EECS
Electrical Engineering and
Computer Sciences

BERKELEY LAB
Lawrence Berkeley National Laboratory

BERKELEY PARLAB

PARALLEL COMPUTING LABORATORY

CORVETTE: Program Correctness, Verification, and Testing for Exascale

PI: **Koushik Sen**, UC Berkeley
co-PI: James W. Demmel, UC Berkeley
co-PI: Costin Iancu, LBNL

Post-doc and students:
Cindy Rubio Gonzalez, Chang-Seo Park, Ahn Cuong Nguyen

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 between two MPI_... calls
- **Things are changing**

Motivation

- High performance scientific computing
  - Exascale: O(10^6) nodes, O(10^3) cores per node
  - Requires asynchrony and “relaxed” memory consistency
  - Shared memory with dynamic task parallelism
  - Languages allow remote memory modification
- Correctness challenges
  - Non-deterministic causes hard to diagnose correctness and performance bugs
    - Data races, atomicity violations, deadlocks …
  - Bugs in DSL
  - Scientific applications use floating-points: non-determinism leads to non-reproducible results
  - Numerical exceptions can cause rare but critical bugs that are hard for non-experts to detect and fix

Goals

Develop correctness tools for different programming models: PGAS, MPI, dynamic parallelism

I. Testing and Verification

- ❑ Identify sources of non-determinism in executions
- ❑ Data races, atomicity violations, non-reproducible floating point results
- ❑ Explore state-of-the-art techniques that use dynamic analysis
- ❑ Develop precise and scalable tools: < 2X overhead

II. Debugging

- ❑ Use minimal amount of concurrency to reproduce bug
- ❑ Support two-level debugging of high-level abstractions
- ❑ Detect causes of floating-point anomalies and determine the minimum precision needed to fix them

Detect bugs

I. Testing and Verification Tools

Scalable Testing of Parallel 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
  - Efficient: reduce overhead from 10x-100x to 2x
  - **Precise**
  - Reproducible
  - **Scalable**
- **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(x) = x

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

foo is an expensive function

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;
  }
}
```

Do you see any problem is this code?

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

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;
   }
}
```

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

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: 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));
```

foo is an expensive function

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

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;
  }
}
```

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

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;
```

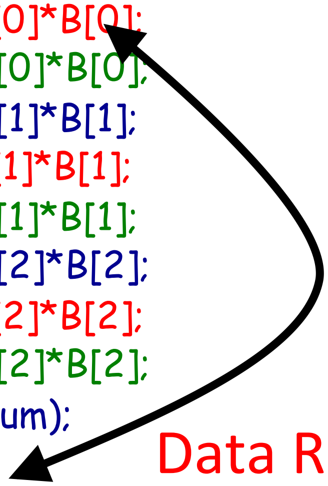
Simple Example in UPC: Trace

```
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?

