메시지전달 프로그램의 디버깅을 위한 경합의 확장적 시각화

(Scalable Race Visualization for Debugging Message-Passing Programs)

박 미 영 † 전 용 기 ††
(Mi-Young Park) (Yong-Kee Jun)

요 약 메시지전달 프로그램에서 발생하는 임의의 메시지경합은 다른 경합의 발생에 영향을 줄 수 있으므로, 효과적인 디버깅을 위해서 영향받지 않은 경합을 탐지하는 것이 중요하다. 이러한 경합을 효율적으로 탐지하기 위한 기존의 기법은 각 프로세스에서 가장 먼저 발생하는 경합의 수신사건에서 수행을 중단하여 경합하는 메시지들을 탐지한다. 그러나 프로세스의 수행 중단은 경합들간에 존재하는 영향관계의 단절을 초래하므로, 탐지된 모든 경합이 영향받지 않은 경합임을 보장하지 못한다. 본 논문은 기존의 두 번째 수행을 위한 알고리즘에 각 프로세스에서 가장 먼저 발생한 경합의 영향관계 정보를 생성하는 알고리즘을 추가하여, 탐지된 경합들간의 관계를 효과적으로 시각화하는 기법을 제안한다. 이러한 시각화는 각 프로세스에서 최초로 발생한 경합들간에 형성된 영향관계를 보임으로써 영향받지 않은 경합을 시각적으로 탐지하는데 효과적이다.

키워드 : 메시지 전달 프로그램, 메시지 경합, 디버깅, 영향받지 않은 경합, 시각화

Abstract Detecting unaffected race conditions is important for debugging message-passing programs effectively, because such races can influence other races to occur or not. The previous technique used in detecting unaffected races detects a race by halting the execution of a process at the receive event of the race that occurs first in the process. However this technique does not guarantee that all of the detected races are unaffected, because halting the execution of processes does disconnect some chains of affects-relations among those races. In this paper, we improved the second pass algorithm of the previous technique by producing information about affects-relations of the races that occur first in each process. Then we effectively visualize affect-relations among the races detected in each process. This visualization is effective in detecting visually unaffected races by simplifying affects-relations among the races which occur first in each process.

Key words : message-passing programs, message race, debugging, unaffected races, visualization

1. 서 론

비동기적 메시지전달(message-passing)[1-3] 프로그램에서 동일한 채널로 두 개 이상의 메시지들이 동일한 수신자에게 도착순서가 보장되지 않고 전송될 수 있을 때 메시지경합(message race)[4-6]은 발생한다. 경합하는 메시지들의 비결정적인 도착 순서는 프로그램의 의도되지 않은 비결정적인 수행 결과를 초래하므로, 효과적인 디버깅을 위해서 메시지경합은 반드시 탐지되어야 한다. 특히 부분 순서관계에서 먼저 발생한 경합이 존재하지 않은 경합인 영향받지 않은 경합은 영향받은 다른 경합들을 나타나게 하거나 숨어있게 하기 때문에 효율적으로 탐지하는 것이 중요하다.

메시지전달 프로그램에서 동적으로 경합을 탐지하는 기법은 탐지시점에 따라서 수행후(post-mortem) 탐지기법[7,8]과 수행중(on-the-fly) 탐지기법[9,6]으로 나눌 수 있다. 경합을 프로그램 수행 중에 탐지하는 기법은 수행후 탐지기법에 비해서 추적파일을 위한 기억공간의 부담이 없으나 존재하는 경합의 부분집합만을 탐지한다. 수행중 경합 탐지기법은 경합의 존재 여부를 검증

· 본 연구의 일부는 한국과학재단 목적기초연구(R05-2003-000-12345-0)의 지원으로 수행되었음
† 비 회 원 : 경상대학교 컴퓨터과학과
park@race.gsnu.ac.kr
†† 종신회원 : 경상대학교 컴퓨터과학과 교수
jun@gsnu.ac.kr
논문접수 : 2004년 2월 5일
심사완료 : 2005년 4월 14일

(verification)하는 기법[6,9]과 영향받지 않은(unaffected) 경합을 탐지하는 기법[10,4,5,11]으로 나눌 수 있다.

영향받지 않은 경합을 가장 효율적으로 탐지하기 위한 기존의 기법[11]은 프로그램의 첫 수행에서 각 프로세스에서 가장 먼저 발생하는 경합의 위치 정보를 수집하고, 두 번째 수행에서는 그 위치 정보를 이용하여 각 프로세서에서 가장 먼저 발생하는 경합의 수신사건에서 수행을 중단함으로써 경합하는 메시지들을 탐지한다. 그러나 경합이 발생된 위치에서 프로세스의 수행이 중단되면 이러한 사실을 반영하는 메시지를 다른 프로세스에게 송신할 수 없고, 이 메시지를 수신하지 못한 프로세스는 자신의 경합을 영향받지 않은 경합으로 잘못 보고할 수 있다. 그러므로 기존의 기법은 탐지된 모든 경합이 영향받지 않은 경합임을 보장하지 못한다.

