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
Moore’s law still continues
Moore’s law still continues

Performance is driven more by parallelism than single-thread
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

SPMD

Barrier
One-to-all
Nearest Neighbor
Pipelining
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

- 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/
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
Our tool: PolyOMP

1) Polyhedral representation + Extensions

2) Debugging (Race detection)

3) Optimization (Redundant barriers)

SPMD-style program

Data races

Optimized code
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/
for(int i = 1; i < M; i++) {
    for(int j = 1; j < N; j++) {
        S;
    }
}
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)
Limitations of Polyhedral Model

(a) An SPMD-style program

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

    #pragma omp barrier //B1

    {S2;}
    {S3;}

    #pragma omp barrier //B2
}
```

(b) Program execution order

![Program execution order diagram]

Chatarasi, Prasanth (Rice University)  Masters Thesis Defense  April 24th, 2017
(a) An SPMD-style program

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

    #pragma omp barrier //B1

    {S2;}
    {S3;}

    #pragma omp barrier //B2

}
```

(b) Program execution order

Currently, there are no approaches to capture partial orders from SPMD programs and express them onto schedules.
Overall workflow (PolyOMP)

Our tool: PolyOMP

1) Polyhedral representation + Extensions

2) Debugging (Race detection)

3) Optimization (Redundant barriers)

SPMD-style program

Data races

Optimized code
What are the important concepts in SPMD execution?

```c
#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?

```c
#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 \( N = 2 \)

<table>
<thead>
<tr>
<th>Thread 0</th>
<th>Thread 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>S1(0, 0)</td>
<td>Phase = 0</td>
</tr>
<tr>
<td></td>
<td>B1(0, 0)</td>
</tr>
<tr>
<td>S2(0, 0)</td>
<td>Phase = 1</td>
</tr>
<tr>
<td>S1(0, 1)</td>
<td>Phase = 1</td>
</tr>
<tr>
<td></td>
<td>B1(0, 1)</td>
</tr>
<tr>
<td>S2(0, 1)</td>
<td>Phase = 2</td>
</tr>
<tr>
<td></td>
<td>B2(0)</td>
</tr>
<tr>
<td>S3(0)</td>
<td>Phase = 3</td>
</tr>
<tr>
<td>S1(1, 0)</td>
<td>Phase = 3</td>
</tr>
<tr>
<td></td>
<td>B1(1, 0)</td>
</tr>
</tbody>
</table>

...
What are the important concepts in SPMD execution?

Important concepts: 1) Threads and 2) Phases
Space Mapping ($\theta^A$)

Assigns a logical processor id to each statement instance
Space Mapping ($\theta^A$)

Assigns a logical processor id to each statement instance

```c
#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)
}
```

<table>
<thead>
<tr>
<th>Thread 0</th>
<th>Thread 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>S1(0, 0)</td>
<td>S1(0, 0)</td>
</tr>
<tr>
<td>B1(0, 0)</td>
<td></td>
</tr>
<tr>
<td>S2(0, 0)</td>
<td>S2(0, 0)</td>
</tr>
<tr>
<td>S1(0, 1)</td>
<td>S1(0, 1)</td>
</tr>
<tr>
<td>B1(0, 1)</td>
<td></td>
</tr>
<tr>
<td>S2(0, 1)</td>
<td>S2(0, 1)</td>
</tr>
<tr>
<td>Phase = 3</td>
<td>Phase = 0</td>
</tr>
<tr>
<td>S3(0)</td>
<td>S1(0, 0)</td>
</tr>
<tr>
<td>B1(1, 0)</td>
<td></td>
</tr>
<tr>
<td>S1(1, 0)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Space Mapping ($\theta^A$)

Assigns a logical processor id to each statement instance

```c
#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, $\theta^A(S3(i)) = 0$
Phase Mapping ($\theta^P$)

Assigns a logical phase id to each statement instance
Phase Mapping ($\theta^P$)

Assigns a logical phase id to each statement instance

