EECS
Electrical Engineering and Computer Sciences

BERKELEY LAB
Lawrence Berkeley National Laboratory

BERKELEY PAR LAB

PARALLEL COMPUTING LABORATORY

Efficient Data Race Detection for Distributed Memory Parallel Programs

Chang-Seo Park and Koushik Sen
University of California Berkeley

Paul Hargove and Costin Iancu
Lawrence Berkeley Laboratory

Supercomputing 2011
Seattle, WA 11/16/2011

Current State of Parallel Programming

- Parallelism everywhere!
  - Top supercomputer has 500K+ cores
  - Quad-core standard on desktop / laptops
  - Dual-core smartphones
- Parallelism and concurrency make programming harder
  - Scheduling non-determinism may cause subtle bugs
- But, limited usage of testing and correctness tools
  - We like hero programmers
  - Hero programmers can find bugs (in sequential code)
  - Tools are hard to find and use

EECS

BERKELEY PAR LAB

BERKELEY LAB

New Landscape for HPC

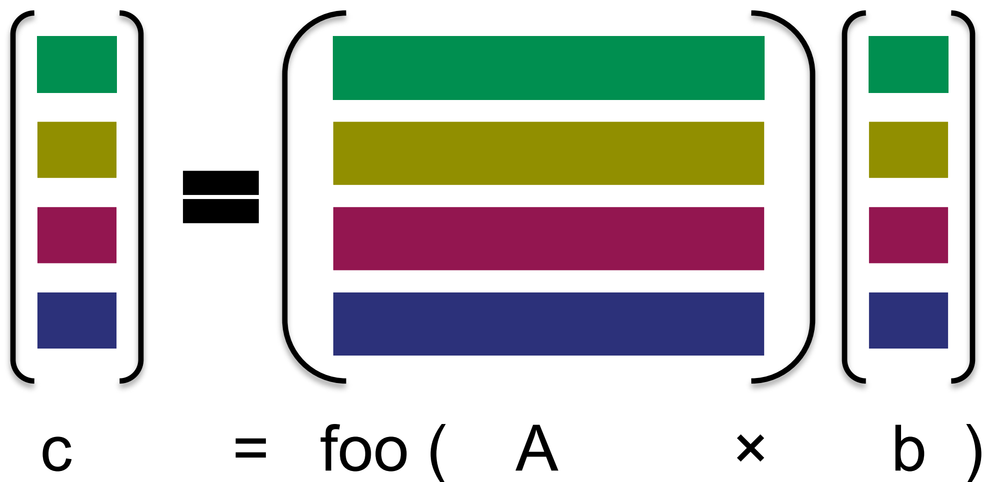
- Shared memory for scalability and utilization
  - Hybrid programming models: MPI + OpenMP
  - PGAS: UPC, CAF, X10, etc.
- Asynchronous access to shared data likely to cause bugs
- Unified Parallel C (UPC)
  - Parallel extensions to ISO C99 standard for shared and distributed memory hardware
  - Single Program Multiple Data (SPMD) + Partitioned Global Address Space (PGAS)
- Shared memory concurrency
  - Transparent access using pointers to shared data (array)
  - Bulk transfers with `memcpy, memput, memget`
  - Fine-grained (lock) and bulk (barrier) synchronization

Outline

- Introduction
- **Example Bug and Motivation**
- Efficient Data Race Detection with Active Testing
  - Prediction phase
  - Confirmation phase
  - Optimizations for scalability
- Experimental Results
- Bugs Found
- Conclusion

Example Parallel Program

- Simple matrix vector multiply and apply F



Example Parallel Program in UPC

```
1: void matvec(shared [N] double A[N][N],
          shared double B[N],
          shared double C[N]) {
2:   double sum[N];
3:   upc_forall(int i = 0; i < N; i++; &C[i]) {
4:     sum[i] = 0;
5:     for(int j = 0; j < N; j++)
6:       sum[i] += A[i][j] * B[j];
7:   }
8:   upc_forall(int i = 0; i < N; i++; &C[i]) {
9:     C[i] = foo(sum[i]);
10:  }
11:} // assert (C == foo(A*B))
```

$$\begin{pmatrix} ? \\ ? \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 1 \end{pmatrix} \begin{pmatrix} 1 \\ 1 \end{pmatrix}$$

C A B

- foo is an expensive function

EECS

BERKELEY PAR LAB   BERKELEY LAB

Example Parallel Program in UPC

```
1: void matvec(shared [N] double A[N][N],
           shared double B[N],
           shared double C[N]) {
 2:   double sum[N];
 3:   upc_forall(int i = 0; i < N; i++; &C[i]) {
 4:     sum[i] = 0;
 5:     for(int j = 0; j < N; j++)
 6:       sum[i] += A[i][j] * B[j];
 7:   }
 8:   upc_forall(int i = 0; i < N; i++; &C[i]) {
 9:     C[i] = foo(sum[i]);
10:   }
11:} // assert (C == foo(A*B))
```

foo(x) = x

$$\begin{pmatrix} 2 \\ 2 \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 1 \end{pmatrix} \begin{pmatrix} 1 \\ 1 \end{pmatrix}$$

C A B

- foo is an expensive function

UPC Example: Problem?

```
1: void matvec(shared [N] double A[N][N],
           shared double B[N],
           shared double C[N]) {
2:   double sum[N];
3:   upc_forall(int i = 0; i < N; i++; &C[i]) {
4:     sum[i] = 0;
5:     for(int j = 0; j < N; j++)
6:       sum[i] += A[i][j] * B[j];
7:   }
8:   upc_forall(int i = 0; i < N; i++; &C[i]) {
9:     C[i] = foo(sum[i]);
10:   }
11:} // assert (C == foo(A*B))
```

