Towards Incremental Static Race Detection in OpenMP Programs

Bradley Swain, Jeff Huang

Parasol Lab, Texas A&M University

Motivating Goal

- Instantaneous feedback on presence of races in OpenMP programs

```
  1  #pragma omp parallel for
  2  for (int i = 0; i < N; i++) {
⚠ 3     A[i] += i;
  4     A[3] = 0;
  5  }
```

Error: Race between lines 3 and 4 when i=3

Existing Work

- Dynamic and Hybrid Tools
  - Archer [IPDPS’16]
  - Sword [IPDPS’18]

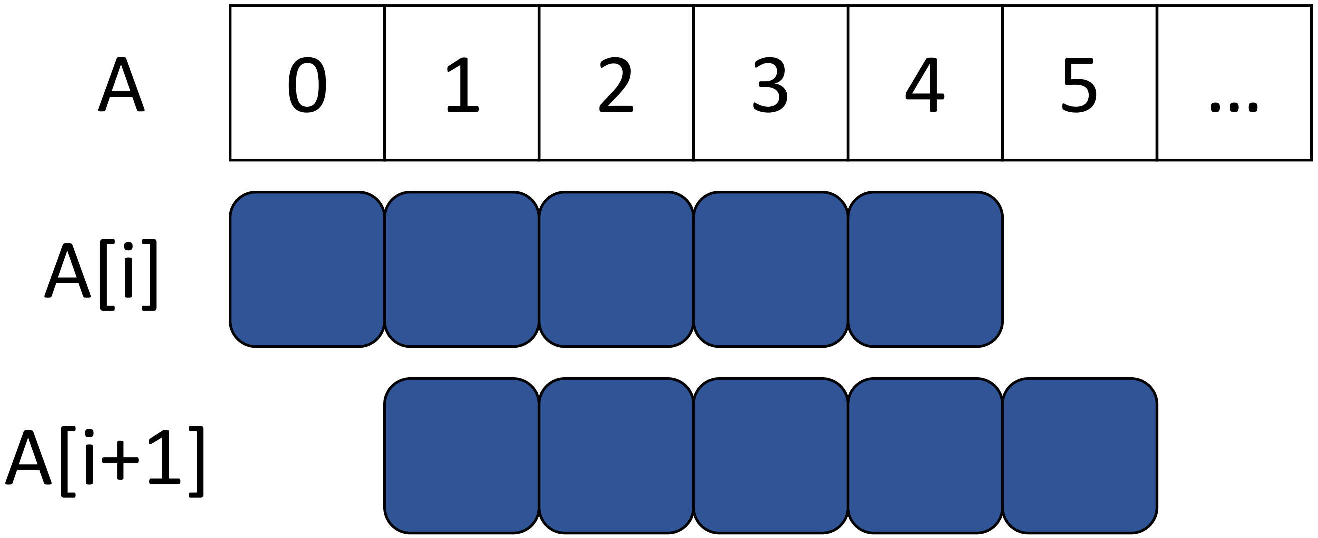
- Static Tools
  - ompVerify [IWOMP’11]
  - PolyOMP [IMPACT’16]

Outline to Achieve Goal

- Array index analysis for simple races
- Phase Graph to extend across synchronizations
- Incrementalization to extend to whole programs