```c
#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)
}
```

<table>
<thead>
<tr>
<th>Thread 0</th>
<th>Phase 0</th>
<th>Thread 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>S1(0, 0)</td>
<td>B1(0, 0)</td>
<td>S1(0, 0)</td>
</tr>
<tr>
<td>S2(0, 0)</td>
<td>Phase = 1</td>
<td>S2(0, 0)</td>
</tr>
<tr>
<td>S1(0, 1)</td>
<td>B1(0, 1)</td>
<td>S1(0, 1)</td>
</tr>
<tr>
<td>S2(0, 1)</td>
<td>Phase = 2</td>
<td>S2(0, 1)</td>
</tr>
<tr>
<td>S1(1, 0)</td>
<td>B1(1, 0)</td>
<td>S1(1, 0)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Phase Mapping ($\theta^P$)
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

<table>
<thead>
<tr>
<th>Thread 0</th>
<th>Thread 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>S1(0, 0)</td>
<td>S1(0, 0)</td>
</tr>
<tr>
<td></td>
<td>Phase = 0</td>
</tr>
<tr>
<td></td>
<td>B1(0, 0)</td>
</tr>
<tr>
<td>S2(0, 0)</td>
<td>S2(0, 0)</td>
</tr>
<tr>
<td>S1(0, 1)</td>
<td>S1(0, 1)</td>
</tr>
<tr>
<td></td>
<td>Phase = 1</td>
</tr>
<tr>
<td></td>
<td>B1(0, 1)</td>
</tr>
<tr>
<td>S2(0, 1)</td>
<td>S2(0, 1)</td>
</tr>
<tr>
<td></td>
<td>Phase = 2</td>
</tr>
<tr>
<td></td>
<td>B2(0)</td>
</tr>
<tr>
<td>S3(0)</td>
<td>S1(1, 0)</td>
</tr>
<tr>
<td>S1(1, 0)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Phase = 3</td>
</tr>
<tr>
<td></td>
<td>B1(1, 0)</td>
</tr>
</tbody>
</table>

\[
RB(S2(0, 1)) = B2(0) \\
RB(S3(0)) = B1(1, 0)
\]
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

<table>
<thead>
<tr>
<th>Thread 0</th>
<th>Thread 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>S1(0, 0)</td>
<td>S1(0, 0)</td>
</tr>
<tr>
<td>S2(0, 0)</td>
<td>S2(0, 0)</td>
</tr>
<tr>
<td>S1(0, 1)</td>
<td>S1(0, 1)</td>
</tr>
<tr>
<td>S2(0, 1)</td>
<td>S2(0, 1)</td>
</tr>
<tr>
<td>S3(0)</td>
<td>S1(1, 0)</td>
</tr>
</tbody>
</table>

\[
\theta^P(S3(0)) = RB(S3(0)) = B1(1, 0)
\]

\[
\theta^P(S1(1, 0)) = RB(S1(1, 0)) = B1(1, 0)
\]
### Observation

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

<table>
<thead>
<tr>
<th>Thread 0</th>
<th>Thread 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>S1(0, 0)</td>
<td>S1(0, 0)</td>
</tr>
<tr>
<td>Phase = 0</td>
<td>Phase = 0</td>
</tr>
<tr>
<td>B1(0, 0)</td>
<td>B1(0, 0)</td>
</tr>
<tr>
<td>S2(0, 0)</td>
<td>S2(0, 0)</td>
</tr>
<tr>
<td>S1(0, 1)</td>
<td>S1(0, 1)</td>
</tr>
<tr>
<td>Phase = 1</td>
<td>Phase = 1</td>
</tr>
<tr>
<td>B1(0, 1)</td>
<td>B1(0, 1)</td>
</tr>
<tr>
<td>S2(0, 1)</td>
<td>S2(0, 1)</td>
</tr>
<tr>
<td>Phase = 2</td>
<td>Phase = 2</td>
</tr>
<tr>
<td>B2(0)</td>
<td>B2(0)</td>
</tr>
<tr>
<td>S3(0)</td>
<td>S3(0)</td>
</tr>
<tr>
<td>S1(1, 0)</td>
<td>S1(1, 0)</td>
</tr>
<tr>
<td>Phase = 3</td>
<td>Phase = 3</td>
</tr>
<tr>
<td>B1(1, 0)</td>
<td>B1(1, 0)</td>
</tr>
</tbody>
</table>

\[
\theta^P(S3(0)) = RB(S3(0)) = B1(1, 0) \\
\theta^P(S1(1, 0)) = RB(S1(1, 0)) = B1(1, 0) \\
\rightarrow \theta^P(S3(0)) = \theta^P(S1(1, 0))
\]
How to compute phase mappings?

Observation

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

<table>
<thead>
<tr>
<th>Thread 0</th>
<th>Thread 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>S1(0, 0)</td>
<td>S1(0, 0)</td>
</tr>
<tr>
<td>Phase = 0</td>
<td>Phase = 0</td>
</tr>
<tr>
<td>B1(0, 0)</td>
<td>B1(0, 0)</td>
</tr>
<tr>
<td>S2(0, 0)</td>
<td>S2(0, 0)</td>
</tr>
<tr>
<td>S1(0, 1)</td>
<td>S1(0, 1)</td>
</tr>
<tr>
<td>Phase = 1</td>
<td>Phase = 1</td>
</tr>
<tr>
<td>B1(0, 1)</td>
<td>B1(0, 1)</td>
</tr>
<tr>
<td>S2(0, 1)</td>
<td>S2(0, 1)</td>
</tr>
<tr>
<td>Phase = 2</td>
<td>Phase = 2</td>
</tr>
<tr>
<td>B2(0)</td>
<td>B2(0)</td>
</tr>
<tr>
<td>S3(0)</td>
<td>S3(0)</td>
</tr>
<tr>
<td>S1(1, 0)</td>
<td>S1(1, 0)</td>
</tr>
<tr>
<td>Phase = 3</td>
<td>Phase = 3</td>
</tr>
<tr>
<td>B1(1, 0)</td>
<td>B1(1, 0)</td>
</tr>
<tr>
<td></td>
<td>S1(1, 0)</td>
</tr>
</tbody>
</table>

\[ \theta^P(S3(0)) = RB(S3(0)) \]
\[ = B1(1, 0) \]
\[ \theta^P(S1(1, 0)) = RB(S1(1, 0)) \]
\[ = B1(1, 0) \]
\[ \implies \theta^P(S3(0)) = \theta^P(S1(1, 0)) \]

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

<table>
<thead>
<tr>
<th>MHP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Two statement instances can run in parallel if they are run by different threads and are in same phase of computation</td>
</tr>
</tbody>
</table>

Now, program order information in polyhedral model

- (Space ($\theta^A$), Phase ($\theta^P$), Schedule ($\theta$))
Overall workflow (PolyOMP)

Our tool: PolyOMP

1) Polyhedral representation + Extensions
2) Debugging (Race detection)
3) Optimization (Redundant barriers)

SPMD-style program

Data races

Optimized code
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
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

