Extending the Polyhedral Compilation Model for Debugging and Optimization of SPMD-style Explicitly-Parallel Programs

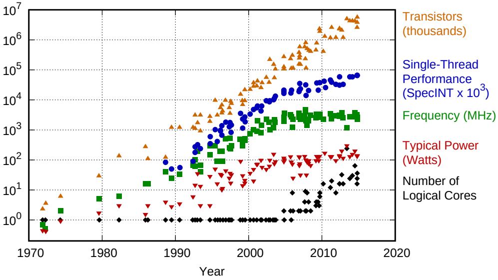
Prasanth Chatarasi

Masters Thesis Defense
Habanero Extreme Scale Software Research Group
Department of Computer Science
Rice University

April 24th, 2017

RICE

40 Years of Microprocessor Trend

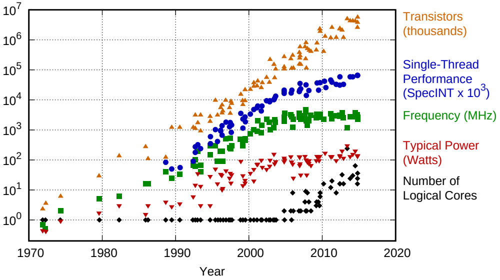


- Moore's law still continues

---

https://www.karlrupp.net/2015/06/40-years-of-microprocessor-trend-data/

40 Years of Microprocessor Trend



- Moore's law still continues
- Performance is driven more by parallelism than single-thread

https://www.karlrupp.net/2015/06/40-years-of-microprocessor-trend-data/

A major challenge facing the overall computer field

- Programming multi-core processors – how to exploit the parallelism in large-scale parallel hardware without undue programmer effort

– Mary Hall et.al., in *Communications of ACM 2009*

A major challenge facing the overall computer field

- Programming multi-core processors – how to exploit the parallelism in large-scale parallel hardware without undue programmer effort

  – Mary Hall et.al., in *Communications of ACM 2009*

- Two major compiler approaches in tackling the challenge
  - Automatic parallelization of sequential programs
    - Compilers extract parallelism
    - Not much burden on programmer but lot of limitations!
  - Manually parallelize programs
    - Full burden on programmer but can get higher performance!

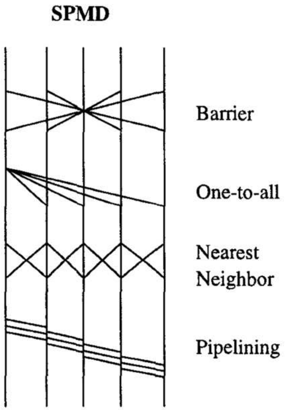
A major challenge facing the overall computer field

- Programming multi-core processors – how to exploit the parallelism in large-scale parallel hardware without undue programmer effort

  – Mary Hall et.al., in *Communications of ACM 2009*

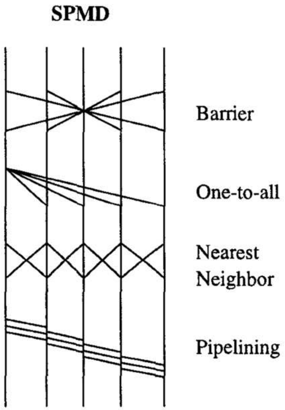
- Two major compiler approaches in tackling the challenge
  - Automatic parallelization of sequential programs
    - Compilers extract parallelism
    - Not much burden on programmer but lot of limitations!
  - Manually parallelize programs
    - Full burden on programmer but can get higher performance!
    - Can the compilers help the programmer?

Focus of this work – SPMD-style parallelism



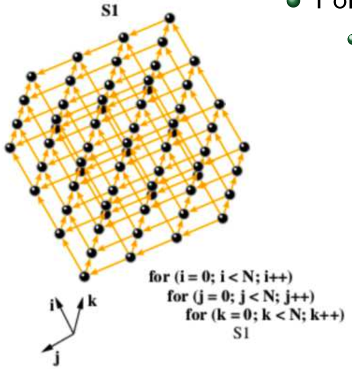
- Focus on SPMD-style parallel programs
  - All processors execute the same program
  - Sequential code redundantly
  - Parallel code cooperatively

Focus of this work – SPMD-style parallelism



- Focus on SPMD-style parallel programs
  - All processors execute the same program
  - Sequential code redundantly
  - Parallel code cooperatively
  - OpenMP for multi-cores, CUDA/ OpenCL for accelerators, MPI for distributed systems

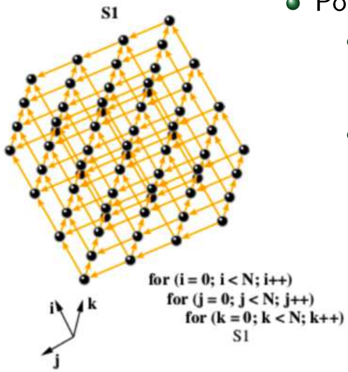
Focus of this work – Polyhedral compilation model



- Polyhedral compilation model
  - Algebraic framework to reason about loop nests

http://pluto-compiler.sourceforge.net/

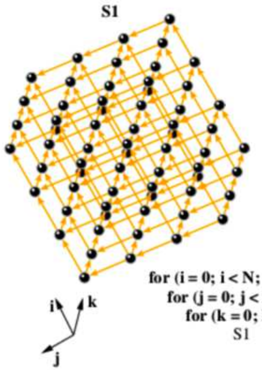
Focus of this work – Polyhedral compilation model



- Polyhedral compilation model
  - Algebraic framework to reason about loop nests
  - Wide range of applications
    - Automatic parallelization
    - High-level synthesis
    - Communication optimizations

http://pluto-compiler.sourceforge.net/

Focus of this work – Polyhedral compilation model




```
for (i = 0; i < N; i++)
  for (j = 0; j < N; j++)
    for (k = 0; k < N; k++)
      S1
```

- Polyhedral compilation model
  - Algebraic framework to reason about loop nests
  - Wide range of applications
    - Automatic parallelization
    - High-level synthesis
    - Communication optimizations
  - Used in
    - Production compilers (LLVM, GCC)
    - Just-in-time compilers (PolyJIT)
    - DSL compilers (PolyMage, Halide)

---

http://pluto-compiler.sourceforge.net/

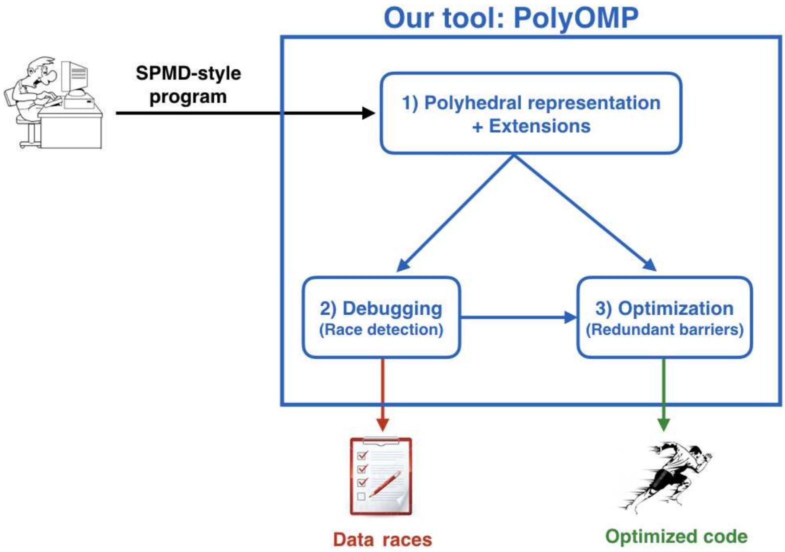
Thesis Statement

Though the polyhedral compilation model was designed for analysis and optimization of sequential programs, our thesis is that it can be extended to enable analysis of SPMD-style explicitly-parallel programs with benefits to debugging and optimization of such programs.