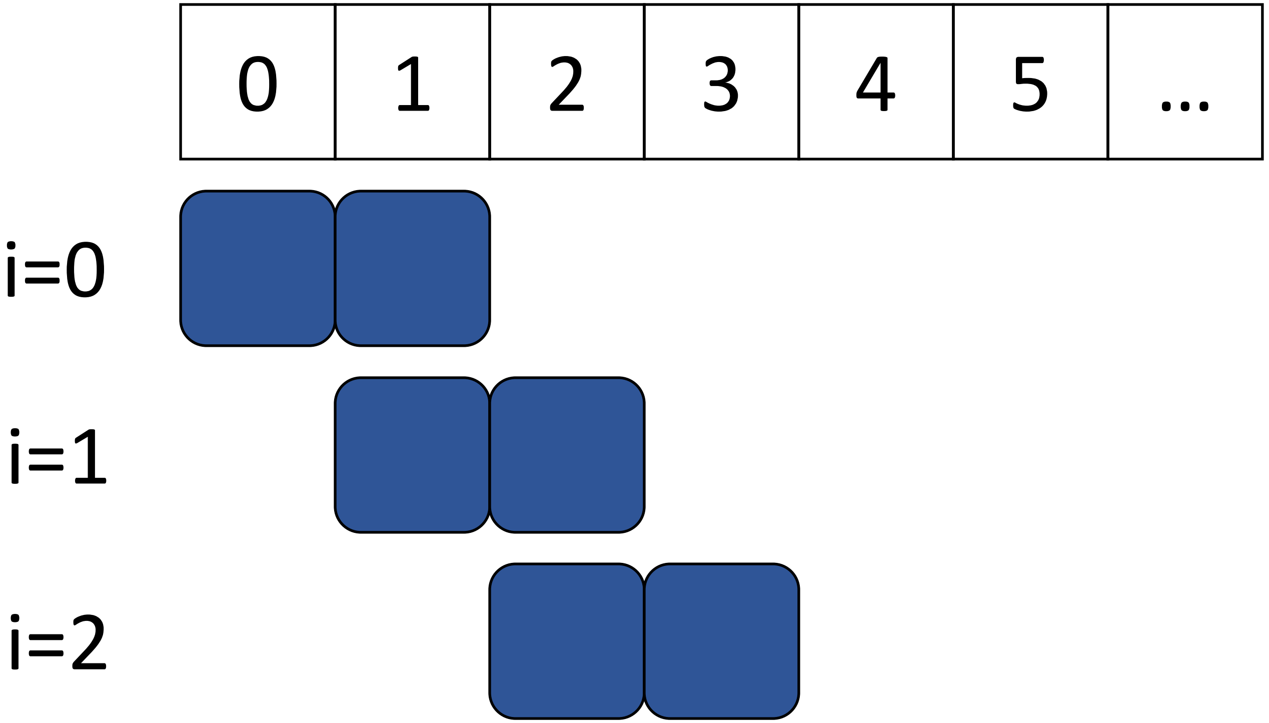
Array Index Analysis

Array accesses can be thought of as a set of elements being accessed



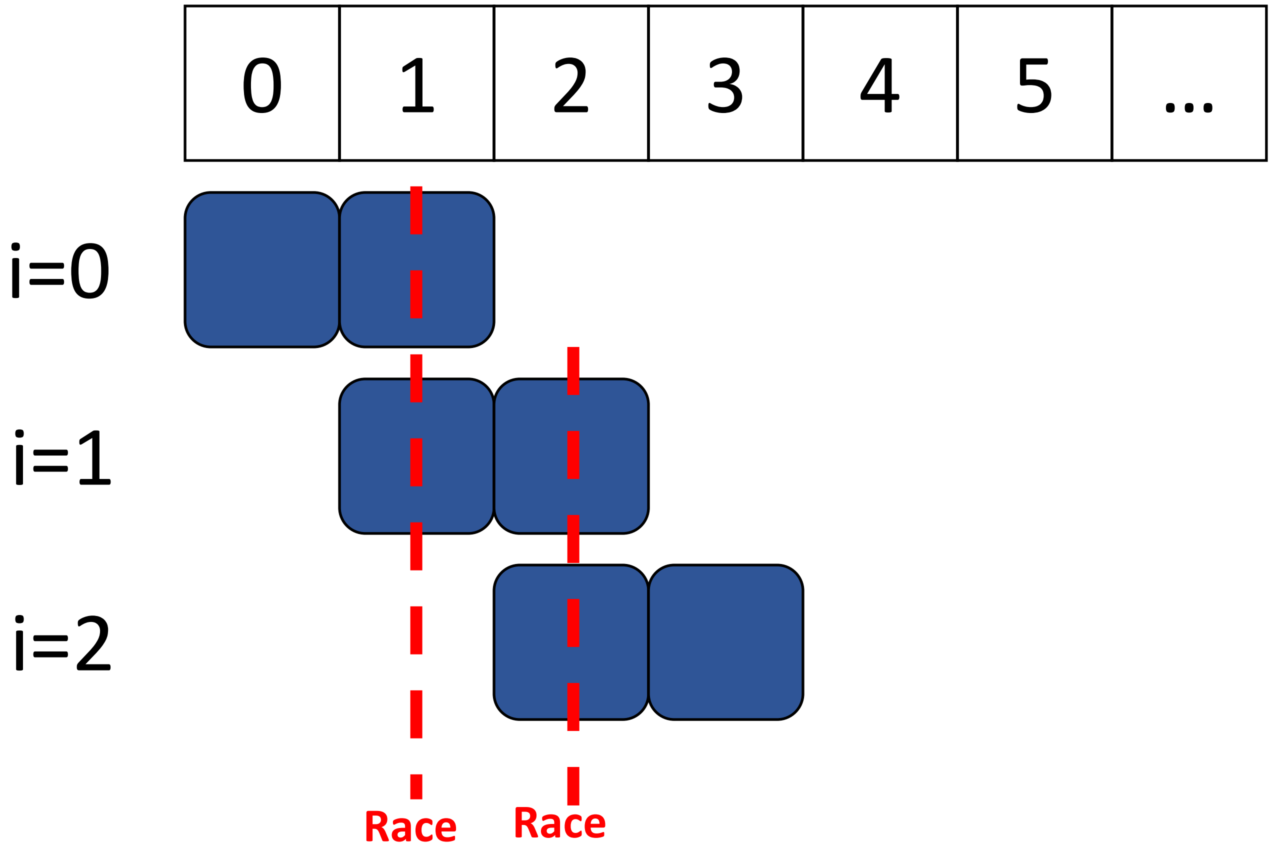
```
#pragma omp parallel for
for (int i = 0; i < 5; i++) {
    A[i + 1] += A[i];
}
```

Array Index Analysis



```
#pragma omp parallel for
for (int i = 0; i < 5; i++) {
    A[i + 1] += A[i];
}
```

Array Index Analysis



