공유 메모리 병렬 프로그램의 수행중 오류 탐지를 위한 루프 분리

송태섭*

Loop Splitting for On-the-fly Race Detection of Sharded-memory Parallel Programs

Tae-seob Song*

요 약

병렬 프로그램은 의도되지 않은 비결정적인 수행을 야기하므로 공유 메모리를 사용하는 병렬 프로그램에서는 경합을 탐지하는 것은 매우 중요하다. 수행 중 기법에서 경합을 탐지하기 위해서 요구되는 기억장소의 부담은 매우 크다. 특히 동기화가 있는 병렬 프로그램에서 경합 탐지에 필요한 기억 공간의 문제는 더욱 심각하다. 그래서, 본 논문에서는 원시 프로그램의 시멘틱을 유지하면서 동기화를 가지는 공유 메모리 병렬 프로그램의 디버깅을 위한 루프 분리 기법을 제시한다. 이것은 동기화를 가지는 병렬 프로그램의 수행 중 경합 탐지에 필요로 하는 기억공간의 복잡성을 줄일 수 있고, 루프 분리된 프로그램을 수행 중에 감시하여 최초 경합들을 탐지할 수 있다.

ABSTRACT

Detecting races is important for debugging shared-memory parallel programs, because the races result in unintended non-deterministic executions of the programs. Previous on-the-fly techniques to detect races in parallel programs with general inter-thread coordination show serious space overhead which depends on the maximum parallelism of the program. Therefore, this paper presents a loop splitting technique for on-the-fly race detection of parallel programs which is more efficient in space complexity than previous techniques. This loop splitting technique is the debugged program which preserves semantics of the original program. Monitering loop splitting program in on-the-fly can detect first races.

키워드

경합 탐지, 디버깅, 루프 분리, 최초 경합

Key word

on-the-fly, race detection, debugging, loop splitting, first race

* 정회원 : 한국국제대학교 조선해양공학과 교수
(교신저자, tss0928@daum.net)

접수일자 : 2011. 11. 15
심사완료일자 : 2011. 12. 12

Open Access http://dx.doi.org/10.6109/jkiice.2012.16.3.391

Ⅰ. 서 론

병렬 프로그램(parallel program)은 순차적 프로그램에 비해 내재된 특성 때문에 오류를 수정하기가 어렵다. 특히, 공유 메모리(shared memory)를 사용하는 병렬 프로그램은 수행 중에 자료경합(data race) 또는 경합(race)이라는 오류를 발생시킨다. 경합은 병행하게 수행되는 스레드(thread)들 간에 실행하는 명령 중에 동일 공유변수에 적어도 하나의 쓰기 접근을 가지고, 적절한 동기화 없이 접근할 때 발생한다. 프로그램의 비결정적(nonderministic) 수행을 야기하는 경합을 탐지하는 것은 병렬 프로그램의 디버깅에서 매우 중요하다.

병렬 프로그램에서 경합을 탐지하기 위한 기법들 중에 모든 탐지 작업을 수행 중에 하는 수행중 분석(on-the-fly analysis) 기법[3, 5, 6, 9, 11]은 다른 분석 기법만큼 많은 경합을 탐지하지는 못하지만 경합이 존재한다면 각 공유변수에 대해 적어도 하나는 탐지를 보장하는 장점이 있으며, 순서적 동기화(ordered synchronization)가 있는 병렬 프로그램에서는 최초 경합을 탐지[9]할 수 있다.

수행중 경합 탐지 기법은 공유변수에 접근하는 스레드들 간의 논리적 병행 여부를 결정하기 위해서, 원시 프로그램에는 경합 탐지에 필요한 부가적인 코드(instrumental code)를 추가한다. 그리고 실행 중에는 공유변수에 접근하는 각 스레드들을 감시하고, 공유변수에 대한 스레드의 접근역사(access history)를 유지하게 된다. 이때 필요한 정보를 생성하고 저장하기 위해 필요로 하는 공간은 최대 병렬성에 의존적으로 증가한다. 더구나 동기화가 있는 공유 메모리 병렬프로그램인 경우는 동기화 정보까지 유지해야 하므로 기억 공간 문제는 더욱 심각해진다. 그래서 이전까지의 수행중 분석 기법에서 경합을 탐지할 때 기억 공간을 줄일 수 있는 방안으로 순차적 감시 기법이 제안되었다. 그러나, 순차적 감시 기법은 적용대상이 제한적이라 동기화가 있는 공유 메모리 병렬프로그램에 대해서는 해결 방안이 되지 못한다. 일반적으로 병렬 프로그램에서 동기화 명령은 빈번한 것이므로, 동기화가 있는 프로그램의 경합 탐지에서도 기억 공간의 부담을 줄일 수 있는 방안이 요구된다.