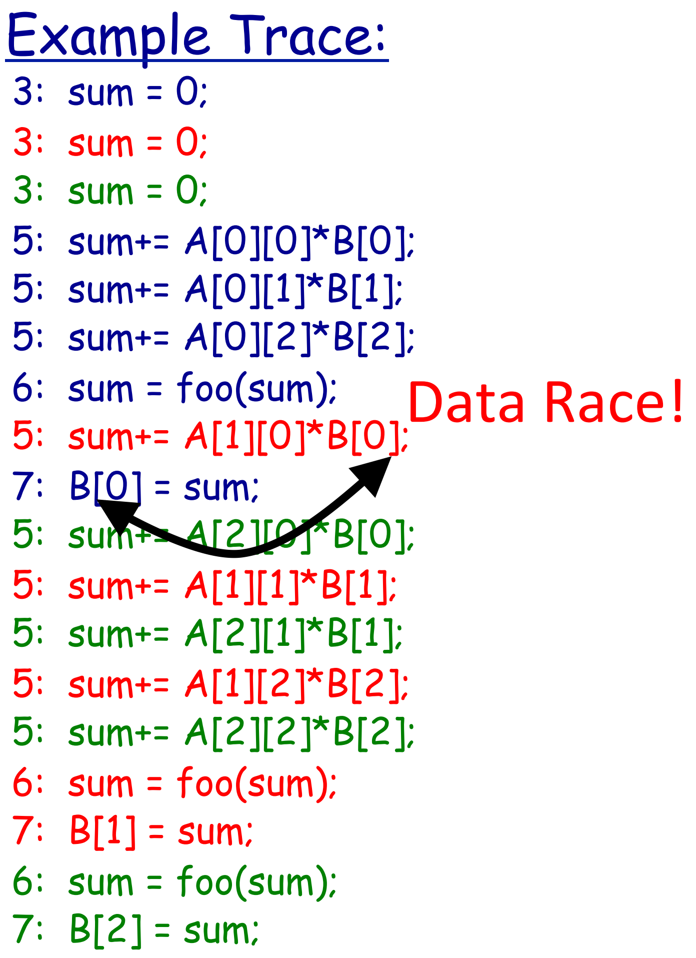
Simple Example in UPC: Trace

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!

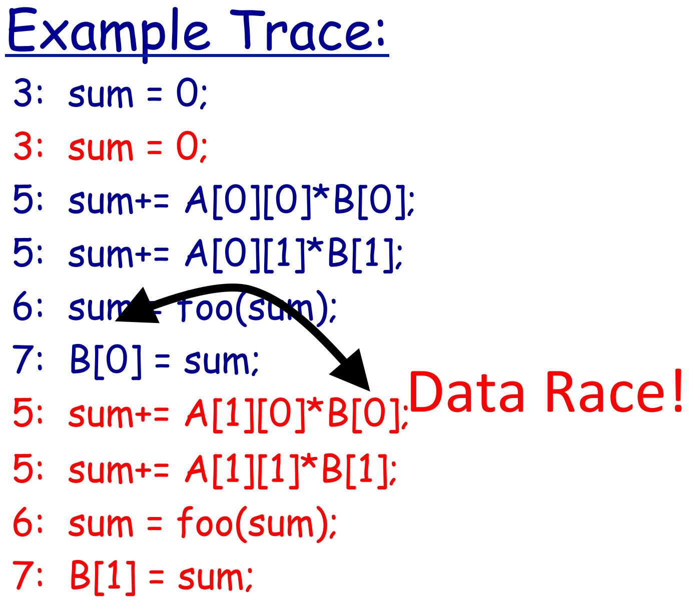
Simple Example in UPC: Trace

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

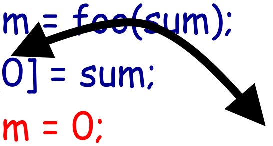
```
1: void matvec(shared [N] int A[N][N],
                          shared int B[N],
                          shared int C[N]) {
2:    upc_forall(int i = 0; i < N; i++; &C[i]) {
3:       int sum = 0;
4:       for(int j = 0; j < N; j++)
5:          sum += A[i][j] * B[j];
6:       sum = foo(sum);
7: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

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

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:
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

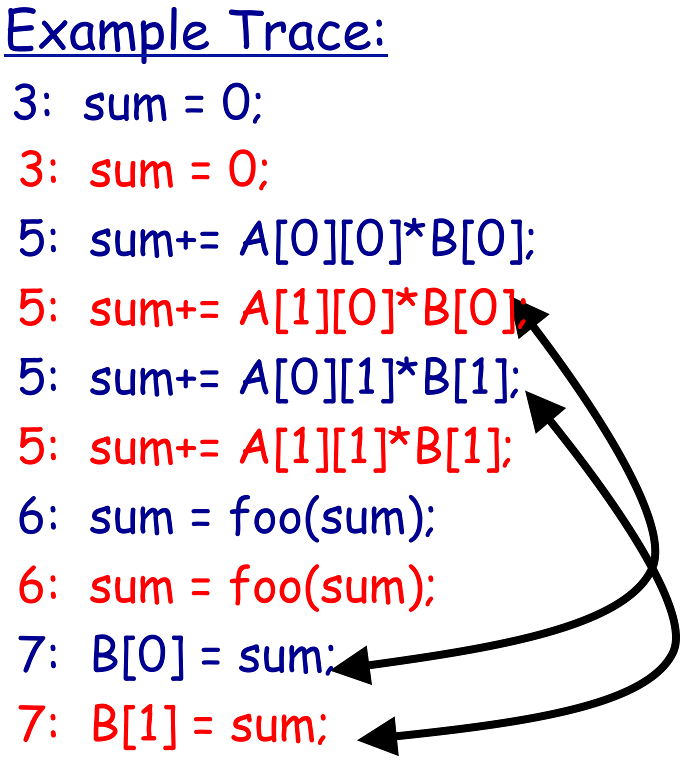
```
             shared int B[N],
             shared int C[N]) {
```

2. 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