Threads are uniquely identified by i

Access sets are created per thread

Overlapping R/W or W/W accesses represent a race

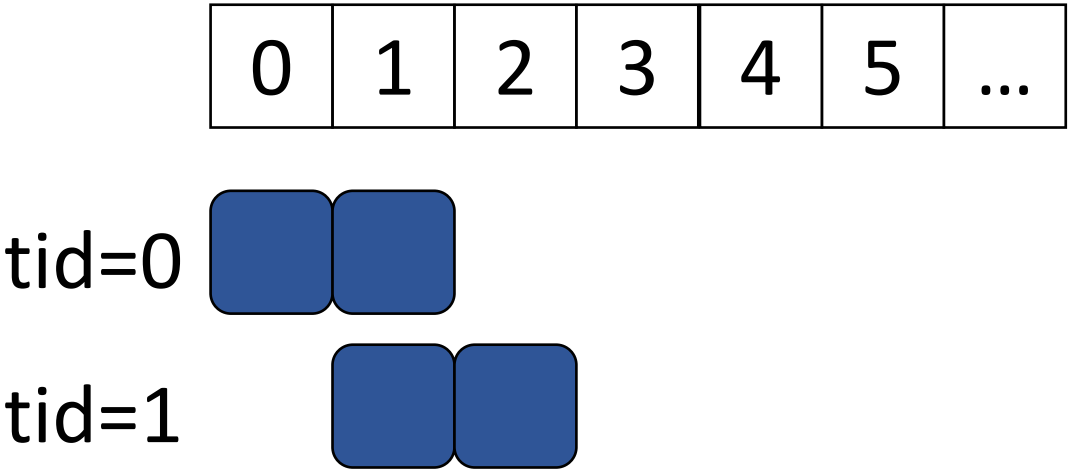
Phases

Must support synchronization

OpenMP barrier blocks threads until all threads have reached the barrier

```
#pragma omp parallel shared(A)
{
   int tid = omp_get_thread_num();

   int v = A[tid];
   #pragma omp barrier
   A[tid + 1] += v;
}
```