- foo is an expensive function

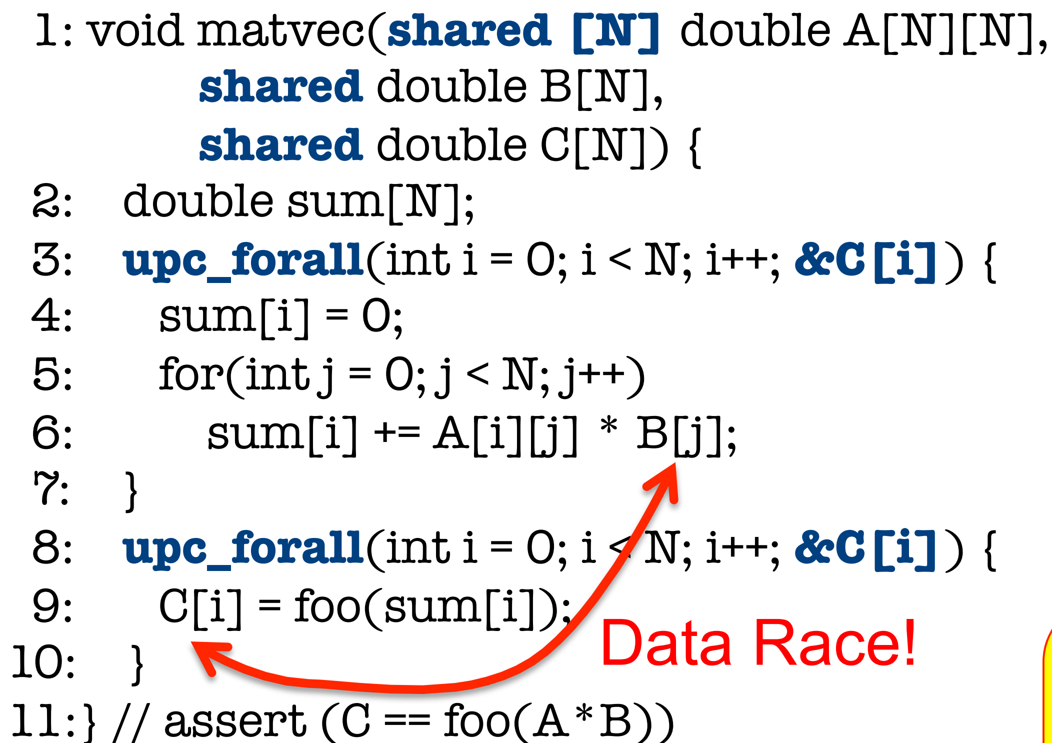
foo(x) = x

$$\begin{pmatrix} 2 \\ 2 \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 1 \end{pmatrix} \begin{pmatrix} 1 \\ 1 \end{pmatrix}$$

C A B

No apparent bug in this program.

UPC Example: Data Race



foo(x) = x

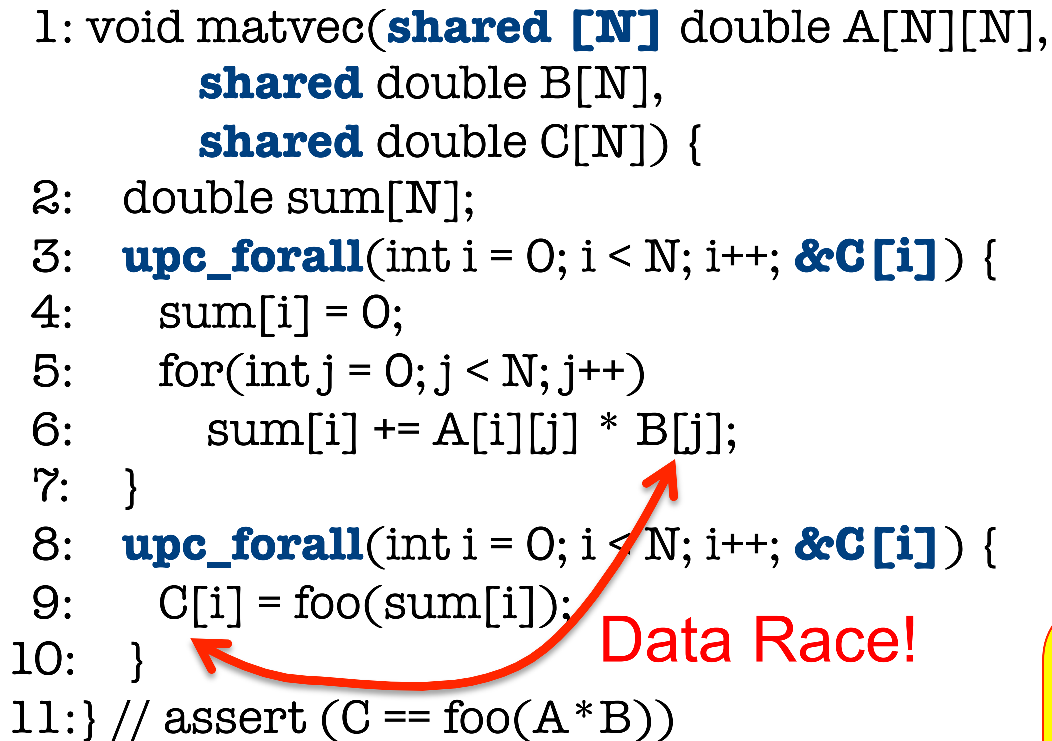
$$\begin{pmatrix} 1 \\ 1 \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 1 \end{pmatrix} \begin{pmatrix} 1 \\ 1 \end{pmatrix}$$

B A B

No apparent bug in this program. But, if we call matvec(A,B,B)?

