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

- Performance is driven 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

- Two major compiler approaches in tackling the challenge
  - Automatic parallelization of sequential programs
    - Compiler extract parallelism
    - Not much burden on programmer but lot of limitations exist!
  - 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

- We 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 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 support SPMD-style parallel programs as input 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)

Data races

Optimized code

SPMD-style program

Overall flow of the talk
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
for(int i = 1; i < M; i++) {
    for(int j = 1; j < N; j++) {
        A[i][j] = MAX(A[i-1][j], A[i-1][j-1], A[i][j-1]); // S
    }
}

Schedule ($\theta$) – 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 general)

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

```
Thread 0
```

```
\[\text{S1} \quad \text{B1} \quad \text{S2} \quad \text{S3}\]
```

```
Thread 1
```

```
\[\text{S1} \quad \text{B1} \quad \text{S2} \quad \text{S3}\]
```
Limitations of Polyhedral Model

Currently, there are no approaches to capture partial orders from SPMD programs and express 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 important in SPMD program execution?

Majorly, two are important, i.e., 1) Threads and 2) Phases

```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>S1(0, 0)</td>
</tr>
<tr>
<td></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>Phase = 1</td>
<td>S1(0, 1)</td>
</tr>
<tr>
<td></td>
<td>B1(0, 1)</td>
</tr>
<tr>
<td>S1(0, 1)</td>
<td>S1(0, 1)</td>
</tr>
<tr>
<td>Phase = 2</td>
<td></td>
</tr>
<tr>
<td>B2(0)</td>
<td></td>
</tr>
<tr>
<td>S3(0)</td>
<td></td>
</tr>
<tr>
<td>Phase = 3</td>
<td></td>
</tr>
<tr>
<td>S1(1, 0)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Extension 1 – Thread/Space/Allocation Mapping

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)
        }
    }
}
```

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

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

<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>S3(0)</td>
<td>Phase 3</td>
<td></td>
</tr>
<tr>
<td>S1(1, 0)</td>
<td>B1(1, 0)</td>
<td>S1(1, 0)</td>
</tr>
</tbody>
</table>

\[ RB(S2(0, 1)) = B2(0) \]
\[ RB(S3(0)) = B1(1, 0) \]
How to compute phase mappings?

**Observation**

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

<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></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>Phase = 1</td>
<td></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 = 2</td>
<td></td>
</tr>
<tr>
<td>B2(0)</td>
<td></td>
</tr>
<tr>
<td>S3(0)</td>
<td>S1(1, 0)</td>
</tr>
<tr>
<td>S1(1, 0)</td>
<td></td>
</tr>
<tr>
<td>Phase = 3</td>
<td></td>
</tr>
<tr>
<td>B1(1, 0)</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>...</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.

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 the same phase of computation.

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)

Data races

Optimized code

SPMD-style program
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 benchmark

1 #pragma omp parallel shared(U, V, k)
2 {
3     while (k <= Max)  // S1
4         {
5             #pragma omp for nowait
6             for(i = 0 to N)
7                 U[i] = V[i];
8             #pragma omp barrier
9
10            #pragma omp for nowait
11            for(i = 1 to N-1)
12                V[i] = U[i-1] + U[i] + U[i+1];
13            #pragma omp barrier
14
15            #pragma omp master
16                { k++;}  // S2
17         }
18     }

- 1-dimensional stencil from OmpSCR suite
- Race b/w S1 and S2 on variable 'k'
- Our goal: Detect such races at compile-time
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 motivating benchmark

```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 S1(x') & S2(x'')

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

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

- Conflict: TRUE (same location 'k')

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

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

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

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

- False positives because of linearized subscripts
Experiments - Polybench-ACC OpenMP Benchmark suite

- Evaluation on 22 benchmarks

- NO False positives (All verified)

- Majority of races are from:
  - Shared scalar variables inside the work-sharing loops
Strengths and Limitations of our approach

Strengths

- Input independent and schedule independent
- Guaranteed to be exact 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>Tool</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 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

SPMD-style program
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.
#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;
```
#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
    }
}

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

- 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., IBM Power 8 and Intel Knights Corner

<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 v15.0.0</td>
<td>IBM XLC v13.1.2</td>
</tr>
<tr>
<td>Compiler flags</td>
<td>-O3 -fast(icc)</td>
<td>-O3</td>
</tr>
</tbody>
</table>

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

Less improvement because of well load-balanced work-sharing loops

Effective IBM XLC barrier implementation than Intel ICC barrier
## 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><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>

Chatarasi, Prasanth (Rice University)  Masters Thesis Defense  April 24th, 2017
Our tool: PolyOMP

1) Polyhedral representation + Extensions
   Clang, LLVM, PET

2) Debugging (Race detection)
   ISL

3) Optimization (Redundant barriers)
   ISL

Data races

Optimized code

PET: Polyhedral Extraction Tool

ISL: Integer Set Library

SPMD-style program
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) 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 and friends*
Finally,

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

Thank you!