Detecting the First Races in Parallel Programs with Ordered Synchronization*

Hee-Dong Park, and Yong-Kee Jun†

Dept. of Computer Science, GyeongSang National University
Chinju, 660-701 South Korea
{hdpark,jun}@nongae.gsnu.ac.kr

Abstract

Detecting races is important for debugging shared-memory parallel programs, because the races result in unintended nondeterministic executions of the programs. Previous on-the-fly techniques to detect races in programs with inter-thread coordination such as ordered synchronization cannot guarantee that the race detected first is not preceded by events that also participate in a race. This paper presents a new two-pass on-the-fly algorithm *to detect the first races in such parallel programs. Detecting the first races is important in debugging, because the removal of such races may make other races disappear including those detected first by the previous techniques. Therefore, this technique makes on-the-fly race detection more effective and practical in debugging parallel programs.*

Keywords: parallel programming, ordered synchronization, debugging, on-the-fly analysis, first race detection, two-pass protocol

1. Introduction

Many errors in shared-memory parallel programs are the result of two or more instructions accessing the same shared variable from different parallel threads when at least one of the accesses is a write and the accesses are not properly synchronized. We refer to such errors as *races*. These races can result in nondeterministic program execution, making debugging parallel programs difficult. Traditional cyclic debugging with breakpoints is often not effective in the presence of races, because the breakpoints can change the execution timing causing the erroneous behavior to disappear.

On-the-fly detection [3, 4, 9, 11] instruments the program to be debugged, and monitors an execution of the program. The monitoring process reports races which occur during the monitored execution. This approach can be a complement to *static analysis* [2, 6], because on-the-fly detection, as described in this paper, can be used to identify feasible (real) races from the potential races reported by static analysis approaches. On-the-fly detection also requires less storage space than *post-mortem detection* [5, 7, 14] because much of the information collected by the monitoring process can be discarded as an execution progresses. Although on-the-fly detection may not report as many races as current post-mortem detection algorithms, at least one race is guaranteed to be reported for each variable involved in a race.

The algorithm reported in this paper is an on-the-fly algorithm that detects the races that "occur first." Intuitively, 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 on-the-fly algorithms would disappear once the first races are removed. Reporting the first races is preferable to previous techniques which might require several iterations of monitoring, because the cost of monitoring a particular execution is still expensive.

This paper introduces an on-the-fly technique to detect the first races in a particular execution of shared-memory programs with ordered synchronization. Section 2 shows some definitions related to our technique, and states problems of the previous on-the-fly tech-

*This work was supported in part by Institute of Information Technology Assessment, and Korea Science and Engineering Foundation (Grant No. 971-0901-012-2).

†In Gyeongsang National University, he is also involved in both Institute of Computer Research and Development, and Information and Communication Research Center, as a research professor.

niques to detect the first races. Our new two-pass algorithm for detecting the first races is presented in Section 3. And, the related works are given in Section 4, before concluding the paper.

2. Background

In this section, we first describe the first races in a program execution which is represented by a directed acyclic graph called *POEG* (*Partial Order Execution Graph*) [4]. And then we describe existing on-the-fly techniques and discuss the problems of the techniques for detecting the first races.

2.1. The First Race

Parallel programs may have parallel loop constructs such as `PARALLEL DO` and `END DO`. In an execution of such programs with parallel loops, multiple threads of control are created at a `PARALLEL DO` and terminated at the corresponding `END DO` statement. These fork and join operations are called *thread operations*. If two threads are coordinated by synchronization primitives such as the `POST` and `WAIT`, the thread that issues `WAIT` may not execute beyond the coordination point until the thread that issues `POST` reached the corresponding point. On the other hand, the posting thread may proceed immediately. These post and wait operations are called *coordination operation*. A *block* is a sequence of instructions, which is executed by a single thread and does not include thread or coordination operations.