```c
#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

```c
#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

```c
#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
   - \( \text{Race}(S, T) = \text{true on 'k'} \)
   - \( \implies \; \text{MHP}(S, T) = \text{true and } S, T \text{ conflict on 'k'} \)
   - \( \implies \; \theta^A(S) \neq \theta^A(T) \text{ and } \theta^P(S) = \theta^P(T) \text{ and } S, T \text{ conflict on 'k'} \)

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
   - \( \text{Race}(S, T) = \text{true on 'k'} \)
   - \( \implies \text{MHP}(S, T) = \text{true and S,T conflict on 'k'} \)
   - \( \implies \theta^A(S) \neq \theta^A(T) \) 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 \textit{data races}
Our approach on the motivating example

```c
#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

```c
#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: $\theta^A(S2) \neq \theta^A(S1)$
- $\land \theta^A(S2) = 0$
Our approach on the motivating example

```c
#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:** $\theta^A(S2) \neq \theta^A(S1)$
  $\land \theta^A(S2) = 0$

- **Phase:** $\theta^P(S2) = \theta^P(S1)$
  $\rightarrow B1(x' + 1) = B1(x'')$
  $\rightarrow x' + 1 = x''$
Our approach on the motivating example

```c
#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: \( \theta^A(S2) \neq \theta^A(S1) \land \theta^A(S2) = 0 \)
- 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

```c
#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: $\theta^A(S2) \neq \theta^A(S1) \\
  \land \theta^A(S2) = 0$

- Phase: $\theta^P(S2) = \theta^P(S1)$
  \rightarrow B1(x' + 1) = B1(x'')
  \rightarrow x' + 1 = x''$

- Conflict: TRUE (same location 'k')

Satisfiable assignment: $(\theta^A(S2) = 0, x' = 0)$ and $(\theta^A(S1) = 1, x'' = 1)$
Our approach on the motivating example

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

```c
#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)
Evaluation on 12 benchmarks

Identified all documented races (5)
Experiments - OmpSCR Benchmark suite

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

![Bar chart showing actual data races and false positives for different benchmarks.]