본 논문에서는 동기화가 존재하는 공유 메모리 병렬 프로그램에서 수행중 경합 탐지를 위한 루프 분리 기법을 보인다. 2절에서는 동기화를 가진 공유 메모리 병렬 프로그램의 수행을 POEG와 함께 설명하고, 이러한 병렬 프로그램을 순차적으로 감시할 때의 문제점에 대해 언급한다. 3절에서는 루프 분리의 기본 원리와 알고리즘을 설명하고, 실험 결과를 통해 루프 분리 이전과 이후의 순차적 감시가 같음을 보인다. 마지막으로 4절에서 본 논문의 결론을 내린다.

Ⅱ. 병렬 프로그램 모델

동기화를 가지는 공유 메모리 병렬 프로그램의 수행에서, 스레드들은 PARALLEL DO 와 END PARALLEL DO 문에 의해 생성 및 종료된다. 하나 이상의 스레드들이 PARALLEL DO 문에서 생성되고, END PARALLEL DO 문에 의해 합류된다. 동일한 생성명령(fork operation)에 의해 생성된 스레드들은 대응하는 합류명령(join operation)이 수행될 때까지 병렬로 수행한다. 이러한 생성과 합류 명령을 스레드 명령이라 한다. 그리고, POST와 WAIT 문은 사건 동기화(event synchronization)를 나타낸다. POST문은 사건의 발생을 나타내고, WAIT 문에 의해 동기화된다.

병렬 프로그램[8]은 병렬 루프(parallel loop)와 병렬 영역(parallel section)[2, 10] 등의 병렬 형태를 가지고 있는데, 본 논문에서 고려하는 병렬 프로그램은 사건 동기화가 있는 병렬 루프이고, 하나의 사건 변수(event variable)에 의한 동기화로 가정한다. 이때 사건 변수는 두 가지 상태(clear, posted)를 가지며, 초기 값은 항상 clear이고 POST에 의해 posted 상태가 된다. posted 상태가 된 값은 WAIT에 의해 비교되어 지는데 WAIT는 사건 변수 값이 POST를 통해 posted 상태가 될 때까지 해당 스레드의 실행을 지연하게 된다. 반면에 POST는 수행후 즉시 다음 실행을 계속한다.

병렬 프로그램의 수행은 POEG(Partial Order Execution Graph)이라 불리는 방향성 비순환 그래프에 의해 나타낼 수 있다. POEG에서의 정점 (vertex)은 스레드 생성 및 종료와 동기화 명령들을 나타낸다. POST를 포함할 수는 있으나 WAIT는 포함되지 않고, 구조화된 블럭 안으로 또는 밖으로 제어 이동이 없는 블럭을 정적블럭(static block)이라 한다. 한편 스레드가 수행하는 정적블럭의 문장들의 연속은 동적블럭(dynamic block)이라 한다.

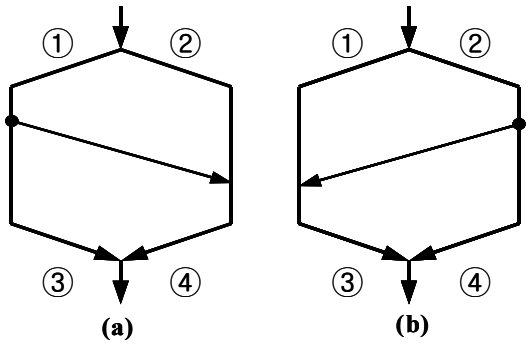

그림 1. 방향성 비순환 그래프
Fig. 1 Partial Order Execution Graph

POEG에서 정점과 정점을 연결하는 각 간선(arc)은 수행 중인 스레드의 동적블럭을 나타내며, 각 스레드들 사이의 간선은 동기화 명령의 시작과 끝을 나타낸다.

