# 내포 병렬성을 가지는 OpenMP 프로그램의 최초 경합 탐지

천 병 규[†]·우 종 정[††]·전 용 기[†††]


## 요 약

공유 변수를 가지는 병렬 프로그램의 오류 수정에서 경합 탐지는 중요하다. 왜냐하면, 경합은 프로그램의 비결정적인 수행을 유발하기 때문이다. 기존에 제시된 병렬 프로그램의 오류 수정 기법인 수행중 탐지 기법은 내포된 병렬 프로그램에서 최초경합 탐지를 보장할 수 없다. 최초 경합을 수정하면 이후에 발생하는 경합들이 나타나지 않을 수 있으므로, 최초경합의 탐지는 중요하다. 본 논문에서는 내포 병렬 루프 프로그램을 대상으로 반복 수행을 통해서 최초경합을 탐지하는 기법을 제시한다. 반복 수행의 횟수는 최악의 경우에 프로그램의 내포 깊이 만큼이며, 각 수행시의 효율성은 공유변수의 개수를 V, 프로그램의 최대 병렬성을 T라 할 때, 공간 복잡도 $O(VT)$와 시간 복잡도 $O(T)$를 가지므로 기존의 수행중 탐지 기법과 동일하다. 그러므로 본 기법은 효과적이고 실용적인 오류 수정을 가능하게 한다.


# Detecting the First Race in OpenMP Program with Nested Parallelism

Byoung-Gyu Chon[†] · Jong-Jung Woo[††] · Yong-Kee Jun[†††]


## ABSTRACT

It is important to detect races for debugging shared-memoy parallel programs, because the races cause unintended nondeterministic program execution. Previous on-the-fly techniques to detect races can not guarantee the first race detection in nested parallel programs. Detecting the first race is important for debugging parallel programs, since the removal of the first race may make the next occurred races disappear. In this paper, we presents an on-the-fly detection technique to detect all of the first races through the reexecution of the debugged programs. We assume that the debugged parallel program may have one-way nested parallel programs. The number of reexecution is at the least the nesting depth of the program in the worst case. The space complexity is $O(VT)$ and the time complexity to detect race in each access of access history is $O(T)$, where $V$ is the number of shared variables and $T$ is the maximum parallelism of the program. This efficiency of our technique in each execution is the same with the previous on-the-fly detection techniques. Therefore, this technique makes debugging parallel programs more effective and practical.


**키워드 : 병렬프로그램(Parallel Program), 디버깅(Debugging), 최초경합(First Race), 수행중 탐지(On-the-fly Detection)**

## 1. 서 론

공유 메모리를 갖는 다중 처리기 시스템에서 수행되는 병렬 프로그램을 작성하고 오류를 수정하는 것은 순차 프로그램에 비해 어렵다[12]. 주된 요인은 동일한 입력에 대해서 수행 시마다 동일한 결과를 보장할 수 없는 병렬 프로그램의 의도되지 않은 비결정적 수행에 의한 오류이다. 이러한 오류는 공유 변수에 대해서 적어도 하나의 쓰기 접근이 동기화 없이 병행하게 수행되는 경합(races)에 의해 발생하므로 경합은 수정되어야 하는 중요한 오류이다. 경합이 제거되어 결정적인 수행이 보장된 병렬 프로그램은 기존의 반복적인 순차 디버거로 그 밖의 논리적인 오류를 수정할 수 있다.

경합 탐지 기법에는 정적 분석 기법(static analysis)[1, 5, 6], 사후추적 분석 기법(postmortem analysis)[4, 14], 수행중 탐지 기법(on-the-fly detection)[3, 7, 8, 10] 등이 있다. 정적 분석 기법은 프로그램을 수행시키지 않고 원시 프로그램을 분석하여 경합을 탐지한다. 그러므로, 실제 수행 시에 나타나지 않는 경합이 너무 많이 탐지된다는 단점을 가지고 있다. 사후추적 분석 기법은 프로그램 수행 후 저장되어 있는 추적 파일을 분석하여 경합을 탐지하는 기법이다. 추적 파일은 프로그램의 한 수행에 의존적이며, 추적 파일을 저장하기 위한 기억 장소의 부담이 크다는 단점이 있다. 수행중 탐지 기법은 정적 분석 기법에 비해서 접근이 발생하는 시점에서 실제적인 경합만을 탐지한다. 그리고, 사후추적 분석 기법에 비해서 경합을 일으킬 수 있는 사건들에 대해

※ 본 연구는 정보통신부에서 지원하는 대학기초 연구지원사업으로 수행되었음.
† 정 회 원 : 한국항공우주산업(주) 주임연구원
†† 종신회원 : 성신여자대학교 컴퓨터정보학부 교수
††† 종신회원 : 경상대학교 컴퓨터과학과 교수, 컴퓨터·정보통신연구소 연구원
논문접수 : 2000년 9월 6일, 심사완료 : 2001년 9월 7일

필요한 정보만을 유지하기 때문에 기억 장소의 부담이 적다. 또한, 경합이 있다면 적어도 하나는 반드시 탐지할 수 있다는 장점이 있다.