본 논문은 프로그램 수행 중에 각 프로세스에서 가장 먼저 발생한 경합에 참여하는 사건정보와 경합탐지 정보를 추적파일에 적재하고, 이를 분석하여 탐지된 경합들과 그들간의 영향관계를 시각화하는 기법을 제안한다. 먼저 각 프로세스에서 최초로 발생한 경합을 탐지하기 위해서는 프로그램 수행 중에 현재 수신사건이 경합에 참여하는지 여부와 그 경합이 현재 프로세스에서 처음으로 발생된 것인지를 판단한다. 그리고 프로세스의 수행 종료 시에 수행 중에 발생한 사건 정보와 프로세스별로 탐지된 경합 및 영향관계 정보를 추적파일에 적재한다. 마지막으로 생성된 추적파일을 분석하여 각 프로세스별로 발생한 경합과 영향관계 정보를 이용하여 탐지된 경합들간의 영향관계를 시각화한다. 이러한 시각화는 각 프로세스에서 최초로 발생한 경합들간에 형성된 영향관계를 보임으로써 영향받지 않은 경합을 시각적으로 탐지하는데 효과적이다.

본 기법의 실험은 산업표준인 MPI(Message Passing Interface) [3]를 구현한 MPICH[12]를 네 개의 Intel 프로세서 노드로 구성된 클러스터 시스템에 설치하여 수행하였다. 그리고 본 기법은 사용자 프로그램과의 투명성을 높이기 위해서 MPI Profiling Interface를 이용한 C 라이브러리로 구현하였으며, 시각화 기법과 사용자 인터페이스는 Java 언어로 구현하였다. 본 기법을 예제 프로그램에 적용한 결과로써, 본 시각화 기법이 영향받지 않은 경합을 시각적으로 탐지하는데 효과적임을 확인하였다.

2절에서는 메시지전달 프로그램의 디버깅에서 영향받지 않은 메시지경합의 중요성을 소개하고, 이러한 경합을 탐지하기 위한 기존 기법들의 문제점을 살펴본다. 3절에서는 경합을 탐지하고 영향관계 정보를 생성하는 알고리즘을 소개하고 추적파일을 분석하여 경합들간의 영향관계를 시각화하는 기법을 소개한다. 마지막으로 4절에서는 본 연구의 중요성 및 향후 과제와 함께 결론을 내린다.

2. 연구 배경

본 절에서는 비동기적 메시지전달 프로그램에서 발생하는 메시지경합들간의 영향관계를 정의하여 영향받지 않은 경합의 개념과 중요성을 설명한다. 그리고 이러한 경합을 탐지하기 위한 기존 기법들을 소개하고, 가장 효율적인 기법이 가진 문제점을 살펴본다.

2.1 메시지경합

분산된 메모리를 가진 병렬컴퓨팅 환경에서 메시지의 전달로써 프로세스간의 통신을 수행하는 병렬프로그램을 메시지전달 프로그램[1-3]이라 한다. 이러한 프로그램에서 프로세스들간에 메시지가 전달되는 형태[1,7,8]는 동기적(synchronous)과 비동기적(asynchronous) 메시지전달로 구분된다. 동기적 메시지전달에서 발생된 송신사건은 대응되는 수신사건에 메시지가 전달된 것을 확인할 때까지 수행이 완료되지 않는다. 반면에 비동기적 메시지전달에서는 송신사건이 송신한 메시지의 수신여부를 확인하지 않고도 수행을 완료한다. 이러한 형태의 메시지전달은 동기적 메시지전달을 구현할 수 있으므로 보다 일반적이면서 높은 병렬성을 제공한다. 그러므로 본 논문은 비동기적 메시지전달 프로그램을 대상으로 한다.

본 논문의 대상 프로그램에서 프로세스간의 송수신은 논리적인 채널 상에서 이루어지며, 각 송수신사건은 메시지를 송신하거나 수신하는 논리적 채널들을 명시하는 것으로 간주한다. 이때 하나 이상의 채널에서 메시지의 수신이 가능하면 수신사건은 임의의 채널을 비결정적으로 선택하여 하나의 메시지를 수신하게 된다. 그리고 임의의 채널로 송신된 메시지는 반드시 하나의 수신사건에서 수신되며, 결과적으로 프로그램 수행 중에 송신된 모든 메시지들은 임의의 대응되는 수신사건에서 모두 수신된다고 가정한다. 이러한 모델은 대부분의 메시지전달 기법[1-3]을 표현할 수 있으므로 일반적인 모델이다.

메시지전달 프로그램의 수행은 수행 중에 발생하는 사건들의 집합과 그들 간의 순서관계(happened-before) [13]로 표현될 수 있다. 모든 수행에서 사건 a가 사건 b보다 항상 먼저 수행하면, 그들간의 순서관계는 (a 발생 후 b) 관계가 만족된다고 하고 $a \rightarrow b$로 표기한다. 예를 들어 하나의 프로세스 내에서 순서적으로 발생하는 두 사건 $\{a, b\}$이 존재하면 $(a \rightarrow b \vee b \rightarrow a)$가 만족된다. 다른 프로세스간에 메시지 전달을 위한 송수신 사건 $\{s, r\}$이 존재하면, 메시지 송신사건 s와 대응되는 수

신사건 r간에는 $s \rightarrow r$를 만족한다. 여기서 $\rightarrow$는 비반사적 추이적 폐쇄(irreflexive transitive closure) 관계이므로, 임의의 세 사건 $\{a, b, c\}$가 존재하고 $(a \rightarrow b \wedge b \rightarrow c)$이 만족되면 $a \rightarrow c$ 이 만족된다.