- foo is an expensive function

UPC Example: Data Race



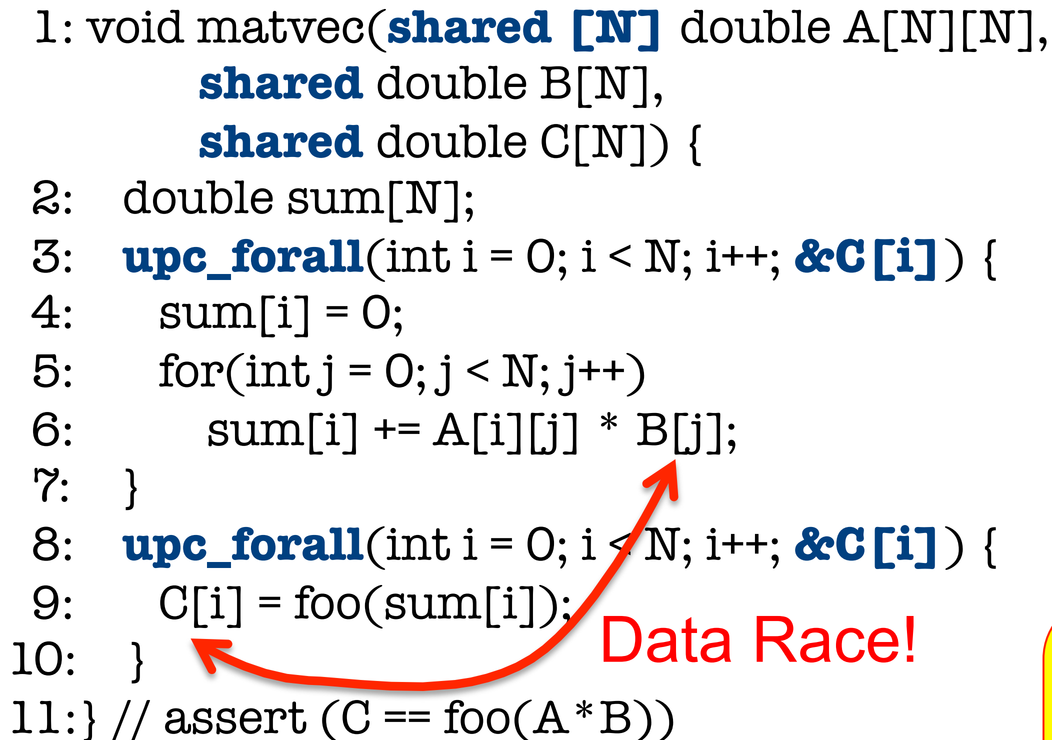
$$foo(x) = x$$

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

No apparent bug in this program. But, if we call matvec(A,B,B)?

- foo is an expensive function

UPC Example: Data Race



$foo(x) = x$

$$\begin{bmatrix} 2 \\ 3 \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} 2 \\ 3 \end{bmatrix}$$

B A B

No apparent bug in this program. But, if we call matvec(A,B,B)?

- foo is an expensive function

UPC Example: Trace

```
1: void matvec(shared [N] double A[N][N],
            shared double B[N],
            shared double C[N]) {
2:    double sum[N];
3:    upc_forall(int i = 0; i < N; i++; &C[i]) {
4:      sum[i] = 0;
5:      for(int j = 0; j < N; j++)
6:        sum[i] += A[i][j] * B[j];
7:    }
8:    upc_forall(int i = 0; i < N; i++; &C[i]) {
9:      C[i] = foo(sum[i]);
10:   }
11:} // assert (C == foo(A*B))
```

- foo is an expensive function

Example Trace:

```
4:  sum[0] = 0;
4:  sum[1] = 0;
6:  sum[0]+= A[0][0]*B[0];
6:  sum[1]+= A[1][0]*B[0];
6:  sum[0]+= A[0][1]*B[1];
6:  sum[1]+= A[1][1]*B[1];
9:  B[0] = foo(sum[0]);
9:  B[1] = foo(sum[1]);
```

Data Race?

UPC Example: Trace

Would be nice to have a trace exhibiting the data race

```
                                      ],
      shared double C[N]) {
 2:  double sum[N];
 3:  upc_forall(int i = 0; i < N; i++; &C[i]) {
 4:    sum[i] = 0;
 5:    for(int j = 0; j < N; j++)
 6:      sum[i] += A[i][j] * B[j];
 7:  }
 8:  upc_forall(int i = 0; i < N; i++; &C[i]) {
 9:    C[i] = foo(sum[i]);
10:  }
11:} // assert (C == foo(A*B))
```

- foo is an expensive function

Example Trace:

```
4: sum[0] = 0;
4: sum[1] = 0;
6: sum[0]+= A[0][0]*B[0];
6: sum[0]+= A[0][1]*B[1];
6: sum[1]+= A[1][0]*B[0];
9: B[0] = foo(sum[0]);
6: sum[1]+= A[1][1]*B[1];
9: B[1] = foo(sum[1]);
```

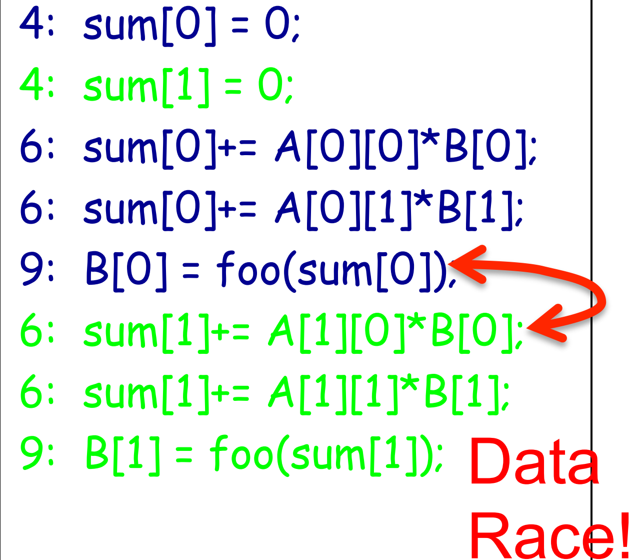
Data Race!