수행중 탐지 기법은 기억 장소 부담을 줄이기 위해 가장 최근에 발생한 접근들만을 유지하기 때문에, 처음으로 탐지된 경합이 처음으로 발생한 경합임을 보장하지 못한다. 본 논문에서는 처음으로 발생한 경합을 최초경합(first race)이라고 한다. 최초경합이 수정되면, 이후에 발생한 경합들은 나타나지 않을 수도 있다. 그러므로, 최초경합의 탐지는 디버깅에 있어서 중요하다. 이전에 제시된 기법[10]은 비내포 병렬 프로그램에서만 최초경합을 탐지할 수 있었다. 본 논문에서는 동기화 없는 다중 내포 구조를 가진 OpenMP[13] 프로그램을 대상으로, 프로그램을 재 수행하면서 반복적인 감시를 통해 최초경합을 효과적으로 탐지하는 기법을 제시한다. 2장에서는 본 연구와 관련된 기본적인 개념을 소개한다. 3장에서는 제시된 반복 수행 기법을 설명하고, 4장에서는 기존의 기법들과 비교 및 분석을 한다. 5장에서 결론을 맺는다.

## 2. 연구 배경

### 2.1 병렬 루프 프로그램의 수행

병렬프로그램의 수행에서 프로그램의 분기 및 종료 명령이 포함되지 않으면서 순차적으로 수행되는 하나의 명령 열을 쓰레드(thread)라고 한다. OpenMP Fortran에서 PARALLEL DO와 같은 병렬 루프의 시작 명령을 만나면 병행하게 수행되는 쓰레드들이 생성되고 대응되는 END DO에 의해 병행한 쓰레드들이 종료되어 단일 쓰레드로 수행을 한다. 병행하게 수행되는 쓰레드들간에 동기화 명령은 없으며, 하나의 쓰레드 내에 또 다른 루프가 나타날 수 있는데 이를 내포되었다 라고 한다. 각각의 병렬 루프가 가지는 내포 수준은 외부 루프의 내포 수준보다 1이 증가한 값을 가진다.

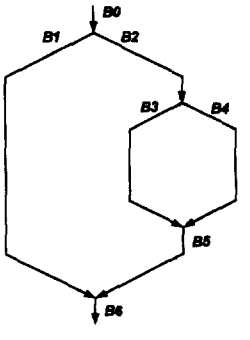
**[정의 2.1]** 병렬 프로그램에서 루프의 한 쓰레드 내에 동일한 내포 수준을 가지는 루프가 하나만 존재하면 단일 방향 내포 루프(one-way nested loop)라 하고, 한 쓰레드 내에 동일한 내포 수준을 가지는 루프가 두개 이상이 존재하면 다중 방향 내포 루프(multi-way nested loop)라 한다.

본 논문에서는 단일 방향 내포 루프를 가진 병렬 프로그램을 대상으로 한다. (그림 1)은 OpenMP Fortran으로 작성된 대상 프로그램의 예를 나타낸 것이다. 병렬 프로그램의 동적인 수행은 방향성이 있는 비순환 그래프인 POEG (Partial Order Execution Graph)[3]으로 나타낼 수 있다. POEG은 병렬 프로그램의 부분적인 실행 순서를 나타내며 사건들간의 발생후(happened-before) 관계[11]를 표현한다. (그림 2)는 대상 프로그램의 수행 흐름을 POEG으로 재 표현한 것이다.

POEG은 정점(vertex)과 간선(edge)으로 구성된다. POEG에서 정점은 쓰레드의 생성(fork) 및 종료(join) 명령을 의미하고, 두 개의 정점을 연결하는 간선은 순차적으로 수행되는 임의의 명령 열을 의미한다. 쓰레드의 생성과 종료 명령이 포함되지 않은 하나의 명령 열을 블록(block)이라고 한다. POEG에서 블록들간에 경로가 존재하면 순서가 보장됨을 의미한다. 예를 들어, (그림 2)에서 블록 B0에서 B1로 경로가 존재하므로 B0은 B1보다 먼저 수행된다. 반면에 B1과 B2 사이에는 경로가 없으므로 두 블록은 병행하게 수행된다.

```
c234567
!$omp PARALLEL SHARED(M)
!$omp DO
      DO I = 1, N
      A(I) = B(I) + M
      IF (I .GT. 1) THEN
!$omp       PARALLEL SHARED(M)
!$omp       DO
            DO J = 1, I
               IF (J .EQ. 1) THEN
                  M = J
               ELSE
            M = M + J
            C(I-1, J) = A(I) + M
               ENDIF
            END DO
!$omp       END PARALLEL
      ENDIF
      END DO
!$omp END PARALLEL
```

(그림 1) OpenMP Fortran 프로그램




(그림 2) POEG

**[정의 2.2]** 임의의 공유 변수에 대한 접근인 $e_i$와 $e_j$가 포함된 POEG에서, 만약 $e_i$에서 $e_j$로의 경로가 존재하면 $e_i$는 $e_j$ 이전에 발생한 사건이며, $e_j$는 $e_i$에 순서화되었다라고 한다. 또한 두 사건간에 경로가

존재하지 않으면 $e_i$와 $e_j$는 병행하다라고 한다.

수행중 탐지 기법은 수행 중에 발생하는 사건들의 병행성을 검사하기 위해 병행성 정보를 사용한다. 병행성 정보는 각 블록에 동적으로 부여되는 레이블이며, 본 논문에 적용되는 레이블은 NR-labeling[8] 기법을 사용해서 생성된다고 가정한다. 두 사건 사이에 보장된 순서가 존재하는지를 검사하기 위해 ordered() 함수를 사용한다. 예를 들어, 임의의 두 사건을 $e_i$와 $e_j$라 할 때 ordered($e_i$, $e_j$)를 만족하면 $e_i$에서 $e_j$로 경로가 존재하거나 $e_j$에서 $e_i$에로 경로가 존재함을 의미한다. 내포 관계는 nested($e_i$, $e_j$) 함수로 식별될 수 있다. 내포 깊이는 nestdepth() 함수에 의해 계산되며 두 사건이 동일 블록에서 발생했는지를 검사하는 함수 sameblock($e_i$, $e_j$)은 병행성 정보가 동일할 때 만족된다.

