i)
  if * && b >= c:
    continue
  cost = compute_cost(i)

  if cost < min_cost:
    min_cost = cost
    min_item = i
```

NDSeq Specification Patterns

- Found three recipes for adding *'s:
    1. Optimistic Concurrent Computation
       (optimistic work with conflict detection)
    2. Redundant Computation Optimization
       (e.g., pruning in branch-and-bound)
    3. Irrelevant Computation
       (e.g., updating a performance counter)
- With these recipes, fairly simple to write NDSeq specifications for our benchmarks.
- See our HotPar 10 and PLDI 11 papers

Testing Parallelism Correctness

Given: an execution of parallel program (e.g. of parallel loop iterations)



Initial State $\mathbf{s}_0$

Final State $\mathbf{s}_1$

Idea: Serializability?

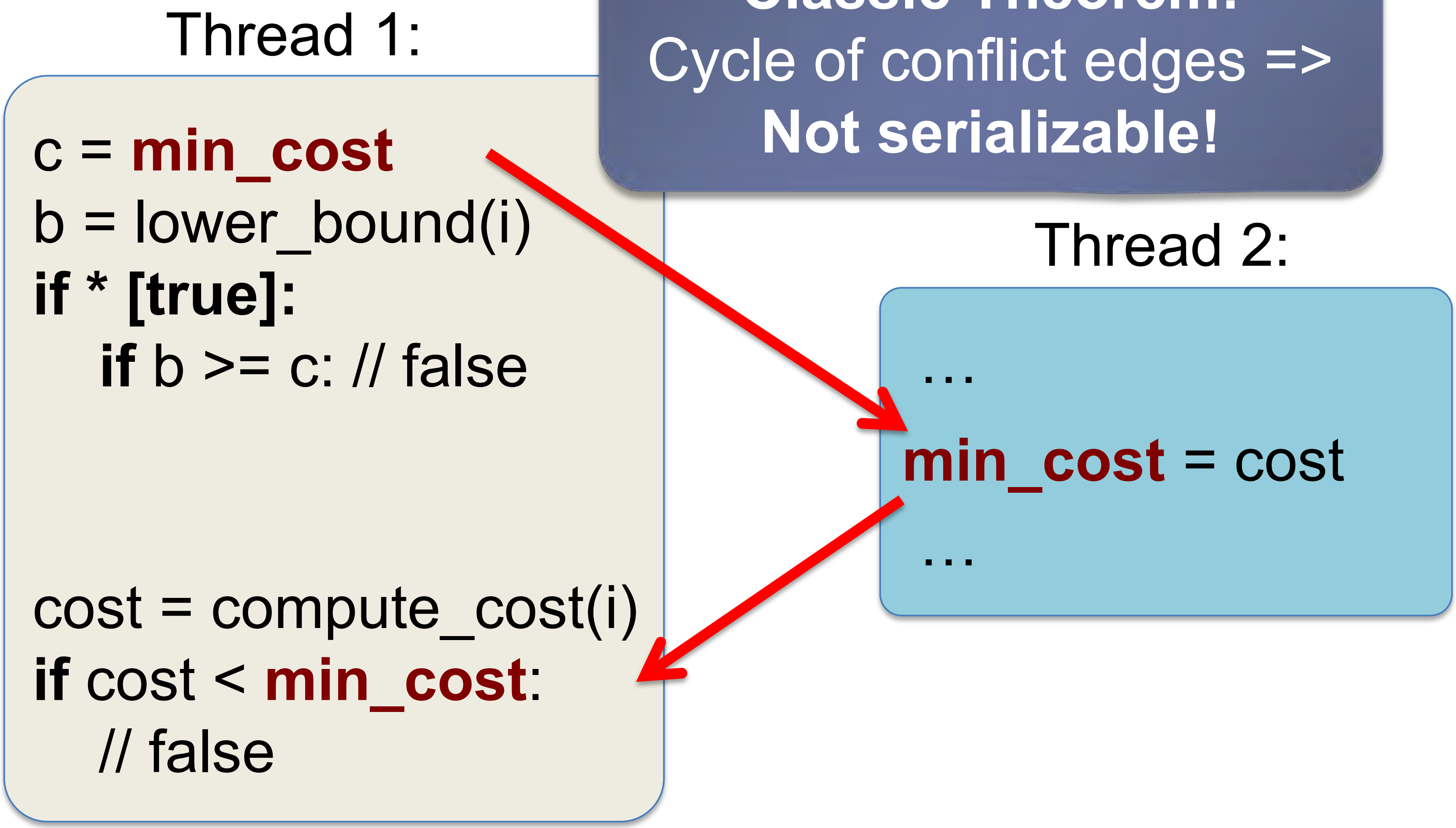
Initial State $\mathbf{s}_0$

Final State $\mathbf{s}_1$

Is there an **equivalent** execution of NDSeq spec?