UPC Example: Trace

Would be nice to have a trace exhibiting the assertion failure

```
                                                  ],
      shared double C[N]) {
 2:  double sum[N];
 3:  upc_forall(int i = 0; i < N; i++; &C[i]) {
 4:    sum[i] = 0;
 5:    for(int j = 0; j < N; j++)
 6:      sum[i] += A[i][j] * B[j];
 7:  }
 8:  upc_forall(int i = 0; i < N; i++; &C[i]) {
 9:    C[i] = foo(sum[i]);
10:  }
11:} // assert (C == foo(A*B))
```

- foo is an expensive function



Desiderata

- 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 (or some other visible artifact)

Active Testing

- 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 (or some other visible artifact)
- Leverage program analysis to make testing quickly find real concurrency bugs
  - Phase 1: Use imprecise static or dynamic program analysis to find *bug patterns* where a potential concurrency bug can happen (Race Detector)
  - Phase 2: *Directed testing* to confirm potential bugs as real (Race Tester)

Active Testing: Phase 1

```
1: void matvec(shared [N] double A[N][N],
           shared double B[N],
           shared double C[N]) {
 2:   double sum[N];
 3:   upc_forall(int i = 0; i < N; i++; &C[i]) {
 4:     sum[i] = 0;
 5:     for(int j = 0; j < N; j++)
 6:        sum[i] += A[i][j] * B[j];
 7:   }
 8:   upc_forall(int i = 0; i < N; i++; &C[i]) {
 9:     C[i] = foo(sum[i]);
10:   }
11:} // assert (C == foo(A*B))
```

Active Testing: Phase 1

```
 1: void matvec(shared [N] double A[N][N],
            shared double B[N],
            shared double C[N]) {
 2:   double sum[N];
 3:   upc_forall(int i = 0; i < N; i++; &C[i]) {
 4:     sum[i] = 0;
 5:     for(int j = 0; j < N; j++)
 6:       sum[i] += A[i][j] * B[j];
 7:   }
 8:   upc_forall(int i = 0; i < N; i++; &C[i]) {
 9:     C[i] = foo(sum[i]);
10:   }
11:} // assert (C == foo(A*B))
```

1. Insert instrumentation at compile time

EECS

BERKELEY PAR LAB BERKELEY LAB

Active Testing: Phase 1

```
1: void matvec(shared [N] double A[N][N],
          shared double B[N],
          shared double C[N]) {
 2:   double sum[N];
 3:   upc_forall(int i = 0; i < N; i++; &C[i]) {
 4:     sum[i] = 0;
 5:     for(int j = 0; j < N; j++)
 6:       sum[i] += A[i][j] * B[j];
 7:   }
 8:   upc_forall(int i = 0; i < N; i++; &C[i]) {
 9:     C[i] = foo(sum[i]);
10:   }
11:} // assert (C == foo(A*B))
```

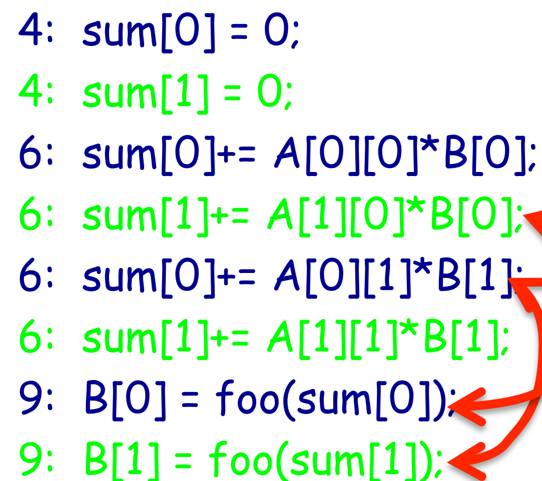
Generated Trace:

```
4: sum[0] = 0;
4: sum[1] = 0;
6: sum[0]+= A[0][0]*B[0];
6: sum[1]+= A[1][0]*B[0];
6: sum[0]+= A[0][1]*B[1];
6: sum[1]+= A[1][1]*B[1];
9: B[0] = foo(sum[0]);
9: B[1] = foo(sum[1]);
```

2. Run instrumented program normally and obtain trace

Active Testing: Phase 1

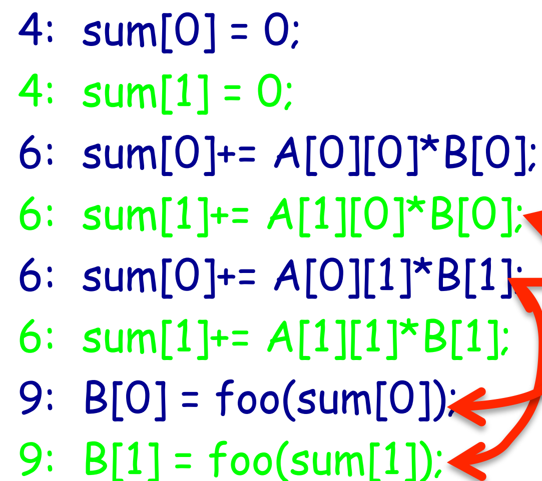
```
1: void matvec(shared [N] double A[N][N],
          shared double B[N],
          shared double C[N]) {
 2:   double sum[N];
 3:   upc_forall(int i = 0; i < N; i++; &C[i]) {
 4:     sum[i] = 0;
 5:     for(int j = 0; j < N; j++)
 6:       sum[i] += A[i][j] * B[j];
 7:   }
 8:   upc_forall(int i = 0; i < N; i++; &C[i]) {
 9:     C[i] = foo(sum[i]);
10:   }
11:} // assert (C == foo(A*B))
```