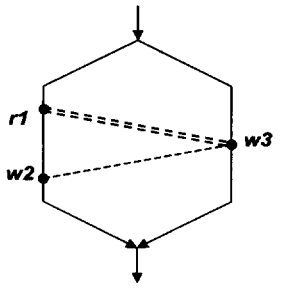
**[정의 2.3]** 임의의 공유 변수에 대한 접근인 $e_i$와 $e_j$가 존재할 때, ordered($e_i$, $e_j$) ∧ (nestdepth($e_i$) ≥ nestdepth ($e_j$)) 를 만족하면, $e_i$는 $e_j$에 내포된 사건(nested event)이다.

수행중 탐지 기법은 프로그램의 수행 중에 경합을 탐지하기 위해 접근 역사를 사용한다. 접근 역사는 공유 변수에 대한 접근들이 발생하는 시점에 유지되며, 이전의 접근들의 병행성 정보를 저장하고 있다. 임의의 공유 변수를 X라 할 때 읽기 접근 역사(AH_R(X))에는 가장 최근의 읽기 접근과 이에 병행한 읽기 접근들의 병행성 정보가 유지된다. 쓰기 접근 역사(AH_W(X))에는 가장 최근의 쓰기 접근의 병행성 정보만 유지된다. 수행중 공유 변수에 대한 접근이 발생하면 접근 역사에 저장된 접근들의 병행성 여부가 검사되고, 적어도 하나의 쓰기가 병행하게 수행되는 접근의 쌍이 경합으로 보고된다. 또한, 이후에 발생하는 경합을 탐지하기 위해 발생한 접근의 병행성 정보가 접근 역사에 저장된다.

병렬 프로그램의 수행에서 임의의 공유변수에 대해서 적어도 하나의 쓰기 사건이 포함된 접근을 충돌 사건(conflicting event)이라 한다. 만약 충돌 사건들이 병행하게 수행되면 프로그램의 비결정적 수행의 원인인 경합이 된다. 최초경합은 동일한 공유 변수에서 발생하는 경합 중 가장 먼저 발생하는 경합을 의미한다. 따라서, 최초경합이 수정되면 이후에 발생하는 경합들은 나타나지 않을 수 있다. 그러므로, 최초경합의 탐지는 프로그램의 오류 수정을 실질적이며 효과적으로 수행할 수 있게 한다.

### 2.2 최초경합 탐지 기법

수행 중에 최초경합을 탐지하는 이전의 탐지 기법[10]은 비내포 병렬 루프 프로그램을 대상으로 한다. [10]에서는 최초경합을 구성할 수 있는 접근을 후보 사건(candidate event)이라 정의한다. 예를 들어, 경합을 $e_i$-$e_j$라 표현 할 때, $e_i$와 $e_j$가 후보 사건이면 $e_i$-$e_j$는 최초경합에 포함된다. 후보 사건은 한 블록에서 처음으로 발생한 읽기 또는 쓰기 접근을 말하며, 각각 읽기 접근 역사와 쓰기 접근 역사에 저장된다. 읽기 후보 사건 이후에 처음으로 나타난 쓰기 접근은 쓰기 후보 사건이 없는 경우에만 후보 사건이 된다. 이러한 접근을 읽기_쓰기 접근이라 하며 새로운 장소인 읽기_쓰기 접근 역사(AH_RW(X))에 저장된다. 추가된 접근 역사를 이용해서 수행 중에 최초경합을 탐지하는 프로토콜은 읽기와 쓰기 접근이 발생하면 호출되는 ***CheckRead(X, current)***와 ***CheckWrite(X, current)*** 함수로 구성된다. 여기서, current는 공유변수 X에 대한 접근을 말한다.




(그림 3) 비내포 루프의 최초경합

(그림 3)은 비내포 병렬 루프의 한 부분을 POEG으로 나타낸 것이다. POEG에서 $r_i$, $w_j$는 각각 읽기와 쓰기 접근을 나타내고 첨자는 임의의 발생 순서를 말한다. r1은 처음으로 발생한 접근이므로 후보 사건이 되며 AH_R(X)에 저장된다. w2는 r1 이후에 처음으로 발생한 접근이므로 AH_RW(X)에 저장된다. 후보 사건인 w3은 r1과 병행하기 때문에 r1-w3은 최초경합에 포함된다. w2-w3은 이전의 r1에 순서화되었으므로 최초경합에 포함되지 않으며 w2는 후보 사건이 아니므로 AH_RW(X)에 저장되지 않는다.

[10]에서는 병행하게 수행되는 쓰레드들에서 발생하는 최초 사건들의 이력을 유지하는 기법을 고안하였다. 이 기법은 동일 쓰레드 내에서 최초경합이 될 수 있는 사건들에 대해서만 실질적인 최초경합 탐지가 가능하게 함으로서, 공유 변수의 접근 역사에 대한 병목현상을 해결하였다.