Chatarasi et.al (LCPC 2016), An Extended Polyhedral Model for SPMD Programs and its use in Static Data Race Detection

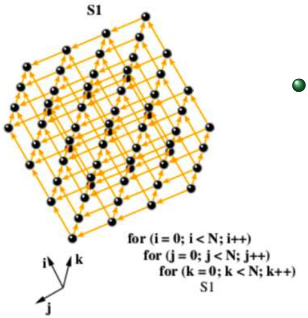
Chatarasi et.al (ACM SRC @ PACT 2015), Extending Polyhedral Model for Analysis and Transformation of OpenMP Programs

Overall flow



Polyhedral Compilation Model

- Compiler (algebraic) techniques for analysis and transformation of codes with nested loops



- Advantages over Abstract Syntax Tree (AST) based frameworks
  - Reasoning at statement instance in loops
  - Unifies many loop transformations into a single transformation
  - Powerful code generation algorithms

http://pluto-compiler.sourceforge.net/

Polyhedral Representation of Programs - Schedule

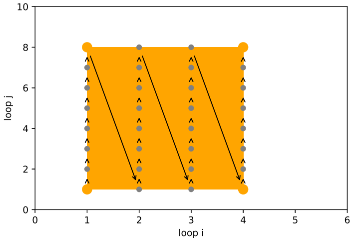
```
for(int i = 1; i < M; i++) {
    for(int j = 1; j < N; j++) {
      S;
    }
}
```

Polyhedral Representation of Programs - Schedule

```
for(int i = 1; i < M; i++) {
    for(int j = 1; j < N; j++) {
      S;
    }
}
```

Schedule (θ) – A key element of polyhedral representation

- Assigns a time-stamp to each statement instance S(i, j)
- Statement instances are executed in increasing order of time-stamps
- Captures program execution order (total order in sequential programs)



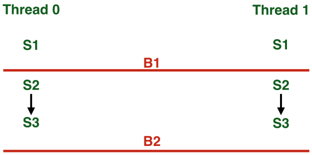
$$\theta(S(i,j)) = (i,j)$$

Limitations of Polyhedral Model

(a) An SPMD-style program

```
#pragma omp parallel num_threads(2)
{
        {S1;}

    #pragma omp barrier //B1

        {S2;}
        {S3;}

    #pragma omp barrier //B2
}
```

(b) Program execution order

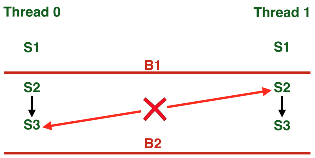


Limitations of Polyhedral Model

(a) An SPMD-style program

```
#pragma omp parallel num_threads(2)
{
        {S1;}

    #pragma omp barrier //B1

        {S2;}
        {S3;}

    #pragma omp barrier //B2
}
```

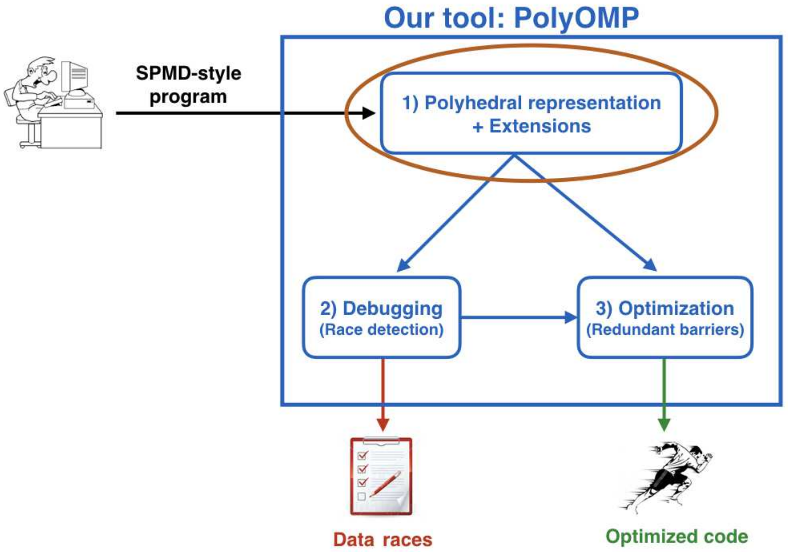
(b) Program execution order



Limitations of Polyhedral Model

Currently, there are no approaches to capture partial orders from SPMD programs and express onto schedules

Overall workflow (PolyOMP)



What are the important concepts in SPMD execution ?

```
#pragma omp parallel
{
    for(int i = 0; i < N; i++)
    {
        for(int j = 0; j < N; j++)
        {
            {S1;} //S1(i, j)
          #pragma omp barrier //B1(i, j)
            {S2;} //S2(i, j)
        }

        #pragma omp barrier //B2(i)

        #pragma omp master
            {S3;} //S3(i)
    }
}
```

What are the important concepts in SPMD execution ?

```
#pragma omp parallel
{
    for(int i = 0; i < N; i++)
    {
        for(int j = 0; j < N; j++)
        {
            {S1;} //S1(i, j)
          #pragma omp barrier //B1(i, j)
            {S2;} //S2(i, j)
        }

        #pragma omp barrier //B2(i)

        #pragma omp master
            {S3;} //S3(i)
    }
}
```

Program execution order for $N = 2$

Thread 0		Thread 1
S1(0, 0)	Phase = 0 B1(0, 0)	S1(0, 0)
S2(0, 0) S1(0, 1)	Phase = 1 B1(0, 1)	S2(0, 0) S1(0, 1)
S2(0, 1)	Phase = 2 B2(0)	S2(0, 1)
S3(0) S1(1, 0)	Phase = 3 B1(1, 0)	S1(1, 0)
. .	. .	. .

