Exploiting Global Optimizations for OpenMP Programs in the OpenUH Compiler

Lei Huang   Deepak Eachempati   Marcus W. Hervey   Barbara Chapman

Department of Computer Science, University of Houston, Houston, TX 77004, USA

{leihuang, dreachem, mwhervey, chapman}@cs.uh.edu

Abstract

The advent of new parallel architectures has increased the need for parallel optimizing compilers to assist developers in creating efficient code. OpenUH is a state-of-the-art optimizing compiler, but it only performs a limited set of optimizations for OpenMP programs due to its conservative assumptions of shared memory programming. These limitations may prevent some OpenMP applications from being fully optimized to the extent of its sequential counterpart. This paper describes our design and implementation of a parallel data flow framework, consisting of a Parallel Control Flow Graph (PCFG) and a Parallel SSA (PSSA) representation in OpenUH, to model data flow for OpenMP programs. This framework enables the OpenUH compiler to perform all classical scalar optimizations for OpenMP programs, in addition to conducting OpenMP specific optimizations.

Categories and Subject Descriptors D.1.3 [*Programming Techniques*]: Concurrent Programming; D.3.4 [*Programming Languages*]: Processors – compilers; optimization

General Terms Language, Performance, Theory

Keywords Compiler Analysis, OpenMP, Parallel SSA

1. Introduction

Multicore hardware has dominated the mainstream computer market from large scale systems to desktops and even laptops. OpenMP is a widely accepted shared memory programming model that is increasingly used for scientific and commercial software development. With OpenMP, a sequential application can be parallelized with the use of compiler directives and a few runtime library routines, typically with little modification to the program's original overall structure. OpenMP's incremental parallelization capability provides great flexibility, and its portability and memory model guarantee correct implementation on different platforms. Compiler technology and well-tuned runtime support are critical to ensure the scalability of OpenMP codes.

The typical translation strategy employed by an OpenMP-aware compiler replaces OpenMP constructs by calls to runtime library routines. It, however, is much easier for a compiler to analyze OpenMP code than the corresponding explicitly multithreaded code. Moreover, OpenMP directives impose a structured programming style that may offer additional, exploitable information to support program optimization. Yet despite a number of related research efforts, we are not aware of any compilers that have implemented the required parallel program analyses.

In order to compare how the compiler performs optimizations for sequential and OpenMP programs we wrote a simple OpenMP code, *pre-example*, containing a parallel loop with redundant computation. We used this code to test two classical compiler optimizations: copy propagation and partial redundancy elimination (PRE). Table 1 shows the execution time difference when we used different compiler flags. In the first case using "-O3" only, the compiler ignores the OpenMP directives, and compiles and optimizes it as a regular sequential program. With "-O3 -mp", the compiler enables OpenMP compilation, and performs the optimizations after the OpenMP directives are translated to threaded library calls. The OpenMP code version exhibited a remarkably poor baseline performance using one thread when compared to its corresponding sequential code. A performance difference can also be seen in the FT and UA applications from the NAS benchmark suite.

benchmark	-O3	-O3 -mp
pre-example	42.46s	87.52s
NAS benchmark FT: CLASS=A	18.45s	26.17s
NAS benchmark UA: CLASS=A	130.31s	220.15s

Table 1. Performance comparison: sequential vs. OpenMP program on one thread

2. Parallel Control Flow Graph

We have designed a Parallel Control Flow Graph (PCFG) for representing OpenMP programs in order to enable aggressive optimizations such as those described in the previous section, while guaranteeing correctness. The PCFG is similar to the Program Execution Graph [1] and the Synchronized Control Flow Graph [2], proposed by other researchers. The distinction between our PCFG and their flowgraphs is that our PCFG is based upon the relaxed memory consistency model of OpenMP, and its barrier and flush synchronizations, instead of event-based synchronizations (such as signal-wait).

OpenMP's relaxed consistency memory model allows each thread to have its own local view of shared data at times during execution: when a synchronization point is reached in the code, consistency is enforced by flushing the values of modified shared data to memory. Most OpenMP worksharing constructs include implicit barriers that ensure the synchronization of thread execution and therefore also serve to keep the data consistent between them. However, the local value of a shared object may or may not be consistent before such a barrier, and OpenMP provides a

Copyright is held by the author/owner(s).

PPoPP'09, February 14–18, 2009, Raleigh, North Carolina, USA.
ACM 978-1-60558-397-6/09/02.

unique feature with the explicit *omp flush* construct to force the value of one or more shared objects to be written back to memory at an arbitrary point during execution. This allows an aggressive optimizing compiler to safely assume that there are no inter-thread data interactions until a flush operation has been reached.

We create the PCFG similar to conventional CFG creation. Fig. 1 shows the PCFG for some OpenMP worksharing constructs. The basic nodes, construct entry and exit nodes, barrier nodes and flush nodes are created by traversing an OpenMP program. The sequential edges and parallel edges are applied based on the OpenMP semantics to connected these nodes. The edges enclosed inside various OpenMP constructs are marked with "must-take" attribute if at least a thread will take the path.