## 3. 반복적인 감시를 통한 반복 수행 기법

### 3.1 루트 사건

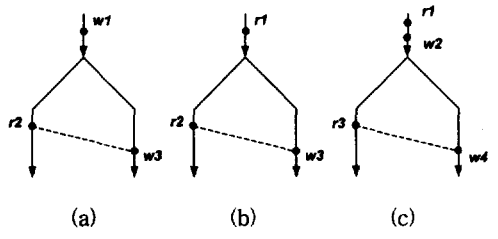
내포 루프 구조에서 최초경합이 후보 사건에 내포된 블록에서 발생했을 때 기존에 제시된 기법으로는 탐지할 수 없다. 즉, 최초경합을 내포하는 후보 사건이 수행 후에 접근 역사에 존재하면 탐지되지 못한 최초경합이 존재함을 의미한다. 이와 같은 후보 사건을 본 논문에서는 루트(root) 사건이라 정의한다.

**[정의 3.1]** 임의 공유 변수 X에 대한 후보 사건인 $e_i$가 다

음의 조건을 만족하면 $e_i$는 X의 루트 사건이다.

1) $e_i$에 내포된 사건들 중 적어도 하나는 쓰기 접근이다.
2) $e_i$에 내포된 사건들로 구성된 경합들의 집합이 최초경합에 포함된다.

프로그램 수행에서 나타날 수 있는 루트 사건은 (그림 4)와 같다. (a)에서 후보 사건 w1에 순서화된 r2-w3은 최초경합에 포함되지만 탐지될 수 없다. 그러므로, w1은 루트 사건이다. (b)에서 최초경합에 포함되는 r2-w3은 탐지될 수 없다. 왜냐하면, 이전에 발생한 후보 사건 r1에 의해 r2가 후보 사건이 될 수 없기 때문이다. 그러므로, r1은 루트 사건이다. (c)에서 r1 이후의 발생한 r2-w3은 최초경합에 포함되는 경합이지만 탐지되지 못한다. 그러므로, 후보 사건 r1은 루트 사건이다.




(그림 4) 루트 사건

**[보조정리 3.1]** 프로그램 수행후 임의의 공유 변수 X에 대한 접근 역사인 AH_W(X)에 하나의 쓰기 접근 $w_i$만 존재하고, AH_R(X) = Ø 일 때, $w_i$는 X의 루트 사건이다.

**[증명]** X에 대한 쓰기 접근 $w_i$에 대해서 i = j 일 때 ordered($w_i$, $w_j$) ∧ (nestdepth($w_i$) ≥ nestdepth($w_j$))를 만족하므로 $w_i$는 자신에 내포된 사건이다. 그러므로, 루트 사건의 조건 1)을 만족한다. 유일한 $w_i$는 AH_R(X) = Ø 이면 병행한 읽기 접근이 없으므로 경합이 될 수 없다. 즉, $w_i$가 구성하는 경합의 집합 $S_x$는 공집합이다. $S_x$는 최초경합에 포함되므로 조건 2)를 만족한다. 그러므로 $w_i$는 X의 루트 사건이다. ■

**[보조정리 3.2]** 프로그램 수행 후에 AH_R(X)와 AH_RW(X)에 포함된 사건들의 집합을 각각 $S_r$과 $S_{rw}$라 하고, AH_W(X) = Ø 일 때, $S_{rw}$가 속한 블록들의 공통 조상 블록에 속한 $r_i \in S_r$ 가 존재하면, $r_i$는 X의 루트 사건이다.

**[증명]** $r_i$가 속한 블록이 $S_{rw}$가 속한 블록들의 공통 조상 블록이므로 $r_i$에 내포된 사건 $rw_j \in S_{rw}$ 가 존재한다. 그러므로, 루트 사건의 정의 1)을 만족한다. AH_W(X) = Ø 이면 $r_i$과 병행한 쓰기는 존재하지 않는다. 즉, $r_i$는 경합을 구성할 수 없다. 그러므로, $r_i$에 내포된 $rw_j$가 구성하는 경합은 최초경합에 포함된다. 따라서, 정의 2)를 만족한다. 그러므로, $r_i$는 루트 사건이다. ■

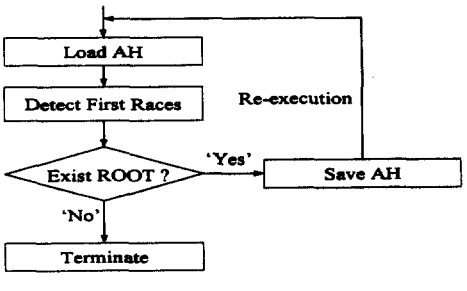
**[보조정리 3.3]** 프로그램 수행 후에 AH_R(X)과 AH_RW(X)에 각각 하나의 읽기 접근 r과 쓰기 접근 rw만 존재하고, AH_W(X) = Ø 일 때, 두 사건이 동일 블록에 속하면 r은 루트 사건이다.

**[증명]** AH_RW(X)에 존재하는 rw는 읽기 이후에 나타난 접근이다. r은 유일한 접근이므로 rw는 ordered(r, w) ∧ (nestdepth(r) ≥ nestdepth(w))를 만족한다. r에 내포된 rw가 존재하므로 r은 루트 사건의 정의 1)을 만족한다. AH_W(X) = Ø이므로 r은 경합을 구성할 수 없다. 그러므로, rw가 구성하는 경합은 최초경합에 포함된다. 따라서 정의 2)를 만족한다. 그러므로 r은 루트 사건이다. ■

**[정리 3.1]** 프로그램 수행후 임의 공유 변수 X의 접근 역사 내에 접근들의 집합 $S_x$가 존재하고, $S_x$내에 적어도 하나의 쓰기 접근이 포함될 때, $S_x$에 포함되면서 경합을 구성하지 않는 사건 $e_i$는 X의 루트 사건이다.