In this work, we consider parallel programs with nested parallel loops and ordered synchronization in which any pair of the corresponding coordination points are on ordered sequence such as *ordered critical section* [15] and *sequence synchronization* for software pipelining [16]. The synchronization constructs in OpenMP Fortran API [15] includes `ORDERED` and `END ORDERED` directives for the ordered critical sections which can appear in the dynamic extent of a `PARALLEL DO` directive. These directives is executed in the order in which iterations would be executed in a sequential execution of the loop, while allowing program outside the section to run in parallel. PCF Fortran [16] provides the primitives for sequence sysnchronization, which is useful for communicating between iterations of a loop, or for communicating between distinct loops. Any sequence of events that can be numbered with an arithmetic sequence can be synchronized using the three primitives: `SET` to define an arithmetic sequence, `POST` to indicate that computation for a particular element of the sequence is complete, and `WAIT` to ensure

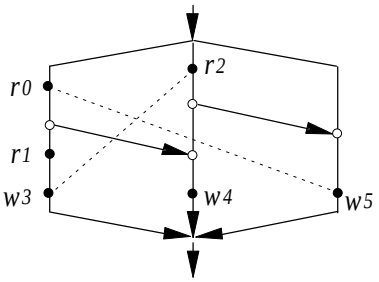

Figure 1. The first races in an execution

that the computation for a particular element of the sequence to complete. Sequence synchronization is more general than the `ORDERED` directives, because anything that can be done with the directives can also be done with sequence synchronization, but the reverse is not true.

The concurrency relationship among blocks is represented by the POEG. A vertex of POEG represents a thread or a coordination operation, and an arc originating from a vertex represents a block or a coordination starting from the corresponding operation. Figure 1 shows a POEG that is an execution instance of a parallel program with ordered synchronization, where a small filled circle on a block represents an access event executed by the block to a shared variable X. In this figure, six accesses are executed by three threads, and two coordination activities occurred between four coordination vertices which is represented with small empty circles.

Because the graph captures the *happened-before* relationship [12], it represents a partial order over a set of the events executed by blocks that make up an execution instance of the program. Concurrency determination is not dependent on the number or the relative execution speeds of processors executing the program.

Definition 2.1 *An event* e_i happened before *another event* e_j, *denoted by* $e_i \rightarrow e_j$, *if there exists a path from* e_i *to* e_j *in a POEG. Any two events are* ordered *if one happened before the other. Two events are* concurrent *with each other, if there exist no paths between them.*

The maximum number of mutually concurrent threads defines the *maximum concurrency* of a program. For example, consider the access events in Figure 1, where ri and wj denote read and write accesses to a shared variable, respectively. The i and j in the preceding are used to indicate the order in which the events are observed in a particular execution. A read access $r0$ happened before a write access $w3$ because there exists a path from $r0$ to $w3$, and we say these two accesses

are ordered. And, $r0$ is concurrent with $w5$, because there exist no paths between them. The maximum concurrency of the graph is three.

Two accesses to a shared variable are *conflicting* accesses if at least one of them is a write. If two access events, e_i and e_j, are conflicting accesses and concurrent, then the two events constitute a *race* denoted e_i-e_j.

Definition 2.2 *An access e_j is* affected *by another access e_i, if $e_i \rightarrow e_j$ and e_i is involved in a race.*

Definition 2.3 *A race e_i-e_j is* unaffected, *if neither e_i nor e_j is affected by any other accesses. The race is* partially-affected, *if only one of e_i or e_j is affected by another access.*

Definition 2.4 *A* tangle *T is a set of partially-affected races such that if e_i-e_j is a race in T then exactly one of e_i or e_j is affected by e_k such that e_k-e_l is also in T.*

Definition 2.5 *A* first to occur race *or simply a* first race *is either an unaffected race or a tangled race.*

The term tangled race was introduced by Netzer and Miller [14] and describes the situation when no single race from a set of tangled races is unaffected by the others. Note that there can never be exactly one tangled race in an execution. Consider the accesses to shared variable X in the POEG shown in Figure 1. There exist seven races in the execution: $r0$-$w5$, $r1$-$w4$, $r1$-$w5$, $r2$-$w3$, $w3$-$w4$, $w3$-$w5$ and $w4$-$w5$. Among them, only two races, $r0$-$w5$ and $r2$-$w3$, are the first races which are tangled races. Eliminating these two races may make the other five affected races disappear.

2.2. On-the-fly Race Detection

In this subsection, we introduce the previous techniques for on-the-fly race detection and then discuss their limitations to detect the first races for the execution model considered here.