Conflict-Serializability is Too Strict

Classic Theorem:
Cycle of conflict edges =>
Not serializable!



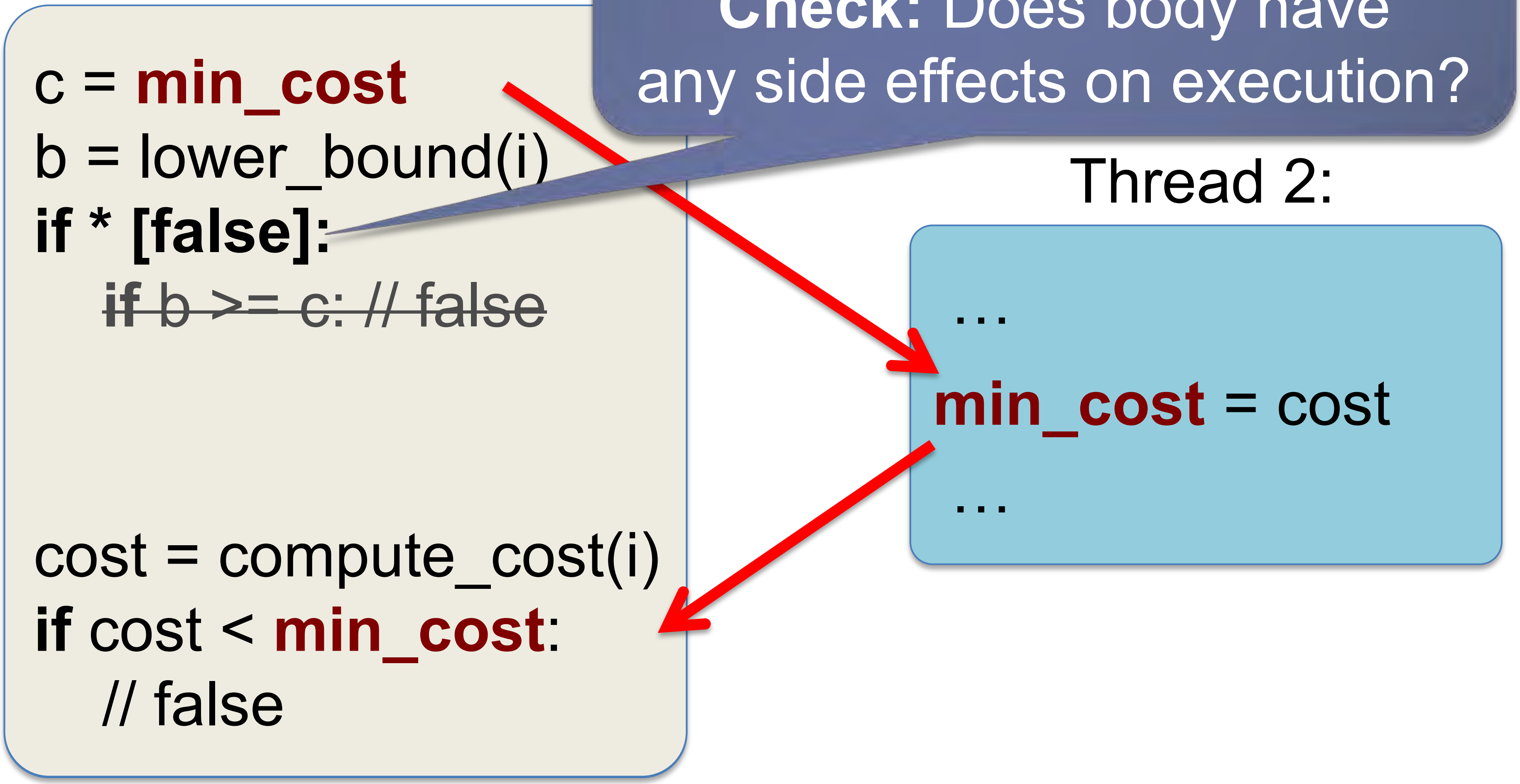
Thread 1:

```
c = min_cost
b = lower_bound(i)
if * [true]:
    if b >= c: // false



cost = compute_cost(i)
if cost < min_cost:
    // false
```

Thread 2:

```
…
min_cost = cost
…
```

Relaxing Conflict-Serializability



Can we set * to false?