3. Algorithm detects data races

Active Testing: Phase 1

```
1: void matvec(shared [N] double A[N][N],
          shared double B[N],
          shared double C[N]) {
 2:   double sum[N];
 3:   upc_forall(int i = 0; i < N; i++; &C[i]) {
 4:     sum[i] = 0;
 5:     for(int j = 0; j < N; j++)
 6:       sum[i] += A[i][j] * B[j];
 7:   }
 8:   upc_forall(int i = 0; i < N; i++; &C[i]) {
 9:     C[i] = foo(sum[i]);
10:   }
11:} // assert (C == foo(A*B))
```



3. ***Potential*** race between statements 6 and 9

Active Testing: Phase 2

```
1: void matvec(shared [N] double A[N][N],
          shared double B[N],
          shared double C[N]) {
2:    double sum[N];
3:    upc_forall(int i = 0; i < N; i++; &C[i]) {
4:       sum[i] = 0;
5:       for(int j = 0; j < N; j++)
6:          sum[i] += A[i][j] * B[j];
7:    }
8:    upc_forall(int i = 0; i < N; i++; &C[i]) {
9:       C[i] = foo(sum[i]);
10:   }
11:} // assert (C == foo(A*B))
```

Goal 1. Confirm races

Goal 2. Create assertion failure

Active Testing: Phase 2

```
1: void matvec(shared [N] double A[N][N],
          shared double B[N],
          shared double C[N]) {
 2:    double sum[N];
 3:    upc_forall(int i = 0; i < N; i++; &C[i]) {
 4:      sum[i] = 0;
 5:      for(int j = 0; j < N; j++)
 6:         sum[i] += A[i][j] * B[j];
 7:    }
 8:    upc_forall(int i = 0; i < N; i++; &C[i]) {
 9:      C[i] = foo(sum[i]);
10:    }
11:} // assert (C == foo(A*B))
```

Generate this execution:

```
4: sum[0] = 0;
4: sum[1] = 0;
6: sum[0]+= A[0][0]*B[0];
6: sum[0]+= A[0][1]*B[1];
9: B[0] = foo(sum[0]);
6: sum[1]+= A[1][0]*B[0];
6: sum[1]+= A[1][1]*B[1];
9: B[1] = foo(sum[1]);
```

Data Race!

Goal 1. Confirm races
Goal 2. Create assertion failure

Active Testing: Phase 2

```
1: void matvec(shared [N] double A[N][N],
            shared double B[N],
            shared double C[N]) {
 2:   double sum[N];
 3:   upc_forall(int i = 0; i < N; i++; &C[i]) {
 4:     sum[i] = 0;
 5:     for(int j = 0; j < N; j++)
 6:       sum[i] += A[i][j] * B[j];
 7:   }
 8:   upc_forall(int i = 0; i < N; i++; &C[i]) {
 9:     C[i] = foo(sum[i]);
10:   }
11:} // assert (C == foo(A*B))
```

Control scheduler knowing that (6,9) could race

Trace:

4: sum[1] = 0;

Active Testing: Phase 2

```
1: void matvec(shared [N] double A[N][N],
          shared double B[N],
          shared double C[N]) {
 2:   double sum[N];
 3:   upc_forall(int i = 0; i < N; i++; &C[i]) {
 4:     sum[i] = 0;
 5:     for(int j = 0; j < N; j++)
 6:       sum[i] += A[i][j] * B[j];
 7:   }
 8:   upc_forall(int i = 0; i < N; i++; &C[i]) {
 9:     C[i] = foo(sum[i]);
10:   }
11:} // assert (C == foo(A*B))
```

Control scheduler knowing that (6,9) could race

Trace:

4: sum[1] = 0;

4: sum[0] = 0;

Active Testing: Phase 2

```
1: void matvec(shared [N] double A[N][N],
          shared double B[N],
          shared double C[N]) {
2:    double sum[N];
3:    upc_forall(int i = 0; i < N; i++; &C[i]) {
4:       sum[i] = 0;
5:       for(int j = 0; j < N; j++)
6:          sum[i] += A[i][j] * B[j];
7:    }
8:    upc_forall(int i = 0; i < N; i++; &C[i]) {
9:       C[i] = foo(sum[i]);
10:   }
11:} // assert (C == foo(A*B))
```

Trace:

```
4:  sum[1] = 0;
4:  sum[0] = 0;
6:  sum[1]+= A[1][0]*B[0];
```

Control scheduler knowing that (6,9) could race

Active Testing: Phase 2

```
1: void matvec(shared [N] double A[N][N],
         shared double B[N],
         shared double C[N]) {
2:   double sum[N];
3:   upc_forall(int i = 0; i < N; i++; &C[i]) {
4:     sum[i] = 0;
5:     for(int j = 0; j < N; j++)
6:       sum[i] += A[i][j] * B[j];
7:   }
8:   upc_forall(int i = 0; i < N; i++; &C[i]) {
9:     C[i] = foo(sum[i]);
10:   }
11:} // assert (C == foo(A*B))
```

Trace:

4: sum[1] = 0;

4: sum[0] = 0;

Postponed: { 6: sum[1]+= A[1][0]*B[0]; }

Do not postpone if there is a deadlock

Control scheduler knowing that (6,9) could race

Active Testing: Phase 2

```
1: void matvec(shared [N] double A[N][N],
          shared double B[N],
          shared double C[N]) {
2:    double sum[N];
3:    upc_forall(int i = 0; i < N; i++; &C[i]) {
4:      sum[i] = 0;
5:      for(int j = 0; j < N; j++)
6:        sum[i] += A[i][j] * B[j];
7:    }
8:    upc_forall(int i = 0; i < N; i++; &C[i]) {
9:      C[i] = foo(sum[i]);
10:   }
11:} // assert (C == foo(A*B))
```