What are the important concepts in SPMD execution ?

```
#pragma omp parallel
{
    for(int i = 0; i < N; i++)
    {
        for(int j = 0; j < N; j++)
        {
            {S1;} //S1(i, j)
          #pragma omp barrier //B1(i, j)
            {S2;} //S2(i, j)
        }

        #pragma omp barrier //B2(i)

        #pragma omp master
            {S3;} //S3(i)
    }
}
```

Program execution order for $N = 2$

Important concepts: 1) Threads and 2) Phases

Extension1 – Thread/Space/Allocation Mapping

Space Mapping (θ^A)

Assigns a logical processor id to each statement instance

Extension1 – Thread/Space/Allocation Mapping

Space Mapping (θ^A)

Assigns a logical processor id to each statement instance

```
#pragma omp parallel
{
    for(int i = 0; i < N; i++)
    {
        for(int j = 0; j < N; j++)
        {
            {S1;} //S1(i, j)
          #pragma omp barrier //B1(i, j)
            {S2;} //S2(i, j)
        }

        #pragma omp barrier //B2(i)

        #pragma omp master
            {S3;} //S3(i)
    }
}
```

Extension1 – Thread/Space/Allocation Mapping

Space Mapping (θ^A)

Assigns a logical processor id to each statement instance

```
#pragma omp parallel
{
    for(int i = 0; i < N; i++)
    {
        for(int j = 0; j < N; j++)
        {
            {S1;} //S1(i, j)
          #pragma omp barrier //B1(i, j)
            {S2;} //S2(i, j)
        }

        #pragma omp barrier //B2(i)

        #pragma omp master
            {S3;} //S3(i)
    }
}
```

For example, $\theta^A(S3(i)) = 0$

Extension2 – Phase Mapping

Phase Mapping (θ^P)

Assigns a logical phase id to each statement instance

Extension2 – Phase Mapping

Phase Mapping (θ^P)

Assigns a logical phase id to each statement instance

```
#pragma omp parallel
{
    for(int i = 0; i < N; i++)
    {
        for(int j = 0; j < N; j++)
        {
            {S1;} //S1(i, j)
          #pragma omp barrier //B1(i, j)
            {S2;} //S2(i, j)
        }

        #pragma omp barrier //B2(i)

        #pragma omp master
            {S3;} //S3(i)
    }
}
```

Extension2 – Phase Mapping

Phase Mapping (θ^P)

Assigns a logical phase id to each statement instance

```
#pragma omp parallel
{
    for(int i = 0; i < N; i++)
    {
        for(int j = 0; j < N; j++)
        {
            {S1;} //S1(i, j)
          #pragma omp barrier //B1(i, j)
            {S2;} //S2(i, j)
        }

        #pragma omp barrier //B2(i)

        #pragma omp master
            {S3;} //S3(i)
    }
}
```

For example, $\theta^P(S3(0)) = 3$

How to compute phase mappings?

We define phase mappings in terms of reachable barriers

Reachable barriers (RB) of a statement instance

Set of barrier instances that can be executed after the statement instance without an intervening barrier instance

How to compute phase mappings?

We define phase mappings in terms of reachable barriers

Reachable barriers (RB) of a statement instance

Set of barrier instances that can be executed after the statement instance without an intervening barrier instance

$$RB(S2(0,1)) = B2(0)$$
$$RB(S3(0)) = B1(1,0)$$

How to compute phase mappings?

Observation

Two statement instances are in same phase if they have *same* set of reachable barrier instances

How to compute phase mappings?

Observation

Two statement instances are in same phase if they have *same* set of reachable barrier instances

$$\theta^P(S3(0)) = RB(S3(0))$$
$$= B1(1,0)$$
$$\theta^P(S1(1,0)) = RB(S1(1,0))$$
$$= B1(1,0)$$

How to compute phase mappings?

Observation

Two statement instances are in same phase if they have *same* set of reachable barrier instances

Thread 0		Thread 1
S1(0, 0)	Phase = 0 B1(0, 0)	S1(0, 0)
S2(0, 0) S1(0, 1)	Phase = 1 B1(0, 1)	S2(0, 0) S1(0, 1)
S2(0, 1)	Phase = 2 B2(0)	S2(0, 1)
S3(0) S1(1, 0)	Phase = 3 B1(1, 0)	S1(1, 0)
. .	. .	. .

$$
\begin{aligned}
\theta^P(S3(0)) &= RB(S3(0)) \\
&= B1(1,0) \\
\theta^P(S1(1,0)) &= RB(S1(1,0)) \\
&= B1(1,0) \\
\Longrightarrow \theta^P(S3(0)) &= \theta^P(S1(1,0))
\end{aligned}
$$

How to compute phase mappings?

Observation

Two statement instances are in same phase if they have *same* set of reachable barrier instances

Thread 0		Thread 1
S1(0, 0)	Phase = 0 B1(0, 0)	S1(0, 0)
S2(0, 0) S1(0, 1)	Phase = 1 B1(0, 1)	S2(0, 0) S1(0, 1)
S2(0, 1)	Phase = 2 B2(0)	S2(0, 1)
S3(0) S1(1, 0)	Phase = 3 B1(1, 0)	S1(1, 0)
. .	. .	. .

$$
\begin{aligned}
\theta^P(S3(0)) &= RB(S3(0)) \\
&= B1(1,0) \\
\theta^P(S1(1,0)) &= RB(S1(1,0)) \\
&= B1(1,0) \\
\implies \theta^P(S3(0)) &= \theta^P(S1(1,0))
\end{aligned}
$$

To compute absolute phase mappings, $\theta^P(S) \; = \; \theta(RB(S))$

Execution order in SPMD-style programs

- In general, partial orders are expressed through May-Happen-in-Parallel (MHP) or Happens-Before (HB) relations

Execution order in SPMD-style programs

- In general, partial orders are expressed through May-Happen-in-Parallel (MHP) or Happens-Before (HB) relations

We define MHP relations in terms of space and phase mappings

MHP

Two statement instances can run in parallel if they are run by different threads and are in same phase of computation

Execution order in SPMD-style programs

- In general, partial orders are expressed through May-Happen-in-Parallel (MHP) or Happens-Before (HB) relations

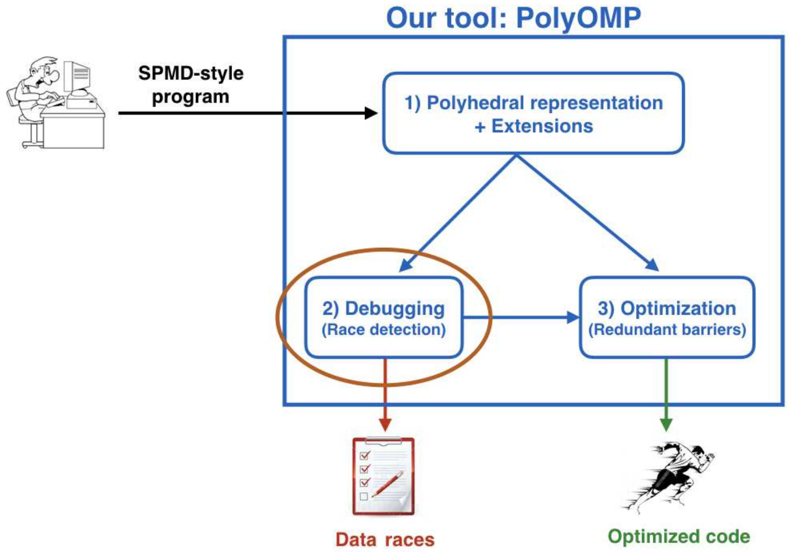
We define MHP relations in terms of space and phase mappings