메시지들은 네트워크의 통신지연이나 프로세스 스케줄링에 의해서 도착하는 순서가 달라질 수 있다. 결정적 수행결과를 보이도록 의도된 프로그램[2,3]에서 메시지의 비결정적인 도착 순서는 예기치 않은 수행결과를 초래하거나 서로 다른 수행 결과를 보일 수 있으므로, 메시지의 경합 조건은 프로그램의 디버깅을 위해서 우선적으로 탐지되어야 한다. 메시지전달 프로그램에서 동일한 채널로 두 개 이상의 메시지들이 동일한 수신자에게 도착순서가 보장되지 않고 전송될 수 있을 때 메시지경합[4-6]이 발생된다. 메시지경합은 하나의 수신사건 r과 경합하는 메시지들의 집합 M으로 구성되므로 $<r, M>$으로 표기한다. 여기서 r은 임의의 수신사건이고, M은 서로 경합하는 메시지들의 집합으로서 r에서 수신 가능한 메시지들로 구성된다. 이때 r은 M에 속한 메시지들 중에서 가장 먼저 도착하는 메시지의 수신사건이며, M에 속한 임의 메시지의 송신사건 s는 $r \rightarrow s$를 만족하지 않는 송신사건이다. 그리고 송신사건 s가 송신한 메시지를 $msg(s)$로 나타낸다.

그림 1은 임의의 메시지전달 프로그램의 수행 중에 발생한 사건들간의 부분적 순서관계를 나타낸 것이다. 그림에서 세로 화살표는 시간적 흐름에 따른 프로세스의 수행을 나타내고, 프로세스간에 비스듬히 그려진 화살표는 메시지의 전송을 나타내며, 해당하는 송수신사건에는 레이블이 붙여져 있다. 그림에서 P_1와 P_3는 메시지 $msg(x_1)$와 $msg(x_2)$를 P_2으로 송신한다. 이때 두 송신사건 x_1과 x_2는 P_2의 수신사건 x에 대해서 $\{x \rightarrow x_1 \wedge x \rightarrow x_2\}$를 만족하지 않으므로, $msg(x_1)$과 $msg(x_2)$는 수신사건 x로 서로 경합한다. 따라서 P_2에 존재하는

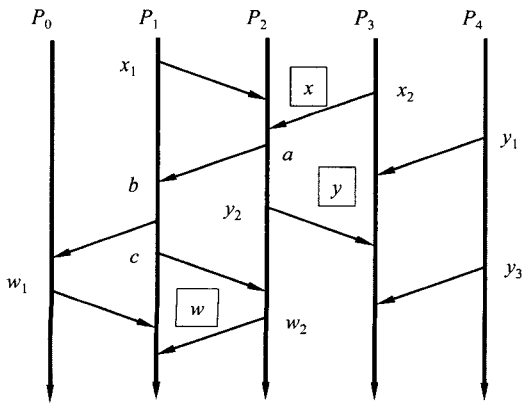

그림 1 메시지 경합

경합은 경합하는 메시지들의 집합 $X = \{msg(x_1), msg(x_2)\}$와 이들 메시지를 수신 가능한 첫 수신사건 x로 구성되므로 $<x, X>$로 나타낼 수 있다.

임의의 프로그램 수행에서 두 개의 메시지경합 $<m, M>$과 $<n, N>$만이 존재한다고 하자. 임의의 메시지 $msg(s) \in N$에 대해서 $(m \rightarrow s \vee m \rightarrow n)$가 만족되면, $msg(s)$를 $<m, M>$로부터 영향받은 메시지라 하고, $<n, N>$은 $<m, M>$으로부터 영향받은(affected) 경합이라 한다. 그리고 $n \rightarrow m$를 만족하지 않고 $n \rightarrow t$를 만족하는 메시지 $msg(t)$가 M에 존재하지 않으므로, $<m, M>$은 영향받지 않은 경합이다. 예를 들어 그림 1에 존재하는 세 개의 경합 $<w, W>$, $<x, X>$, $<y, Y>$들 중에서 $<w, W>$와 $<y, Y>$는 $<x, X>$로부터 영향받은 경합이다. 왜냐하면 $<y, Y>$에 대해서 $x \rightarrow y_2$가 만족되고, $<w, W>$에 대해서는 $x \rightarrow w$가 만족되기 때문이다. 영향받은 $<w, W>$와 $<y, Y>$의 발생은 $<x, X>$의 발생 결과에 의존적일 수 있으며, $<x, X>$을 디버깅하면 사라질 수도 있는 경합이다. 따라서 $<x, X>$와 같이 영향받지 않은 경합은 동일한 입력에 대한 모든 수행에서 나타나는 경합으로서 영향받은 다른 경합을 나타나게 하거나 숨어있게 할 수 있으므로 효과적인 디버깅을 위해서 반드시 탐지해야 한다.

임의의 프로세스별로 가장 먼저 발생하는 경합을 지역적 최초경합(locally-first race) 이라 한다. 이러한 지역적 최초경합은 한 프로세스 내에서 발생한 경합들 중에서는 영향을 받지 않은 경합이지만, 다른 프로세스에서 발생한 경합으로부터 영향을 받지 않은 경합임을 보장하지는 못한다. 그림 1에 나타난 모든 경합은 지역적 최초경합으로서, 두 개의 지역적 최초경합 $\{<w, W>, <y, Y>\}$는 다른 지역적 최초경합 $<x, X>$로부터 영향을 받은 경합이다

2.2 관련 연구

메시지전달 프로그램에서 동적으로 경합을 탐지하는 기법은 탐지시점에 따라서 수행후(post-mortem) 탐지기법[7,8]과 수행중(on-the-fly) 탐지기법[9,6]으로 나눌 수 있다. 수행후 탐지기법은 프로그램을 수행하면서 수행정보를 추적파일에 기록하고, 프로그램의 수행이 끝난 후에 추적파일을 분석하여 경합을 탐지한다. 이 기법은 한 번의 수행에서 발생한 모든 경합을 탐지하지만, 소규모의 병렬프로그램에서도 추적파일을 위한 기억공간의 요구가 비현실적으로 크다. 수행중 탐지기법은 프로그램의 수행을 감시하면서 수신사건을 수행할 때마다 경합의 발생여부를 검사한다. 이 기법은 감시되는 프로그램에 경합이 존재한다면 적어도 한 개의 경합은 반드시 탐지하며, 경합탐지를 위해 사용되는 기억공간을 수행중에 재사용하므로 대규모의 병렬프로그램에서도 적용

할 수 있는 현실적인 기법이다.