Trace:

4: sum[1] = 0;

4: sum[0] = 0;

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

Do not postpone
if there is a deadlock

Control scheduler knowing
that (6,9) could race

Postponed: { 6: sum[1]+= A[1][0]*B[0]; }

Active Testing: Phase 2

```
1: void matvec(shared [N] double A[N][N],
          shared double B[N],
          shared double C[N]) {
2:    double sum[N];
3:    upc_forall(int i = 0; i < N; i++; &C[i]) {
4:       sum[i] = 0;
5:       for(int j = 0; j < N; j++)
6:          sum[i] += A[i][j] * B[j];
7:    }
8:    upc_forall(int i = 0; i < N; i++; &C[i]) {
9:       C[i] = foo(sum[i]);
10:   }
11:} // assert (C == foo(A*B))
```

Trace:

4: sum[1] = 0;

4: sum[0] = 0;

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

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

Postponed: { 6: sum[1]+= A[1][0]*B[0]; }

Do not postpone
if there is a deadlock

Control scheduler knowing
that (6,9) could race

Active Testing: Phase 2

```
1: void matvec(shared [N] double A[N][N],
          shared double B[N],
          shared double C[N]) {
 2:   double sum[N];
 3:   upc_forall(int i = 0; i < N; i++; &C[i]) {
 4:     sum[i] = 0;
 5:     for(int j = 0; j < N; j++)
 6:       sum[i] += A[i][j] * B[j];
 7:   }
 8:   upc_forall(int i = 0; i < N; i++; &C[i]) {
 9:     C[i] = foo(sum[i]);
10:   }
11:} // assert (C == foo(A*B))
```

Trace:

```
4: sum[1] = 0;
4: sum[0] = 0;
6: sum[0]+= A[0][0]*B[0];
6: sum[0]+= A[0][1]*B[1];
9: B[0] = foo(sum[0]);
```

Postponed: { 6: sum[1]+= A[1][0]*B[0]; }

Control scheduler knowing that (6,9) could race

Active Testing: Phase 2

```
1: void matvec(shared [N] double A[N][N],
          shared double B[N],
          shared double C[N]) {
2:    double sum[N];
3:    upc_forall(int i = 0; i < N; i++; &C[i]) {
4:      sum[i] = 0;
5:      for(int j = 0; j < N; j++)
6:        sum[i] += A[i][j] * B[j];
7:    }
8:    upc_forall(int i = 0; i < N; i++; &C[i]) {
9:      C[i] = foo(sum[i]);
10:   }
11:} // assert (C == foo(A*B))
```

Trace:

```
4:  sum[1] = 0;
4:  sum[0] = 0;
6:  sum[0]+= A[0][0]*B[0];
6:  sum[0]+= A[0][1]*B[1];
9:  B[0] = foo(sum[0]);
```

Race? yes

Postponed: { 6: sum[1]+= A[1][0]*B[0]; }

Control scheduler knowing that (6,9) could race

Active Testing: Phase 2

```
1: void matvec(shared [N] double A[N][N],
            shared double B[N],
            shared double C[N]) {
2:    double sum[N];
3:    upc_forall(int i = 0; i < N; i++; &C[i]) {
4:       sum[i] = 0;
5:       for(int j = 0; j < N; j++)
6:          sum[i] += A[i][j] * B[j];
7:    }
8:    upc_forall(int i = 0; i < N; i++; &C[i]) {
9:       C[i] = foo(sum[i]);
10:   }
11:} // assert (C == foo(A*B))
```

Trace:

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

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

Postponed: {}

Control scheduler knowing that (6,9) could race

Active Testing: Phase 2

```
1: void matvec(shared [N] double A[N][N],
          shared double B[N],
          shared double C[N]) {
 2:   double sum[N];
 3:   upc_forall(i                                  {
 4:     sum[i] = 0;
 5:     for(int j = 0; j
 6:       sum[i] += A[i][j] * B[j];
 7:   }
 8:   upc_forall(int i = 0; i < N; i++; &C[i]) {
 9:     C[i] = foo(sum[i]);
10:   }
11:} // assert (C == foo(A*B))
```

Racing Statements Temporally Adjacent

Trace:

```
4:  sum[1] = 0;
4:  sum[0] = 0;
6:  sum[0]+= A[0][0]*B[0];
6:  sum[0]+= A[0][1]*B[1];
9:  B[0] = foo(sum[0]);
6:  sum[1]+= A[1][0]*B[0];
```

Achieved Goal 1:
Confirmed race.

EECS

BERKELEY PAR LAB

BERKELEY LAB

Active Testing: Phase 2

```
1: void matvec(shared [N] double A[N][N],
          shared double B[N],
          shared double C[N]) {
 2:   double sum[N];
 3:   upc_forall(int i = 0; i < N; i++; &C[i]) {
 4:     sum[i] = 0;
 5:     for(int j = 0; j < N; j++)
 6:       sum[i] += A[i][j] * B[j];
 7:   }
 8:   upc_forall(int i = 0; i < N; i++; &C[i]) {
 9:     C[i] = foo(sum[i]);
10:   }
11:} // assert (C == foo(A*B))
```

Achieved Goal 2:
Assertion failure.