The existence of a race involving a shared variable is solely a function of which events access the variable and the concurrency relationship between the events. Therefore, we can consider races for each shared variable independently. Current on-the-fly race detection algorithms use a *race detection protocol* [4, 9, 11, 13] which maintains an *access history* for each shared variable and reports the detected races, and a *labeling scheme* [1, 4, 8, 13] to generate the concurrency information called *label* of each thread. The labels for two threads can be compared to determine if the events in the threads are ordered or concurrent.

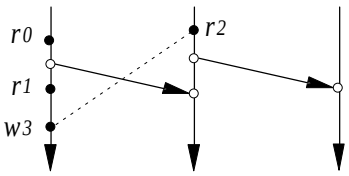

Figure 2. Accesses with a race

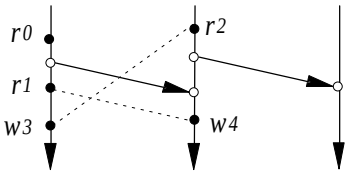

Figure 3. A tangled race

The race detection protocols can be classified into two groups: *first race detection* [9, 11], and *race verification* [4, 13]. With the protocols of race verification, at least one race is guaranteed to be reported for each shared variable if it is involved in a race, but a race detected first can not be guaranteed to be a race that occurred first. The first race detection provides the protocols capable of detecting the first races, although their programming models are more or less restrictive: only non-nested parallelism [11], and nested parallelism with no other inter-thread coordination [9]. In this work, we present a new two-pass protocol which extends the previous protocols again to first race verification in the first pass, and detects the first races in the second pass for the parallel program which has inter-thread coordination of ordered synchronization.

The protocols for first race detection [9, 11] are based on identifying *key* accesses that could participate in a first race. The protocol keeps in the access history three types of key accesses: *read key*, *write key*, and *read-write key*. $AH(X, R)$, $AH(X, W)$, and $AH(X, RW)$ denote each set of these keys, respectively, in the access history for a shared variable X. Races are detected by checking three kinds of pairs among the concurrent and conflicting key accesses: write and write, read and write, and read and read-write. These protocols, however, do not work for the programs with inter-thread coordination.

Consider a series of POEGs shown in Figures 2 and 3. In this case, the above protocols will fail to find the race $r1$-$w4$. First, look at the case of Figure 2. After $r0$ is executed, $AH(X, R)$ will contain $r0$ since it is a key. The following $r1$ is not a key, because there exists $r0$ that happened before $r1$, so $r1$ will be discarded. Also $r2$ is a key and will be added to $AH(X, R)$, and $w3$ be added to $AH(X, RW)$ because $r0$ is a key and happened before $w3$, and it can find the race $r2$-$w3$.

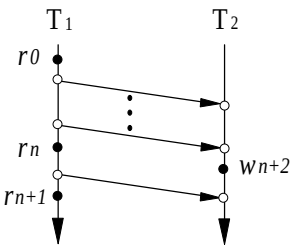

Figure 4. Synchronized events

Second, look at the next case in Figure 3. When $w4$ is executed, the protocol adds it to $AH(X, RW)$, because $r2$ is a key and happened before $w4$. But, the protocol fails to find $r1$-$w4$, because $r0 \rightarrow r1$ and $r1$ has already been discarded. This kind of limitation is addressed in the protocol presented in this paper.

3. The First Race Detection

To present our two-pass monitoring protocol that is an on-the-fly technique to detect the first races in parallel programs with ordered synchronization, in this section, we define first the set of *candidate accesses* which can be involved in the first races. Then, we introduce a new protocol which collects a subset of candidates in the first pass, and then completes the set of candidates in the second pass.

3.1. The Candidate Accesses

To detect the first races in one monitored execution of parallel programs with ordered synchronization, we must store all the accesses that occurred before the current access monitored, in the worst-case, which may incur impractical space complexity. For example, consider Figure 4 which shows two threads and many inter-thread coordinations between them. In this case, the first race involves r_n and w_{n+2}. To wait until w_{n+2} appears, we have to keep into the appropriate access history all the accesses considering the synchronization events of thread T_1, because we can not expect when the access w_{n+2} occurs. This method therefore can incur potentially unbounded space, and then impractical. To cope with this problem, we identify w_{n+2} in the first pass, and thereafter compares every read with w_{n+2} during the reexecution of the second pass to determine another candidate r_n. This is the main idea of our two pass on-the-fly technique.