POEG은 발생후(happened-before) 관계를 나타내기 때문에 병렬 루프의 수행에서 발생되는 스레드들 사이의 부분순서(partial order)를 표현한다. POEG에서 스레드 T_i 발생 후에 스레드 T_j가 발생되면 발생 후 관계에 있다고 하며, $T_i \rightarrow T_j$로 나타낸다. $T_i \rightarrow T_k \wedge T_k \rightarrow T_j$를 만족하는 T_k가 존재하지 않는다면, T_i는 T_j의 부모 스레드(parent thread)라고 한다.

예를 들어, Fig.1-(a)에서 ① → ② 관계가 성립하고, 이것은 ①은 반드시 ② 보다 먼저 실행되며 ②의 부모 스레드임을 나타낸다. 동기화 명령도 발생후 관계를 유지한다. 즉, Fig. 1-(a)에서 간선이 없는 ①, ③은 발생후 관계가 없으나, 동기화로 연결된 ①, ④는 ① → ④ 관계를 만족한다.

Fig. 1-(a)와 같은 동기화가 있는 POEG에서 순차적 감시를 하게 되면 그 순서는 ① -> ② -> ③ -> ④ 이다. 다시 말해 블록 ①을 감시하고 연속해서 ② ③ ④를 순차적으로 감시하더라도 동기화로 생기는 순서가 그대로 유지되므로 논리적인 병행성을 체크하여 경합을 탐지해 낼 수 있다. 그러나, Fig. 1-(b)와 같은 형태의 동기화에 대해서는 올바른 탐지가 되지 못한다. 왜냐하면, Fig. 1-(b) POEG를 순차적 감시 기법에 따라 ①을 수행한 후 WAIT 문에 의해 ②를 수행하지 못하므로 교착상태(deadlock)가 발생된다. 그러나 동기화를 가지는 공유 메모리 병렬 프로그램에서도 기존의 순차적 감시로 경합을 탐지할 수 있다면, 공간 복잡도를 현저히 감소시킬 수 있다. 예를 들어 감시 대상이 되는 공유변수의 수를 V라 하고, 최대 병렬성을 T라 할 때 TR 기법[2]의 경우 최악의 공간 복잡도는 $O(VT + T^2)$이다. 그러나 순차적인 감시를 하게 되면 최악의 공간 복잡도는 O(VT)가 된다. 경합을 탐지하는 기존의 수행중 기법은 사건 동기화를 가지는 공유 메모리 병렬 프로그램에서의 최초 경합 탐지를 보장하지 못한다. 그러나 루프 분리를 하면 순서적 동기화에서 최초 경합을 탐지하는 알고리즘[9]을 사용할 수 있어 최초 경합을 탐지해 낼 수 있다. 디버깅에 있어 최초 경합의 의미는 중요하다. 왜냐하면, 최초 경합의 제거에 의해 다른 여러 경합들도 자동적으로 제거 될 수도 있기 때문이다.

Ⅲ. 알고리즘

루프 분리란 동기화가 있는 하나의 병렬 루프를 프로그램의 시멘틱(semantics)을 그대로 유지하면서 순차적 감시가 가능하도록 각 사건 별로 순서적인 루프로 만드는 것을 말한다. 이때 루프 분리된 순서적인 루프들은 감시 코드가 부가된 원시 병렬 프로그램의 WAIT를 기준으로 before-wait loop 과 after-wait loop으로 구분된다. before-wait loop은 WAIT이전의 정적블럭으로 구성되고 after-wait loop은 WAIT이후의 정적블록으로 구성된다. 예를 들어, Fig. 2-(a)는 동기화를 가지고 있는 병렬 프로그램이고, (b) 와 (c)는 병렬 프로그램인 (a)를 루프 분리한 형태로 (b)는 before-wait loop 이고 (c)는 after-wait loop 이다.

먼저 before-wait loop을 구성하는 과정을 살펴보면, 병렬 프로그램의 WAIT는 GOTO로 바뀌며 이때의 GOTO 분기는 루프가 마치는 곳으로 이동이 된다. 이렇게 해서 before-wait loop은 wait 수행 전의 명령들만을 수행한다. 그리고 Fig. 2-(c)에서 보여주고 있는 after-wait loop은 waited라는 변수 값을 세팅해서 WAIT 이후의 정적블럭만을 수행하도록 구성된다. 본 논문이 기반으로 하는 루프 분리 기법은 Fig. 2와 같은 한 개의 WAIT 뿐만 아니라 여러 개의 WAIT 경우에도 동일하게 적용되며, 이러한 경우에도 병렬 프로그램의 시멘틱은 그대로 유지된다.