Phases



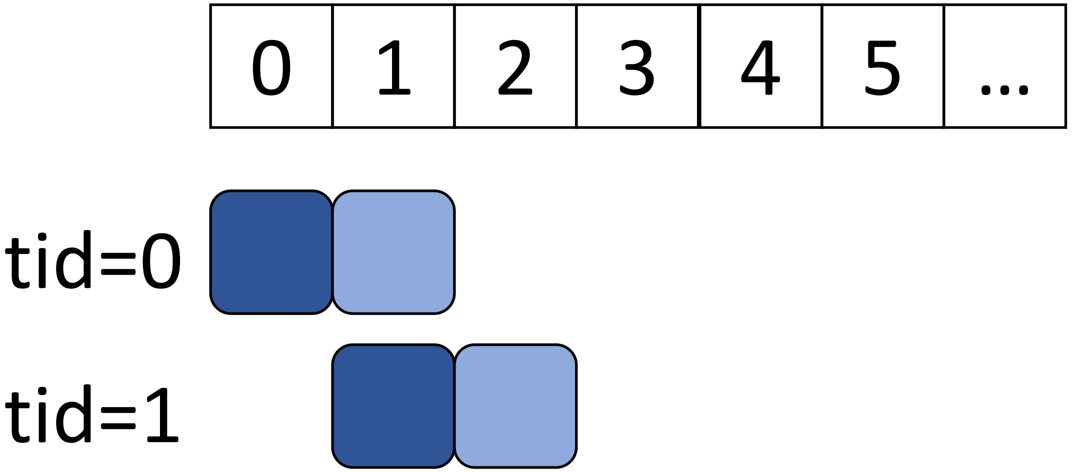
0	1	2	3	4	5	…

tid=0

tid=1

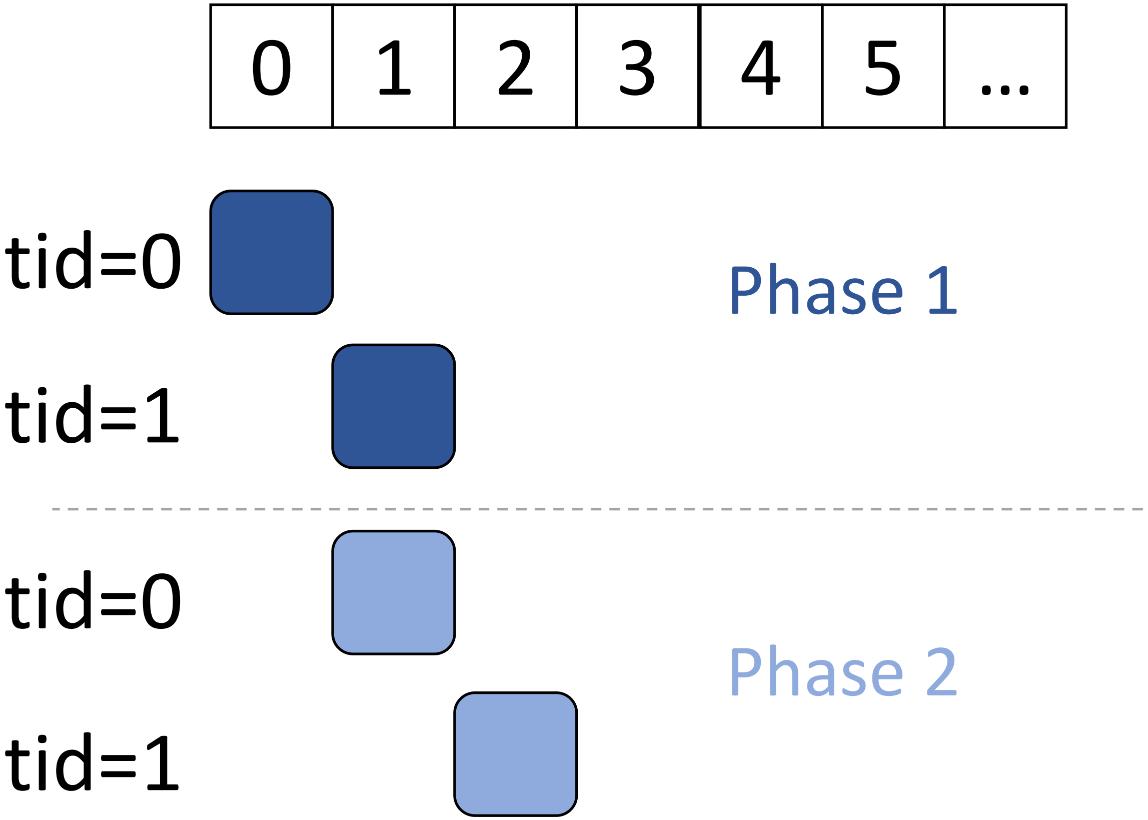
```
#pragma omp parallel shared(A)
{
   int tid = omp_get_thread_num();

   int v = A[tid];

   A[tid + 1] += v;
}
```