수행중 탐지기법은 탐지되는 경합의 성격에 따라서 경합의 존재 여부를 검증하는 기법[9,6]과 영향받지 않은 경합을 탐지하는 기법[10,4,5,11]으로 나눈다. 경합검증 기법은 매 수신사건마다 이전 수신사건과 현재 송신사건간의 병행성을 검사하여 경합을 탐지한다. 그러나 이러한 기법은 경합들간의 영향관계를 고려하지 않고 수행 중에 존재하는 모든 경합을 대상으로 하기 때문에 디버깅에는 효과적이지 않다. 영향받지 않은 경합을 탐지하는 기법은 매 수신사건마다 경합의 발생여부를 검사하고, 경합이 발생한 경우에는 해당 프로세스에서 가장 먼저 발생한 지역적 최초경합인지를 검사한다. 탐지된 경합은 그 프로세스내에서 발생된 경합으로부터는 영향받지 않은 경합임을 보장하므로 경합 디버깅에 효과적이다.

메시지전달 프로그램의 수행 중에 영향받지 않은 경합을 탐지하고자 하는 기존의 기법은 프로그램 감시작업의 병렬성에 따라 OtOt(One-thread-at-One-time) 기법[10,4]과 MtOt (Multi-threads-at-One-time) 기법[5,11]으로 구분된다. OtOt 기법은 한 번의 수행에서 하나의 프로세스만을 감시하여 지역적 최초경합만을 탐지하기 때문에, 적어도 프로세스 수 만큼을 반복해서 수행해야 한다. 그러나 MtOt 기법은 한 번의 수행으로 모든 프로세스를 감시하여 지역적 최초경합을 탐지하는 기법으로서, 필요한 수행 횟수에 따라서 1-Pass 기법[5]과 2-Pass 기법[11] 기법으로 분류된다. 1-Pass 기법은 매 수신사건마다 이전에 발생한 모든 수신사건을 검사하여 관련된 경합들을 모두 탐지하기 때문에, 메시지 수에 비례하는 복잡성을 보이므로 비현실적이다. 반면에 2-Pass 기법은 첫 수행에서 각 프로세스에서 채널별로 가장 먼저 발생하는 경합의 위치 정보를 탐지하고, 두 번째 수행에서는 그 위치 정보를 이용해서 최초경합이 발생한 수신사건에서 해당 프로세스의 수행을 중단하여 경합하는 메시지를 탐지한다. 이러한 2-Pass 기법은 메시지의 수와 무관한 시간 및 공간 복잡도를 가지므로, 1-Pass 기법에 비해서 효율적이다.

기존의 기법들 중에서 가장 효율적인 2-Pass 기법은 프로세스별로 지역적 최초경합을 탐지하지만, 탐지된 모든 경합이 영향받지 않은 경합임을 보장하지는 못한다. 왜냐하면, 경합이 발생된 위치에서 프로세스의 수행이 중단되면 이러한 사실을 반영하는 메시지를 다른 프로세스들에게 송신할 수 없으며, 이 메시지를 수신하지 못한 프로세스는 탐지된 자신의 경합을 영향받지 않은 경합으로 잘못 보고할 수 있기 때문이다. 예를 들어, 그림 1의 수행에 기존의 2-Pass 기법을 적용하자. 먼저 Pass-1 단계에서는 프로세스별로 지역적 최초경합이 발생하는 수신사건 $\{w \in P_1, x \in P_2, y \in P_3\}$를 찾기 위한 위치정보로서 $\{c \in P_1, \text{null} \in P_2, x_2 \in P_3\}$를 추적파일에 기록한다. 그리고 두 번째 수행에서는 $\{w, x, y\}$ 위치에서 해당 프로세스의 실행을 중단하고, 경합하는 메시지들을 수신버퍼에 저장한다. 이때 P_1의 수신버퍼는 P_2의 수행 중단으로 비어 있는 상태가 되고, P_2의 수신버퍼는 $X = \{msg(x_1), msg(x_2)\}$, P_3의 수신버퍼는 $Y = \{msg(y_1), msg(y_3)\}$ 등으로 구성된다. 결과적으로 2-Pass 기법은 $<x, X>$와 $<y, Y>$를 영향받지 않은 경합으로 보고한다. 그러나 그림 1에서 알 수 있듯이 $<y, Y>$는 $<x, X>$로부터 영향받은 경합이다. 이러한 잘못된 탐지는 P_2의 수행이 x 위치에서 중단됨으로써 $msg(y_2)$가 P_3로 송신되지 못하여 두 경합간의 영향관계가 반영되지 않았기 때문이다. 그러므로 기존의 2-Pass 기법은 탐지된 모든 경합이 영향받지 않은 경합임을 보장하지 못한다.