동기화를 가진 병렬 프로그램을 루프 분리하기 위해서 두 개의 순서적인 루프로 만드는 기준은 if절 안의

WAIT이다. Fig. 2는 가장 기본적인 WAIT 형태를 보여주고 있다. 예에서 if절 안에 오직 WAIT만이 있을 경우는 위에서 설명한 대로 구성하면 되지만, WAIT 이외의 정적블럭이 있을 경우는 루프 분리하기 이전에 그러한 if절을 처리하는 단계가 필요하다. 즉 WAIT을 갖고 있는 if절이 Fig. 2와 같이 단순하지 않고, 정적블럭을 포함하는 경우, 루프 분리 이전에 WAIT가 있는 if문을 Fig. 2에서와 같은 기본 형태로 단순화시키는 과정이 필요하다.

다음은 Fig. 2와 같이 루프 분리하는 알고리즘을 보여준다.

```
 < before-wait loop algorithm >
do while(!EOF)
 {
   Read a Line from Original Program
   GetToken(token) /* token is a word */
   if ( token == "WAIT")
   WAIT statement GOTO statement transform.
   Line is stored in the sentence was  transformed.
   end if
   Write the Line in Before-wait
      /* Before-wait is a temporary file */
 }

 < after-wait loop construction algorithm >
do while( ! EOF)
 {
   Read a Line from Original Program
   GetToken(token) /* token is a word */
   if (first static block )
      goto First
   end if
   Write the Line in After-wait
   if (token == "WAIT")
     In the previous corresponding quarter of the

   WAIT statement
      inserts continue statement
       Write " waited = .true. "
       Write " if .not. waited then goto "
   end if
First :
  continue
 }
```

```
   parallel do I = 1, 3
     B0
     if C0  then wait e
     B1
   end parallel do
```

(a)

```
   parallel do I = 1, 3
     B0
     if C0  then goto 100
     B1
100  continue
   end parallel do
```

(b)

```
   parallel do I = 1, 3
     if C0 then
      wait e                          endif
     if .not. waited then goto 10
       B1
10 continue
   end parallel do
```

그림 2. 루프 분리 전후 병렬 프로그램
(a) 분리되기 전의 루프 (b) wait문 이전의 루프
(c) wait문 이후의 루프
Fig. 2 Original Parallel Loop
(a) original parallel loop (b) before-wait loop
(c) after-wait loop

[정의 3.1] 두 개의 동적블럭 B_i와 B_j가 주어지고, B_i의 마지막 수행문이 B_j의 첫 번째 수행문의 시작 전에 종료된다면 시간적 순서(temporally ordered) 관계에 있다고 하고, $(B_i \Rightarrow B_j)$로 나타낸다. $(B_i \Rightarrow B_j \vee B_j \Rightarrow B_i)$가 만족되지 않으면 B_i와 B_j는 시간적 병

행(temporally concurrent) 관계에 있다고 하고, $(B_i \blacksquare B_j)$ 로 나타낸다.

[보조정리 3.1] 두 개의 동적블럭 B_i와 B_j가 주어졌을 때, $(B_i \rightarrow B_j)$ 이면, $(B_i \Rightarrow B_j)$ 이다. 그리고 $(B_i \parallel B_j)$ 이면, $(B_i \Rightarrow B_j) \vee (B_j \Rightarrow B_i) \vee (B_i \blacksquare B_j)$ 이다.

[보조정리 3.2] 두 개의 동적블럭 B_i와 B_j가 주어졌을 때, $(B_i \Rightarrow B_j)$ 이면, $(B_i \rightarrow B_j) \vee (B_i \parallel B_j)$ 이다. 그리고, $(B_i \blacksquare B_j)$ 이면, $(B_i \parallel B_j)$ 이다.

(보조정리 3.1)과 (보조정리 3.2)에 의해 $(B_i \rightarrow B_j) \vee (B_i \parallel B_j)$ 이면, $\{(B_i \Rightarrow B_j) \wedge \neg(B_i \parallel B_j)\} \vee [\{(B_i \Rightarrow B_j) \vee (B_j \Rightarrow B_i) \vee (B_i \blacksquare B_j)\} \wedge \neg(B_i \rightarrow B_j)] \vee (B_i \Rightarrow B_j)$ 이므로, 두 동적블럭의 모든 수행 경우를 $(B_i \Rightarrow B_j)$ 으로 대응시킬 수 있다.