MHP

Two statement instances can run in parallel if they are run by different threads and are in same phase of computation

- Now, program order information in polyhedral model
  - (Space (θ^A), Phase (θ^P), Schedule (θ))

Overall workflow (PolyOMP)



Debugging of SPMD-style programs - Data races

- Data races are common bugs in SPMD shared memory programs
- Definition:
  - *A race occurs when two or more threads perform a conflicting accesses to a shared variable without any synchronization*
- Data races result in non-deterministic behavior

Debugging of SPMD-style programs - Data races

- Data races are common bugs in SPMD shared memory programs
- Definition:
  - *A race occurs when two or more threads perform a conflicting accesses to a shared variable without any synchronization*
- Data races result in non-deterministic behavior
- Occurs only in few of the possible schedules of a parallel program
  - Extremely hard to reproduce and debug!

Motivating example from OmpSCR Suite

```
#pragma omp parallel shared(U, V, k)
{
    while (k <= Max)  // S1
    {
        #pragma omp for nowait
        for(i = 0 to N)
            U[i] = V[i];
        #pragma omp barrier

        #pragma omp for nowait
        for(i = 1 to N-1)
            V[i] = U[i-1] + U[i] + U[i+1];
        #pragma omp barrier

        #pragma omp master
         { k++;} // S2
    }
}
```

- 1-dimensional stencil (c_jacobi3.c) from OmpSCR suite

Motivating example from OmpSCR Suite

```
#pragma omp parallel shared(U, V, k)
{
    while (k <= Max)  // S1
    {
      #pragma omp for nowait
      for(i = 0 to N)
          U[i] = V[i];
      #pragma omp barrier

      #pragma omp for nowait
      for(i = 1 to N-1)
          V[i] = U[i-1] + U[i] + U[i+1];
      #pragma omp barrier

      #pragma omp master
       { k++;} // S2
    }
}
```

- 1-dimensional stencil (c_jacobi3.c) from OmpSCR suite
- Race b/w S1 and S2 on variable 'k'

Motivating example from OmpSCR Suite

```
#pragma omp parallel shared(U, V, k)
{
    while (k <= Max)  // S1
    {
       #pragma omp for nowait
       for(i = 0 to N)
           U[i] = V[i];
       #pragma omp barrier

       #pragma omp for nowait
       for(i = 1 to N-1)
           V[i] = U[i-1] + U[i] + U[i+1];
       #pragma omp barrier

       #pragma omp master
        { k++;} // S2
    }
}
```

- 1-dimensional stencil (c_jacobi3.c) from OmpSCR suite
- Race b/w S1 and S2 on variable 'k'
- Our goal: Detect such races at compile-time

Motivating example from OmpSCR Suite

```
#pragma omp parallel shared(U, V, k)
{
    while (k <= Max)  // S1
    {
        #pragma omp for nowait
        for(i = 0 to N)
            U[i] = V[i];
        #pragma omp barrier

        #pragma omp for nowait
        for(i = 1 to N-1)
            V[i] = U[i-1] + U[i] + U[i+1];
        #pragma omp barrier

        #pragma omp master
         { k++;} // S2
    }
}
```

- 1-dimensional stencil (c_jacobi3.c) from OmpSCR suite
- Race b/w S1 and S2 on variable 'k'
- Our goal: Detect such races at compile-time

ARCHER: "The data race in c_jacobi3.c highly influences the execution time of the benchmark, varying it by a factor of 1000 from run to run."

Our approach for race detection

1. Generate race conditions for every pair of read/write accesses of all statements
   - *Race(S, T) = true on 'k'*
   - $\implies$ $MHP(S,T)$ = *true and S,T conflict on 'k'*
   - $\implies$ $\theta^A(S) \neq \theta^A(T)$ *and* $\theta^P(S) = \theta^P(T)$ *and S,T conflict on 'k'*

Chatarasi et.al (LCPC 2016), An Extended Polyhedral Model for SPMD Programs and its use in Static Data Race Detection

Our approach for race detection

1. Generate race conditions for every pair of read/write accesses of all statements
   - *Race(S, T) = true on 'k'*
   - $\implies MHP(S, T) \ = \ true$ *and S,T conflict on 'k'*
   - $\implies \theta^A(S) \neq \theta^A(T)$ *and* $\theta^P(S) = \theta^P(T)$ *and S,T conflict on 'k'*
2. Solve the race conditions for existence of solutions.
   - If there are no solutions, there are no *data races*

Chatarasi et.al (LCPC 2016), An Extended Polyhedral Model for SPMD Programs and its use in Static Data Race Detection

Our approach on the motivating example

```
#pragma omp parallel shared(U, V, k)
{
    while (k <= Max)  // S1 (loop-x)
    {
       #pragma omp for nowait
       for(i = 0 to N)
           U[i] = V[i];
       #pragma omp barrier // B1

       #pragma omp for nowait
       for(i = 1 to N-1)
           V[i] = U[i-1] + U[i] + U[i+1];
       #pragma omp barrier

       #pragma omp master
        { k++;} // S2
    }
}
```

Race cond. b/w S2(x') & S1(x'')

Our approach on the motivating example

```
#pragma omp parallel shared(U, V, k)
{
    while (k <= Max)  // S1 (loop-x)
    {
        #pragma omp for nowait
        for(i = 0 to N)
            U[i] = V[i];
        #pragma omp barrier // B1

        #pragma omp for nowait
        for(i = 1 to N-1)
            V[i] = U[i-1] + U[i] + U[i+1];
        #pragma omp barrier

        #pragma omp master
         { k++;} // S2
    }
}
```

Race cond. b/w S2(x') & S1(x'')

- Space: $\theta^A(S2) \neq \theta^A(S1)$
  $\wedge\, \theta^A(S2) = 0$

Our approach on the motivating example

```
#pragma omp parallel shared(U, V, k)
{
    while (k <= Max)  // S1 (loop-x)
    {
        #pragma omp for nowait
        for(i = 0 to N)
            U[i] = V[i];
        #pragma omp barrier // B1

        #pragma omp for nowait
        for(i = 1 to N-1)
            V[i] = U[i-1] + U[i] + U[i+1];
        #pragma omp barrier

        #pragma omp master
         { k++;} // S2
    }
}
```

Race cond. b/w S2(x') & S1(x")

- Space: $\theta^A(S2) \neq \theta^A(S1)$
  $\wedge\, \theta^A(S2) = 0$
- Phase: $\theta^P(S2) = \theta^P(S1)$
  $\rightarrow$ B1(x' + 1) = B1(x")
  $\rightarrow$ x' + 1 = x"