Trace:

```
4: sum[1] = 0;
4: sum[0] = 0;
6: sum[0]+= A[0][0]*B[0];
6: sum[0]+= A[0][1]*B[1];
9: B[0] = foo(sum[0]);
6: sum[1]+= A[1][0]*B[0];
6: sum[1]+= A[1][1]*B[1];
9: B[1] = foo(sum[1]);
```

C != foo(A*B)

UPC-Thrille

- Thread Interposition Library and Lightweight Extensions
  - Framework for active testing UPC programs
- Instrument UPC source code at compile time
  - Using macro expansions, add hooks for analyses
- Phase 1: Race detector
  - Observe execution and predict which accesses may potentially have a data race
- Phase 2: Race tester
  - Re-execute program while controlling the scheduler to create actual data race scenarios predicted in phase 1

Phase 1: Checking for Conflicts

- To predict possible races,
  - Need to check all shared accesses for conflicts
  - Collect information through instrumentation

EECS
BERKELEY PAR LAB
BERKELEY LAB
Lawrence Berkeley National Laboratory

Phase 1: Checking for Conflicts

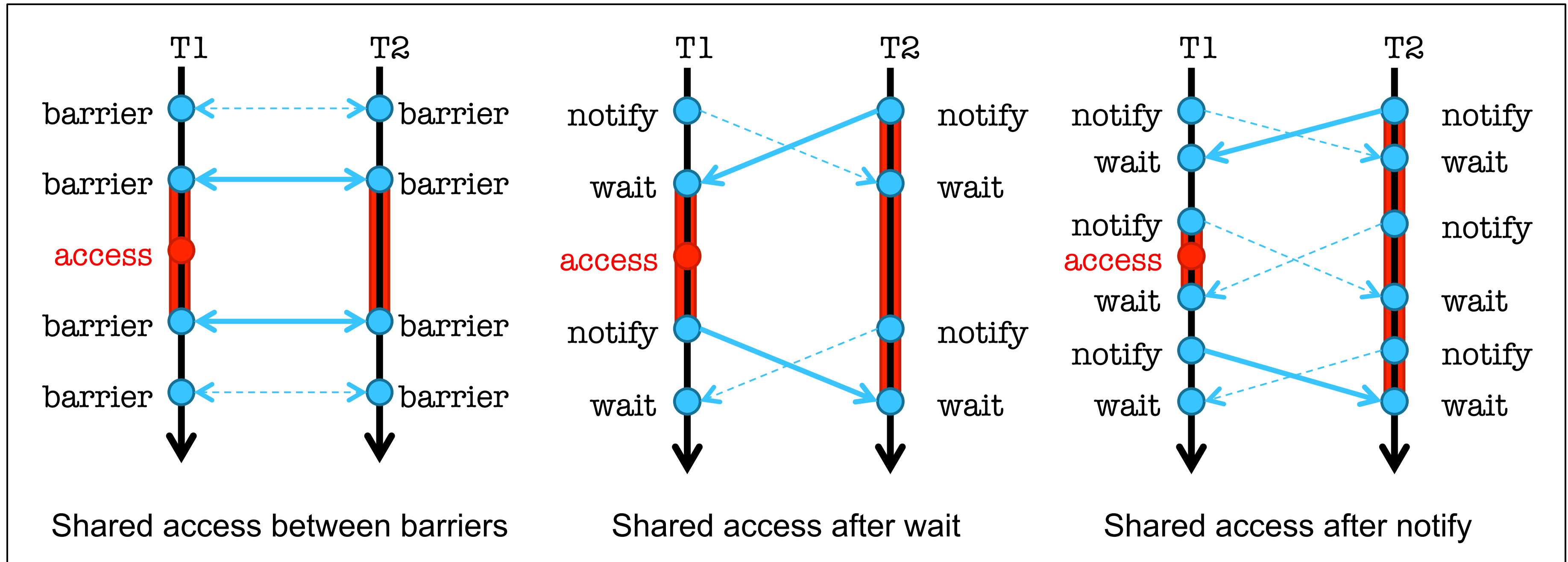
- To predict possible races,
  - Need to check all shared accesses for conflicts
  - Collect information through instrumentation
- Two accesses $e_1 = (m_1, t_1, L_1, a_1, p_1, s_1)$ and $e_2 = (m_2, t_2, L_2, a_2, p_2, s_2)$ are in conflict when
  - memory range overlaps $(m_1 \cap m_2 \neq \varnothing)$
  - accesses from different threads $(t_1 \neq t_2)$
  - no common locks held $(L_1 \cap L_2 = \varnothing)$
  - at least one write $(a_1 = \text{Write} \lor a_2 = \text{Write})$
  - may happen in parallel w.r.t. barriers $(p_1 \parallel p_2)$

  $\Rightarrow (s_1, s_2)$ is a potential data race pair

Differences and Challenges for UPC

- Previous work on Java and pthreads programs
  - Synchronization with locks and condition variables
  - Single node
- UPC has different programming model (SPMD)
  - Large scale
  - Bulk communication (memory regions)
  - Non-blocking communication
  - Collective operations with data movement
  - Different memory consistency model

Differences and Challenges for UPC



- Different memory consistency model
- Optimizations for scalability
  - **Distribute analysis and coalesce queries**
  - Efficient data structures for memory interval reasoning
  - Reduce communication through filtering and sampling

Differences and Challenges for UPC

- Previous w … ms
  - Synchron … s
  - Single no
- UPC has d … MD)
  - Large sca
  - Bulk com
  - Non-bloc
  - Collective
  - Different
- Optimizati
  - Distribute
  - **Efficient … easoning**
  - Reduce … pling

10 20 30 40 50

{I1} {I3} {I2} {I2} {I4}

{I1, I3} {I2} {I2, I4}

Interval Skip List

I1	I2	I3	I4	...
[10, 50)	[20, 50)	[10, 30)	[30, 40)	...

node node node node

L1 L2 L3 L4

data data data

thread: 2
phase: 11
access: READ
label: hello.upc: 11

Lock Trie

EECS

BERKELEY PAR LAB

BERKELEY LAB

Differences and Challenges for UPC

- (Extended) Weaker-than relation
  - Only keep the *least protected* accesses
  - Prune provably redundant accesses [Choi et al ’02]
  - Also reduces superfluous race reports
- $e_1 \sqsubseteq e_2$ (access e_1 is weaker-than e_2) iff
  - larger memory range $(e_1.m \supseteq e_2.m)$
  - accessed by more threads $(e_1.t = * \vee e_1.t = e_2.t)$
  - smaller lockset $(e_1.L \subseteq e_2.L)$
  - weaker access type $(e_1.a = Write \vee e_1.a = e_2.a)$
- Optimizations for scalability
  - Distribute analysis and coalesce queries
  - Efficient data structures for memory interval reasoning
  - **Reduce communication through filtering and sampling**