Definition 3.1 *A read (write) access a_i is a* read (write) candidate, *if and only if (1) a_i is involved in a race, and (2) there exists no other access a_h such that $a_h \rightarrow a_i$ and a_h is involved in a race.*

Definition 3.2 *A write access w_i is a* read-write candidate *or* r-write candidate, *if and only if (1) w_i is involved in a race, (2) there exists a read candidate r_h which happened before w_i, and (3) there exists no other write access w_x such that $w_x \rightarrow w_i$ and $r_h \rightarrow w_x$.*

Definition 3.3 *The* candidate set *for a shared variable X, denoted by $CS(X)$, is a set of candidates which occurred in an execution for a shared variable X: $CS(X, R)$ for a set of read candidates, $CS(X, W)$ for a set of write candidates, and $CS(X, RW)$ for a set of r-write candidates.*

For example, consider the accesses to a shared variable X shown in Figure 3. The read access $r0$ is not a candidate, because $r0$ is not involved in a race. The read $r1$ and $r2$ is a read candidate, however, because $r1$ is involved in a race with $w4$, and $r2$ with $w3$. The write accesses $w3$ and $w4$ is an r-write candidate because they are involved in the race $r2$-$w3$ and $r1$-$w4$, respectively. In summary, Figure 3 shows five accesses, but candidates are four: $r1$, $r2$, $w3$, and $w4$.

All these candidates, however, is not always involved in the first races. For example, assume that Figure 3 does not have $r2$. Then, the access $w4$ is not an r-write candidate but a write candidate. In this case, the race $w3$-$w4$ that involves an r-write candidate $w3$ is not a first race, because $w3$ is concurrent with a write candidate $w4$ which is involved in $r1$-$w4$. Therefore, $w3$ is an r-write candidate which is not *effective*.

Definition 3.4 *A read (write) candidate is* effective *by itself. An r-write candidate is* effective, *if there does not exist a write candidate.*

An r-write candidate which is effective is not always involved in the first races. For example, consider the access $w4$ shown in Figure 1. In this case, $w4$ is an r-write candidate which is effective, because there does not exist any write candidate. This effective r-write candidate $w4$, however, is involved in races with $r1$, $w3$, and $w5$, which are not first races. This is because all the read candidates happened before $w4$, and then there does not exist a read candidate which is concurrent with $w4$; all the races that involve $w4$ involves the accesses which is affected by the read candidates. Therefore, $w4$ is a *pseudo* r-write candidate although it is effective.

Definition 3.5 *A read (write) candidate is not* peudo *by itself. An r-write candidate rw is* pseudo, *if there does not exist a read candidate which is concurrent with rw.*

Definition 3.6 *Given two concurrent candidates in $CS(X)$, a_i and a_j, which are effective and not pseudo,*

	pass-I	pass-II
$AH(X,R)$	r_1^*, r_2^*	
$AH(X,W)$	w_3^*, w_4^*, w_5^*	
$CS(X,R)$		r_0, r_2
$CS(X,W)$	w_3, w_4, w_5	
$CS(X,RW)$		w_3, w_4, w_5

Table 1. AH and CS for Figure 1

a_i-a_j is a candidate race, *if and only if a_i and a_j are in one of the following two cases: (1) both a_i and a_j are in $CS(X,W)$; or (2) one of a_i and a_j is in $CS(X,R)$, and the other is in $CS(X,RW) \cup CS(X,W)$.*

3.2. Two-Pass Protocol

To detect first races in two pass, our protocol collects a subset of the candidates in the first pass, and then completes the set of candidates in the second pass. By the definitions of candidates, they are reproduced in the second pass of monitored execution, but the order in which two candidates are observed may be different if they are concurrent. Our algorithm works in this case. And, each current access halts the current thread if it is a write or an r-write candidate in each pass of the protocol. It is of no use to proceed more than the current point in this case, because a write or an r-write candidate does not happen before any other candidate.