Afterward, we conduct the concurrency analysis to determine the concurrency code phases in the program. Flush operations in a same phase may be executed concurrently and thus shared variables may be synchronized between different threads. We create a conflict edge to connect these flush operations to model the inter-thread data propagation.

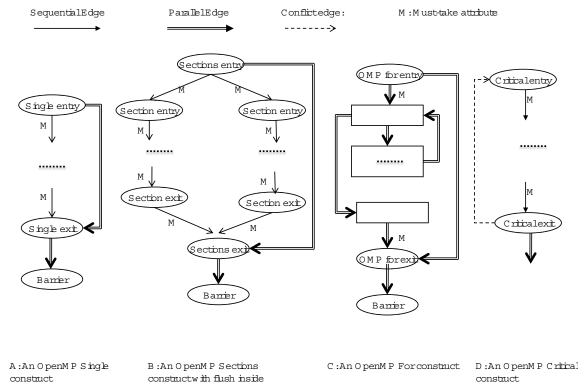

Figure 1. OpenMP Worksharing Constructs in PCFG

3. Parallel SSA

Parallel SSA (PSSA) form is an extension of SSA that represents more complex data flow that occurs in an OpenMP parallel program. We incorporate ψ- and π-functions into our representation, based in part on work by Lee *et al.* [5]. We use a ψ-function with a single operand to represent a special assignment to some shared variable *v* at a *barrier* node in the PCFG. The π-function is used to represent a special assignment to some shared variable *v*, where the operands refer to the reaching definition of *v* from the current thread and all potentially reaching definitions of *v* from other executing threads based on PCFG *conflict edges*. The primary goal of PSSA is to more accurately encode data flow information with respect to shared variables so that the compiler has more scope for optimizations.

4. Experiments

We have tested the performance of the PRE example on the SMP with 8 AMD Opteron(2.0GHz) processors and 32GB RAM Linux box using the OpenUH compiler with our PDFA implementation. The code achieved linear speedup with 8 threads after these optimizations, while the code only reached 3.8 times speedup with 8 threads without these optimizations compared with the sequential version execution. We are currently working on obtaining performance metrics with other benchmarks such as NAS and SPEC OpenMP versions.

5. Related Work

There have been a number of efforts that develop parallel dataflow analysis. Shasha and Snir showed that a parallel program may violate sequential consistency if $E \bigcup P$ contains a cycle, where E is the execution order and P is the order of variable accesses. Based on Shash and Snir's work, Krishnamurthy and Yelick [4] proposed a compiler framework to analyze parallel programs with explicit barriers, post-wait and lock synchronization. Knoop *et. al* [3] has developed a theory of Parallel Data Flow Analysis and proved that it is possible to perform it for parallel programs [6]. Other work [5] use the SSA form to solve data flow problems for parallel programs, however this research does not target OpenMP programs.

6. Conclusions and Future Work

This paper describes the design and implementation of a compiler framework that enables high-level data flow analysis and optimizations for OpenMP. The framework creates and/or increases the opportunities for performing a range of traditional global optimizations on OpenMP code before it is lowered to explicitly threaded code. By taking OpenMP semantics into consideration, it also enables more aggressive optimizations that are specific to this programming interface. In the future, we plan to complete the implementation of this framework in the OpenUH compiler, evaluate our work using more sophisticated benchmarks, investigate how it may support the implementation of OpenMP on clusters, and use it to statically detect data races in OpenMP program. We believe that this framework will improve overall OpenMP program performance and support our long-term goal of providing a highly productive, effective parallel programming model for a wide range of shared memory platforms.

Acknowledgments

This work was supported by DOE Pmodels project under Grant Number DE-FC02-06ER25759.

References

[1] Vasanth Balasundaram and Ken Kennedy. Compile-time detection of race conditions in a parallel program. In ICS '89: Proceedings of the 3rd international conference on Supercomputing, pages 175–185, Crete, Greece, June 1989. ACM Press.

[2] 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, pages 21–30, Seattle, Washington, United States, March 1990. ACM Press.

[3] Jens Knoop, Bernhard Steffen, and Jurgen Vollmer. Parallelism for free: efficient and optimal bitvector analyses for parallel programs. ACM Trans. Program. Lang. Syst., 18(3):268–299, 1996.

[4] Arvind Krishnamurthy and Katherine A. Yelick. Optimizing parallel programs with explicit synchronization. In SIGPLAN Conference on Programming Language Design and Implementation, pages 196–204, La Jolla, California, United States, June 1995.

[5] J. Lee, D. A. Padua, and S. P. Midkiff. Basic compiler algorithms for parallel programs. In Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'99), pages 1–12, Atlanta, Georgia, United States, August 1999. ACM SIGPLAN.

[6] Jurgen Vollmer. Data flow analysis of parallel programs. In PACT '95: Proceedings of the IFIP WG10.3 working conference on Parallel architectures and compilation techniques, pages 168–177, Manchester, United Kingdom, 1995. IFIP Working Group on Algol.