Check: Does body have any side effects on execution?

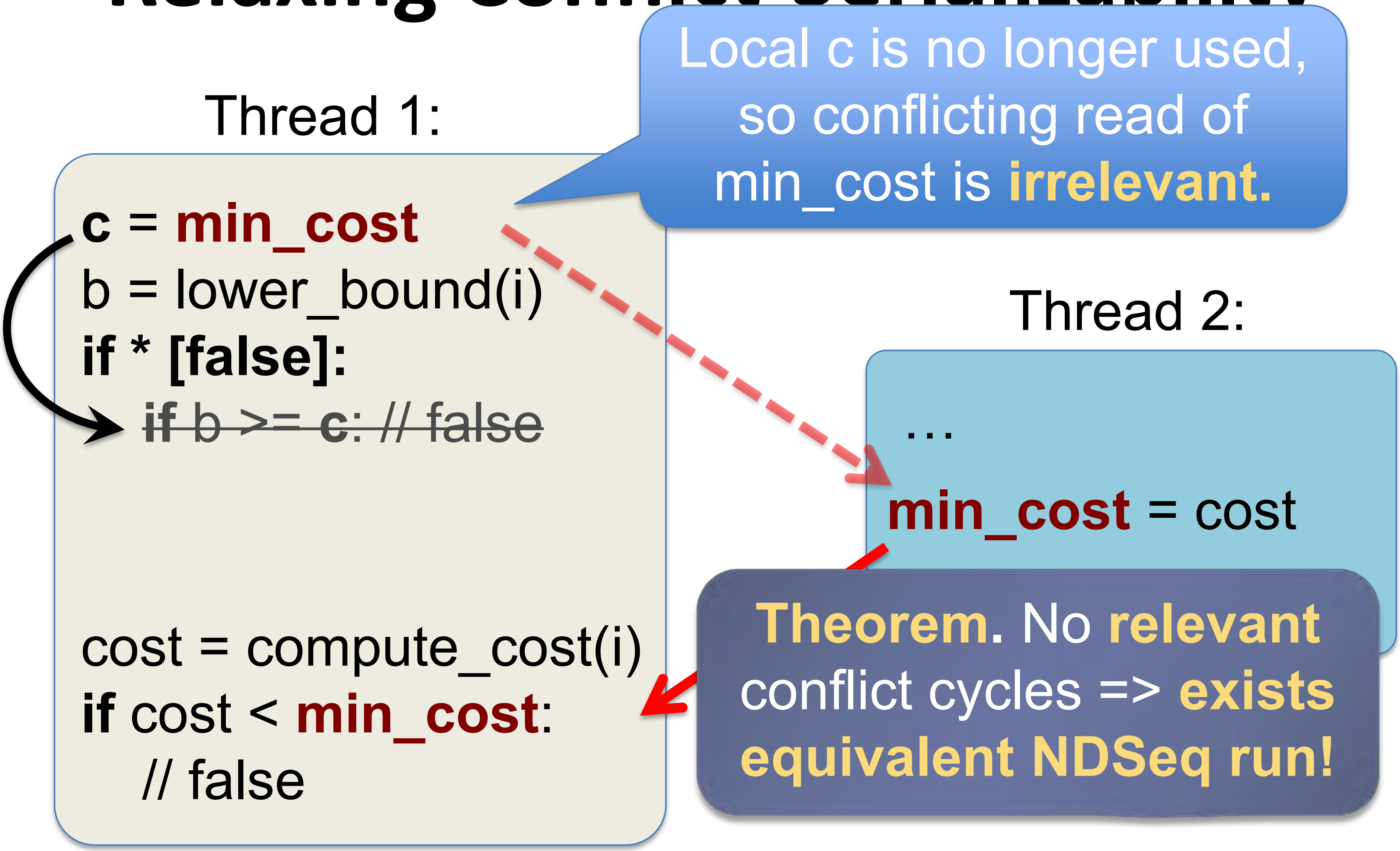
Thread 1:

```
c = min_cost
b = lower_bound(i)
if * [false]:
    ~~if b >= c: // false~~


cost = compute_cost(i)
if cost < min_cost:
    // false
```

Thread 2:

```
…
min_cost = cost
…
```

Relaxing Conflict-Serializability



Local c is no longer used, so conflicting read of min_cost is **irrelevant.**