The protocol checks in the first pass if every current access is a candidate to get a subset of $CS(X)$. The protocol is similar with the race verification protocol using access histories, but it uses a special bit, called *affecting bit*, for each access stored in access history. The affecting bit is set if its access a_i get involved in a race. Affecting bits are shown in Table 1, which are coded with asterisks attached to the accesses. And, in this pass, the protocol does not discriminate write candidates from the r-write candidates which are filtered out in the second pass.

Algorithm 1 *The first pass uses $AH(X)$ which has two sets of read and write accesses, and produces $CS(X)$ which has two sets of candidates: one set for read candidates, and the other for both write and r-write candidates.*

1. Update $AH(X)$: *(1) For all accesses in $AH(X)$, if there is an access a_h which is happened before the current and the affecting bit of a_h is true, return, otherwise delete a_h from $AH(X)$; and (2) Add the current to the corresponding set of $AH(X)$.*
2. Determine candidate: *(1) For all accesses in $AH(X)$, if there is an access a_i which is involved in a race with the current, set to true the affecting bits of both a_i and the current. (2) Return, if the affecting bit of the current is false.*
3. Update $CS(X)$: *Add the current to the $CS(X,R)$ if it is a read access, or to the $CS(X,W)$ if it is a write access.*
4. Halt: *Halt the current thread, if the current is a write access.*

In the above first-pass algorithm, the step 1 updates the corresponding access history $AH(X,R)$ or $AH(X,W)$ with the current access. This step makes $AH(X,R)$ and $AH(X,W)$ contain the accesses which are mutually concurrent. Any access which arrives in the step 3 is a candidate, and then added to the corrensponding $CS(X)$. For example, consider the case shown in the Table 1 for the Figure 1 again. The first access $r0$ is added to $AH(X,R)$, and deleted when the next read access $r1$ is observed in the step 1 of $r1$, The write access $w3$ is added to $AH(X,W)$ in its step 1. $w3$ is also added to $CS(X,W)$ in its step 2 and 3, because $w3$ is concurrent with $r2$ which has been kept in $AH(X,R)$, and the affecting bits of both $w3$ and $r2$ is set to true. This halts the thread which executes $w3$ in the step 4. The subsequent accesses $w4$ and $w5$ are also added to $CS(X,W)$, and halt the corresponding threads.

Theorem 1 *An access is a candidate which is effective and not pseudo in an execution of program, if and only if the access is involved in a first race.*

Proofs of theorems appear in the appendix.

To complete $CS(X)$, the second pass protocol checks if the current is a candidate, and filters an r-write candidate which is same with the current out of the set of write candidates. In this case, the current access was the r-write candidate in the first pass. Filtering r-write candidates is important because the candidates may not be effective or may be pseudo. After the second pass, we report the candidate races as the first races from the corresponding candidate set by Definition 3.6.

Algorithm 2 *The second pass does not use $AH(X)$, but only uses $CS(X)$. $CS(X)$ has three sets of candidates including $CS(X,RW)$.*

1. Determine candidate: *(1) For all accesses in $CS(X)$, if there is no access which is involved in a race with the current, return; (2) In case of the*

read current, return if there is a read candidate in $CS(X, R)$ which happened before the current; and (3) In case of the write current, determine if the current is an r-write candidate, by checking $CS(X, R)$.

2. Update $CS(X)$: *Add the current to the corresponding set of $CS(X)$, if the current is a new candidate; or move the current from $CS(X, W)$ to $CS(X, RW)$, if the current is an r-write candidate.*

3. Halt: *Halt the current thread, if the current is a write or an r-write canidate.*

For example, consider the case shown in the Table 1 and Figure 1. The first access $r0$, in the second execution, is added to $CS(X, R)$, because $r0$ is involved in a race with $w5$ which has been kept in $CS(X, W)$ in the first pass. The next read access $r1$ is returned in its step 1. Another read access $r2$ is added to $CS(X, R)$, because $r2$ is involved in a race with $w3$ which has been kept in $CS(X, W)$ in the first pass. The subsequent accesses $w3$ and $w4$ which have been kept in $CS(X, W)$ are moved to $CS(X, RW)$, because there exist a read candidate $r0$ which is happened before $w3$ and $w4$. And, $w5$ is also moved to $CS(X, RW)$, because of the read candidate $r2$ in $CS(X, R)$ which is happened before $w5$. Finally, five candidates are detected as shown in Table 1: two read candidates and three r-write candidates. When we report the first races with this result in $CS(X)$, we can identify that $w4$ is a pseudo r-write candidate, because there does not exists any read candidate which is concurrent with $w4$. Therefore, the remaining four candidates are involved in two tangled races: $r0$-$w5$ and $r2$-$w3$.