Phases



```
#pragma omp parallel shared(A)
{
  int tid = omp_get_thread_num();

  int v = A[tid];
  #pragma omp barrier
  A[tid + 1] += v;
}
```

Phases



0	1	2	3	4	5	…

tid=0

tid=1

Phase 1

tid=0

tid=1

Phase 2

```
#pragma omp parallel shared(A)
{
   int tid = omp_get_thread_num();

   int v = A[tid];
   #pragma omp barrier
   A[tid + 1] += v;
}
```

Phases

Phases
(B0, ??):

```
#pragma omp parallel shared(A)
B0: { // Pseudo-Barrier

S1:    int v = A[tid];
B1:    #pragma omp barrier
S2:   A[tid + 1] += v;

B2: } // Implicit Barrier
```

Phases

Phases

(B0, ??): S1

```
#pragma omp parallel shared(A)
B0: { // Pseudo-Barrier

S1:    int v = A[tid];
B1:    #pragma omp barrier
S2:   A[tid + 1] += v;

B2: } // Implicit Barrier
```

Phases

Phases

(B0, B1): S1

```
#pragma omp parallel shared(A)
B0: { // Pseudo-Barrier

S1:    int v = A[tid];
B1:    #pragma omp barrier
S2:   A[tid + 1] += v;

B2: } // Implicit Barrier
```

Phases

Phases

(B0, B1): S1

(B1, ??): S2

```
#pragma omp parallel shared(A)
B0: { // Pseudo-Barrier

S1:    int v = A[tid];
B1:    #pragma omp barrier
S2:   A[tid + 1] += v;

B2: } // Implicit Barrier
```

Phases

Phases

(B0, B1): S1

(B1, B2): S2

```
#pragma omp parallel shared(A)
B0: { // Pseudo-Barrier

S1:    int v = A[tid];
B1:    #pragma omp barrier
S2:   A[tid + 1] += v;

B2: } // Implicit Barrier
```

Phases

Phases
(B0, B1): S1
(B1, B2): S2

Building all phases is slow

We want instantaneous updates

```
#pragma omp parallel shared(A)
B0: { // Pseudo-Barrier

S1:    int v = A[tid];
B1:    #pragma omp barrier
S2:   A[tid + 1] += v;

B2: } // Implicit Barrier
```

Incremental Race Detection

- Addition
  - New statement
  - New Synchronization
- Deletion
  - Statements
  - Synchronization

Incremental Race Detection

Phases

(B0, B1): S1

(B1, B2): S2

```
#pragma omp parallel shared(A)
B0: { // Pseudo-Barrier

S1:    int v = A[tid];
B1:    #pragma omp barrier
S2:   A[tid + 1] += v;

B2: } // Implicit Barrier
```

Incremental Race Detection

Phases

(B0, B1): S1

(B1, B2): S2

```
#pragma omp parallel shared(A)
B0: { // Pseudo-Barrier

S1:    int v = A[tid];
B1:    #pragma omp barrier
S2:   A[tid + 1] += v;
S3:   v = 0;
B2: } // Implicit Barrier
```