3. 메시지 경합의 시각화

본 절에서는 각 프로세스에서 수신되는 메시지들의 경합 여부와 메시지에 포함된 영향관계 정보를 검사하여 영향받지 않은 경합을 효과적으로 시각화하는 기법을 제안한다. 먼저 프로세스별로 발생하는 지역적 최초경합과 그들간의 영향관계 정보를 효율적으로 관리하는 알고리즘을 소개하고, 수행 후 추적파일의 영향관계 정보를 이용하여 영향받지 않은 경합을 효과적으로 시각화하는 예를 보인다.

3.1 지역적 최초경합의 탐지

본 기법은 지역적 최초경합을 효율적으로 탐지하기 위해서 매 수신사건에서 2-Pass 알고리즘을 수행한다. Pass-1에서는 기존의 기법과 같이 프로그램 수행 중에 현재 프로세스에서 발생한 지역적 최초경합의 위치정보를 탐지하고, 본 기법의 Pass-2에서는 Pass-1에서 탐

```
00: CenerateAffecters(send, recv, Msg)
01: for all i in Channels do
02:    if (cutoff → recv ∧ firstChan = i ∧ ¬affecting) then
03:          firstRecv := recv;
04:          affecting := true;
05:    endif
06: endfor
07: affecting := affecting ∨ Msg(affecting);
08: if (Msg(affecting) = true) then
09:    Affecter[send] = true;
10: endif
```

그림 2 Pass-2 알고리즘

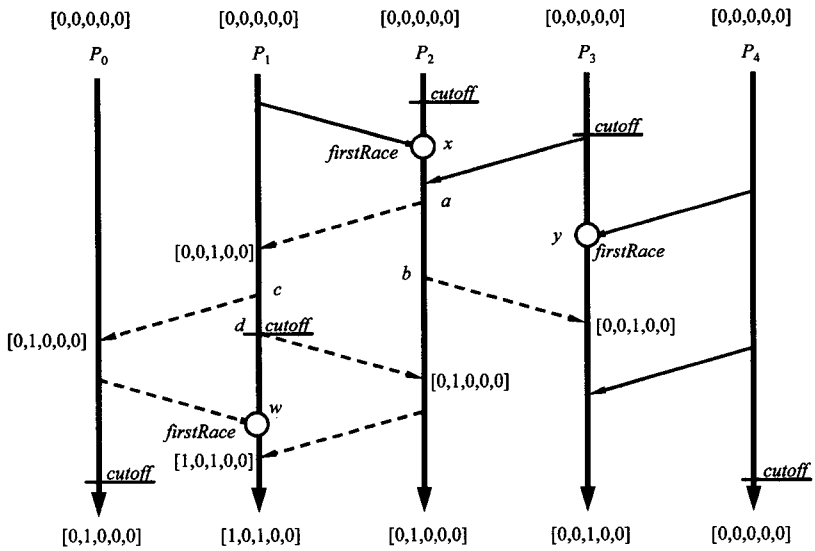

그림 3 지역적 최초경합과 Affecters

지된 위치정보를 이용하여 지역적 최초경합이 발생된 수신사건과 경합하는 메시지들을 탐지하고, 그들 간의 영향관계 정보를 추적파일에 적재한다. Pass-1에서는 각 수신사건에서 timestamp로 병행성을 검사하여 경합의 존재여부를 판단하고, 존재하는 경합에 대해서 가장 먼저 발생한 지역적 최초경합에 대한 위치정보를 *cutoff*, 해당 경합이 발생한 채널 정보는 *firstChan*에 저장한다.

지역적 최초경합을 탐지하고 영향관계 정보를 생성하는 Pass-2 알고리즘은 그림 3과 같다. 그림의 1-2행에서는 지역적 최초경합의 위치정보인 *cutoff*와 현재 수신사건의 위치간의 순서관계를 검사하고, 현재의 수신사건에 관련된 모든 채널 중에서 *firstChan*이 포함되어 있는지를 검사하고, 현재 수신사건의 영향받은 여부를 검사한다. 왜냐하면 현재의 수신사건이 *cutoff* 이후에 처음으로 발생한 영향받지 않은 사건이면서 *firstChan*에서도 메시지를 수신할 수 있는 수신사건이면, 현재 수신사건은 해당 프로세스에서 발생한 지역적 최초경합의 첫 수신사건이기 때문이다. 이러한 사실을 활용하기 위해서, 알고리즘의 3-4행에서는 현재 수신사건 정보를 *first-Recv*에 저장하고, *affecting* 값을 참으로 한다.

7행에서는 현재 프로세스에서 송신할 메시지가 다른 프로세스에서 발생한 경합에 미칠 영향관계 정보를 생성하기 위해서 현재의 *affecting*과 *Msg*(*affecting*)의 논리합을 수행한다. 왜냐하면 송신되는 메시지는 그 프로세스에서 경합이 발생하였거나 영향받은 메시지를 수신한 경우에는 다른 프로세스에게 영향을 주기 때문이다.

8행에서는 수신된 메시지에 첨부된 *affecting*을 검사하여 true이면 *Affecter*의 *send* 위치 값을 true로 설정한다. 왜냐하면 수신된 메시지에 첨부된 *affecting*이 true이면 현재의 수신자 프로세스가 *send* 프로세스로부터 영향받은 메시지를 받은 것이며, 이렇게 함으로써 수신자 프로세스는 자신에게 영향을 미친 프로세스들의 정보를 *Affecter*에 유지할 수 있기 때문이다.