**[증명]** 접근 역사에 적어도 하나의 쓰기 접근이 포함된 경우는 1) AH_W(X) ≠ Ø 이거나 혹은, 2) AH_RW(X) ≠ Ø 이다. 1)의 경우에 경합을 구성하지 않는 사건은 AH_R(X) = Ø 일 때, 유일한 쓰기 접근인 경우이다[보조 정리 3.1]. 2)의 경우에는 AH_RW(X)에 속한 모든 접근에 순서화된 유일한 읽기 사건[보조정리 3.2] 와 각각 하나의 읽기 와 동일 블록의 읽기_쓰기 접근의 경우이다[보조 정리 3.3]. 그러므로, $e_i \in S_x$ 는 X의 루트 사건이다. □

프로그램의 수행 후에 루트 사건의 존재 여부로 탐지되지 못한 최초경합이 존재하는지를 판단할 수 있다. 루트 사건 이후에 발생한 후보 사건을 검사함으로써 최초경합은 탐지될 수 있으며, 만약 루트 사건이 존재하지 않으면 더 이상의 최초경합은 존재하지 않음을 의미한다. 본 논문에 제시된 기법은 루트 사건이 존재하면 재 수행을 함으로서

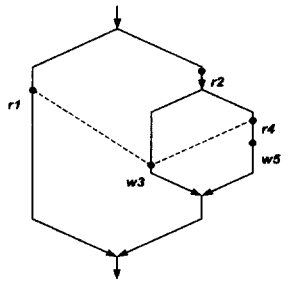



(그림 5) 최초경합 탐지 과정

최초경합을 탐지한다. 또한, 루트 사건은 추가된 기억 장소인 ROOT(X)에 저장되어 재 수행 시에 루트 사건 이후의 후보 사건이 검사될 수 있게 한다.

### 3.2 내포된 최초경합 탐지

재 수행을 통해서 최초경합을 탐지하는 과정은 (그림 5)와 같다. 프로그램 수행 전에 이전의 수행에서 저장된 접근 역사가 적재된다. 수행 중에 공유 변수에 대한 접근이 발생하면 후보 사건인지를 판단하고, 접근 역사에 저장된 이전의 접근과 비교하여 최초경합을 보고한다. 후보 사건들은 이후에 발생하는 접근들과 비교하기 위해 접근 역사에 저장된다. 수행 후에 루트 사건이 존재하는지 판단하기 위해 접근 역사를 검사한다. 루트 사건이 존재하지 않으면 최초경합 탐지가 종료되고, 그렇지 않으면 재 수행을 한다.




(그림 6) 내포 루프의 최초경합

(그림 6)은 대상 프로그램에서 발생한 최초경합을 POEG으로 표시한 것이다. 최초 수행에서 r1과 r2는 후보 사건이므로 AH_R(X)에 저장된다. w3이 발생하면 AH_R(X)에 저장된 r1과의 경합인 ***r1-w3***은 최초경합으로 보고되고, w3은 r2 이후에 발생한 접근이므로 AH_RW(X)에 저장된다. r4는 후보사건이 아니므로 저장되지 않는다. w5가 발생하면 ***r1-w5***는 최초경합으로 보고되고 w5는 읽기_쓰기 접근이므로 AH_RW(X)에 저장된다. 수행 후에 AH_RW(X)에 저장된 w3과 w5의 공통 조상 블록에서 발생한 r2는 루트 사건이 된다. 왜냐 하면, r2에 각각 내포된 사건으로 구성된 경합인 ***w3-r4***는 최초경합에 포함되지만 보고될 수 없기 때문이다. 재 수행 시에 r2 이후의 블록에서 w3이 발생하면 AH_RW(X)에 포함된 접근으로 구성된 경합인 ***r1-w5***는 최초경합에서 삭제된다. r4가 발생하면 ***w3-r4***가 최초경합으로 보고되며, w5는 후보 사건이 아니므로 무시되고 더

| | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| Root | | | | | |
| R | r1 | r1, r2 | r1, r2 | r1, r2 | r1, r2 |
| RW | | | w3 | w3 | w3, w5 |
| W | | | | | |

| | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| Root | r2 | r2 | r2 | r2 | |
| R | r1 | r1 | r1 | r1, r4 | r1, r4 |
| RW | | | | | |
| W | | | w3 | w3 | w3 |

(그림 7) 접근 역사

이상의 루트 사건이 없으므로 종료하게 된다. (그림 7)은 사건들이 발생할 때마다 변화되는 접근 역사의 내용을 보인 것이다. 여기서 프로그램은 두 번 수행되어지며 각 사건들의 발생 순서는 임의의 한 수행 예를 의미한다.