Theorem 2 *There exists a first race a_i-a_j, if and only if a_i-a_j is a candidate race.*

The space complexity of our algorithm depends on V which denotes the number of monitored shared variables, and T which denotes the maximum parallelism of the program. The access history requires $O(T)$ space for each shared variable, because it has at most two accesses for each thread. And then the required space for candidate set is also $O(T)$. The total space complexity of our algorithm is therefore $O(VT)$.

4. Related Works

Some works are reported to detect the first races in parallel programs in [3, 9, 11, 14]. In this section, we describe the existing techniques and compare them with our technique.

Choi and Min [3] propose an on-the-fly technique for reexecuting monitored programs to reproduce undetected races. The undetected races are such the races that are not detected in the first execution of monitored program, and then include the first races. To detect the undetected races, they propose a method which guarantees deterministic reexecution of the program up to the point of the races detected first, allowing additional instrumentation that can locate the previously undetected races. With this method, programmers repeat reproducing undetected races and debugging the parallel program until the program does not include any other races. This method, therefore, requires additional efforts of the programmer to detect the first races in parallel programs, allowing a cyclical debugging such as breakpoint or execution replay. And, such cyclical debugging leads to another drawback, because it may require iterations of monitoring and the cost of monitoring a particular execution is still expensive.

Kim and Jun [11] present a scalable on-the-fly technique for detecting the first races to reduce the central bottleneck to serializing at most two accesses of each thread. This technique checks each access, and compares it with the other accesses to the same shared variable. The main power of the technique is detecting the first races in one monitored execution, but the programs considered in the technique is restricted to programs that have no other inter-thread coordination or synchronization. Jun and McDowell [9] present an on-the-fly technique to detect efficiently the first races in programs that have nested parallelism and no other inter-thread coordination. This technique also detects the first races in one monitored execution.

Netzer and Miller [14] introduce a notion of the first race, called *non-artifact race*, which uses the event-control dependences to define how accesses affect each other. They show a post-mortem method to detect the races by validation and ordering of data races. Data race validation determines the feasible races which involve accesses that either did execute concurrently or could have. Data race ordering identifies the races that did not occur only as a result of other races.

5. Conclusion

In this paper, we present a two-pass on-the-fly algorithm to detect the first races in programs which may have ordered synchronization. This algorithm also can be used for the nondeterministic programs such as Ada programs with *select* statements, if we keep in the first pass the sequence of accesses appeared until the candi-

dates, and then control the second pass using the access sequence.

The main idea of the technique is identifying the set of candidate accesses which includes all accesses involved in first races. In the first pass, a subset of the candidates is gathered. And, the second pass completes the candidate set by detecting the candidates not detected yet in the previous pass. And then, we report the candidate races as the first races.

This technique makes on-the-fly race detection more effective and practical in debugging a large class of shared-memory parallel programs, since the removal of the first races may make the races affected by the first races disappear. We have been implementing our technique in a prototype system called *Race Stand* [10] for debugging races in parallel programming environment.

References

[1] Audenaert, K., *"Clock Trees: Logical Clocks for Programs with Nested Parallelism,"* Tr. on Software Engineering, 23(10): 646-658, IEEE, Oct. 1997.

[2] Callahan, D., K. Kennedy, and J. Subhlok, *"Analysis of Event Synchronization in a Parallel Programming Tool,"* 2nd Symposium on Principles and Practice of Parallel Programming, pp. 21-30, ACM, March 1990.

[3] Choi, J., and S. L. Min, *"Race Frontier: Reproducing Data Races in Parallel-Program Debugging,"* 3rd Symposium on Principles and Practice of Parallel Programming, pp. 145-154, ACM, April 1991.