Thread 1:

c = **min_cost**
b = lower_bound(i)
if * [false]:
    ~~if b >= **c**: // false~~

cost = compute_cost(i)
if cost < **min_cost**:
    // false

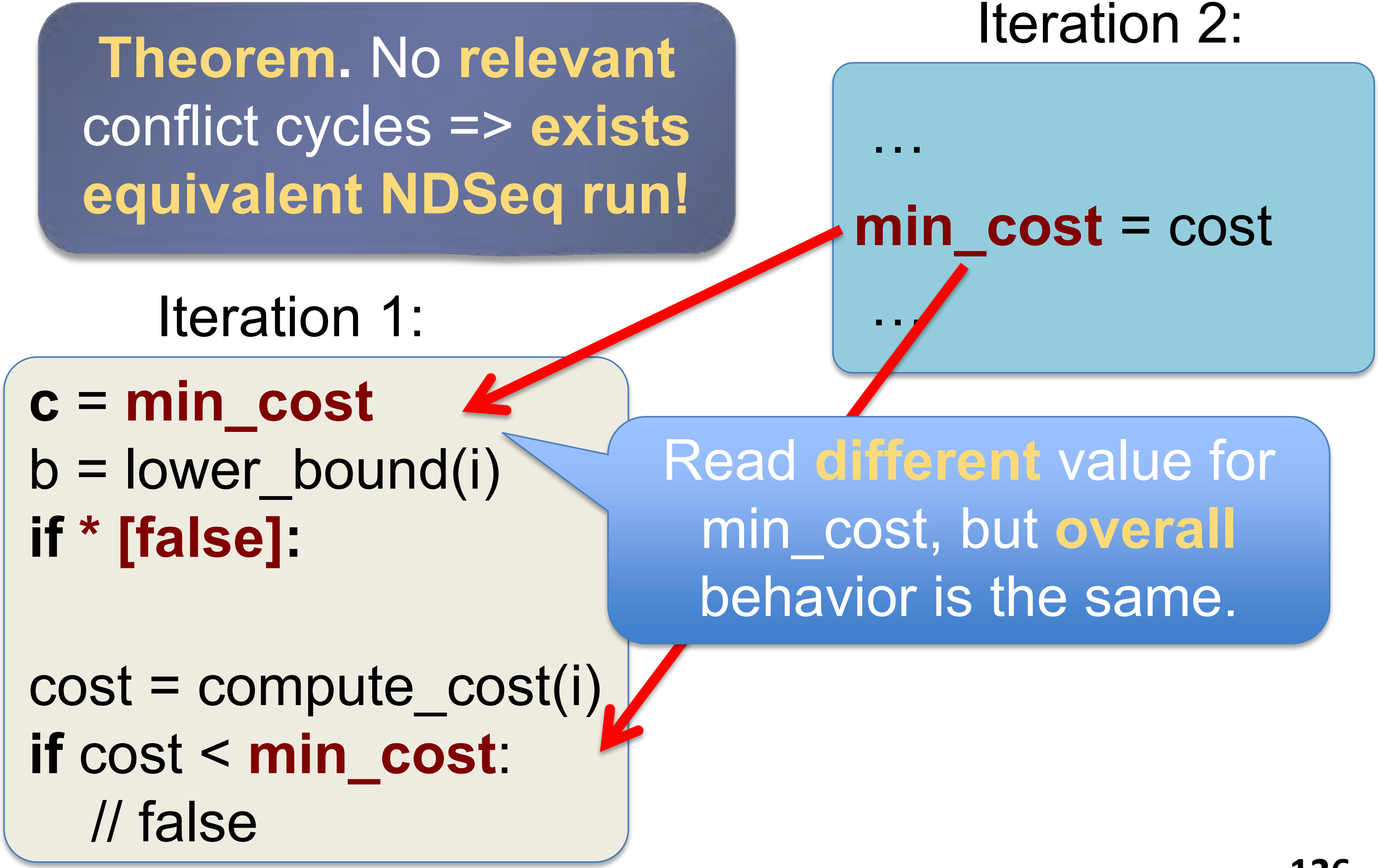
Thread 2:

…
min_cost = cost

Theorem. No **relevant** conflict cycles => **exists equivalent NDSeq run!**

Relaxing Conflict-Serializability