```
              shared int B[N];
              shared int C[N]) {
```

2. Run instrumented program normally -> Trace

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

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 Tes

Goals. 1. Confirm races
2. Check Assertion Failure

1. Insert Instrumentations at compile time

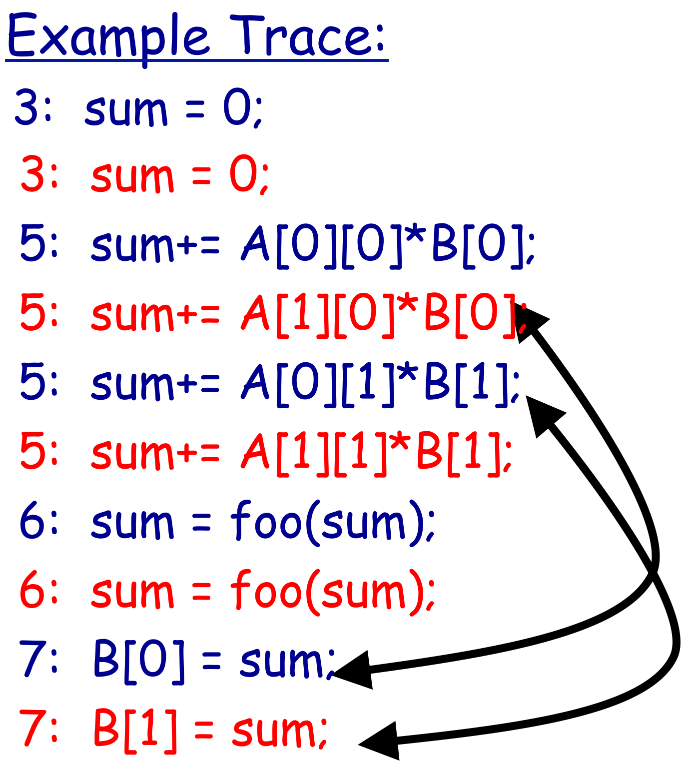
```
              shared int B[N],
              shared int C[N]) {
```

2. Run instrumented program normally -> Trace

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

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

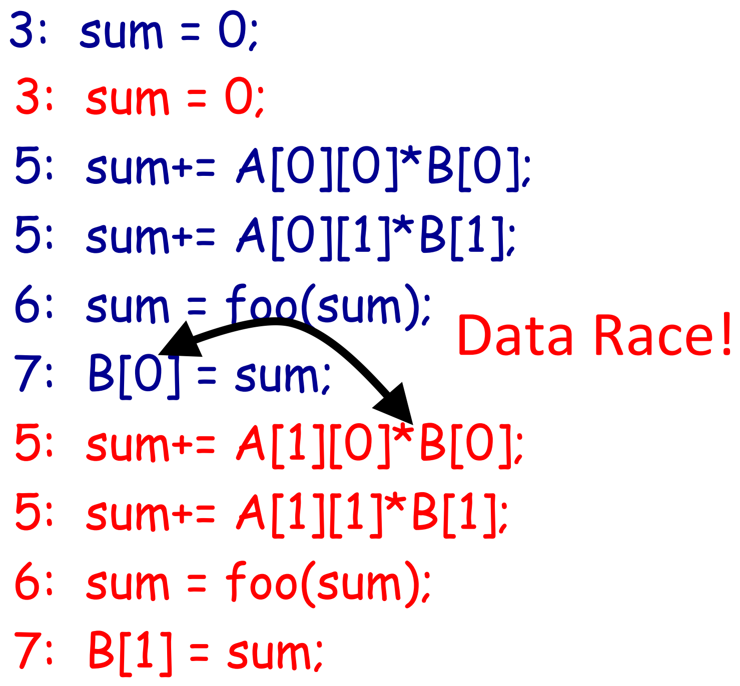
Control Scheduler using knowledge that (5,7) could race

