Extending Polyhedral Model for Analysis and Transformation of OpenMP programs

Prasanth Chatarasi
Rice University
prasanth@rice.edu

Vivek Sarkar (Academic Advisor)
Rice University
vsarkar@rice.edu

1. Motivation & Introduction

The polyhedral model is a powerful algebraic framework that has enabled significant advances in analysis and transformation of *sequential affine (sub)programs*, relative to traditional AST-based approaches. However, given the rapid growth of parallel software, there is a need for increased attention to using polyhedral compilation techniques to analyze and transform *explicitly parallel programs*. In our PACT'15 paper titled "Polyhedral Optimizations of Explicitly Parallel Programs" [1, 2], we addressed the problem of analyzing and transforming programs with explicit parallelism that satisfy the *serial-elision* property, i.e., the property that removal of all parallel constructs results in a sequential program that is a valid (albeit inefficient) implementation of the parallel program semantics.

In this poster, we address the problem of analyzing and transforming more general OpenMP programs that do not satisfy the *serial-elision* property. Our contributions include the following: 1) An extension of the polyhedral model to represent input OpenMP programs, 2) Formalization of May Happen in Parallel (MHP) and Happens before (HB) relations in the extended model, 3) An approach for static detection of data races in OpenMP programs by generating race constraints that can be solved by an SMT solver such as Z3, and 4) An approach for transforming OpenMP programs.

2. Extended Polyhedral Representation

Loop nests amenable to polyhedral representation are called *Static Control Parts* (SCoPs). A SCoP consists of a set of consecutive statements, and each statement is represented using three elements 1) Iteration domain $\mathcal{D}^S(\vec{i})$ to capture a set of the statement instances $S(\vec{i})$, 2) Access relations $\mathcal{A}^S(\vec{i})$ to identify memory locations accessed in the statement instances $S(\vec{i})$, and 3) Scattering function to describe the order in which statement instances have to be executed relative to each other, and contains a schedule map of the form $\Theta^S(\vec{i})$, which assign logical timestamps to the statement instances $S(\vec{i})$.

A major difference between a sequential program and an explicitly parallel program (such as OpenMP) is that a sequential program specifies a total execution order among statement instances, and an explicitly parallel program specifies a partial execution order. The existing schedule function ($\Theta^S(\vec{i})$) for sequential programs captures the total execution order very effectively. The same schedule function ($\Theta^S(\vec{i})$) can also specify parallelism by 1) Not specifying any ordering information, which means statement instances can be executed in any order, 2) Assigning same logical timestamp, which means statement instances can be executed simultaneously at the same time. But, this representation is not sufficient to specify the kinds of parallelism and synchronization constructs present in OpenMP parallel programs (e.g., barriers, point-to-point synchronizations).

In our work, the scattering (ordering) function of a statement is extended with additional mapping information such as allocation (*space*) and computational phase (*phase*) to capture OpenMP constructs. Allocation (space) mapping assigns processor ids, expressing on which logical processor a statement instance $S(\vec{i})$ has to be executed. A key property of OpenMP programs is that their execution can be partitioned into a sequence of phases separated by barriers. The *phase* mapping assigns a logical identifier, that we refer to as a *phase stamp*, to each statement instance $S(\vec{i})$.

3. Static detection of data races

Data races are a major source of errors in parallel programs. Complicating matters, data races may occur only in few of the possible schedules of a parallel program, thereby making them extremely hard to detect and reproduce. Our race detection algorithm considers a read/write or write/write pair on the same shared variable in the same parallel region and generate race constraints as mentioned below.