Theorem. No **relevant** conflict cycles => **exists equivalent NDSeq run!**



Iteration 1:

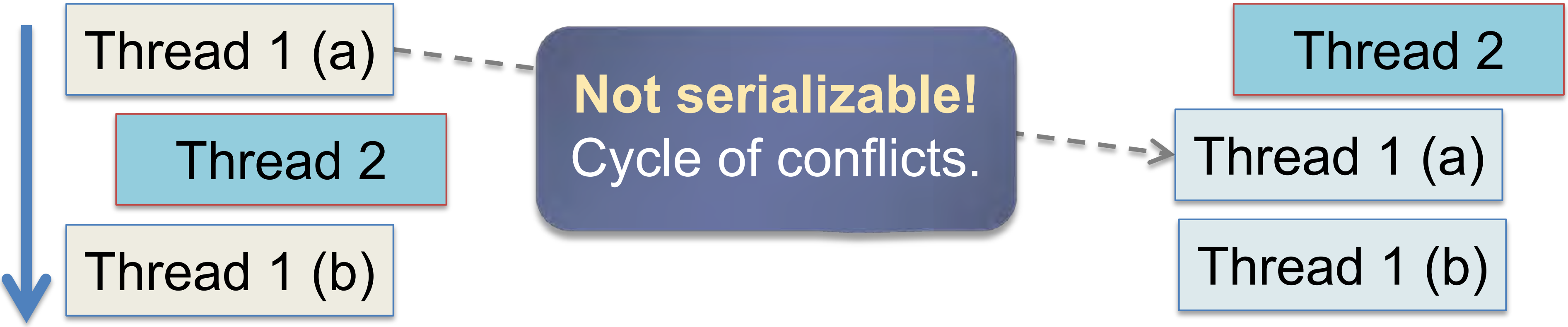
```
c = min_cost
b = lower_bound(i)
if * [false]:

cost = compute_cost(i)
if cost < min_cost:
   // false
```

Iteration 2:

```
…
min_cost = cost
…
```

Read **different** value for min_cost, but **overall** behavior is the same.