읽기_쓰기 접근들의 공통 조상 블록에서 발생한 읽기 접근을 식별하는 함수는 (그림 8)과 같다. 읽기 접근들 중에서 임의의 한 읽기_쓰기 접근과 순서화된 접근은 반드시 존재한다. 왜냐 하면, 읽기_쓰기 접근은 읽기 이후에 발생한 접근이기 때문이다. AH_RW[1]과 순서화된 접근은 임시 장소로 옮겨진다(3 행). 이 접근이 모든 읽기_쓰기 접근들과 순서화 되었으면 공통 조상 블록의 접근이고(7행), 하나라도 순서화 되지 않았거나 동일 블록이면 공통 조상 블록의 접근은 존재하지 않게 된다(5,6행). 예를 들어, AH_R(X)에 포함된 접근을 r1, r2라 하고 AH_RW(X)에 포함된 접근을 rw3, rw4라 할 때, r1이 rw3과는 순서화 되고 rw4와는 순서화 되지 않았다면, rw4와 순서화된 r2가 존재하더라도 r1과 순서화된 rw4와는 순서화 되지 못한다. 왜냐 하면, 접근 역사에 저장된 r1과 r2는 병행한 접근이며, r2에 순서화된 rw4도 역시 r1과 병행한 접근이기 때문이다. 즉, 읽기 접근 역사에 포함된 접근들 중에서 읽기_쓰기 접근 역사 내에 포함된 임의의 접근과 비교해서 처음으로 발견된 순서화된 읽기 접근이 읽기_쓰기 접근들의 공통 조상 블록에서 발생한 읽기 사건이 되거나, 아니면 공통 조상 블록에서 발생한 읽기 접근은 존재하지 않게 된다. 그러므로, c_ances() 함수의 효율성을 나타내는 접근들 간의 비교 횟수는 최악의 경우에도 접근 역사에 저장된 접근 개수의 두 배로 한정된다.

```
1 c_ances(AH_RW(X), AH_R(X))
2   for all a ∈ AH_R(X) do
3     if ordered(a, AH_RW[1]) then move a to temp ;
4   for all a ∈ AH_RW(X) do
5     if not ordered(a, temp) ∨ sameblock(a, temp) then
6       begin move temp to AH_R(X) ; return ; end ;
7   return temp ;
8 end c_ances
```

(그림 8) c_ances() 함수

### 3.3 최초경합 탐지를 위한 반복 수행 알고리즘

경합을 탐지하기 위해서는 프로그램 내에 프로토콜을 호출하는 삽입문이 위치해야 된다. (그림 9)는 대상 프로그램의 예이다. 병행성 정보와 관련된 함수들인 init_label(), fork_label(), join_label() 등은 프로그램 초기와 병행한 블록의 생성과 종료 시에 호출된다. 대상 프로그램에서 공유 변수는 첫 번째 루프에서의 변수 M과 내포된 루프에서의 변수 M이다. 접근이 발생한 직전의 위치에서 프로토콜이 호출되는데 읽기 혹은 쓰기에 따라 공유 변수와 접근의 병행성 정보를 인자로 한다. checkroot()에서 모든 공유 변수에 대해서 루트 사건이 있는지 검사되고, 프로그램의 시작과 끝

에는 접근 역사를 적재 및 저장을 위한 함수가 위치하고 있다.

```
c234567
      init_label()
      load_root()
!$omp PARALLEL SHARED(M)
!$omp DO
      DO I = 1, N
      fork_label(i)
      checkread(M, i_label)
        A(I) = B(I) + M
        IF (I .GT. 1) THEN
!$omp     PARALLEL SHARED(M)
!$omp     DO
          DO J = 1, I
      fork_label(j)
            IF (J .EQ. 1) THEN
      checkread(M, j_label)
              M = J
            ELSE
      checkread(M, j_label)
      checkwrite(M, j_label)
              M = M + J
              C(I-1, J) = A(I) + M
            ENDIF
          END DO
!$omp END PARALLEL
      join_label(j)
        ENDIF
      END DO
!$omp END PARALLEL
      join_label(i)
      ckeckroot()
      save_root()
```

(그림 9) 삽입문이 첨가된 대상 프로그램

반복 수행 프로토콜은 (그림 10)과 같이 5개의 함수로 구성된다. load_root() 함수(1 행)는 수행 전에 접근 역사를 적재하고, checkread() 및 checkwrite() 함수(3, 7 행)는 수행 중에 공유 변수에 대한 읽기 및 쓰기 접근이 발생했을 때 호출된다. checkroot() 함수(11 행)는 수행 후에 접근 역사 내에 루트 사건이 존재하는지를 검사하고, save_root() 함수(30 행)는 접근 역사를 저장한다.

```
1 load_root(X)
2 /* root file을 AH_ROOT(X) 에 load */

3 checkread(X, current)
4   if nested(ROOT(X), current) then return ;
5   s_checkread(X, current) ;
6 end checkread

7 checkwrite(X, current)
8   if nested(ROOT(X), current) then return ;
9   s_checkwrite(X, current) ;
10 end checkwrite

11 checkroot(X)
12   clear ROOT(X) ;
13   if AH_W(X) ≠ Ø then
14     if single(AH_W(X)) ∧ AH_R(X) = Ø then
15     begin move AH_W(X) to ROOT(X) ; return ; end ;
16   else if ROOT(X) : = c_ances(AH_RW(X), AH_R(X)) then
17     begin clear AH_RW(X) ; return ; end ;
18   else if single(AH_RW(X)) ∧ single(AH_R(X)) then
19     begin move AH_R(X) to ROOT(X) ; clear AH_RW(X) ;
       return ; end ;
20   endif
21   exit ;
22 end checkroot

23 save_root(X)
24 /* AH_ROOT(X)를 root file 에 save */
```

(그림 10) 반복 수행 프로토콜

수행 중에 임의의 공유 변수에 대한 접근이 발생하면 checkread() 와 checkwrite() 함수가 호출된다. 루트 사건을 내포하는 접근은 경합에 포함되지 않는다. 그러므로, 발생한 접근에 대해 루트 사건이 내포되었는지를 검사해야 한다(4, 8 행). 발생된 접근들이 후보 사건인지를 판단하고 최초경합을 탐지하여 보고하는 s_checkread() 와 s_checkwrite()는 기존의 최초경합 탐지 기법[10]의 함수와 동일하다(5, 9행).