```
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

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;
```

Data Race!

Goal. Generate this execution

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

Challenges for Exascale

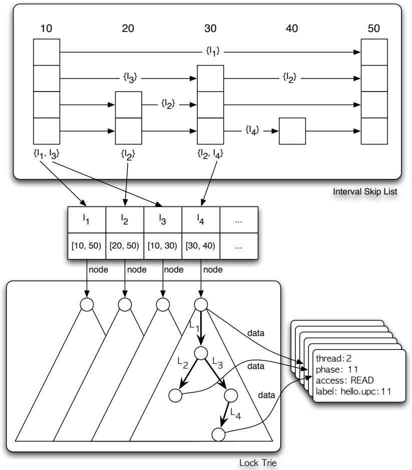
- Java and pthreads programs
  - Synchronization with locks and condition variables
  - Single node
- Exascale has different programming models
  - Large scale
  - Bulk communication
  - Collective operations with data movement
  - Memory consistency
  - Distributed shared memory
- Cannot use centralized dynamic analyses
- Cannot instrument and track every statement

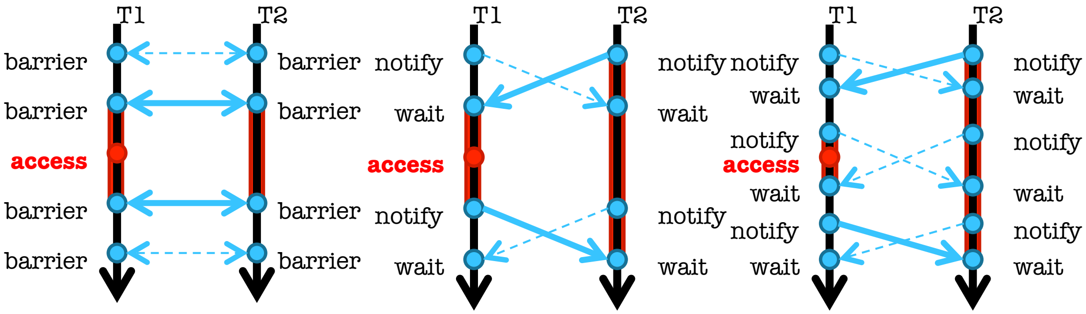
Further Challenges!

- 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, Shared Distributed Memory
  - Need to correlate non-determinism across paradigms