Our approach on the motivating example

```
#pragma omp parallel shared(U, V, k)
{
    while (k <= Max)  // S1 (loop-x)
    {
        #pragma omp for nowait
        for(i = 0 to N)
            U[i] = V[i];
        #pragma omp barrier // B1

        #pragma omp for nowait
        for(i = 1 to N-1)
            V[i] = U[i-1] + U[i] + U[i+1];
        #pragma omp barrier

        #pragma omp master
         { k++;} // S2
    }
}
```

Race cond. b/w S2(x') & S1(x'')

- Space: $\theta^A(S2) \neq \theta^A(S1)$
  $\wedge\, \theta^A(S2) = 0$
- Phase: $\theta^P(S2) = \theta^P(S1)$
  $\rightarrow$ B1(x' + 1) = B1(x'')
  $\rightarrow$ x' + 1 = x''
- Conflict: TRUE (same location 'k')

Our approach on the motivating example

```
#pragma omp parallel shared(U, V, k)
{
    while (k <= Max)  // S1 (loop-x)
    {
        #pragma omp for nowait
        for(i = 0 to N)
            U[i] = V[i];
        #pragma omp barrier // B1

        #pragma omp for nowait
        for(i = 1 to N-1)
            V[i] = U[i-1] + U[i] + U[i+1];
        #pragma omp barrier

        #pragma omp master
         { k++;} // S2
    }
}
```

Race cond. b/w S2(x') & S1(x'')

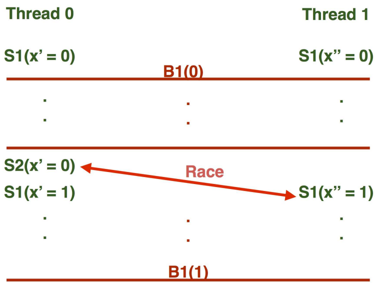
- Space: $\theta^A(S2) \neq \theta^A(S1)$
  $\wedge\ \theta^A(S2) = 0$
- Phase: $\theta^P(S2) = \theta^P(S1)$
  $\rightarrow$ B1(x' + 1) = B1(x'')
  $\rightarrow$ x' + 1 = x''
- Conflict: TRUE (same location 'k')

Satisfiable assignment: (θ^A(S2) = 0, x' = 0) and (θ^A(S1) = 1, x'' = 1)

Our approach on the motivating example

Satisfiable assignment: $(\theta^A(\text{S2}) = 0, \text{x'} = 0)$ and $(\theta^A(\text{S1}) = 1, \text{x''} = 1)$

```
#pragma omp parallel shared(U, V, k)
{
    while (k <= Max)  // S1 (loop-x)
    {
       #pragma omp for nowait
       for(i = 0 to N)
           U[i] = V[i];
       #pragma omp barrier // B1

       #pragma omp for nowait
       for(i = 1 to N-1)
           V[i] = U[i-1] + U[i] + U[i+1];
       #pragma omp barrier

       #pragma omp master
        { k++;} // S2
    }
}
```



Experimental Setup

- Quad core-i7 machine (2.2GHz) of 16GB main memory on macOS

Experimental Setup

- Quad core-i7 machine (2.2GHz) of 16GB main memory on macOS
- Benchmark suites
  - OmpSCR Benchmarks Suite,
  - Polybench-ACC OpenMP Benchmarks Suite

Experimental Setup

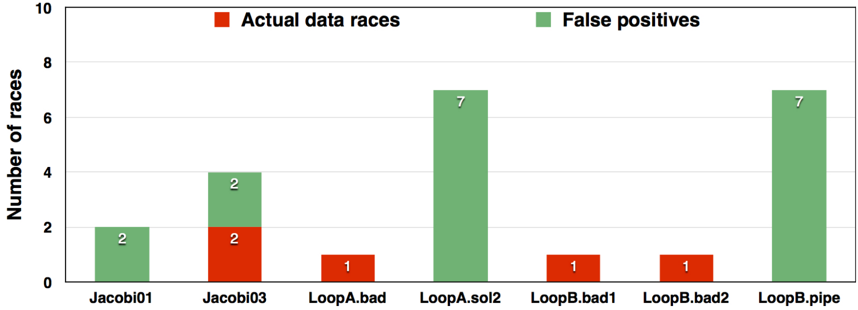
- Quad core-i7 machine (2.2GHz) of 16GB main memory on macOS
- Benchmark suites
  - OmpSCR Benchmarks Suite,
  - Polybench-ACC OpenMP Benchmarks Suite
- Comparisons with existing tools
  - ARCHER (Static + Dynamic)
  - Intel Inspector XE (Dynamic)
  - ARCHER (Static)

Experiments - OmpSCR Benchmark suite

- Evaluation on 12 benchmarks
- Identified all documented races (5)

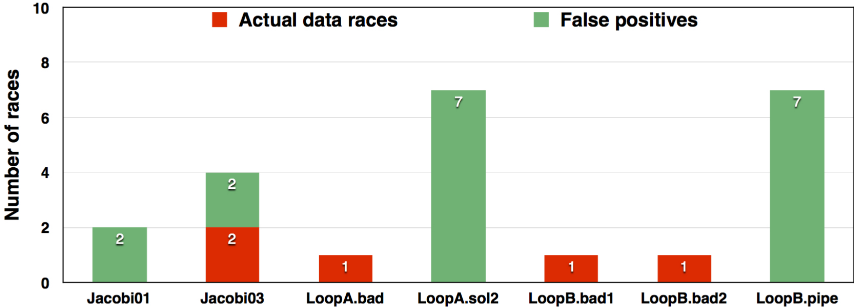
Experiments - OmpSCR Benchmark suite

- Evaluation on 12 benchmarks
- Identified all documented races (5)



Experiments - OmpSCR Benchmark suite

- Evaluation on 12 benchmarks
- Identified all documented races (5)



- False positives because of linearized array subscripts

Experiments - OmpSCR Benchmark suites

Tool	ARCHER (Static + Dynamic)	Intel Inspector XE (Dynamic)	ARCHER (Static)	PolyOMP (Static)
True races	5	5	3	5
False +ves	0	5	*	27

- Static part of ARCHER focuses on worksharing loops but not SPMD!
    - Remaining 2 races incurred significant overhead in dynamic analysis

Experiments - OmpSCR Benchmark suites

Tool	ARCHER (Static + Dynamic)	Intel Inspector XE (Dynamic)	ARCHER (Static)	PolyOMP (Static)
True races	5	5	3	5
False +ves	0	5	*	27