본 기법을 그림 1에 적용했을 때 각 프로세스에서 발생한 지역적 최초경합과 각 프로세스의 *Affecter* 정보의 변화 과정을 보면 그림 3과 같다. 먼저 Pass-1 수행 결과로써 각 프로세스에서 발생한 지역적 최초경합의 위치정보인 *cutoff*을 탐지한다. 그리고 Pass-2에서는 각 프로세스 *Affecter*의 모든 항목의 값을 0으로 초기화하고, Pass-1의 *cutoff*이후에 발생한 최초의 수신사건에서 지역적 최초경합을 탐지한다. 그림 3에서 지역적 최초경합은 w, x 그리고 y이고 지역적 최초경합이후에 송신되는 메시지는 영향받은 메시지로서 점선으로 표시하였다. 그리고 각 프로세스의 *Affecter*는 영향받은 메시지를 수신할 때 마다 해당 메시지를 송신한 *send*의 위치 값을 1로 설정함으로써 *Affecter*값을 유지된다. 예를 들어 P_1에서는 영향받은 메시지 a를 수신하면서 P_1의 *Affecter*의 값은 [0,0,0,0,0]에서 [0,0,1,0,0]으로 바뀐다. 왜냐하면 영향받은 메시지를 P_2가 송신하였기 때문이다. 수행이 종료되었을 때 각 프로세스의 *Affecter*값을 살펴보면 P_0와 P_2는 P_1으로부터 영향을 받았고, P_1은 P_0와 P_2로부터, P_3는 P_2로부터 영향받은 메시지를 수신했음을 알 수 있다.

3.2 경합의 확장적 시각화

각 프로세스에서 최초로 발생하는 경합들을 탐지하는 본 기법은 프로그램 수행 중에 발생한 송수신 사건 정보와 경합 정보를 각각 사건 추적파일과 경합 추적파일에 적재한다. 이러한 두 가지 유형의 추적파일들은 각 프로세스별로 생성되고, 프로그램 수행이 종료된 후에 각각 하나의 추적파일로 통합된다. 먼저, 사건 추적파일은 프로그램 수행 중에 발생한 송신 및 수신 사건에 대한 정보를 가진 파일이다. 이러한 사건 추적파일을 생성하기 위해서 프로그램 수행 중에 모든 송수신 사건에 대한 현재 프로세스 번호, 사건발생에 대한 순서정보, 사건 유형, 송신자 프로세스 번호, 메시지 식별자, 논리적 전송채널 식별자, 병행성 정보인 벡터 타임스탬프[14, 15] 등의 정보를 저장한다. 경합 추적파일은 각 프로세스에서 발생한 지역적 최초경합에 대한 위치정보, *affecting*, *Affecter* 등을 가진다. 각 프로세스의 수행 종료 시에는 지역적 최초경합에 참여한 사건 정보를 사건 추적파일의 내용과 동일한 유형으로 *affecting*, *Affecter*와 함께 경합 추적파일에 적재한다.

추적파일을 분석하여 제공되는 시각화에서는 사건 뷰, 최초경합 뷰, 영향관계 뷰, 추상적 영향관계 뷰 등 네 가지의 시각적 뷰등 있다. 사건 추적파일를 이용한 사건 뷰에서는 사건 추적파일에 저장된 모든 송수신 사건간의 부분순서를 시각적으로 보여주는 뷰이고, 최초경합 뷰에서는 경합 추적파일을 이용하여 지역적 최초경합에 포함된 사건만을 시각적으로 보여준다. 영향관계 뷰에서는 지역적 최초경합들간의 영향관계를 영향받은 메시지들을 통해서 보이고, 추상적 영향관계 뷰에서는 프로세스별로 추상화시켜 영향관계를 보인다.

*Affecter*를 이용한 추상적 시각화는 그림 4와 같다. 그림 4(a)는 Pass-2 수행 후 각 프로세스의 *Affecter*를 보이고 있고, 그림 4(b)는 *Affecter*를 이용하여 프로세스별로 추상화한 영향관계를 보이고 있다. 그림 4(b)에서 경합이 발생한 프로세스는 색이 있는 정점으로 표시되었고, 각 프로세스간의 영향관계의 방향은 간선으로 표시되었다. 즉, 그림에서 P_1, P_2, P_3에서 지역적 최초경합이 발생하였고, P_3에서 발생한 경합은 P_2에서 발생한 경합으로부터 영향을 받고, P_1과 P_2는 서로 영향을 미치는 것을 알 수 있다. P_1과 P_2처럼 서로 영향을 미치는 관계는 하나로 추상화될 수 있고, 이것은 그림에서 사각형으로 표시되었다.

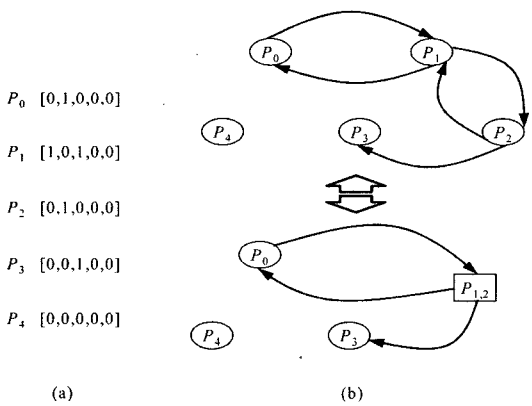

그림 4 Affecters를 이용한 영향관계

3.3 구현 및 실험

본 기법의 구현을 위한 환경으로는 네 개의 Intel 프로세서 노드로 구성된 클러스터 시스템을 구축하고, 각 노드에 커널 버전이 2.4.18인 Linux RedHat 8.0을 설치하였다. 그리고 메시지전달 프로그램의 수행 환경으로는 산업표준인 MPI(Message Passing Interface)[3]를 선택하고, 각 노드에 MPI를 구현한 MPICH[12]를 설치하였다. 본 기법은 사용자 프로그램을 수정하지 않고 투명하게 적용할 수 있도록 표준 C 언어와 MPI에서 제공하는 Profiling Interface를 이용하여 함수 단위로 구현하였고, 추적파일을 분석하고 시각화하는 부분은 Java 언어와 Soot API를 이용하여 구현하였다.