- Statement instance S, T should touch same memory location and one of them should be a write.
  - Captured from *access relations*
- S, T should happen in parallel
  - S, T should be in same phase of computation
    - Captured from *phase* mappings
  - S, T should be executed by different threads
    - Captured from *space* mappings

PACT '15, October 18–21, 2015, San Francisco, CA, USA.

http://dx.doi.org/10.1145/nnnnnnn.nnnnnnn

```
// T - total number of threads,
// tid - thread ID
#pragma omp parallel  {
    A[tid] =  .... // S1
    #pragma omp for schedule(static, 2)
    for(i : 0 to N)
        A[i] = .... // S2
    #pragma omp barrier
    A[tid] = .... // S3
}
/* Extended Schedule (Phase, Space, Time):
S1: (0, tid, 0), S2: (0, (i % 2T)/2, 1, i),
S3: (1, tid, 0) */
```

Figure 1: An OpenMP program with a data race (b/w S1 and S2) involving a loop worksharing construct with static schedule and chunk size value of 2.

These race constraints encode necessary conditions for conflicting accesses to that shared variable by two threads. Then, the race constraint is passed onto a Z3 SMT solver for the existence of solutions. If the race constraint is not satisfiable, then we conclude that there are no races on those pair of accesses. Our approach is guaranteed to be exact (with neither false positives nor false negatives) if the input program satisfies all the standard preconditions of the polyhedral model (without any non-affine constructs) and the race constraints are decidable by Z3 SMT solver. If the conservative estimations (for non-affine constructs) are used during representation and analysis such as may-access relations, then this approach may induce false positives. The race conditions between statements `S1` and `S2` in Figure 1 is $\{(tid, tid', i') \mid (tid = i') \wedge (tid != tid') \wedge (tid' = (i'\%2T)/2) \wedge (0 = 0)\}$. The possible solution from the Z3 SMT solver is (T = 2, tid = 1, tid′ = 0, i′ = 1). It can be interpreted as a race between statement `S1` executed by thread with id as 1, and statement `S2` executed by thread with id as 2 and iterator value as 1. These kinds of races (dependent on schedule techniques and chunk sizes) are often unreported by many static analysis techniques.

4. Transformation of OpenMP Programs

As software with explicit parallelism is on the rise, transforming explicitly parallel programs is very crucial in obtaining speedups and scalability over a variety of machines. So, we extend the definition of data dependence with the happens-before relations for ordering among statement instances.

- Statement instance S, T should touch same memory location and one of them should be a write.
  - Captured from *access relations*
- S should happen-before T
  - S, T should be in different phase of computation
    - Captured from *phase* mappings
  - (or) S, T should be executed by same thread and S occurs before T in the same phase
    - Captured from *space, time* mappings

Based on the data dependence relations between `S1` and `S2` in Figure 2, the explicit barrier between loop worksharing constructs can be removed for performance and preserving semantics of the program.

```
// T - total number of threads,
#pragma omp parallel  {
 #pragma omp for schedule(static, 2) nowait
 for(i : 0 to N)
    A[i] = .... // S1
 #pragma omp barrier
 #pragma omp for schedule(static, 2)
 for(i : 0 to N)
    ..... = A[i] // S2
}
/* Extended Schedule (Phase, Space, Time):
S1: (0, (i % 2T)/2, 0, i),
S2: (1, (i % 2T)/2, 0, i) */
```

Figure 2: An OpenMP program, where explicit barrier between loop worksharing constructs can be removed for performance.

The producer thread of `A[i]` in the statement `S1` is same as consumer thread of `A[i]` in the statement `S2` even after removal of barrier statement because of same static schedule and chunk size present in both worksharing constructs.

5. Acknowledgments

We acknowledge Jun Shirako for the discussions, and members of the Habanero Extreme Scale Software Research Group at Rice University for the feedback on earlier drafts of this work.

References

[1] P. Chatarasi, J. Shirako, and V. Sarkar. Polyhedral optimizations of explicitly parallel programs. In *Proc. of The 24th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, San Francisco, CA, USA, 2015.

[2] P. Chatarasi, J. Shirako, and V. Sarkar. Polyhedral transformations of explicitly parallel programs. In *5th International Workshop on Polyhedral Compilation Techniques (IMPACT)*, Amsterdam, Netherlands, Jan. 2015.