- Jacobi01: 2 actual, 2 false positives
- Jacobi03: 2 actual, 2 false positives
- LoopA.bad: 1 actual, 1 false positive
- LoopA.sol2: 7 actual
- LoopB.bad1: 1 actual, 1 false positive
- LoopB.bad2: 1 actual, 1 false positive
- LoopB.pipe: 7 actual
Experiments - OmpSCR Benchmark suite

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

\[ \text{Number of races} \]

<table>
<thead>
<tr>
<th></th>
<th>Actual data races</th>
<th>False positives</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jacobi01</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>Jacobi03</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>LoopA.bad</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>LoopA.sol2</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>LoopB.bad1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>LoopB.bad2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>LoopB.pipe</td>
<td>7</td>
<td>7</td>
</tr>
</tbody>
</table>

- False positives because of linearized array subscripts
<table>
<thead>
<tr>
<th>Tool</th>
<th>ARCHER (Static + Dynamic)</th>
<th>Intel Inspector XE (Dynamic)</th>
<th>ARCHER (Static)</th>
<th>PolyOMP (Static)</th>
</tr>
</thead>
<tbody>
<tr>
<td>True races</td>
<td>5</td>
<td>5</td>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>False +ves</td>
<td>0</td>
<td>5</td>
<td>*</td>
<td>27</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Tool</th>
<th>ARCHER (Static + Dynamic)</th>
<th>Intel Inspector XE (Dynamic)</th>
<th>ARCHER (Static)</th>
<th>PolyOMP (Static)</th>
</tr>
</thead>
<tbody>
<tr>
<td>True races</td>
<td>5</td>
<td>5</td>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>False +ves</td>
<td>0</td>
<td>5</td>
<td>*</td>
<td>27</td>
</tr>
</tbody>
</table>

- 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

![Bar chart showing actual data races and false positives for different benchmarks.]

- Actual data races
- False positives

- Number of races:
  - Atax: 2
  - Bicg: 2
  - Cholesky: 28
  - Symm: 5
  - Trmm: 1
  - Durbin: 6
  - Gramschmidt: 12
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
Shared scalar variables inside the work-sharing loops (Eg: cholesky.c)

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

```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)

```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

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

Our tool: PolyOMP

1) Polyhedral representation + Extensions

2) Debugging (Race detection)

3) Optimization (Redundant barriers)

Data races

Optimized code
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)
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.
A sequence of matrix multiplications, i.e., $E = A \times B$; $F = C \times D$; $G = E \times F$;
Overall execution with data dependences

\[ E = A \times B \]

\[ B_1 \]

\[ F = C \times D \]

\[ B_2 \]

\[ G = E \times F \]

thread\(_1\)  \hspace{1cm} \text{thread} 2
Overall execution with data dependences

Barrier B1 is redundant 😊
Optimization of SPMD-style programs - Redundant barriers

```c
#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
    } //B1

    #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
    }
}

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

<table>
<thead>
<tr>
<th></th>
<th>Intel KNC</th>
<th>IBM Power 8</th>
</tr>
</thead>
<tbody>
<tr>
<td>Micro architecture</td>
<td>Xeon Phi</td>
<td>Power PC</td>
</tr>
<tr>
<td>Total threads</td>
<td>228</td>
<td>192</td>
</tr>
<tr>
<td>Compiler</td>
<td>Intel ICC v15.0</td>
<td>IBM XLC v13.1</td>
</tr>
<tr>
<td>Compiler flags</td>
<td>-O3</td>
<td>-O5</td>
</tr>
</tbody>
</table>
Experimental Setup

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

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

<table>
<thead>
<tr>
<th></th>
<th>Intel KNC</th>
<th>IBM Power 8</th>
</tr>
</thead>
<tbody>
<tr>
<td>Micro architecture</td>
<td>Xeon Phi</td>
<td>Power PC</td>
</tr>
<tr>
<td>Total threads</td>
<td>228</td>
<td>192</td>
</tr>
<tr>
<td>Compiler</td>
<td>Intel ICC v15.0</td>
<td>IBM XLC v13.1</td>
</tr>
<tr>
<td>Compiler flags</td>
<td>-O3</td>
<td>-O5</td>
</tr>
</tbody>
</table>

- 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
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

| 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 \textit{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

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

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

1) Polyhedral representation
   + Extensions
   Clang, LLVM, PET

2) Debugging
   (Race detection)
   ISL

3) Optimization
   (Redundant barriers)
   ISL

SPMD-style program

PET: Polyhedral Extraction Tool
ISL: Integer Set Library

Data races
Optimized code

Chatarasi, Prasanth (Rice University)
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!
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.

New programming models such as Chapel, X10 have barriers enclosed in parallel loops.

Parallel Loops with Barriers do not satisfy Serial Elision

```plaintext
forall (i = ...)
S1;
barrier;
S2;

1
2
3
4
5
```
SPMD programs satisfy serial elision if they can be run on 1 thread

```c
#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

```cpp
#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