[4] Dinning, A., and E. Schonberg, *"An Empirical Comparison of Monitoring Algorithms for Access Anomaly Detection,"* 2nd Symposium on Principles and Practice of Parallel Programming, pp. 1-10, ACM, March 1990.

[5] Emrath, P. A., S. Ghosh, and D. A. Padua, *"Detecting Nondeterminacy in Parallel Programs,"* Software, 9(1): 69-77, IEEE, Jan. 1992.

[6] Grunwald, D., and H. Srinivasan, *"Efficient Computation of Precedence Information in Parallel Programs,"* 6th Workshop on Languages and Compilers for Parallel Computing, pp. 602-616, Springer-Verlag, Aug. 1993.

[7] Helmbold, D. P., and C. E. McDowell, *"A Class of Synchronization Operations that Permit Efficient Race Detection,"* 2nd Int'l Conf. on Parallel and Distributed Processing Techniques and Applications, pp. 1537-1548, CSREA, August 1996.

[8] Jun, Y., and K. Koh, *"On-the-fly Detection of Access Anomalies in Nested Parallel Loops,"* 3rd Workshop on Parallel and Distributed Debugging, pp. 107-117, ACM, May 1993.

[9] 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, pp. 1549-1560, CSREA, August 1996.

[10] Kim, D., and Y. Jun, *"An Effective Tool for Debugging Races in Parallel Programs"* 3rd Intl. Conf. on Parallel and Distributed Processing Techniques and Applications, pp. 117-126, CSREA, July 1997.

[11] Kim, J., and Y. Jun, *"Scalable On-the-fly Detection of the First Races in Parallel Programs,"* 12nd Intl. Conf. on Supercomputing, pp. 345-352, ACM, July 1998.

[12] Lamport, L., *"Time, Clocks, and the Ordering of Events in a Distributed System"* Communications of the ACM, 21(7): 558-565, ACM, July 1978.

[13] Mellor-Crummey, J., *"On-the-fly Detection of Data Races for Programs with Nested Fork-Join Parallelism,"* Supercomputing '91, pp. 24-33, ACM/IEEE, Nov. 1991.

[14] Netzer, R. H. B., and B. P. Miller, *"Improving the Accuracy of Data Race Detection,"* 3rd Symp. on Principles and Practice of Parallel Programming, pp. 133-144, ACM, April 1991.

[15] OpenMP Architecture Review Board, *OpenMP Fortran Application Program Interface: Version 1.0,* October 1997.

[16] Parallel Computing Forum, *PCF Parallel Fortran Extensions,* Fortran Forum, 10(3), ACM, Sept. 1991.

Appendix: Proofs of Theorems

Theorem 1 *An access is a candidate which is effective and not pseudo in an execution of program, if and only if the access is involved in a first race.*

Proof ($\Rightarrow$) *If an access a_t is a candidate which is effective and not pseudo, it is involved in a first race.* There are two cases in which the access is an r-write candidate or not.

Case A: *a_t is a read or a write candidate.* An access a_t is effective and not pseudo by Definition 3.4 and

3.5. And by Definition 3.1, there does not exist any access a_h such that $a_h \rightarrow a_t$ and a_h is involved in a race. This implies that there exists no other race that involves an access which happened before a_t. Therefore if a_t is a read or a write candidate, a_t is unaffected by Definition 2.2.

