Detecting Accesses of First Data Races in Parallel Programs with Random Synchronization

Hee-Dong Park[1], and Yong-Kee Jun[2]

[1] Joongbu University, Kumsan, Korea
hdpark@joongbu.ac.kr
[2] Gyeongsang National University, Jinju, Korea
jun@gnu.ac.kr

Abstract. Detecting data races is important in debugging shared memory parallel programs, because the races could result in unintended non-deterministic executions of the programs. Unfortunately, previous race detection techniques cannot guarantee to detect at least one access involved in the first races to occur in parallel programs with random synchronization. This paper presents a program monitoring algorithm which collects all of the current key accesses of their local thread blocks that are involved in races with the latest key accesses in the other concurrent thread blocks.

Keywords: race detection, parallel program debugging, first race, random synchronization, program monitoring

1 Introduction

One of the inherently fundamental problems encountered when debugging a parallel program is resolving the data race conditions in the program. A *data race* occurs in a parallel or multi-threaded program when two threads access the same memory location without proper synchronization constraints between the accesses, such that at least one of the accesses is a write [1]. Incorrect synchronization leads to incorrect ordering between accesses to shared memory. Data race could result in either unpredictable results or paths of events in different executions on the same input.

Previous race detection techniques [4, 6, 5, 7] cannot locate the candidate accesses which contain at least one access involved in the first races for parallel programs with random synchronization. Such parallel programs could exhibit different sequence of events when executed repeatedly, that is, can not guarantee the access sequence in one execution to that of in another execution due to small timing variations in execution of synchronization event, and could result in non-deterministic event ordering for debugging. It has been proved that detecting races in program executions that have synchronization powerful enough to support mutual exclusion is NP-hard [2]. Thus detecting actual races is of practical use in a particular execution of such parallel programs.

The races that occur first are races between two accesses that are not causally preceded by any other accesses also involved in races. The first races are important in debugging because the removal of such races may make other races disappear. It is even possible that all races reported by other algorithms would disappear once the first races are removed.

The main result of our on-the-fly algorithm is to collect filtered candidate accesses in which at least one access be included in first races for a parallel program execution.

2 Candidate Access Filtering for Race Detection

The concurrency relation among threads in an execution of shared-memory parallel program can be represented by a directed acyclic graph called POEG (Partial Order Execution Graph) [3] which captures the happens-before relation [1]. An access a_i happened before another access a_j, denoted as $a_i \rightarrow a_j$, and it is concurrent with each other if there exist no paths between them. There can be many races in an execution of parallel program, and first race to occur or simply a first race is either an unaffected race or a tangled race.

In a parallel program, a synchronization block or a block is an access sequence between inter-thread coordination events(such as $POST, WAIT$, *fork* and *join*). We define a write (*read*) access a_i as a *key access* if there does not exist any other write (*read* or *write*) access a_j within a block such that $a_j \rightarrow a_i$. And Block Access-history for a shared variable X and thread identifier T, denoted by $BA(X,T)$ is a set of key accesses in a block of thread T. A $BA(X,T)$ has maximum two accesses(*read* and/or *write*) and is cleared at each synchronization event. A read (*write*) access a_i in the corresponding access history(AH) is a read (*write*) candidate, if a_i is involved in a race and there exists no other access a_h such that $a_h \rightarrow a_i$ and a_h is involved in a race.

A BA(X,T) is maintained as local variable of corresponding thread T, which can be free from mutual exclusion with shared memory of other threads or processors. The key accesses are not always involved in the race, and maintaining all the key accesses to detect first races is also inefficient, therefore we filter out the key accesses to make candidate accesses and only the candidate accesses would be involved in the first race.

The Candidate Set for a shared variable X, denoted by $CS(X)$, is a set of candidates which are involved in the race for a shared variable X : $CS(X,R)$ for a set of read candidates, $CS(X,W)$ for a set of write candidates. To get a subset of $CS(X)$, every key access from $BA(X,T)$ is checked through the logical concurrency with the access in $AH(X)$ in an execution of program. The algorithm is as follows:

1. *Collect key accesses :* Check if the *current* access is key or not which from $BA(X,T)$ in thread T. If not key, then return.
2. *Update AH(X) :* For all accesses in $AH(X)$, if there is an access a_i which is $a_i \rightarrow$ *current* and the race bit of a_i is true, return, otherwise delete a_i from $AH(X)$; and add the *current* to the corresponding set of $AH(X)$.

3. *Determine CS(X) :* For all accesses in $AH(X)$, if there is an access a_i which is involved in a race with the *current*, set the race bits of both accesses, otherwise return. Any *current* access concurrent with the accesses in $AH(X)$ is added to the corresponding $CS(X)$.
4. *Halt :* Halt the *current* thread, if the *current* is a write access.

In step 1 and 2, we monitor all memory operations executed during a particular execution and filter out the accesses to get key accesses in each block access history, and update access history which contain mutually concurrent accesses. In step 3, we inspect access history for getting a subset of the candidates in which at least one access is included in the first races. The accesses which are not key are discarded in processing race condition determination, which makes more efficient in time and space complexity.

3 Conclusion

In this paper, we present an algorithm to collect filtered accesses on a particular execution of parallel program with random synchronization, by extracting key accesses and collecting candidate sets in which at least one access is involved in the first races.

It could be required huge amount of space to detect the first races in one monitored execution of parallel programs with synchronization, so our technique to collect filtered accesses for race detection is more efficient and practical in debugging a large class of shared-memory parallel programs.

References

1. Lamport, L.: Time, Clocks, and the Ordering of Events in a Distributed System, Communications of the ACM, pp. 558-565, July 1978.
2. Netzer, R.H.B. and B.P. Miller: On the Complexity of Event Ordering for Shared-Memory Parallel Program Executions, Intl. Conf. on Parallel Processing, pp. II-93-II-97 (1990).
3. Dinning, A., and Schonberg E., An Empirical Comparison of Monitoring Algorithms for Access Anomaly Detection, 2nd Symp. on Principles and Practice of Parallel Programming, ACM, pp. 1-10, March 1990.
4. Jun, Y., and C. E. McDowell, On-the-fly Detection of the First Races in Programs with Nested Parallelism, 2nd Int'l Conf. on Parallel and Distributed Processing Techniques and Applications, CSREA, pp. 1549-1560, August 1996.
5. Park, H., and Y. Jun, Detecting the Firtst Races in Parallel Programs with Ordered Synchronization, 6th Int'l Conference on Parallel and Distributed Systems (ICPADS), pp. 201-208, IEEE, Tainan, Taiwan, December 1998.
6. Kim, J. and Jun, Y., Scalable On-the-fly Detection of the First Races in Parallel Programs, Proc. of the 12nd Int'l Conf. on Supercomputing, ACM, pp. 345-352, July 1998
7. Ha, K., Y. Jun, and K. Yoo, Efficient On-the-fly Detection of First Races in Nested Parallel Programs, Workshop on State-of-the-Art in Scientific Computing (PARA), pp. 75-84, Copenhagen, Denmark, June 2004.