수행 후에 checkroot() 함수가 호출되면 접근 역사에 루트 사건이 존재하는지 검사된다. 가장 최근의 루트 사건만을 유지하기 위해 이전의 루트 사건은 삭제된다(12행). single() 함수는 접근 역사에 하나의 접근만 포함되어 있을 때 만족한다. 읽기 후보 사건이 없을 때 유일한 쓰기 후보 사건이 존재하면 루트 사건이 된다(14행). 쓰기 후보 사건이 없을 때, 모든 읽기_쓰기 접근의 공통 조상 블록의 읽기 후보 사건은 루트 사건이 된다(16행). 쓰기 후보 사건이 없을 때, 각각 하나의 읽기 후보 사건과 읽기_쓰기 접근이 동일한 블록이면 읽기 후보 사건은 루트 사건이다(18행). 루트 사건이 존재하면 ROOT(X)에 저장된 후에 checkroot() 함수가 종료되고(15, 16, 19행), 접근 역사를 저장하는 save_ah() 함수(23행)가 호출되고, 재 수행을 하게 된다. 루트 사건이 존재하지 않으면 더 이상의 최초경합이 없으므로 프로토콜은 종료된다(21행).

반복 수행 프로토콜은 비내포 병렬프로그램을 대상으로 하는 이전의 기법을 내포된 프로그램으로 확장한 것이다. 본 기법에서는 내포된 프로그램에서 최초경합을 탐지하기 루트사건의 개념을 정의하였으며, 발생되는 접근들 중에서 루트사건을 지정함으로서 내포되어 있는 모든 최초경합을 탐지한다. 루트사건이 존재하면, 루트사건에 내포된 최초경합이 있을 수 있으므로 루트사건 이후에 나타나는 접근들을 재 수행하여 반복적인 감시를 통해서 최초경합을 탐지

한다. 최악의 경우 루트 사건이 각 내포 블록에서 순서적으로 발생할 수 있다. 그러므로, 최초경합 탐지를 위한 재 수행 횟수는 프로그램의 최대 내포 깊이 이하이다.

**[정리 3.2]** 내포 깊이가 n인 프로그램에서 최초경합을 탐지하기 위한 재 수행 횟수는 기껏해야 n번이다.

**[증명]** 내포 깊이가 i인 블록 $B_i$에서 루트 사건이 존재한다면 $root_i$로 나타낼 수 있다. 루트 사건의 정의에 의해 루트 사건을 내포한 블록 $B_j(j \leq i)$에서는 최초경합이 존재하지 않는다. 즉, $root_i$로 인해 탐지되지 못한 최초경합이 존재한다면 $B_k(i < k \leq n)$내에 있다. 한번의 수행 이후에 발생할 수 있는 가장 근접한 루트 사건은 $root_{i+1}$이다. 만약 루트 사건이 $root_n$이라면, $k = n + 1$ 인 $B_k$는 존재하지 않으므로 더 이상의 최초경합은 없다. 따라서, 탐지되지 못한 모든 최초경합은 $root_k$가 발생할 수 있는 $(n - i)$번의 재 수행을 통해서 탐지될 수 있다. 최악의 경우 최초경합이 존재하지 않지만 루트 사건이 n번 발생하는 경우 n번의 재 수행이 필요하다. 그러므로, 재 수행 횟수는 기껏해야 n번이다. □

## 4. 관련 연구 및 분석

처음으로 발생하는 경합을 탐지하기 위해 제시된 최초의 기법은 Race Frontier[2]이다. 이 기법은 수행중 기법에 의해 경합이 탐지되면 경합을 유발한 사건직전에 중단점(break point)을 설치한 후 재 수행을 통해서 최초경합을 탐지할 수 있도록 유도해 주는 전통적인 순환적 디버깅 분야에 속한다. 또한, 공유 변수에 대한 모든 접근을 유지해야 하며, 재 수행시에 결정적 수행을 보장하기 위한 기억 장소의 부담이 크다는 단점이 있다. [10]에서 비내포 병렬 프로그램을 대상으로 수행중 최초경합 탐지 기법이 제시되었으며, 접근 역사에서 발생하는 병목현상을 해결하였다. [7]에서는 내포된 프로그램을 대상으로 하지만 실질적인 알고리즘이 제시되지 않았다. 본 논문에서 제시된 기법은 단일 방향 내포 프로그램에서 최초경합 탐지를 보장하는 최초의 기법이다. 단일 수행에서 감시 비용은 기억 장소의 부담과 수행 시간의 부담에 의존적이다. 기억 장소의 부담을 나타내는 공간 복잡도(complexity)는 공유 변수의 개수를 나타내는 $V$, 프로그램의 최대 병렬성을 나타내는 $T$에 의해 결정된다. 본 기법에서는 NR-Labeling을 사용하는 것으로 가정하므로 레이블이 차지하는 공간은 $O(1)$이다. 접근 역사는 모든 공유 변수에 대해 독립적으로 유지되며 최악의 경우 최대 병렬성 만큼의 레이블이 유지된다. 또한, 이전에 제시된 기법[10]에 비해 상수 크기의 새로운 접근 역사 AH_ROOT(X)가 추가되었다. 그러므로, 공간 복잡도는 최악의 경우에 $O(VT)$이다. 수행 시간에 영향을 미치는 시간 복잡도는 공유 변수에 대한 접근이 발생될 때 현재의 접근과 접근 역사에 저장된 접근들과의 비교 횟수에 의해 결정된다. 따라서 시간 복잡도는 최악의 경우에 프로그램의 최대 병렬성인 $O(T)$이다. 그러므로, 기존의 수행중 기법[10]과 동일한 감시 비용을 갖는다. 반복 수행의 횟수는 최악의 경우에도 내포 깊이 만큼으로 제한된다. 일반적인 병렬 프로그램의 내포 깊이는 기껏해야 3 혹은 4이므로 프로그램의 수행에 대한 부담은 무시할 수 있다.