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





Found a Bug. Now what?

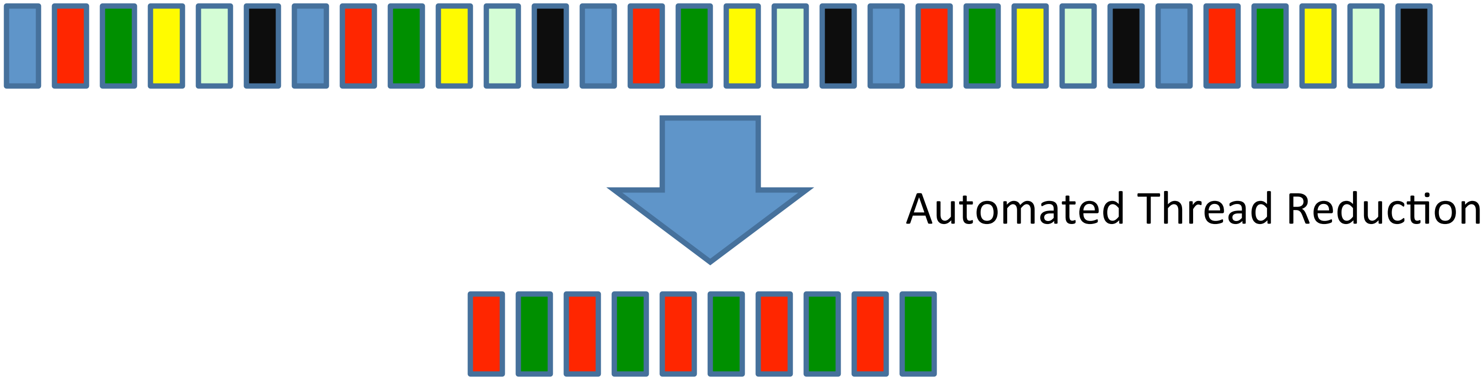
II. Debugging Tools

Debugging project I

Detect bug with fewer threads and fewer context switches

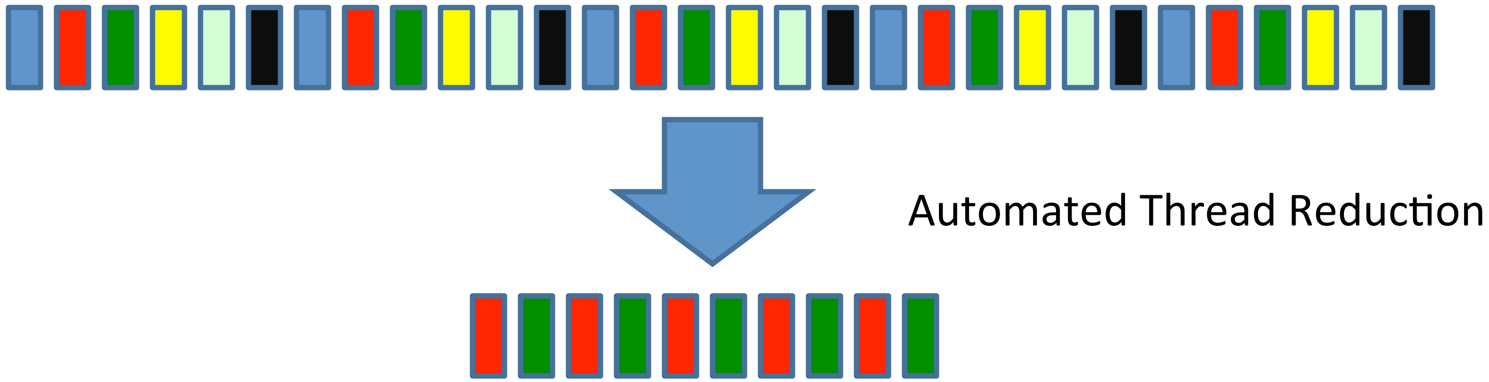
Found a Bug. Now what?

Goal 3: Show a buggy trace having fewer threads



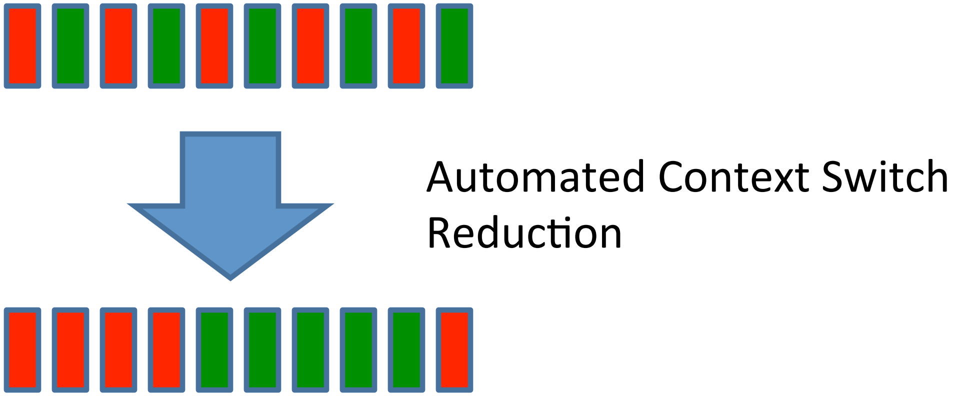
Found a Bug. Now what?

Goal 3: Show a buggy trace having fewer threads



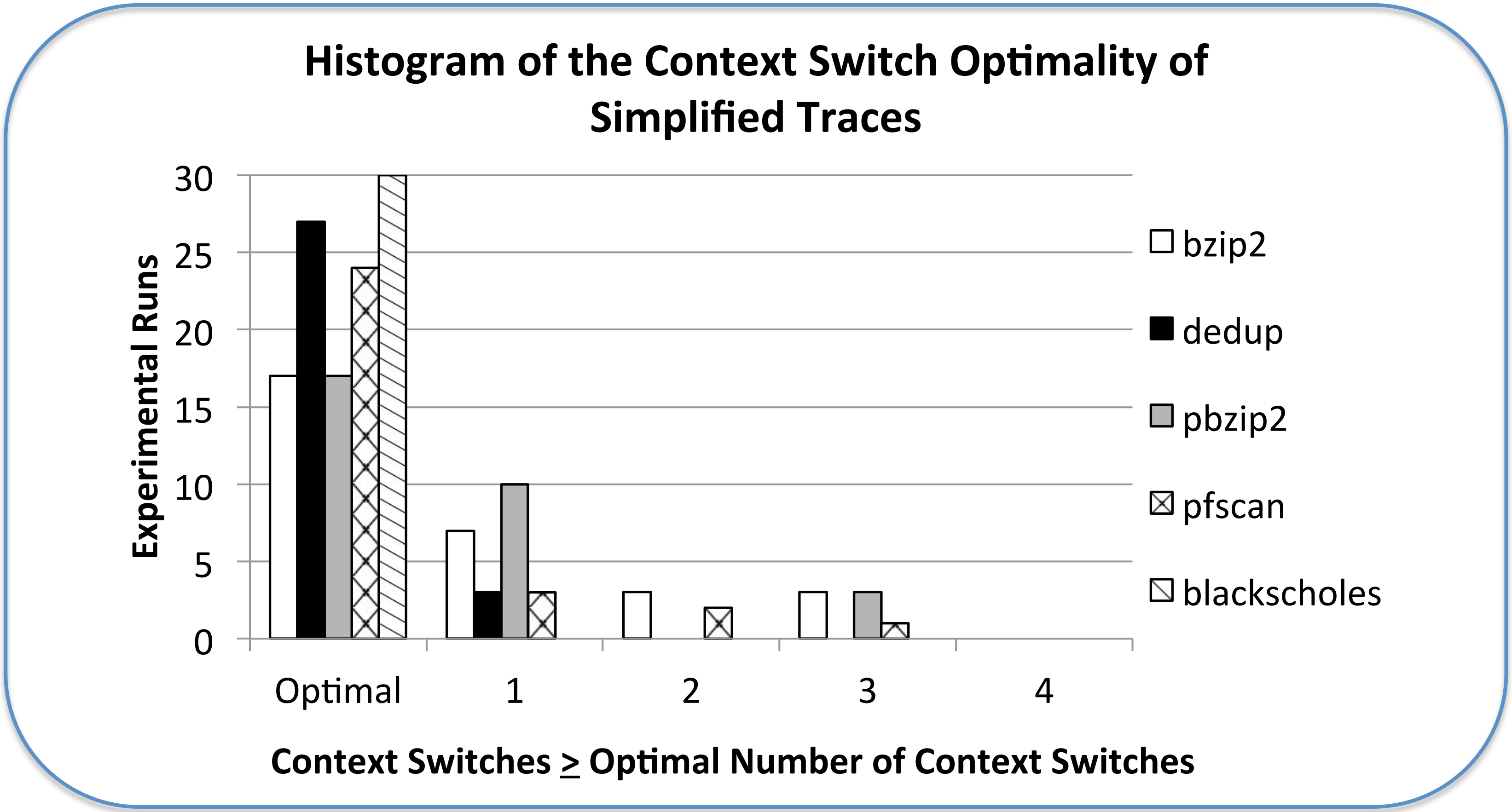
Automated Thread Reduction

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.

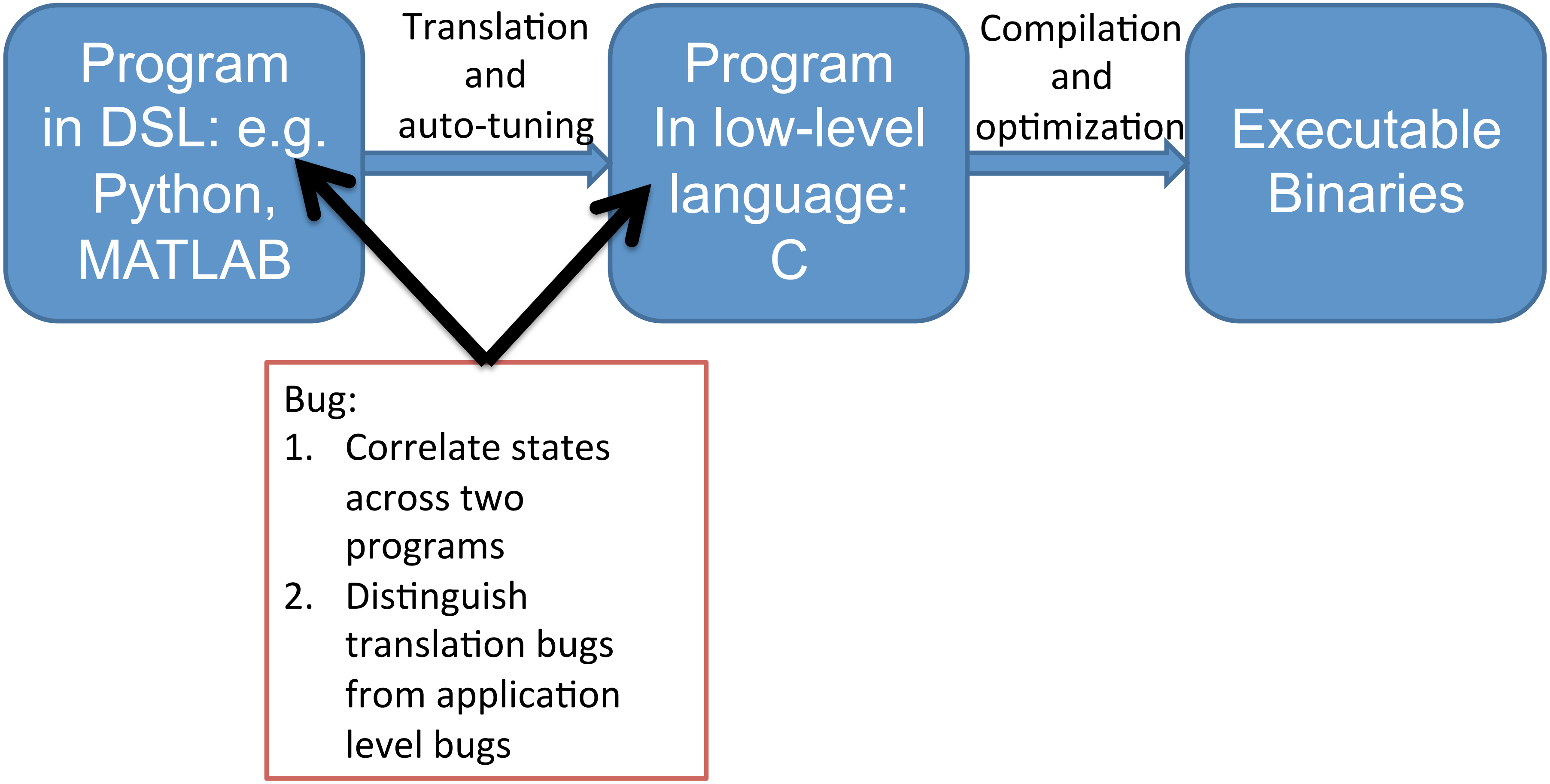
Small model hypothesis

- 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

Debugging project II

Two-level debugging of DSLs.
Correlate program state across
program versions

Two level debugging for DSLs



Program in DSL: e.g. Python, MATLAB

Translation and auto-tuning