[정의 3.2] 대기조건(wait condition) C_i는 i 번째 wait 문이 수행되기 위한 조건값이다. 그리고 $\beta(C_i)$는 대기조건 C_i가 만족될 때 해당되는 wait 문으로부터 수행을 시작하는 동적블럭을 나타낸다.

[정의 3.3] 설치된 루프(instrumented loop) L_I는 수행중 경합 탐지를 위한 레이블링 함수들을 포함하는 부가적인 코드가 설치된 병렬 루프이다. L_I가 m 개의 쓰레드를 생성하고, n 개의 동적블럭을 포함하는 임의의 쓰레드 I_I^i 내의 $(j+1)$ 번째 동적블럭을 B_{ij}라고 할 때,

$$I_I^i = \{ B_{ij}, B_{st} \mid B_{ij} = \beta\left(\bigwedge_{k=1,j} \neg C_k\right), B_{i(j-1)} \rightarrow, B_{ij}$$
$$B_{st} = \beta\left(\bigvee_{r=1,j} C_r\right), B_{s(t-1)} \rightarrow B_{st}, B_{ij} \rightarrow B_{st}, 1 \le i, s \le m, 1 \le j, t \le n \}$$
$$L_I = \{ I_I^i \mid 1 \le i \le m \}$$

[정의 3.4] 루프 분리 (loop splitting) L_R은 순차적 수행을 위해서 L_I를 루프 분리한 루프로서, 대기전 루프(before-wait loop) L_{RB}와 대기후 루프(after-wait loop) L_{RA}로 구성된다. L_{RB}는 L_I를 구성하는 각 쓰레드에서 첫 번째 wait 문 이전에 수행되는 동적블럭들로 구성된 루프이고, L_{RA}는 각 쓰레드의 첫 번째 wait 문과 그 이후에 수행되는 동적블럭들로 구성된 루프이다. I_{RB}^i는 L_{RB}내의 i번째 쓰레드에서 수행되는 동적블럭들의 집합이고, I_{RA}^i는 L_{RA} 내의 i번째 쓰레드에서 수행되는 동적블럭들의 집합이라 하자. 그리고, I_{RB}^i를 구성하는 동적블럭의 수를 p 개라 하고, I_{RA}^i를 구성하는 동적블럭의 수를 q 개라 하면, I_I^i를 구성하는 동적블럭의 수 $n = (p+q)$ 이다. 그러면,

$$I_{RB}^i = \{ B_{i0}, B_{ij} \mid B_{ij} = \beta\left(\bigwedge_{k=1,j} \neg C_k\right), B_{i0} \Rightarrow B_{i1}, B_{ij} \Rightarrow B_{i(j+1)}, 1 \le i \le m, 1 \le j \le (p-1) \}$$
$$I_{RA}^i = \{ B_{ij} \mid B_{ij} = \beta\left(\bigvee_{r=1,j} C_r\right), B_{ij} \Rightarrow B_{i(j+1)}, 1 \le i \le m, p \le j \le (n-1) \}$$
$$L_{RB} = \{ I_{RB}^i \mid 1 \le i \le m \}$$
$$L_{RA} = \{ I_{RA}^i \mid 1 \le i \le m \}$$

[정리 3.1] $(L_R \Rightarrow L_I)$ L_R의 순차적 수행에서 두 동적블럭 B_i'와 B_j'가 존재하면 $(i < j)$, L_I의 수행에서 B_i'와 B_j'에 대응하는 두 동적블럭 B_i와 B_j가 존재한다.

[증명] 우리는 여기에서 B_i'와 B_j'가 A) 같은 쓰레드에서 수행되는 경우와, B) 다른 쓰레드에서 수행되는 경우 등 두 가지 경우로 나누어서 증명한다.

A의 경우) B_i'와 B_j'이 수행되는 쓰레드를 I_R^x라 하고, $(1 \le x \le m)$을 만족한다고 하자. B_i'와 B_j'의 수행은 (정의 3.4)에 의해서 3 가지 경우 이외에는 존재하지 않는다. 첫째는 B_i'와 B_j'가 I_{RB}^x에서 수행되는 경우이다. 둘째는 B_i'와 B_j'가 L_{RA}^x에서 수행되는 경

우이다. 셋째는 B_i' 는 I_{RB}^x에서 수행되고, B_j' 는 L_{RA}^x에서 수행되는 경우이다.