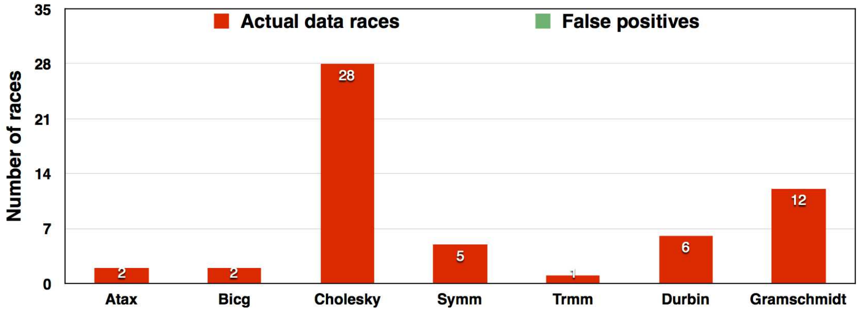
- Static part of ARCHER focuses on worksharing loops but not SPMD!
  - Remaining 2 races incurred significant overhead in dynamic analysis
- Intel Inspector reported false +ves on worksharing loop iterators

Experiments - Polybench-ACC OpenMP Benchmark suite

- Evaluation on 22 benchmarks

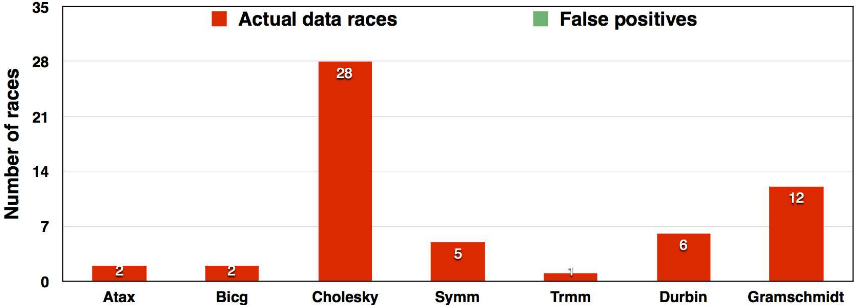
Experiments - Polybench-ACC OpenMP Benchmark suite

- Evaluation on 22 benchmarks



Experiments - Polybench-ACC OpenMP Benchmark suite

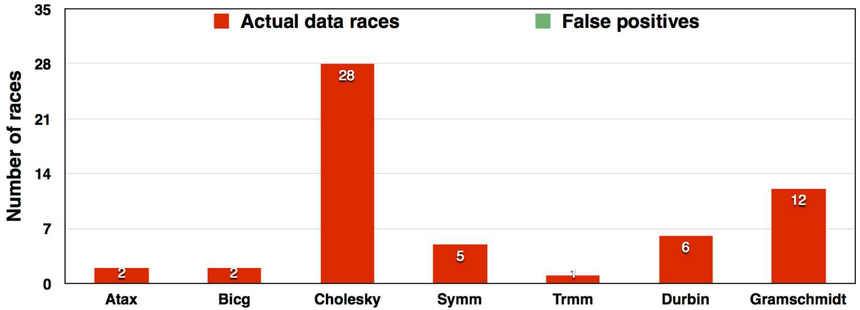
- Evaluation on 22 benchmarks



- Our tool found 61 races and no False positives (All verified)

Experiments - Polybench-ACC OpenMP Benchmark suite

- Evaluation on 22 benchmarks



- Our tool found 61 races and no False positives (All verified)
- Intel Inspector found only 31 races

Source of races

- Shared scalar variables inside the work-sharing loops (Eg: cholesky.c)

```
int x;
#pragma omp parallel
{
    #pragma omp for private (j,k)
    for (i = 0; i < _PB_N; ++i) {
        x = A[i][i];
        for (j = 0; j <= i - 1; ++j)
            x = x - A[i][j] * A[i][j];
    } ......
}
```

Source of races

- Shared scalar variables inside the work-sharing loops (Eg: cholesky.c)

```
int x;
#pragma omp parallel
{
    #pragma omp for private (j,k)
    for (i = 0; i < _PB_N; ++i) {
        x = A[i][i];
        for (j = 0; j <= i - 1; ++j)
            x = x - A[i][j] * A[i][j];
    } ......
}
```

- Accessing common elements of arrays in parallel (Eg: trmm.c)

```
#pragma omp parallel
{
    #pragma omp for private (j, k)
    for (i = 1; i < _PB_NI; i++)
        for (j = 0; j < _PB_NI; j++)
            for (k = 0; k < i; k++)
                B[i][j] += alpha * A[i][k] * B[j][k];
}
```

Strengths and Limitations of our approach

- Strengths
  - Input independent and schedule independent
  - Guaranteed to be exact (No false +ves and No false -ves) if the input program satisfies all the standard preconditions of the polyhedral model

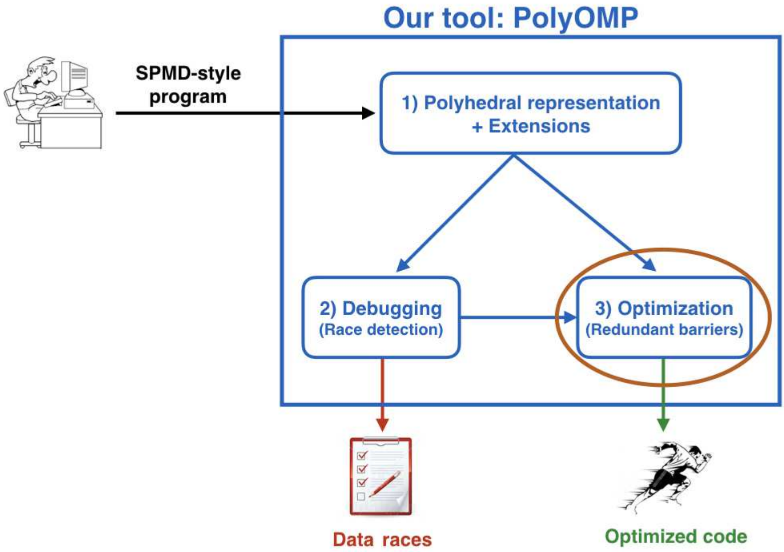
Strengths and Limitations of our approach

- Strengths
  - Input independent and schedule independent
  - Guaranteed to be exact (No false +ves and No false -ves) if the input program satisfies all the standard preconditions of the polyhedral model
- Limitations
  - Textually aligned barriers
    - *All threads encounter same sequence of barriers*
  - Pointer aliasing

Closely related static approaches for race detection

	Supported Constructs	Approach	Guarantees
Pathg (Yu et.al)	OpenMP worksharing loops, Simple Barriers, Atomic	Thread automata	Per number of threads
OAT (Ma et.al)	OpenMP worksharing loops, Barriers, locks, Atomic, single, master	Symbolic execution	Per number of threads
ompVerify (Basupalli et.al)	OpenMP ‘parallel for’	Polyhedral (Dependence analysis)	Per worksharing loop
ARCHER (static) (Atzeni et.al)	OpenMP ‘parallel for’	Polyhedral (Dependence analysis)	Per worksharing loop
PolyOMP Our Approach	OpenMP worksharing loops, Barriers in arbitrary nested loops, Single, master	Polyhedral (MHP relations)	Per program