Traditional conflict serializability:



Thread 1 (a)

Thread 2

Thread 1 (b)

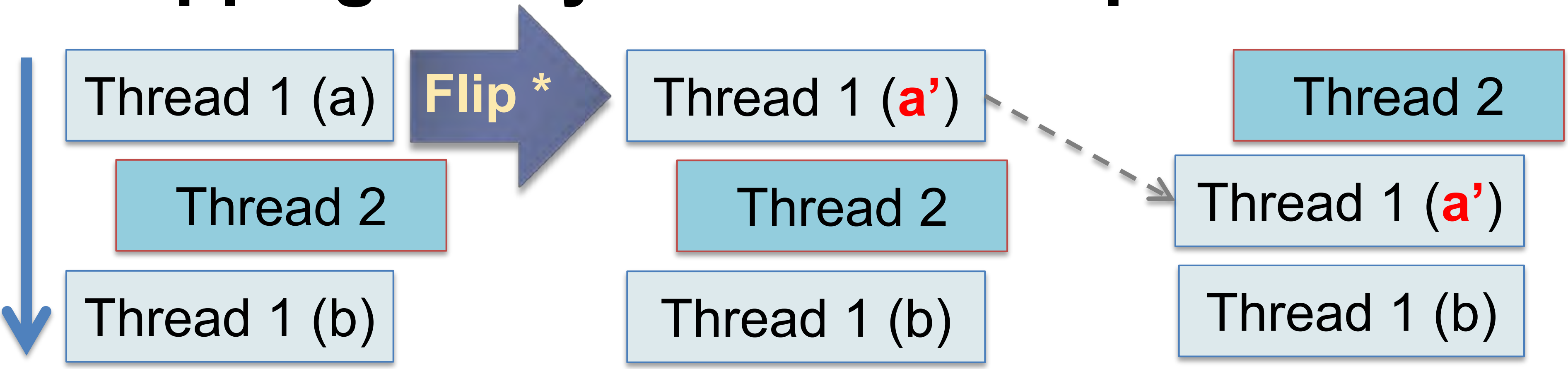
Not serializable!
Cycle of conflicts.

Thread 2

Thread 1 (a)

Thread 1 (b)

+ flipping * + dynamic data dependence:



Thread 1 (a)

Thread 2

Thread 1 (b)

Flip *

Thread 1 (**a’**)

Thread 2

Thread 1 (b)

Thread 2

Thread 1 (**a’**)

Thread 1 (b)

Experimental Evaluation

- Wrote and tested NDSeq specifications for:
  - Java Grande, Parallel Java, Lonestar, DaCapo, and nonblocking data structure.
  - **Size:** 40 to 300K lines of code.
  - Tested 5 parallel executions / benchmark.
- **Two claims:**
  1. Easy to write NDSeq specifications.
  2. Our technique serializes significantly more executions than traditional methods.

	Benchmark	Lines of Code	# of Parallel Constructs	# of if(*)
	stack	40	**1**	**2**
	queue	60	**1**	**2**
	meshrefine	1K	**1**	**2**
DaCapo	sunflow	24K	**4**	**4**
	xalan	302K	**1**	**3**
PJ	keysearch3	200	**2**	0
	mandelbrot	250	**1**	0
	phylogeny	4.4K	**2**	**3**
JGF	series	800	**1**	0
	crypt	1.1K	**2**	0
	raytracer	1.9K	**1**	0
	montecarlo	3.6K	**1**	0

	Benchmark	**Size of Trace**	**Serializability Warnings**	
			Traditional	Our Technique
	stack	1,744	**5 (false)**	0
	queue	846	**9 (false)**	0
	meshrefine	747K	**30 (false)**	0
DaCapo	sunflow	24,250K	**28 (false)**	**3 (false)**
	xalan	16,540K	**6 (false)**	**2 (false)**
PJ	keysearch3	2,059K	**2 (false)**	0
	mandelbrot	1,707K	**1 (false)**	0
	phylogeny	470K	**6**	**6**
JGF	series	11K	0	0
	crypt	504K	0	0
	raytracer	6,170K	**1**	**1**
	montecarlo	1,897K	**2 (false)**	0

	Benchmark	Size of Trace	Serializability Warnings	
			Traditional	Our Technique
	stack	1,744	**5 (false)**	0
	queue	846	**9 (false)**	0
	meshrefine	747K	**30 (false)**	0
DaCapo	sunflow	24,250K	**28 (false)**	**3 (false)**
	xalan	16,540K	**6 (false)**	**2 (false)**
PJ	keysearch3	2,059K	**2 (false)**	0
	mandelbrot	1,707K	**1 (false)**	0
	phylogeny	470K	**6**	**6**
JGF	series	11K	0	0
	crypt	504K	0	0
	raytracer	6,170K	**1**	**1**
	montecarlo	1,897K	**2 (false)**	0

Conclusions

- Build testing tools
  - Close to what programmers use
  - Hide program analysis under testing
- Develop light-weight specification mechanisms
  - Bridge predicates and NDSeq
- Claim: We have made enough progress in finding concurrency bugs
  - Our tools can find important bugs quickly
  - Time to focus on sequential test generation