첫째, $\{B_i', B_j'\} \subset I_{RB}^x$인 경우로서, 첫 번째 wait 문 수행 이전의 블럭들이며, (정의 3.4)에 의해 각각 $(i+1)$ 와 $(j+1)$ 번째 수행되는 블럭들로 대기조건 $(\bigwedge_{k=1,i} \neg C_k)$와 $(\bigwedge_{k=1,j} \neg C_k)$을 만족하는 블럭들이다. 이는 $(i < j)$이므로, (정의 4.3)에 의해 I_I^x에서 B_i' 와 B_j' 에 대응되는 두 동적블럭 B_i와 B_j가 존재한다.

둘째, $\{B_i', B_j'\} \subset I_{RA}^x$인 경우로서, 각각 $(i+1)$ 번째와 $(j+1)$ 번째 수행되며, 대기조건 $(\bigvee_{r=1,i} C_r)$와 $(\bigvee_{r=1,j} C_r)$을 만족하는 블럭들이다. 그래서 B_i' 와 B_j' 사이에는 하나 이상의 wait 문이 존재한다. 이는 $(i < j)$이므로, (정의 4.3)에 의해 I_I^x에서 B_i' 와 B_j' 에 대응되는 두 동적블럭 B_i와 B_j가 존재한다.

셋째, $B_i' \in I_{RB}^x$인 경우로 대기조건 $(\bigwedge_{k=1,i} \neg C_k)$을 만족하고, $B_j' \in I_{RA}^x$인 경우로 대기조건 $(\bigvee_{r=1,j} C_r)$을 만족하는 경우이다. 그러면 B_i' 와 B_j' 사이에는 하나 이상의 wait 문이 존재한다. 그러므로, (정의 4.3)에 의해 B_i' 와 B_j' 에 대응되는 B_i와 B_j가 I_I^x에 존재한다.

B의 경우) B_i' 와 B_j' 이 수행되는 쓰레드를 각각 I_I^x와 I_I^y라 하고, $(1 \le x, y \le m)$을 만족한다고 하자. 여기에서 B_i' 와 B_j' 의 수행은 (A의 경우)와 마찬가지로 (정의 3.4)에 의해서 3 가지 이외에는 존재하지 않는다. 첫째는 B_i' 와 B_j' 가 L_{RB}에서 수행되는 경우이다. 둘째는 B_i' 와 B_j' 가 L_{RA}에서 수행되는 경우이다. 셋째는 B_i' 는 L_{RB}에서 수행되고, B_j' 는 L_{RA}에서 수행되는 경우이다.

첫째, $\{(B_i' \in I_{RB}^x) \wedge (B_j' \in I_{RB}^y)\}$인 경우로서, 첫 번째 wait 문 수행 이전의 블럭들이며, (정의 3.4)에 의해 각각 I_{RB}^x에서 $(i+1)$ 번째와 I_{RB}^y에서 $(j+1)$ 번째 수행되며 대기조건 $(\bigwedge_{k=1,i} \neg C_k)$와 $(\bigwedge_{k=1,j} \neg C_k)$을 만족하는 블럭들이다. 이는 (정의 4.3)에 의해 L_I에서 B_i' 와 B_j' 에 대응되는 두 동적블럭 B_i와 B_j가 존재한다.

둘째, $\{(B_i' \in I_{RA}^x) \wedge (B_j' \in I_{RA}^y)\}$인 경우로서, 첫 번째 wait 문 수행 이후의 블럭들이며, (정의 3.4)에 의해 각각 I_{RA}^x에서 $(i+1)$ 번째와 I_{RA}^y에서 $(j+1)$ 번째 수행되며 대기조건 $(\bigvee_{r=1,i} C_r)$와 $(\bigvee_{r=1,j} C_r)$을 만족하는 블럭들이다. 이는 (정의 4.3)에 의해 L_I에서 B_i' 와 B_j' 에 대응되는 두 동적블럭 B_i와 B_j가 존재한다.