Overall workflow (PolyOMP)



Optimization of SPMD-style programs - Redundant barriers

- Redundant usage of barriers is a common performance issue
- Definition:
  - *A barrier is redundant if its removal doesn't change the program semantics (No data races)*

Optimization of SPMD-style programs - Redundant barriers

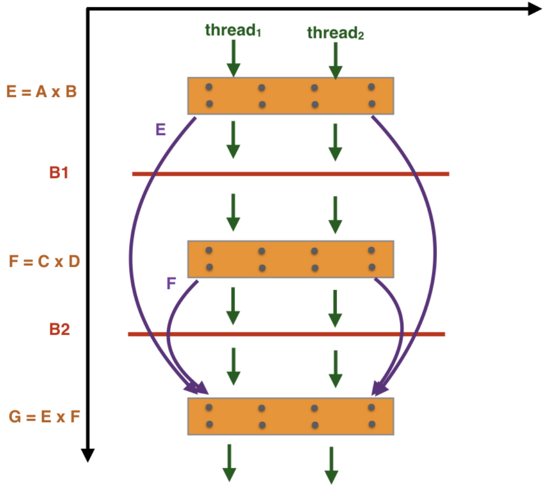
- Redundant usage of barriers is a common performance issue
- Definition:
  - *A barrier is redundant if its removal doesn't change the program semantics (No data races)*
- Hence, we assume input programs to be data-race-free.

Optimization of SPMD-style programs - Redundant barriers

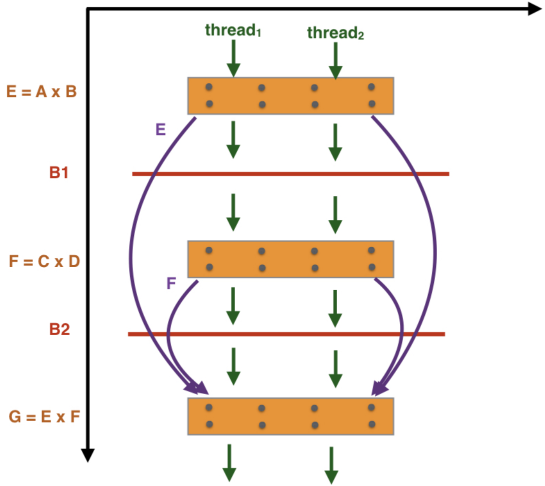
```
#pragma omp parallel
{
    #pragma omp for
    for(int i = 0; i < N; i++) {
        for(int j = 0; j < N; j++)
            for(int k = 0; k < N; k++)
                E[i][j] = A[i][k] * B[k][j]; //S1
    }

    #pragma omp for
    for(int i = 0; i < N; i++) {
        for(int j = 0; j < N; j++)
            for(int k = 0; k < N; k++)
                F[i][j] = C[i][k] * D[k][j]; //S2
    }

    #pragma omp for
    for(int i = 0; i < N; i++) {
        for(int j = 0; j < N; j++)
            for(int k = 0; k < N; k++)
                G[i][j] = E[i][k] * F[k][j]; //S3
    }
}
```

A sequence of matrix multiplications, i.e., $E = A \times B$; $F = C \times D$; $G = E \times F$;

Overall execution with data dependences



Overall execution with data dependences



Barrier B1 is redundant ☺

Optimization of SPMD-style programs - Redundant barriers

```c
1  #pragma omp parallel
2  {
3      #pragma omp for
4      for(int i = 0; i < N; i++) {
5          for(int j = 0; j < N; j++)
6              for(int k = 0; k < N; k++)
7                  E[i][j] = A[i][k] * B[k][j]; //S1
8      } //B1
9
10     #pragma omp for
11     for(int i = 0; i < N; i++) {
12         for(int j = 0; j < N; j++)
13             for(int k = 0; k < N; k++)
14                 F[i][j] = C[i][k] * D[k][j]; //S2
15     }
16
17     #pragma omp for
18     for(int i = 0; i < N; i++) {
19         for(int j = 0; j < N; j++)
20             for(int k = 0; k < N; k++)
21                 G[i][j] = E[i][k] * F[k][j]; //S3
22     }
23 }
```

Implicit barrier on line 8 is redundant ☺

Our approach for identification of redundant barriers

- Remove all barriers from the program and compute data races
    - Races are computed with our race detection approach

Our approach for identification of redundant barriers

- Remove all barriers from the program and compute data races
  - Races are computed with our race detection approach
- Map each barrier to a set of races that can be fixed with that barrier
  - For each barrier, our approach computes *phases* again, and see whether source and sink of the race are in different phases

Our approach for identification of redundant barriers

- Remove all barriers from the program and compute data races
  - Races are computed with our race detection approach
- Map each barrier to a set of races that can be fixed with that barrier
  - For each barrier, our approach computes *phases* again, and see whether source and sink of the race are in different phases
- Greedily pick up set of barriers from the map so that all races are covered.
  - Subtract the required barriers from set of initial barriers

Experimental Setup

- Benchmark suites
  - OmpSCR Benchmark Suite, Polybench-ACC OpenMP Benchmark suite

- Two platforms, i.e., Intel Knights Corner and IBM Power 8

	Intel KNC	IBM Power 8
Micro architecture	Xeon Phi	Power PC
Total threads	228	192
Compiler	Intel ICC v15.0	IBM XLC v13.1
Compiler flags	-O3	-O5

Experimental Setup

- Benchmark suites
  - OmpSCR Benchmark Suite, Polybench-ACC OpenMP Benchmark suite
- Two platforms, i.e., Intel Knights Corner and IBM Power 8

	Intel KNC	IBM Power 8
Micro architecture	Xeon Phi	Power PC
Total threads	228	192
Compiler	Intel ICC v15.0	IBM XLC v13.1
Compiler flags	-O3	-O5

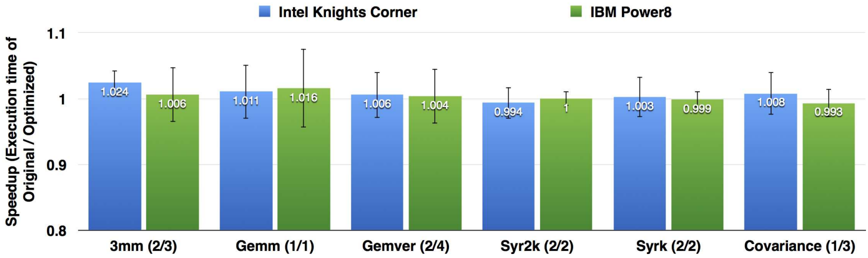
- Two variants:
  - Original OpenMP program
  - OpenMP program after removing redundant barriers

Experiments - OmpSCR Benchmark suite

- Evaluation on 12 benchmarks
- Detected 4 benchmarks as race-free
  - All barriers are necessary to respect program semantics

