An Extended Polyhedral Model for SPMD Programs and its use in Static Data Race Detection

Prasanth Chatarasi, Jun Shirako, Martin Kong, Vivek Sarkar

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

The 29th International Workshop on Languages and Compilers for Parallel Computing (LCPC 2016)

September 29, 2016

Introduction

- Productivity and scalability in parallel programming models
  - Demand for new compilation techniques

- In this talk, we consider SPMD-style parallelism
  - *All logical processors (worker threads) execute the same program, with sequential code executed redundantly and parallel code (worksharing constructs, barriers, etc.) executed cooperatively*

  - OpenMP for multicores, CUDA/ OpenCL for accelerators, MPI for distributed

  - Data races, deadlocks are common issues in SPMD programs

Data races in shared memory SPMD models

- *Conflicting access to shared memory location without synchronization*

```
#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 jacobi from OmpSCR
- Race b/w S1 and S2 on variable 'k'
- Our Goal: Detect such races in SPMD programs at compile-time

Data races in shared memory SPMD models

- *Conflicting access to shared memory location without synchronization*

```
#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 jacobi from OmpSCR
- Race b/w S1 and S2 on variable 'k'
- Our Goal: Detect such races in SPMD programs at compile-time

Data races in shared memory SPMD models

- *Conflicting access to shared memory location without synchronization*

```
#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 jacobi from OmpSCR
- Race b/w S1 and S2 on variable 'k'
- Our Goal: Detect such races in SPMD programs at compile-time

Prasanth Chatarasi, Jun Shirako, Martin Kong, Vivek Sarkar

Data races in shared memory SPMD models

- Assumption: Textually aligned barriers
  - *SPMD execution can be partitioned into a sequence of phases separated by barriers.*

- There exists a race between S & T iff
  - Access same memory location and at-least one is write
  - May happen in parallel
    - Run by different threads
    - In the same phase of computation

Data races in shared memory SPMD models

- Assumption: Textually aligned barriers
  - *SPMD execution can be partitioned into a sequence of phases separated by barriers.*

- There exists a race between S & T iff
  - Access same memory location and at-least one is write
  - May happen in parallel
    - Run by different threads
    - In the same phase of computation

Data races in shared memory SPMD models

- *Conflicting access to shared memory location without synchronization*

```
#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

       #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
    }
}
```

- Access same memory location and write ??
- Yes, on variable 'k'
- Run by different threads ??
- Yes, if thread (S1) != 0
- In same phase of computation ??
- Yes, S2(x) and S1(x+1) where x is iteartor of while loop.

Data races in shared memory SPMD models

- *Conflicting access to shared memory location without synchronization*

```
#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

       #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
    }
}
```

- Access same memory location and write ??
- Yes, on variable 'k'

- Run by different threads ??
- Yes, if thread (S1) != 0

- In same phase of computation ??
- Yes, S2(x) and S1(x+1) where x is iteartor of while loop.

Data races in shared memory SPMD models

- *Conflicting access to shared memory location without synchronization*

```
#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

       #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
    }
}
```

- Access same memory location and write ??
- Yes, on variable 'k'

- Run by different threads ??
- Yes, if thread (S1) != 0

- In same phase of computation ??
- Yes, S2(x) and S1(x+1) where x is iteartor of while loop.

Phase computation

How to compute phases for a given statement (S) ??

- Numbering is hard !
- We compute phase of S in terms of "Reachable barriers":
  - *Set of barrier instances that can be executed after S without an intervening barrier*
- Two statements are in same phase iff they have same reachable barrier instances

Phase computation

How to compute phases for a given statement (S) ??

- Numbering is hard !
- We compute phase of S in terms of "Reachable barriers":
  - *Set of barrier instances that can be executed after S without an intervening barrier*
- Two statements are in same phase iff they have same reachable barrier instances

Phase computation

How to compute phases for a given statement (S) ??

- Numbering is hard !
- We compute phase of S in terms of “Reachable barriers”:
  - *Set of barrier instances that can be executed after S without an intervening barrier*
- Two statements are in same phase iff they have same reachable barrier instances

Reachable barriers

```
#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 // B2

       #pragma omp master
        { k++;} // S2
    }
} // B3
```

Reachable barriers of S1(x)

- B1(x) if x lies in loop range
- B3 else

Reachable barriers of S2(x)

- B1(x+1) if x+1 lies in loop range
- B3 else

Hence, S1(x+1) & S2(x) in same phase

Reachable barriers

```
#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 // B2

       #pragma omp master
        { k++;} // S2
    }
} // B3
```

Reachable barriers of S1(x)

- B1(x) if x lies in loop range
- B3 else

Reachable barriers of S2(x)

- B1(x+1) if x+1 lies in loop range
- B3 else

Hence, S1(x+1) & S2(x) in same phase

Reachable barriers