MPI Profiling Interface는 일반 사용자가 MPI의 함수 호출에 개입하여 별도의 작업을 수행하도록 허락하는 인터페이스로서, 모든 MPI 함수 호출이 다른 이름으로 호출 가능하도록 한 것이다. MPI_xxx 형태의 모든 함수 호출은 PMPI_xxx 함수 호출로 대체 가능하며, 이것을 사용해서 사용자는 그들 자신만의 MPI 수행환경을 실험할 수 있다. 본 연구에서는 추적파일 생성 모듈을 일대일 통신에 관련된 MPI 함수와 대응되는 PMPI 함수 내에 추가함으로써, MPI 함수 수행 시에 사건 추적파일 및 경합 추적파일을 생성하는 함수가 수행할 수 있도록 구현하였다. 본 연구의 실험을 위해서 수정된 MPI 함수들로는 MPI_Init(), MPI_Comm_size(), MPI_Comm_rank(), MPI_Send(), MPI_Recv(), MPI_Finalize() 등이 있다.

그림 5는 그림 1 예제 프로그램의 수행 중에 발생한 모든 송수신 사건들을 나타낸 사건 뷰이다. 이 뷰에서는 발생한 모든 사건들을 시각적으로 나타내기 때문에 전체 수행에서 전송된 메시지는 알 수 있으나, 그들 간에 경합이 존재하는지를 식별하기는 어렵다. 그림 6은 예제 프로그램의 수행 중에 발생한 지역적 최초경합을 시각화한 경합 뷰이다. 그림을 통해서 P_1, P_2, P_3 등에서 지역적 최초경합이 발생하였음을 알 수 있다. P_1에서는 5번째 사건에서 경합이 발생하였으며, 2개의 메시지가 경합한다. 그리고 P_2에서는 1번째 사건에서 2개의 메시지가 경합하고, P_3에서는 2번째 사건에서 3개의 메시지가 경합한다. 그림 7은 예제 프로그램 수행에서 탐지된 지

