Parallel Data Flow Analysis for OpenMP Programs

Lei Huang, Girija Sethuraman, and Barbara Chapman

University of Houston, Houston TX 77004, USA,
{leihuang,girija,chapman}@cs.uh.edu

Abstract. The paper presents a compiler framework for analyzing and optimizing OpenMP programs. The framework includes Parallel Control Flow Graph and Parallel Data Flow equations based on the OpenMP relaxed memory consistency model. It enables traditional compiler analyses as well as specific optimizations for OpenMP. Based on the framework, we describe dead code elimination and barrier elimination algorithms. An OpenMP code example is showed in the paper to illustrate the optimizations. The framework guarantees that the traditional optimizations can be performed safely to OpenMP programs, and it further increases the opportunities for more aggressive optimizations.

1 Introduction

OpenMP [7] and PThreads [2] are the most widely used programming models for parallelizing applications in shared memory systems. Whereas PThreads often requires major reorganization of a program's structure, the insertion of OpenMP directives is often straightforward. OpenMP directives impose a structured programming style with a simple means for synchronization that helps avoid some kinds of programming errors. A compiler translates an OpenMP code to threaded C/C++, Fortran code that will be linked with a thread library. It is easier for a compiler to analyze an OpenMP code than its corresponding threaded code due to its very structured style. However, most compilers do not exploit the fact to analyze an OpenMP code. There is little or no optimization in most OpenMP compilers [8, 6] before an OpenMP code is translated to a threaded code. Fig. 1 shows an OpenMP code and a compiler generated threaded code after the translation. Based on OpenMP semantics, it is known that k is equal to 1 after the OpenMP *single* construct. However, in the translated code, a compiler is not sure if k is equal to 1 or not at the *if(k==1)* statement since the value of *mpsp_status* is unknown at compile time.

```
#pragma omp single
  {
    k = 1;
  }
  if(k==1) ...
```

(a) An OpenMP program with single construct

```
mpsp_status = __ompc_single(__ompv_temp_gtid);
if(mpsp_status == 1)
{
  k = 1;
}
__ompc_end_single(__ompv_temp_gtid);

if(k==1) ...
```

(b) The corresponding compiler translated threaded code

Fig. 1. A compiler translated OpenMP code

2 OpenMP Memory Model

As described in [4] and OpenMP 2.5 specification, OpenMP is based on the relaxed consistency memory model, which means that each thread is allowed to have its own local view of shared data. The local value of a shared object may or may not be consistent unless there is a flush operation to force the value to be consistent over all threads. Besides the OpenMP flush directive, OpenMP synchronization mechanism (*omp barrier, omp critical, omp atomic, and locks*) contain implicit flush to make the data consistent. Most OpenMP constructs have implicit barriers at the end of them to ensure the synchronization of threads execution and keep the data consistent between them. An aggressive optimizing compiler can safely assume that there is no inter-thread data interactions until a flush operation has been reached. The OpenMP memory model simplifies the compiler analysis for parallel programs since a compiler can perform traditional analysis and optimizations safely between two synchronization operations. Therefore, most OpenMP compiler performs compiler optimizations after OpenMP has been translated to a threaded code, and limits the sequential analysis and optimizations to be performed only between two synchronizations.

3 Parallel Data Flow Analysis Framework for OpenMP

The idea of PCFG is similar to Program Execution Graph [1] and the Synchronized Control Flow Graph [3]. The distinction between our PCFG with them is that our PCFG is based on barrier and flush synchronizations, instead of event based synchronizations (such as post-wait). The PCFG is a directed graph (N, E, s, e), where N is the set of nodes including basic nodes, composite nodes, super nodes, and barrier nodes; E is a set of directed edges including sequential edges and parallel edges; s and e represent the entry and exit of a parallel region, respectively. A basic node is a basic block, or contains a **omp flush** directive. A composite node is composed of an OpenMP worksharing or synchronization construct and the basic nodes associated with it. A barrier node contains **omp barrier** directive only. A sequential edge indicates the sequential control flow within a thread. A parallel edge indicates a branch that more threads may take.