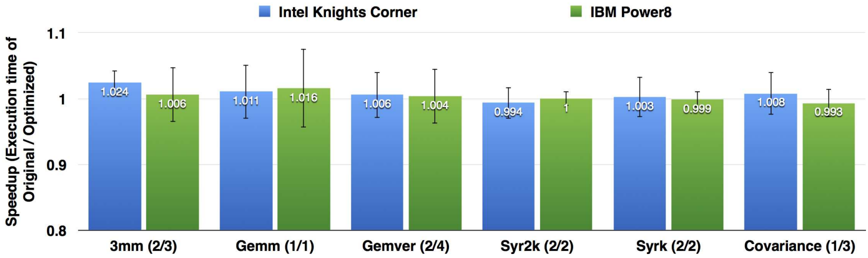
Experiments - Polybench-ACC OpenMP Benchmark suite

- Evaluation on 22 benchmarks
- Detected 14 benchmarks as race-free



Experiments - Polybench-ACC OpenMP Benchmark suite

- Evaluation on 22 benchmarks
- Detected 14 benchmarks as race-free



- Less improvement because of well load-balanced work-sharing loops
- More effective IBM XLC barrier implementation than Intel ICC

Closely related work in barrier analysis

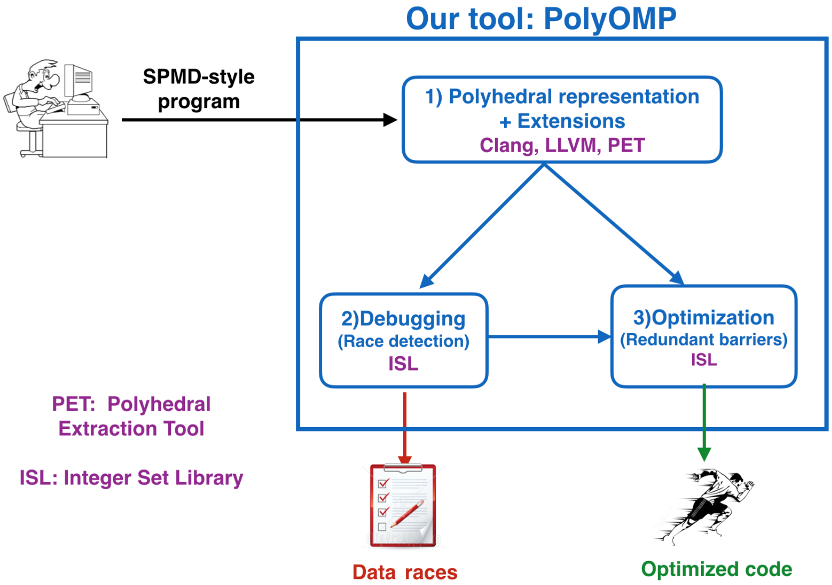
	Style	Key idea	Limitations
Kamil et.al LCPC'05	SPMD	Tree traversal on concurrency graph	Conservative MHP in case of barriers enclosed in loops
Tseng et.al PPoPP'95	SPMD + fork-join	Communication analysis b/w computation partitions	Structure of loops enclosing barriers
Zhao et.al PACT'10	fork-join	SPMDization by loop transformations	Join (barrier) synchronization from only for-all loops
Surendran et.al PLDI'14	fork-join	Dynamic programming on scoped dynamic structure trees	Limited to *finish* construct but the finish placement algorithm is optimal
Our approach	SPMD	Precise MHP analysis with extensions to Polyhedral model	Can support barriers in arbitrarily nested loops

Closely related work in barrier analysis

	Style	Key idea	Limitations
Kamil et.al LCPC'05	SPMD	Tree traversal on concurrency graph	Conservative MHP in case of barriers enclosed in loops
Tseng et.al PPoPP'95	SPMD + fork-join	Communication analysis b/w computation partitions	Structure of loops enclosing barriers
Zhao et.al PACT'10	fork-join	SPMDization by loop transformations	Join (barrier) synchronization from only for-all loops
Surendran et.al PLDI'14	fork-join	Dynamic programming on scoped dynamic structure trees	Limited to *finish* construct but the finish placement algorithm is optimal
Our approach	SPMD	Precise MHP analysis with extensions to Polyhedral model	Can support barriers in arbitrarily nested loops

Limitations in our approach: Greedy heuristic on barrier selection may not be optimal

PolyOMP Infrastructure



PET: Polyhedral Extraction Tool

ISL: Integer Set Library

Conclusions

- Extensions (Space and Phase mappings) to the polyhedral model to capture partial order in SPMD-style programs
- Formalization of May-Happen-in-Parallel (MHP) relations from the extensions
- Approaches for static data race detection and redundant barrier detection in SPMD-style programs
- Demonstration of our approaches on 34 OpenMP programs from the OmpSCR and PolyBench-ACC benchmark suites

Future work

- Enhancing OpenMP dynamic analysis tools for race detection with our MHP analysis
- Replacing barriers with fine grained synchronization for better performance
- Repair of OpenMP programs with barriers
- Enabling classic scalar optimizations (code motion) on concurrency constructs in OpenMP programs

Acknowledgments

- Thesis Committee
  - Prof. Vivek Sarkar,
  - Prof. John M. Mellor-Crummey,
  - Prof. Keith D. Cooper, and
  - Dr. Jun Shirako
- Co-author: Dr. Martin Kong
- Rice Habanero Extreme Scale Software Research Group
- Polyhedral research community
- Family, friends and department staff

Finally,

“Extending the polyhedral compilation model for explicitly parallel programs is a new direction to multi-core programming challenge.”

Thank you!

Serial-elision

A Cilk program, when run on one processor, is semantically equivalent to the C program that results from the deletion of the three keywords. Such a program is called the serial elision or C elision of the Cilk program.

https://arxiv.org/pdf/cs/0608122.pdf

Parallel Loops with Barriers do not satisfy Serial Elision

- New programming models such as Chapel, X10 have barriers enclosed in parallel loops.
- Parallel Loops with Barriers do not satisfy Serial Elision

```
forall (i = ...) {
    S1;
    barrier;
    S2;
}
```

SPMD programs satisfy serial elision if they can be run on 1 thread

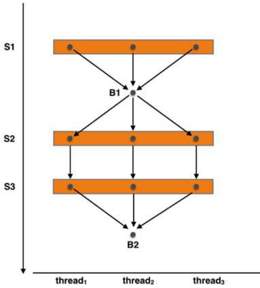
- SPMD programs satisfy serial elision if they can be run on 1 thread

```
#pragma omp parallel
{
        {S1;}

    #pragma omp barrier //B1

        {S2;}
        {S3;}

    #pragma omp barrier //B2
}
```

What about SPMD programs with fixed number of threads?

- May not have an execution on one thread

```
#pragma omp parallel num_threads(2)
{
        {S1;}

    #pragma omp barrier //B1

        {S2;}
        {S3;}

    #pragma omp barrier //B2
}
```



What about SPMD programs with fixed number of threads?

- May not have an execution on one thread