Program In low-level language: C

Compilation and optimization

Executable Binaries

Bug:

1. Correlate states across two programs
2. Distinguish translation bugs from application level bugs

Debugging project III

Find floating point anomalies. Recommend safe reduction of precision.

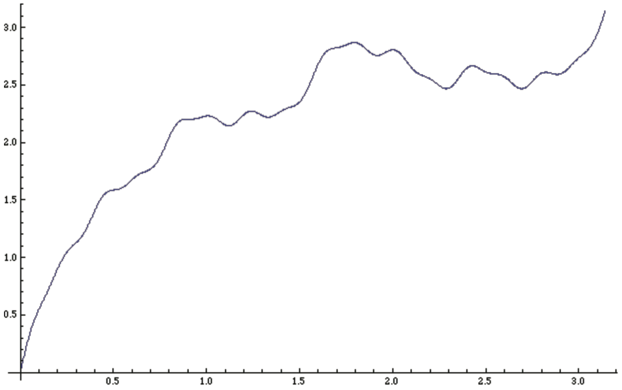
Floating point Debugging: Why do we care?

- Usage of floating point programs has been growing rapidly
  - HPC
  - Cloud, games, graphics, finance, speech, signal processing
- Most programmers are not expert in floating-point!
  - Why not use highest precision everywhere
- High precision wastes
  - Energy
  - Time
  - Storage

FP Debugging Problem 1: Reduce unnecessary precision

- Consider the problem of finding the arc length of the function

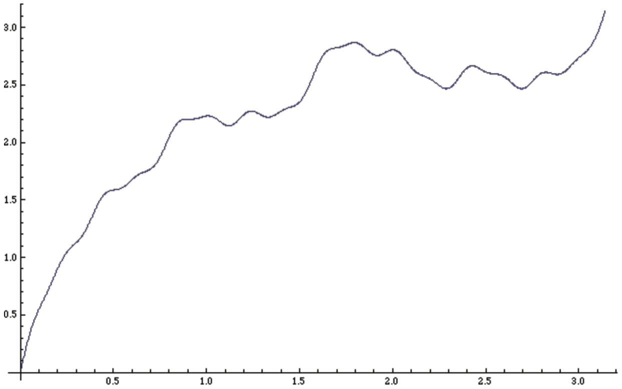
$$g(x) = x + \sum_{0 \leq n \leq 5} 2^{-k} \sin(2^k x)$$