Results on Single Node

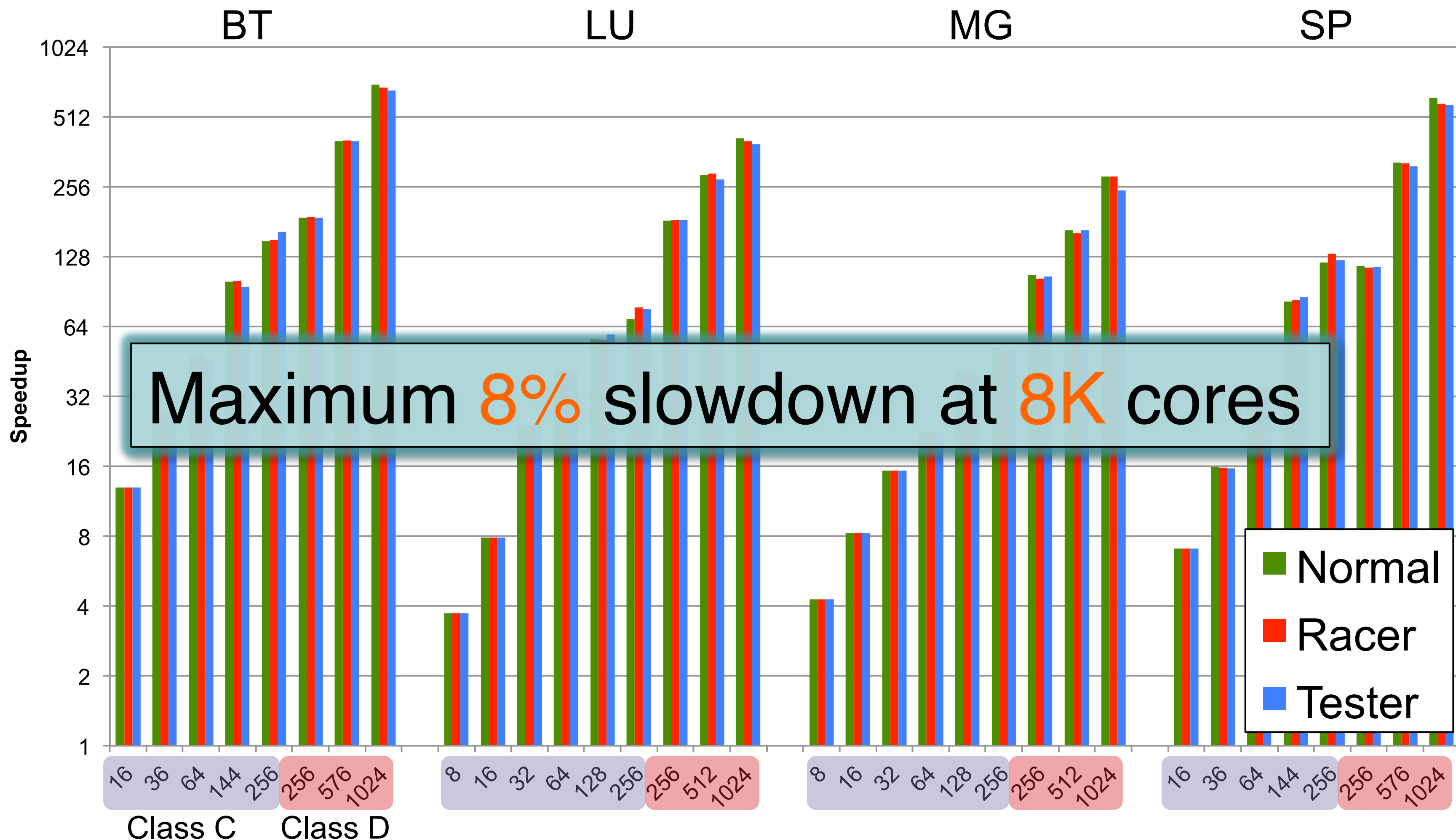
Benchmark	LoC	Runtime	ThrilleRacer		ThrilleTester	
			Overhead	Pot. race	Overhead	Conf. race
guppie	227	2.094s	12%	2	1.7%	2
knapsack	191	2.099s	14.9%	2	1.8%	2
lapalce	123	2.101s	**16.3%**	0	-	-
mcop	358	2.183s	0.7%	0	-	-
psearch	777	2.982s	1.8%	**3**	3.8%	**2**
FT 2.3	2306	8.711s	6.1%	2	**4.8%**	2
CG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

* 4 threads on quad-core 2.66GHz CPU / 8GB RAM

Low overhead: < 20%

Unconfirmed bugs due to custom synchronization

Scalability Results on Franklin*



* Cray XT4 Supercomputer at NERSC

Quad-core 2.3GHz CPU / 8GB RAM per node / Portals interconnect

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
  - Different results on different executions
  - 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
        }
    }
}
```

EECS

BERKELEY PAR LAB BERKELEY LAB Lawrence Berkeley National Laboratory

Conclusion

- Need correctness tool support for HPC
  - Scarcity of effective correctness tools
- Our proposal: Active testing
  - Combine dynamic analysis with testing
  - Low overhead (<10%)
  - Scalable (>8K cores)
  - General algorithm: applicable to other programming models
    - MPI, CUDA, OpenMP

http://upc.lbl.gov/thrille

PGAS @ Booth 124