셋째, $B_i' \in I_{RB}^x$인 경우로 대기조건 $(\bigwedge_{k=1,i} \neg C_k)$을 만족하고, $B_j' \in I_{RA}^y$인 경우로 대기조건 $(\bigvee_{r=1,j} C_r)$을 만족하는 경우이다. 그러면 B_i' 와 B_j' 사이에는 하나 이상의 wait 문이 존재한다. 그러므로, (정의 4.3)에 의해 B_i' 와 B_j' 에 대응되는 $B_i \in I_I^x$와 $B_j \in I_I^y$가 존재한다.

[정의 3.5] L_R의 순차적 수행에서 두 동적블럭 B_i' 와 B_j' 가 주어졌을 때, L_R에 대한 L_I의 대응된 병렬 수행(Corresponding Parallel Execution) E_C는 L_I의 수행 경우로서, E_C에서 동적블럭 B_i와 B_j가 존재한다면, 대응되는 B_i' 와 B_j' 가 L_R의 순차적 수행에서 존재한다.

[정리 3.2] $(L_I \Rightarrow L_R)$ L_R의 순차적 수행에서 두 동적블럭 B_i' 와 B_j' 가 존재하고, L_R에 대한 L_I의 대응된 병렬 수행 E_C에서 대응되는 동적블럭 B_i와 B_j가 주어졌을 때 $(i < j)$, $(B_i \rightarrow B_j)$ 이면 $(B_i' \Rightarrow B_j')$ 이다.

[증명] [정리 3.1]에서와 마찬가지로 B_i와 B_j가 A) 같은 쓰레드에서 수행되는 경우와 B) 다른 쓰레드에서 수행되는 경우 등 두 가지 경우로 나누어서 증명한다.

A의 경우) B_i와 B_j이 수행되는 쓰레드를 I_I^x라 하고, $(1 \le x \le m)$을 만족한다고 하자. $(B_i \rightarrow B_j)$를 만족하는 B_i와 B_j의 수행은 (정의 3.3)의 I_I^x에서 3 가지 경우 이외에는 존재하지 않는다. 첫째는 B_i와 B_j가 I_I^x에서 대기조건 $(\bigwedge_{k=1,i} \neg C_k)$과 $(\bigwedge_{k=1,j} \neg C_k)$을 만족하는 경우이다. 둘째는 B_i와 B_j가 I_I^x에서 대기조건 $(\bigvee_{r=1,i} C_r)$과 $(\bigvee_{r=1,j} C_r)$을 만족하는 경우이다. 셋째는 B_i는 I_I^x에서 대기조건 $(\bigwedge_{k=1,i} \neg C_k)$을 만족하고, B_j는 I_I^x에서 대기조건 $(\bigvee_{r=1,j} C_r)$을 만족하는 경우이다.

첫째, $\{B_i, B_j\} \subset I_I^x$에서 첫 번째 wait 문 수행 이전의 블럭들로서, (정의 3.3)에 의해 각각 $(i+1)$ 번째와 $(j+1)$ 번째 수행되며 대기조건 $(\bigwedge_{k=1,i} \neg C_k)$와 $(\bigwedge_{k=1,j} \neg C_k)$을 만족하는 블럭들이다. 이는 $(i < j)$이므로, (정의 4.4)에 의해 I_{RB}^x에서 B_i와 B_j에 대응되는 B_i'와 B_j'이 존재하고, $(B_i' \Rightarrow B_j')$가 만족된다.

둘째, $\{B_i, B_j\} \subset I_I^x$에서 첫 번째 wait 문 수행 이후의 블럭들로서, (정의 3.3)에 의해 각각 $(i+1)$ 번째와 $(j+1)$ 번째 수행되며 대기조건 $(\bigvee_{r=1,i} C_r)$와 $(\bigvee_{r=1,j} C_r)$을 만족하는 블럭들이다. 이는 $(i < j)$이므로, (정의 4.4)에 의해 I_{RA}^x에서 B_i와 B_j에 대응되는 B_i'와 B_j'이 존재하고, $(B_i' \Rightarrow B_j')$가 만족된다.

셋째, $B_i \in I_I^x$에서 첫 번째 wait 문 수행 이전의 블럭이고 $B_j \in I_I^x$에서 첫 번째 wait 문 수행 이후의 블럭으로서, (정의 3.3)에 의해 각각 $(i+1)$ 번째와 $(j+1)$ 번째 수행되며 B_i는 대기조건 $(\bigwedge_{k=1,i} \neg C_k)$을 만족하고, B_j는 대기조건 $(\bigvee_{r=1,j} C_r)$을 만족하는 블럭들이다. 이는 $(i < j)$이므로, (정의 4.4)에 의해 I_{RB}^x에서 B_i에 대응되는 B_i'와 I_{RA}^x에서 B_j에 대응되는 B_j'가 존재하고, $(B_i' \Rightarrow B_j')$가 만족된다.