FP Debugging Problem 1: Reduce unnecessary precision

- Consider the problem of finding the arc length of the function

$$g(x) = x + \sum_{0\leq n\leq 5} 2^{-k}\sin(2^k x)$$

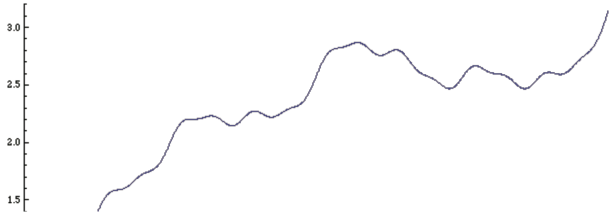


Precision	Slowdown	Result	
double-double	20X	5.795776322413031	✓
double	1X	5.795776322412856	✗
summation variable is double-double	< 2X	5.795776322413031	✓

FP Debugging Problem 1: Reduce unnecessary precision

- Consider the problem of finding the arc length of the function

$$g(x) = x + \sum_{0 \leq n \leq 5} 2^{-k} \sin(2^k x)$$



How can we find a minimal set of code fragments whose precision must be high?

Precision	Slowdown	Result	
double-double	20X	5.79577632241 3031	✓
double	1X	5.79577632241 2856	✗
summation variable is double-double	< 2X	5.79577632241 3031	✓

FP Debugging Problem 2: Detect Inaccuracy and Anomaly

Precondition: x[i] > 0 for all i

