Tracing Logical Concurrency for Dynamic Race Detection in OpenMP Programs[*]

In-Bon Kuh, Ok-Kyoon Ha, and Yong-Kee Jun[**]

Dept. of Informatics,
Gyeongsang National University,
The Republic of Korea
{inbon,jassmin,jun}@gnu.ac.kr

Abstract. OpenMP is an industry standard supporting a serialized program to be executed in parallel with simple compiler directives and libraries. OpenMP covers only user-directed parallelization and does not enforce to fix concurrency bugs, such as data races. However it is difficult to locate data races occurred in an execution of the program, because they may lead the program execution to be non-deterministic. Previous work considers only physically concurrent threads to detect data race in OpenMP programs dynamically, and it misses apparent data races among logically concurrent threads. This work presents a fine-grained tool which traces implicit threading points with empirical investigation. We implemented the tool on top of Pin instrumentation framework and experimented the correctness of our tracer with a data race detector.

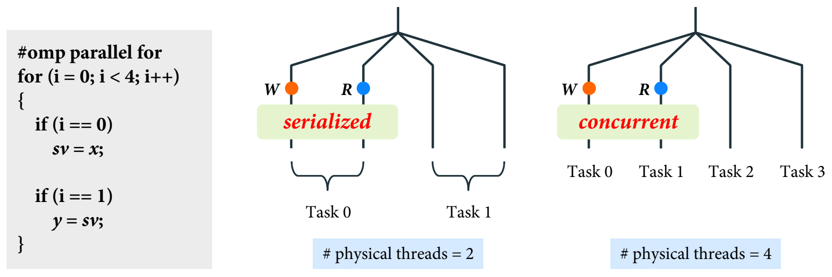
Keywords: OpenMP, dynamic race detection, logical concurrency, instrumentation framework

1 Introduction

OpenMP [5] is an industry standard supporting a serialized program to be executed in parallel with simple compiler of directives and libraries supporting standard C/C++ and Fortran 77/90. Because OpenMP covers only user-directed parallelization and does not enforce to fix concurrency bugs, such as data races, application developers should be responsible for correctly using OpenMP. However it is difficult to locate data races occurred in an execution of the program, because they may lead the program execution to be non-deterministic. Data race detection techniques are categorized static analysis which leads to many false positives and dynamic analysis which reports data races occurred in an

[*] This research was supported by the MKE(The Ministry of Knowledge Economy), Korea, under the ITRC(Information Technology Research Center) support program (NIPA-2012-H0301-12-3003) supervised by the NIPA(National IT Industry Promotion Agency)

[**] Corresponding author: In Gyeongsang National University, he is now the director of GNU Embedded Software Center for Avionics (GESCA) which is a national IT Research Center (ITRC) of Republic of Korea



```
#omp parallel for
for (i = 0; i < 4; i++)
{
  if (i == 0)
    sv = x;

  if (i == 1)
    y = sv;
}
```

Fig. 1. An example of concurrency variation by OpenMP task scheduling

execution of the program [3]. Dynamic analysis classifies again into trace based post-mortem technique and on-the-fly technique. Usually, a dynamic detector employs an instrumentation frameworks to insert monitoring codes for memory accesses and thread operations.

Unfortunately, previous work [3, 6] monitors only physically concurrent threads to detect data race in OpenMP programs, and it misses apparent data races among logically concurrent threads [2]. Fig.1 shows an example of concurrency variation by OpenMP scheduling. One expect the shared variable *sv* to be accessed individually on concurrent threads according to the `parallel for` directive in the source code of Fig.1. However, OpenMP programs determine how many tasks are assigned by consulting the pre-defined *internal control variables* and the number of the physical processors, when execution encounters a *parallel region*. This work presents a fine-grained tool which traces implicit threading points on top of Pin instrumentation framework.

2 Tracing Logical Concurrency

A *task* represents a basic component generated by the implicit parallel region or generated when a `parallel` construct is encountered [5]. Prior detectors miss apparent data races occurred during an execution in OpenMP programs, because they regard the logically concurrent threads of a parallel region as a set of sequential threads by a task. To divide a set of sequential threads on a *task region* into logically concurrent threads, we locate implicit threading points by empirical investigation with following programs:

– **In-the-lab synthetic programs** were developed for testing all kinds of scheduling policies and multilevel nested parallelism.
– **OmpSCR** and **PARSEC benchmark suites** were employed to analyze implicit threading points in divers applications of the real world.

Afters the investigation, we found a common point which can divide a set of sequential threads into logically concurrent threads in machine instruction level. The point is located around the next instruction of the initialization code block in a routine for a task region.

We implemented a Pin-tool with this technique on top of Pin instrumentation framework [4] which is widely used to analyze binary executable. The Pin-tool traces implicit threading points when an OpenMP program is loaded into system memory before the main thread is started. We experimented the correctness of the tool with a data race detector which identifies the concurrency of logical threads. Our tool calls detection routines at the points by instrumented monitoring codes. We replaced the instrumentation part of the detector with our Pin-tool and evaluated the revised detector with in-the-lab synthetic programs. The tool correctly identifies a set of sequential threads on a task as logically concurrent threads for detecting data races. We empirically confirmed that the revised detector still does not miss apparent data races by OpenMP task scheduling.

3 Conclusion

To detect data races in OpenMP programs dynamically, previous work considers only physically concurrent threads, missing apparent races among logically concurrent threads. We presented a fine-grained tool which traces implicit threading points with empirical investigation. We implemented the tool on top of Pin instrumentation framework and experimented the correctness of our tool with a data race detector. Our tool correctly identifies a set of sequential threads as logically concurrent threads for detecting data races. We believe that this work makes detectors not to miss apparent data races disappeared by OpenMP task scheduling.

References

1. Ha, Ok-Kyoon, Young-Joo Kim, Mun-Hye Kang and Yong-Kee Jun, "Empirical Comparison of Race Detection Tools for OpenMP Programs," Grid and Distributed Computing, pp. 108-116, Springer, 2009
2. Jannesari, Ali and Kaibin Bao, and Victor Pankratius and Walter F. Tichy, "Helgrind+: An efficient dynamic race detector," Proceedings of the 2009 IEEE International Symposium on Parallel and Distributed Processing, pp. 1-13, IEEE, 2009
3. Luk, Chi-Keung, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, StevenWallace, Vijay Janapa Reddi and Kim Hazelwood, "Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation," Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation, pp. 190-200, ACM, June 2005
4. OpenMP Application Program Interface, `http://www.openmp.org/mp-documents/OpenMP3.1.pdf`, 2011
5. Petersen, Paul. and Sanjiv Shah, "OpenMP Support in the Intel Thread Checker," Proceedings of the OpenMP applications and tools 2003 international conference on OpenMP shared memory parallel programming, pp. 1-12, Springer, 2003