Incremental Race Detection

Phases

(B0, B1): S1

(B1, B2): S2

```
#pragma omp parallel shared(A)
B0: { // Pseudo-Barrier

S1:    int v = A[tid];
B1:    #pragma omp barrier
S2:   A[tid + 1] += v;
S3:   v = 0;
B2: } // Implicit Barrier
```

Incremental Race Detection

Phases

(B0, B1): S1

(B1, B2): S2, S3

```
#pragma omp parallel shared(A)
B0: { // Pseudo-Barrier

S1:    int v = A[tid];
B1:    #pragma omp barrier
S2:   A[tid + 1] += v;
S3:   v = 0;
B2: } // Implicit Barrier
```

Incremental Race Detection

Phases

(B0, B1): S1

(B1, B2): S2, S3

```
#pragma omp parallel shared(A)
B0: { // Pseudo-Barrier

S1:    int v = A[tid];
B1:    #pragma omp barrier
S2:   A[tid + 1] += v;
S3:   v = 0;
B2: } // Implicit Barrier
```

Incremental Race Detection

Phases

(B0, B1): S1

(B1, B2): S2, S3

```
#pragma omp parallel shared(A)
B0: { // Pseudo-Barrier

S1:    int v = A[tid];
B1:    #pragma omp barrier
S2:   A[tid + 1] += v;
S3:   v = 0;
B2: } // Implicit Barrier
```

Incremental Race Detection

Phases

(B0, B1): S1

(B1, B2): S2

```
#pragma omp parallel shared(A)
B0: { // Pseudo-Barrier

S1:    int v = A[tid];
B1:    #pragma omp barrier
S2:   A[tid + 1] += v;

B2: } // Implicit Barrier
```

Incremental Race Detection

Phases

(B0, B1): S1

(B1, B2): S2

```
#pragma omp parallel shared(A)
B0: { // Pseudo-Barrier

S1:    int v = A[tid];
B1:    #pragma omp barrier
S2:   A[tid + 1] += v;

B2: } // Implicit Barrier
```

Incremental Race Detection

Phases

(B0, *): S1

(*, B2): S2

```
#pragma omp parallel shared(A)
B0: { // Pseudo-Barrier

S1:    int v = A[tid];

S2:   A[tid + 1] += v;

B2: } // Implicit Barrier
```

Incremental Race Detection

Phases

(B0, *   ): S1

(*, B2):   S2

```
#pragma omp parallel shared(A)
B0: { // Pseudo-Barrier

S1:   int v = A[tid];

S2:  A[tid + 1] += v;

B2: } // Implicit Barrier
```

Incremental Race Detection

Phases

(B0, * , B2): S1, S2

```
#pragma omp parallel shared(A)
B0: { // Pseudo-Barrier

S1:    int v = A[tid];

S2:   A[tid + 1] += v;

B2: } // Implicit Barrier
```

Incremental Race Detection

Phases

(B0, B2): S1, S2

```
#pragma omp parallel shared(A)
B0: { // Pseudo-Barrier

S1:    int v = A[tid];

S2:   A[tid + 1] += v;

B2: } // Implicit Barrier
```

Incremental Race Detection

Phases
(B0, B2): S1, S2

New Phases
(??, B1):
(B1, ??):

```
#pragma omp parallel shared(A)
B0: { // Pseudo-Barrier

S1:    int v = A[tid];
B1:    #pragma omp barrier
S2:   A[tid + 1] += v;

B2: } // Implicit Barrier
```

Incremental Race Detection

Phases

(B0, B2): S1, S2

New Phases

(??, B1): S1

(B1, ??):

```
#pragma omp parallel shared(A)
B0: { // Pseudo-Barrier

S1:    int v = A[tid];
B1:    #pragma omp barrier
S2:   A[tid + 1] += v;

B2: } // Implicit Barrier
```

Incremental Race Detection

Phases

(B0, B2): S1, S2

New Phases

(B0, B1): S1

(B1, ??):

```
#pragma omp parallel shared(A)
B0: { // Pseudo-Barrier

S1:   int v = A[tid];
B1:   #pragma omp barrier
S2:  A[tid + 1] += v;

B2: } // Implicit Barrier
```