B의 경우) B_i와 B_j가 수행되는 쓰레드를 각각 I_I^x와 I_I^y라 하고, $(1 \le x, y \le m)$을 만족한다고 하자. $(B_i \rightarrow B_j)$를 만족하는 B_i와 B_j의 수행은 I_I^x와 I_I^y에서 오직 한가지 경우밖에 존재하지 않는다. B_i는 I_I^x에서 대기조건 $(\bigwedge_{k=1,i} \neg C_k)$을 만족하고, B_j는 I_I^y에서 대기조건 $(\bigvee_{r=1,j} C_r)$을 만족하는 경우이다.

$B_i \in I_I^x$는 첫 번째 wait 문 수행 이전의 블럭이고, $B_j \in I_I^y$는 첫 번째 wait 문 수행 이후의 블럭으로서, (정의 3.3)에 의해 각각 I_I^x에서는 $(i+1)$ 번째와 I_I^y에서는 $(j+1)$ 번째 수행되며, 대기조건 $(\bigwedge_{k=1,i} \neg C_k)$와 $(\bigvee_{r=1,j} C_r)$을 만족하는 블럭들이다. 이는 (정의 4.4)에 의해 B_i에 대응되는 병렬 수행 B_i'는 I_{RB}^x에서 수행되고, B_j에 대응되는 병렬 수행 B_j'는 I_{RB}^y에서 수행된다. 그러므로 $(B_i' \Rightarrow B_j')$가 만족된다.

Ⅳ. 결 론

병렬 프로그램에서 경합을 탐지하는 것은 중요하나, 수행중 기법으로 경합을 탐지할 때 필요로 하는 기억 공간의 부담이 크다. 기존의 수행중 기법에는 이러한 기억 공간의 부담을 줄이고자 순차적 감시기법을 대안으로 삼고 있으나, 동기화가 있는 병렬 프로그램에는 사용하지 못한다.

본 논문에서는 동기화를 가지는 공유 메모리 병렬 프로그램을 순서적인 루프들로 루프 분리함으로써 동기화가 있는 공유 메모리 병렬 프로그램에서도 순차적인 감시가 가능하게 하였다. 그러므로 이 기법을 이용하면 수행중 경합탐지를 위한 공간 복잡도를 병렬성과 무관하게 유지할 수 있다. 또한 순서적 동기화에서 최초 경합을 탐지하는 알고리즘을 이용하여 최초 경합 탐지도 가능하다.

참고문헌

[1] Callahan, D. Kennedy K., Subhlok, J., "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] Dagum, L., and R. Menon, "OpenMP: An Industry-Standard API for Shared-Memory Programming," Computational Science and Engineering, Vol. 5(1), pp. 46-55, IEEE, Jan.-Mar. 1998.

[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, Mar. 1990.

[4] Grunwald, D., 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.

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

[6] 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, June 1997.

[7] Netzer, R. H. B., S. Ghosh, "Efficient Race Condition Detection for Shared- Memory Programs with Post/Wait Synchronization", Int'l. Conf. on Parallel Processing, pp. II-242-246, Penn. State Univ., Aug. 1992.

[8] Netzer, R. H. B., 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.

[9] Park, H., Y. Jun, "Detecting the First Races in Parallel Programs with Ordered Synchronization," 6th Intl. Conf. on Parallel and Distributed Systems, pp. 201-208, IEEE, Dec. 1998.

[10] Parallel Computing Forum, PCF Parallel Fortran Extensions, Vol. 10(3), ACM, July 1991.

[11] Kim, Y., and Y. Jun, "Restructuring Parallel Programs for On-the-fly Race Detection", 5th Int'l. Conf. Parallel Computing Technologies (PaCT-99), pp. 446-452, RAS, Sept. 1999.

저자소개

송태섭(Tae-seob Song)

1992년 3월 ~ 현재 한국국제대학교 조선해양공학과 교수
1999년 2월 경남대학교 대학원 컴퓨터공학과(공학박사)

※관심분야 : 컴퓨터시스템 및 정보보안, 멀티미디어 및 컴퓨터그래픽스