Fig 2 shows how the composite nodes containing worksharing directives are connected by parallel and sequential edges. In Fig 2 A and B, different threads may take different paths, so that parallel edges represent the branches for different threads. In Fig. 2 C, the *omp for* loop will be executed by all threads, and we use a sequential edge to connect it. Based on the *omp for* directive semantics, the enclosed loop should not have any data dependence. We treat the loop as a sequential loop in the PCFG. Fig. 2 D presents the PCFG for *omp critical* construct. A critical section is executed by multiple threads one by one, but never at same time. It is similar to the execution of a loop in a sequential program in terms of data propagation. We create a backward edge in the critical construct, so that a data defined in the critical region will be visible by next thread when it executes the critical section.

We introduce new equations in Parallel Data Flow Analysis(PDFA) to handle the distinctions between sequential and parallel data flow analysis as follows.

Super Node (S) Equations: A super node contains one or more composite nodes between two barrier nodes. The equation *In(s)* is the *Out(entry)* or *Out(Previous Barrier)*, and the Out(S) is a union of all composite nodes that directly reach the end of the super node. A flush set *Flush(S)* needs to be gathered in a super node, since a flush operation performs inter-thread data flow inside a super node.

$$Flush(S) = \bigcup_{b \in BasicNode(S)} Flush(b) \qquad (1)$$

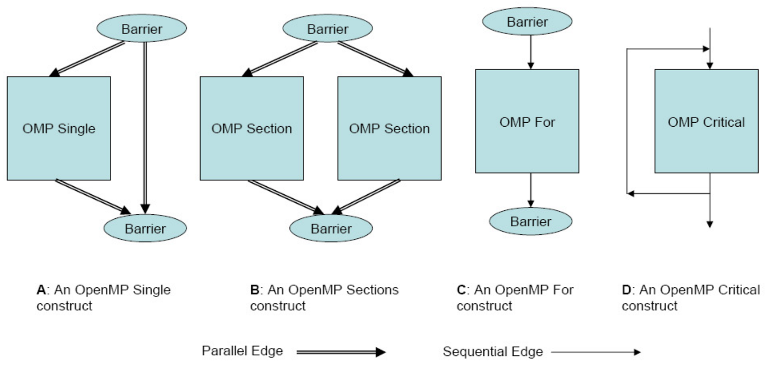

Fig. 2. OpenMP Worksharing Constructs in PCFG

We also need to compute the definitions that are killed in a super node. In a parallel program, if a definition has been killed in at least one of composite nodes inside a super node, it cannot reach the next super node. We need to perform a union operation for all must-be-killed definitions of all composite nodes in a super node. We define *MustKill(S)* of a super node to be a set of must-be-killed definitions in its all enclosed composite nodes.

$$MustKill(S) = \bigcup_{C \in CompositeNode(S)} MustKill(C) \tag{2}$$

Composite Node (C) Equations: A composite node contains one or more basic nodes. We are interested in the intra-thread data flow, and the must-be-killed definitions in a composite node. The equation *In(C)* is a union of Out(C) of all previous composite nodes or In(S) of its super node. *Out(C)* is a union of its basic nodes that directly reach the end of the composite node. Intuitively, if a definition appears in the beginning of of a super node, but does not reach the end of a composite node inside the super node, it is included in the *MustKill(C)* set.

$$MustKill(C) = In(S) - Out(C)\ (where\ S\ is\ the\ super\ node\ of\ C) \tag{3}$$

Basic Node (b) Equations: We compute data flow equations for each basic node similarly with them in a sequential DFA. In addition, we need to handle inter-thread data flow by flush operations. We have the following modifications of equations for each basic node b:

$$Out(b) = (In(b) - Kill(b)) \bigcup Gen(n) \bigcup Flush(S)\ (Where\ S\ is\ the\ super\ node\ enclosing\ b) \tag{4}$$

$$Flush(b) = \begin{cases} \bigcup v & (v \in variables\ specified\ in\ the\ flush\ directive) \\ all\ shared\ variables & (if\ the\ flush\ does\ not\ specify\ any\ variables) \end{cases} \tag{5}$$