Incremental Race Detection

Phases

(B0, B2): S1, S2

New Phases

(B0, B1): S1

(B1, ??): S2

```
#pragma omp parallel shared(A)
B0: { // Pseudo-Barrier

S1:    int v = A[tid];
B1:    #pragma omp barrier
S2:   A[tid + 1] += v;

B2: } // Implicit Barrier
```

Incremental Race Detection

Phases

(B0, B2): S1, S2

New Phases

(B0, B1): S1

(B1, B2): S2

```
#pragma omp parallel shared(A)
B0: { // Pseudo-Barrier

S1:    int v = A[tid];
B1:    #pragma omp barrier
S2:   A[tid + 1] += v;


B2: } // Implicit Barrier
```

Incremental Race Detection

Phases

(B0, B2): S1, S2

New Phases

(B0, B1): S1

(B1, B2): S2

```
#pragma omp parallel shared(A)
B0: { // Pseudo-Barrier

S1:    int v = A[tid];
B1:    #pragma omp barrier
S2:   A[tid + 1] += v;

B2: } // Implicit Barrier
```

Incremental Race Detection

Phases

(B0, B1): S1

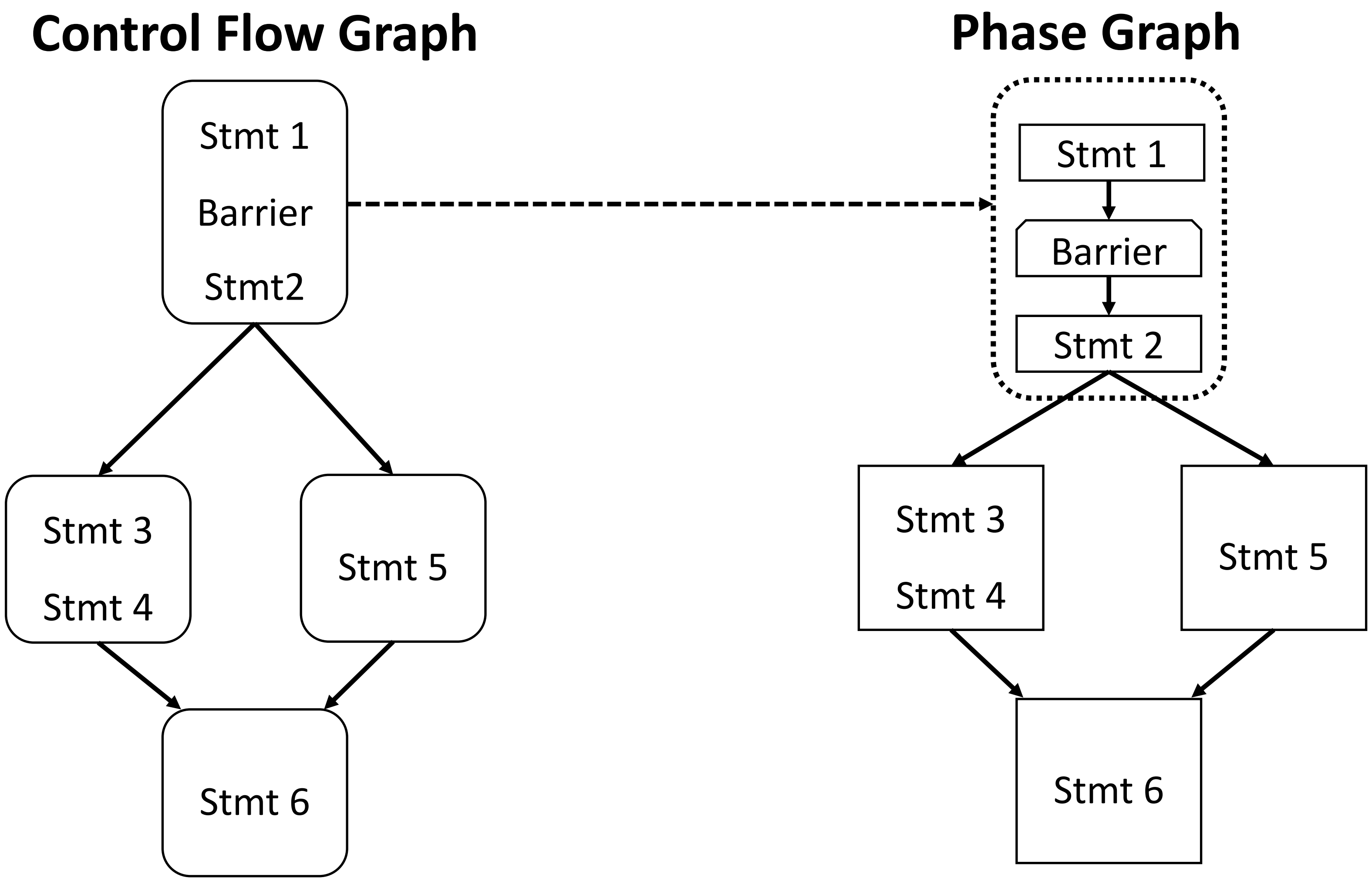
(B1, B2): S2

```
#pragma omp parallel shared(A)
B0: { // Pseudo-Barrier

S1:    int v = A[tid];
B1:    #pragma omp barrier
S2:   A[tid + 1] += v;

B2: } // Implicit Barrier
```