Case B: *a_t is an r-write candidate.* Given a race involving a write access a_t, if a_t is an r-write candidate which effective and not pseudo, there exists a read candidate r_t such that $r_t \rightarrow a_t$, and it satisfies the following conditions: by Definition 3.2, 3.4 and 3.5. Case B.1) *There does not exist write candidates.* If there exists a write candidate w_a, r_t is involved in a race with w_a. This means that a_t is affected by r_t which is a contradiction that a_t is involved in a first race. From this, we can see that w_a must be an r-write candidate. And there exists a read candidate r_a which happened before w_a. In this case, there exists a race r_t-w_a, and then a_t is affected by a read candidate r_t. So a_t-r_a and w_a-r_t are in a tangled. Therefore, a_t is affected by an access r_t which is involved in a tangled race. Case B.2) *There exists a read candidate which is concurrent with a_t.* This is implied in the above Case B.1. Case B.3) *There does not exist any write access w_h such that $w_h \rightarrow a_t$.* If there exists a write access w_h which happened before a_t, there are two subcases: (1) *In the case of $r_t \rightarrow w_h$.* There exists a read candidate r_a and an r-write candidate w_a such that r_a which happened before w_a and w_a which is concurrent with r_t. This implies that w_h is also concurent with w_a because of $r_t \rightarrow w_h$. The read candidate r_t is not happened before r_a because r_t is concurrent with w_a. Therefore w_h is concurrent with r_a. This means that w_h is an r-write candidate which is a contradiction that a_t is an r-write candidate because of $w_h \rightarrow a_t$. (2) *In the case of $w_h \rightarrow r_t$.* From the above subcase (1), if there exists an r-write candidate w_a which is concurrent with r_t, w_h is also concurrent with w_a. This implies that w_h is a candidate, and both r_t and a_t cannot be candidates which are contradiction. Therefore if an access is a candidate which is effective and not pseudo, it is involved in a first race.

($\Leftarrow$) *If an access a_t is involved in a first race, it is a candidate which is effective and not pseudo.* The first race is either an unaffected race or involved in a tangled race by Definition 2.5. There are two cases.

Case A: *a_t is a read access.* Unaffected race with a_t means that there does not exist any access a_s such that $a_s \rightarrow a_t$. This implies that a_t is a read candidate by Definition 3.1, and is effective and not pseudo by Definition 3.4 and 3.5.

Case B: *a_t is a write access.* Given an access a_s such that $a_s \rightarrow a_t$, there are two subcases. Case B.1) *If a_s is a write access.* In this case, by above Case A, a_t is write candidate with effective and not pseudo because a_s should not involved in any race. Case B.2) *a_s is a read access.* (1) *a_t is not involved in a race.* This means that there does not exist any write candidate which is concurrent with a_s. So a_t is effective, and there exists a read candidate which is concurrent with a_t because a_s is not involved in any race, which implies that a_t is not pseudo. (2) *a_s is involved in a race.* If there exists a write candidate w_a which is involved in a race with a_s, this implies that a_t is affected by a_s. If a_t is, however, involved in a first race, w_a should be an r-write candidate. Therefore a_t is effective, and there exists a read candidate r_a which is concurrent with a_t which means not pseudo. Therefore, if an access is involved in a first race, it is a candidate which is effective and not pseudo. *Q.E.D.*

Theorem 2 *There exists a first race a_i-a_j, if and only if a_i-a_j is a candidate race.*

Proof ($\Rightarrow$) *If there exists a first race a_i-a_j, then a_i-a_j is a candidate race.* We prove this with its contrapositive: *If a_i-a_j is not a candidate race, a_i-a_j is not a first race.* If a_i-a_j is not a candidate race, by Definition 3.6, there are two cases.

Case A: *Ether a_i or a_j is not a candidate.* By Theorem 1, a_i or a_j is affected by a race which is not a tangled race.

Case B: *Both a_i and a_j are candidates, and a_i-a_j is not a candidate race.* By Definition 3.6, there are two subcases: Case B.1) *Both a_i and a_j is read candidates.* In this case, a_i-a_j is not a race because the two accesses are not conflicting. Case B.2) *Both a_i and a_j is r-write candidates.* In this case, by Theorem 1, both a_i and a_j are affected. Therefore, If a_i-a_j is not a candidate race, a_i-a_j is not a first race.

($\Leftarrow$) *If there exists a candidate race, a_i-a_j is a first race.* From Definition 3.6, there are three cases. Case A: *Both a_i and a_j are write candidates.* Case B: *One of a_i and a_j is a read candidate and the other is a write candidate.* In the above two cases, neither a_i nor a_j is affected by Theorem 1. Case C: *One of a_i and a_j is a read candidate and the other is an r-write candidate.* In this case, one read candidate is not affected, and the other r-write candidate is not a pseudo candidate and is affected by an access which is involved in a tangled race. Therefore, if there exists a candidate race, a_i-a_j, a_i-a_j is unaffected or in a tangle, so a_i-a_j is a first race. *Q.E.D.*