## 5. 결 론

공유 변수를 가지는 병렬 프로그램의 오류 수정에서 경합 탐지는 중요하다. 탐지된 경합이 수정되면 프로그램의 결정적인 수행이 보장되며, 이후 기존의 순환적인 디버거를 사용하여 프로그램의 논리적 오류를 수정할 수 있기 때문이다. 기존의 수행중 경합 탐지 기법은 내포된 병렬 프로그램에서 최초경합 탐지를 보장할 수 없다. 최초경합을 수정하면 이후에 발생하는 경합들이 나타나지 않을 수 있으므로, 최초경합의 탐지는 중요하다.

본 논문에서 제시된 기법은 단일 방향 내포 병렬 프로그램을 대상으로 수행 중에 최초경합 탐지를 보장하는 최초의 기법이다. 본 기법은 이전에 제시된 비내포 병렬 프로그램을 대상으로 하는 기법과 동일한 감시 비용을 가지며 현실적인 재 수행 횟수만 요구된다. 따라서, 본 기법은 실제 디버깅의 대상이 되는 경합만을 탐지하는 효과적이고 실용적인 수행중 탐지 기법이다. 향후 연구 과제는 좀 더 일반적인 형태인 다중 방향 내포 및 동기화가 있는 프로그램에도 적용될 수 있도록 프로토콜을 확장하는 것이며, 현재 개발 중인 경합 탐지 디버거인 Race Stand[9]에 적용하는 것이다.

## 참 고 문 헌

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

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

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

[4] Emrath, P. A., S. Ghosh, and D. A. Padua, "Detecting Nondeterminacy in Parallel Programs," Software, 9(1) :

pp.69-77, IEEE, Jan. 1992.

[5] Emrath, P. A., and D. A. Padua, "Automatic Detection of Nondeterminacy in Parallel Programs," 1st Workshop on Parallel Distributed Debugging, pp.89-99, ACM, May, 1988.

[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] Jun, Y., and C. E., McDowell, "On-the-fly Detection of First Races in Programs with Nested Parallelism," 2nd Int'l Conf. on Parallel and Distributed Processing Techniques and Applications pp.1549-1560, CSREA, Sunnyvale, California, August, 1996.

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

[9] Kim, D., and Y. Jun, "An Effective Tool for Debugging Races in Parallel Programs," 3rd Int'l Conf. on Parallel and Distributed Processing Techniques and Applications, pp. 117-126, CSREA, Las Vegas, Nevada, July, 1997.

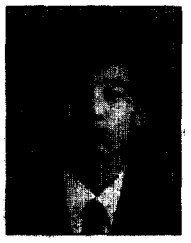
[10] Kim, J., and Y. Jun, "Scalable On-the-fly Detection of the First Races in Parallel Programs," 12nd Intl. Conf. on Supercomputing, pp.345-352, ACM, Melbourne, Australia, July, 1998.

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

[12] McDowell, C. E., and D. B. Helmbold, "Debugging Concurrent Programs," Computing Surveys, 21(4) : pp.593-622, ACM, Dec. 1989.

[13] OpenMP Architecture Review Board, OpenMP Fortran Application Program Interface, version 2.0, Jun. 2000.

[14] Ramanujam, J., and A. Mathrew, "Analysis of Event Synchronization in Parallel Programs," Languages and Compilers for Parallel Computing, Eds. K. Pingali, et. al., Lecture Notes in Computer Science(LNCS), 892 : pp.300-315, Springer-Verlag, 1995.



**천 병 규**

e-mail : chon@koreaaero.com

1993년 경상대학교 컴퓨터과학과 졸업 (학사)

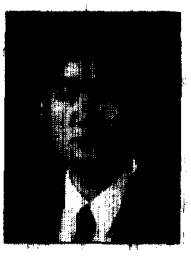
1995년 경상대학교 컴퓨터과학과 졸업 (석사)

1995년 POSDATA(주)

1996년~1999년 대우중공업(주) 우주항공연구소

1999년~현재 한국항공우주산업(주)

관심분야 : 병렬디버깅, 운영체제, Network



**우 종 정**

e-mail : jwoo@cs.sungshin.ac.kr

1982년 경북대학교 컴퓨터공학과 졸업 (학사)

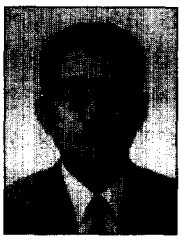
1982~1988년 산업연구원 책임연구원

1988~1993년 Univ. of Texas at Austin 전기컴퓨터공학과 졸업 석사 및 박사

1998~1999년 Univ. of Texas at Austin 전기컴퓨터공학과 방문교수

1993년~현재 성신여자대학교 컴퓨터정보학부 교수.

관심분야 : 컴퓨터구조, 멀티미디어시스템, 전자상거래, 원격교육, CAD



**전 용 기**

e-mail : jun@nongae.gsnu.ac.kr

1980년 경북대학교 컴퓨터공학과 졸업 (학사)

1982년 서울대학교 컴퓨터공학부 졸업 (석사)

1993년 서울대학교 컴퓨터공학부 졸업 (박사)

1982년~1985년 한국전자통신연구소 연구원

1985년~현재 경상대학교 컴퓨터과학과 교수, 컴퓨터·정보통신연구소 연구원

관심분야 : 운영체제, 분산 및 병렬처리, 프로그래밍 환경