Implementation

- Clang AST
- Clang CFG
- Clang Static Analyzer
  - Symbolic execution engine within Clang

Phase Implementation



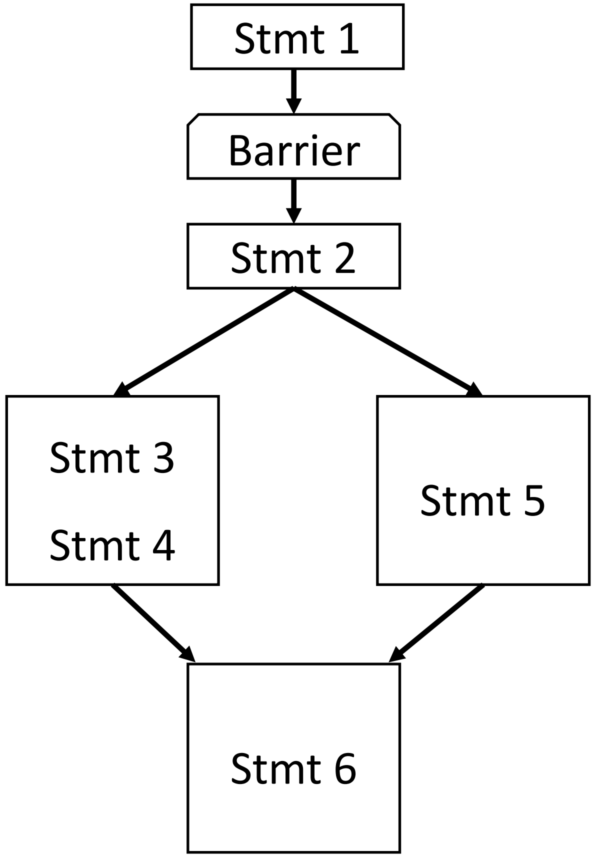
Phase Implementation

Phases

(Begin, Barrier): S1

(Barrier, End): S2, S3, S4, S5, S6

Phase Graph



Array Index Analysis Implementation

- Recursive AST Visitor to find array accesses
    - Classify access as either read or write
- Over approximate access sets
    - Use upper/lower bounds
- Run Clang static analysis checker
    - Check for overlapping access sets

Results

- Only testing Array Index Analysis

- DataRaceBench microbenchmarks
  - 33 could be evaluated (~28%)
  - All 21 real races identified
  - 4 false positives in 3 benchmarks

Conclusions

- Simple Array Index Analysis
  - Useful on micro-benchmarks
- Extend simple analysis to non trivial OpenMP programs
  - Phase Map to model OpenMP synchronizations
- Allow for fast feedback to user
  - Incremental updates to the Phase Map
  - Potential to give feedback within seconds

Thank You!