```
float f(float * x, size_t nel, float * y) {
        float sum = 0.0;
        for (int i = 0; i < nel; i ++) {
                sum = sum + x[i]*x[i];
        sum = sqrt(sum);
        for (i = 0; i < nel; i++) {
                y[i] = x[i]*x[i]/sum;
        }
}
```

Can lead to NaN even when given strictly positive inputs.

Can we generate such an input?

What we can do?

- We can reduce precision “safely”
  - reduce power, improve performance, get better answer
- Automated testing and debugging techniques
  - To recommend “precision reduction”
  - Formal proof of “safety” can be replaced by concolic testing
- Approach: automate previously hand-made debugging
  - Concolic testing
  - Delta debugging [Zeller et al.]

Implementation

- Prototype implementation for C programs
  - Uses CIL compiler framework
  - http://perso.univ-perp.fr/guillaume.revy/index.php?page=debugging
- Future plans
  - Build on top of LLVM compiler framework

Summary

Detect Data Races

Bug Simplification

Reproducibility in FP programs

Precision Reduction

2-Level Debugging for DSLs

Concolic Testing for Input Generation

Partial restart for debugging

Handle CUDA, OpenMP

Potential Collaboration

- Dynamic analyses to find bugs - dynamic parallelism, unstructured parallelism, shared memory
  - DEGAS, XPRESS, Traleika Glacier
- Floating point debugging
  - Co-design centers
- 2-level debugging
  - DTEC

Conclusions

- Build testing tools
  - Close to what programmers use
  - Hide formal methods and program analysis under testing
- If you are not obsessed with formal correctness
  - Testing and debugging can help you solve these problems with high confidence