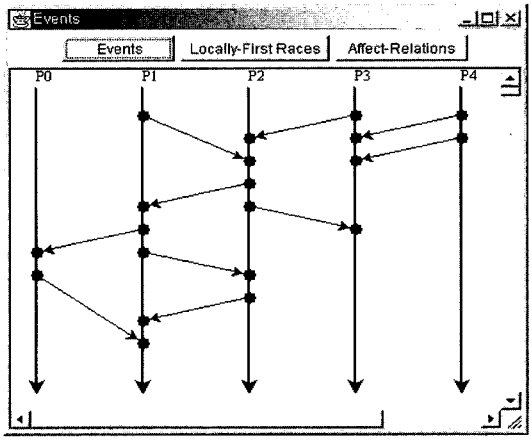

그림 5 사건 뷰

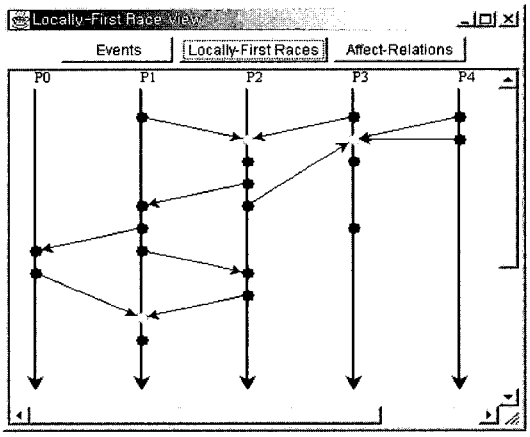

그림 6 지역적 최초경합 뷰

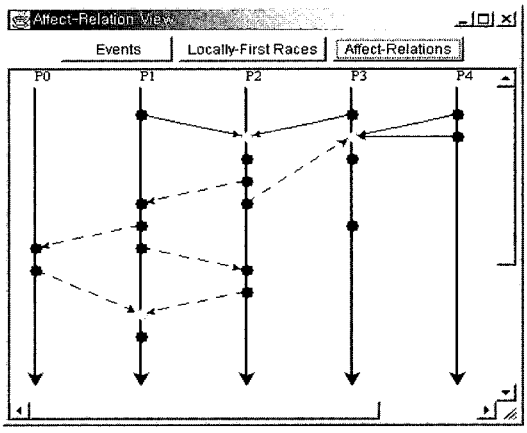

그림 7 영향관계 뷰

역적 최초경합들 간의 영향관계를 보인 것으로 영향받은 메시지는 점선으로 시각화되었다. 그림에서 P_1의 5

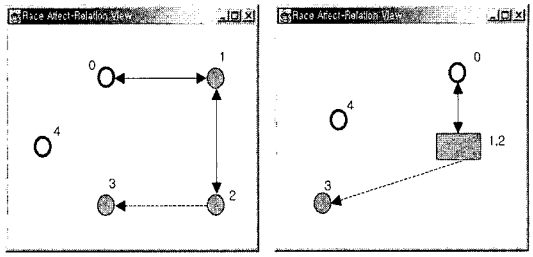

그림 8 추상적 영향관계 뷰

번째 사건에서 3개의 메시지가 경합하고 이들 모두가 영향받은 메시지로서 P_1의 지역적 최초경합은 영향받은 경합이다. 이와 마찬가지로 P_3에서 발생한 경합도 영향받은 경합이며, P_2에서 발생한 경합만이 영향받지 않은 최초경합임을 알 수 있다. 그림 8은 그림 7에 대한 추상적 영향관계 뷰를 보이고 있다. 그림에서 그림 8을 통해서 탐지된 경합들간의 영향관계를 쉽게 식별할 수 있다.

4. 결 론

본 연구에서는 각 프로세스에서 가장 먼저 발생한 경합인 지역적 최초경합과 영향관계 정보를 탐지하고 이를 분석하여 탐지된 경합들간의 영향관계를 시각화하는 기법을 제안하였다. 기존의 기법은 탐지된 모든 경합이 영향받지 않은 경합임을 보장하지 않았지만, 본 시각화 기법은 각 프로세스에서 최초로 발생한 경합들 간의 영향관계를 추상적으로 시각화함으로써 영향받지 않은 경합을 탐지하는데 효과적이다. 그러나 본 연구에서 제시된 기법은 독립적으로 사용될 수 없고, 기존의 첫 수행 알고리즘 Pass-1과 두 번째 수행 알고리즘 Pass-2와 결합해서 사용해야 합니다. 향후 연구에서는 본 기법을 독립적으로 사용할 수 있도록 개선될 필요가 있습니다.

참 고 문 헌

[1] Cypher, R., and E. Leu, "The Semantics of Blocking and Nonblocking Send and Receive Primitives," *8th Intl. Parallel Processing Symp.*, pp. 729-735, IEEE, Apr. 1994.

[2] Geist, A., A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. "PVM: Parallel Virtual Machine," *A Users' Guide and Tutorial for Networked Parallel Computing*, Cambridge, MIT Press, 1994.

[3] Snir, M., S. Otto, S. Huss-Lederman, D. Walker, *MPI: The Complete Reference*, MIT Press, 1996.

[4] Damodaran-Kamal, S. K. and J. M. Francioni, "Testing Races in Parallel Programs with an OtOt Strategy," *Int'l Symp. on Software Testing and Analysis*, pp. 216-227, ACM, Aug. 1994.

[5] Kilgore, R. and C. Chase, "Re-execution of

Distributed Programs to Detect Bugs Hidden by Racing Messages," *30th Annual Hawaii Int'l. Conference on System Sciences*, Vol. 1, pp. 423-432, Jan. 1997.

[6] Netzer, R. H. B., and B. P. Miller, "Optimal Tracing and Replay for Debugging Message-Passing Parallel Programs," *Supercomputing*, pp. 502-511, IEEE/ACM, Nov. 1992.

[7] Tai, K. C. "Reachability Testing of Asynchronous Message-Passing Programs," *Int'l. Symp. on Software Engineering for Parallel and Dist. Systems*, pp. 50-61, IEEE, May 1997.

[8] Tai, K. C. "Race Analysis of Traces of Asynchronous Message-Passing Programs," *Int'l. Conf. Distributed Computing Systems*, pp. 261-268, IEEE, May 1997.

[9] Cypher, R., and E. Leu, "Efficient Race Detection for Message-Passing Programs with Nonblocking Sends and Receives," *7th Symp. on Parallel and Distributed Processing*, pp. 534-541, IEEE, San Antonio, Texas, 1995.

[10] Damodaran-Kamal, S. K., and J. M. Francioni, "Nondeterminacy: Testing and Debugging in Message Passing Parallel Programs," *ACM/ONR Workshop on Parallel and Distributed Debugging*, Sigplan Notices, 28(12): 118-128, ACM, Dec. 1993.

[11] Netzer, R. H. B., T. W. Brennan, and K. D. Suresh, "Debugging Race Conditions in Message-Passing Programs," *SIGMETRICS Symp. on Parallel and Distributed Tools*, ACM, May 1996.

[12] Gropp, W. and E. Lusk, *User's Guide for Mpich, A Portable Implementation of MPI*, TR-ANL-96/6, Argonne National Laboratory, 1996.

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

[14] Fidge, C. J., "Partial Orders for Parallel Debugging," *SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging*, pp. 183-194, ACM, May 1988.

[15] Mattern, F., "Virtual Time and Global States of Distributed Systems," *Parallel and Distributed Algorithms*, pp. 215-226, Elsevier Science, North holland, 1989.

박 미 영

1999년 동서대학교 컴퓨터공학과 졸업(학사). 2001년 경상대학교 컴퓨터과학과 졸업(석사). 2005년 경상대학교 컴퓨터과학과 졸업(박사). 관심분야는 운영체제, 병렬 컴퓨팅 시스템, Grid 컴퓨팅

전 용 기

1980년 경북대학교 컴퓨터공학과 졸업(학사). 1982년 서울대학교 컴퓨터공학부 졸업(석사). 1993년 서울대학교 컴퓨터공학부 졸업(박사). 1982~1985년 한국전자통신연구소 연구원. 1985년~현재 경상대학교 컴퓨터과학과 교수, 컴퓨터·정보통신연구소 연구원