Barrier Node (Barrier) Equations: At a barrier point, threads wait until all threads reaching the point. It then flushes all shared variables before all threads execute the next super node. The *In(Barrier)* equation is a union of all predecessors. And the Out(barrier) set should exclude all definitions that must be killed in the previous super node.

$$Out(Barrier) = (In(barrier) - MustKill(S)) \quad (where\ S\ is\ the\ previous\ super\ node\ of\ the\ barrier) \tag{6}$$

3.1 Compiler Optimizations

Based on the above Parallel Control Flow Graph and Parallel Data Flow equations, we are able to perform traditional optimizations before an OpenMP code is lowered to a threaded code. A lowered threaded code may lose the structure of threads execution and interactions, and make it difficult for a compiler to optimize it globally. We can compute the data flow information such as reaching definitions based on the PDFA. Inter-thread and intra-thread use-definition and definition-use chain can be calculated and traditional optimizations such as copy propagation, dead code elimination, and partial redundancy elimination(PRE) etc. can be performed. Moreover, Barrier elimination is an optimization for parallel programs to remove redundant barriers so as to improve the performance. We can perform optimizations specific to parallel programs such as Barrier eliminataion based our PCFG and PDFA equations. Due to the page limit in this paper, we will present the algorithm of these optimizations in the future.

4 Conclusion and Future Work

The contribution of the paper is to present a compiler framework that enables high-level data flow analysis and optimizations for OpenMP by taking its semantics into consideration. The framework represents the intra- and inter-thread data flow in OpenMP based on the relaxed memory model. It enables classical global optimizations to be performed before an OpenMP code is lowered to a threaded code. Moreover, a compiler is able to perform more aggressive optimizations specific to OpenMP programs. In the future work, We will implement it into the OpenUH compiler [6] to further evaluate the work. We will explore more compiler optimizations based on the framework in cluster OpenMP implementation [5]. It could also be used in static analysis for detecting race conditions of an OpenMP program. We believe that the framework will lead to more aggressive optimizations and analysis for OpenMP.

References

1. V. Balasundaram and K. Kennedy, "Compile-time detection of race conditions in a parallel program," in *ICS '89: Proceedings of the 3rd international conference on Supercomputing.* New York, NY, USA: ACM Press, 1989, pp. 175–185.
2. D. Buttlar, B. Nichols, and J. P. Farrell, *Pthreads Programming.* O'Reilly & Associates, Inc., 1996.
3. D. Callahan, K. Kennedy, and J. Subhlok, "Analysis of event synchronization in a parallel programming tool," in *PPOPP '90: Proceedings of the second ACM SIGPLAN symposium on Principles & practice of parallel programming.* New York, NY, USA: ACM Press, 1990, pp. 21–30.
4. J. P. Hoeflinger and B. R. de Supinski, "The openmp memory model," in *the 1st International Workshop on OpenMP (IWOMP 2005)*, 2005.
5. L. Huang, B. Chapman, and Z. Liu, "Towards a more efficient implementation of OpenMP for clusters via translation to Global Arrays," *Parallel Computing*, vol. 31, no. 10-12, 2005.
6. C. Liao, O. Hernandez, B. Chapman, W. Chen, and W. Zheng, "OpenUH: An optimizing, portable OpenMP compiler," *Concurrency and Computation: Practice and Experience, Special Issue on CPC'2006 selected papers*, 2006.
7. "OpenMP: Simple, portable, scalable SMP programming," http://www.openmp.org, 2006.
8. X. Tian, A. Bik, M. Girkar, P. Grey, H. Saito, and E. Su, "Intel OpenMP C++/Fortran compiler for hyper-threading technology: Implementation and performance," *Intel Technology Journal*, vol. 6, pp. 36–46, 2002.