```
#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 // B2

       #pragma omp master
        { k++;} // S2
    }
} // B3
```

Reachable barriers of S1(x)

- B1(x) if x lies in loop range
- B3 else

Reachable barriers of S2(x)

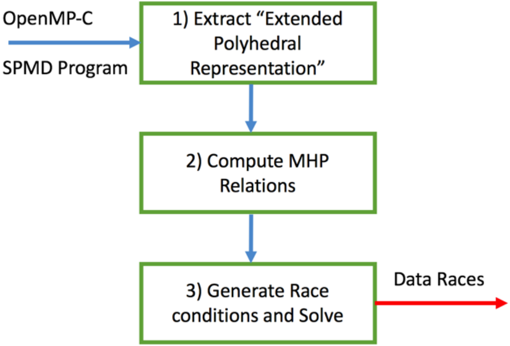
- B1(x+1) if x+1 lies in loop range
- B3 else

Hence, S1(x+1) & S2(x) in same phase

Extensions to Polyhedral Model

- For each statement ‘S’ in polyhedral representation
  - Domain - Set of statement instances
  - Access relations - Memory location touched
  - Schedule - execution time stamp
- Existing “Schedule” is not sufficient for SPMD programs
  - Captures only serial execution order
- We add the following to each statement ‘S’ :
  - Space - executing thread id
  - Phase - execution time stamp of reachable barriers

Overall workflow (PolyOMP)



- Polyhedral Extraction Tool (PET)
  - CLANG 3.5 with support of OpenMP 4.0
- Integer Set Library (ISL)

Race conditions

```
#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 // B2

       #pragma omp master
        { k++;} // S2
    }
} // B3
```

Race condition b/w S1(x_{S1}) & S2(x_{S2})

- Thread(S1) != 0 and
- $x_{S1} = x_{S2} + 1$
- TRUE (same memory location)

Experiments - OmpSCR Benchmark suites

Benchmarks		12
Documented races		5
Static: PolyOMP	Overall Detection time[1] (sec)	13.03
	Reported races	32
	False Positives	27
Dynamic: Intel Inspector XE Reported races		10
Hybrid: ARCHER Reported races		5

- False positives in case of PolyOMP : Linearized subscripts
- False negatives in case of Inspector : Races on worksharing loop iteartors

[1] On a quad core-i7 machine (2.2GHz) with 16GB memory

Experiments - PolyBench-ACC Benchmark suites

Benchmarks		22
Static: PolyOMP	Overall Detection time[2] (sec)	14.41
	Reported races	61
	False Positives	0
Dynamic: Intel Inspector XE Reported races		31

- NO False positives in case of PolyOMP : Affine programs
- Majority of races are from :
  - Non-privatized scalar variables inside the worksharing loops
  - Updating common array elements in sequential loops of SPMD

[2]On a quad core-i7 machine (2.2GHz) with 16GB memory

Recent static approaches for race detection in case of OpenMP

	Supported Constructs	Approach	Guarantees	False +Ves	False -Ves
Pathg (Yu et.al) LCTES'12	OpenMP worksharing loops, Simple Barriers, Atomic	Thread automata	Per number of threads	Yes	No
OAT (Ma et.al) ICPP'13	OpenMP worksharing loops, Barriers, locks, Atomic, single, master	Symbolic execution	Per number of threads	Yes	No
ompVerify (Basupalli et.al) IWOMP'11	OpenMP 'parallel for'	Polyhedral (Dependence analysis)	Per 'parallel for' loop	No - (Affine subscripts)	No - (Affine subscripts)
Our Approach	OpenMP worksharing loops, Barriers in arbitrary nested loops, Single, master	Polyhedral (MHP relations)	Per SPMD region	No - (Affine subscripts) Yes - (Non affine)	No

Conclusions

- Extensions to the polyhedral compilation model for SPMD programs
- Formalization of May Happen in Parallel (MHP) relations in the extended model
- An approach for static data race detection in SPMD programs
- Demonstration of our approach on 34 OpenMP programs from the OmpSCR and PolyBench-ACC benchmark suites.

Future work

- Debugging:
  - Deadlock detection in MPI (SPMD-Style)
  - Hybrid Race detection for SPMD
- Optimizations:
  - Redundant barrier removal optimization in SPMD
  - Fusion of CUDA Kernel calls (Fusion of SPMD regions)

Finally,

- *Compiler Analysis for Debugging and Optimizations of explicitly parallel programs is an important direction to improve productivity and scalability of parallel programs.*

- Acknowledgments
  - Rice Habanero Extreme Scale Software Research Group
  - LCPC 2016 Program Committee
  - IMPACT 2016 Program Committee
  - PACT 2015 ACM SRC Committee

- Thank you!