메시지 전달 프로그램에서의 수행 중 경합탐지
(On-the-fly Detection of Race Conditions in Message-Passing Programs)

박 미 영 † 강 문 혜 †† 전 용 기 ††† 박 혁 로 ††††
(Mi-Young Park) (Moon-Hye Kang) (Yong-Kee Jun) (Hyuk-Ro Park)

요 약 메시지전달 프로그램에서 발생하는 메시지경합은 프로그램의 비결정적 수행결과를 초래하므로 효과적인 디버깅을 위하여 탐지되어야 한다. 특히 각 프로세스에서 가장 먼저 발생하는 최초경합은 동일한 프로세스 내에서 다른 경합의 발생을 초래할 수 있으므로, 효과적인 경합탐지를 위해서 우선적으로 탐지되어야 한다. 이러한 경합을 탐지하기 위한 기존의 기법들은 적어도 프로그램을 두 번 수행하거나, 메시지들의 수에 비례한 크기의 추적 파일의 분석을 요구한다. 본 논문은 추적파일을 생성하지 않으면서 단 한번의 프로그램 수행으로 각 프로세스에서 발생하는 최초경합을 탐지하는 기법을 제시하고, 실험을 통해서 본 기법이 최초경합을 정확히 탐지함을 보인다.

키워드 : 메시지전달 프로그램, 디버깅, 메시지 경합, 수행 중 탐지, 최초경합

Abstract Message races should be detected for debugging message-passing parallel programs because they can cause non-deterministic executions. Specially, it is important to detect the first race in each process because the first race can cause the occurrence of the other races in the same process. The previous techniques for detecting the first races require more than two monitored runs of a program or analyze a trace file which size is proportional to the number of messages. In this paper we introduce an on-the-fly technique to detect the first race in each process without generating any trace file. In the experiment we test the accuracy of our technique with some benchmark programs and it shows that our technique detects the first race in each process in all benchmark programs.

Key words : message-passing program, debugging, message races, on-the-fly detection, first races

1. 서 론

메시지전달 프로그램[1-6]에서 프로세스들 간에 전송되는 메시지들은 네트워크의 통신지연이나 프로세스 스케줄링에 의해서 도착하는 순서가 달라질 수 있으며, 이것은 프로그램의 비결정적인 수행 결과를 초래한다. 경합하는 메시지들의 비결정적인 도착순서는 프로그램의 의도되지 않은 비결정적인 수행 결과를 초래하여, 병렬 프로그램의 정확성을 검증하고 재 수행을 이용한 디버깅 작업을 매우 어렵게 한다.

· 본 연구는 한국학술진흥재단의 신진연구자연수지원으로 수행되었음
† 정 회 원 : 전남대학교 유비쿼터스 정보가전 사업단 연구원
openmp@korea.com
†† 학생회원 : 경상대학교 정보과학과
kturtle@hanmai.net
††† 종신회원 : 경상대학교 정보과학과 교수
jun@gsnu.ac.kr
†††† 종신회원 : 전남대학교 전산과 교수
hyukro@chonnam.ac.kr
논문접수 : 2007년 1월 23일
심사완료 : 2007년 5월 8일

비록 성능향상을 위해서 일부 병렬 프로그램은 프로그래머가 의도적으로 경합을 가지도록 프로그램을 작성하기도 하지만, 다음과 같은 이유로 메시지 경합의 탐지는 디버깅 작업에서 매우 중요한다. 첫째 비결정적 수행[7]을 보이는 병렬 프로그램에서는 기존의 재수행을 통한 디버깅 작업이 거의 불가능하다. 둘째 병렬 프로그램의 모든 가능한 수행의 검증이 어렵다. 그러므로 메시지전달 프로그램의 오류를 검사하고 디버깅하기 위해서는 반드시 메시지들의 경합을 탐지해야 한다.

비동기적 메시지전달[1-6] 프로그램에서 메시지경합[8-11]은 두 개 이상의 메시지들이 도착순서가 보장되지 않고 동일한 채널로 전송될 때 발생한다. 메시지경합은 다른 경합과의 영향관계에 따라 영향받지 않은 경합[8,10,12]과 영향받은 경합[8,10,12]으로 나눌 수 있다. 영향받은 경합은 이전에 발생한 경합을 디버깅함으로써 자연히 사라질 수 있는 경합인 반면에, 영향받지 않은 경합은 다른 경합의 발생을 초래하기도 한다. 따라서 영향받지 않은 경합을 탐지하는 것이 디버깅에

효과적이다.

영향받지 않은 경합을 효율적으로 탐지하는 것은 각 프로세스에서 처음으로 발생한 경합을 탐지하는 것이다. 왜냐하면 각 프로세스에서 처음으로 발생한 최초경합[10,12]은 동일한 프로세스 내에서 발생된 다른 경합들로부터 영향을 받지 않은 경합임을 보장할 수 있기 때문이다. 영향받지 않은 경합을 탐지하는 기존의 기법[8-10,12,13]은 적어도 프로그램을 두 번 수행하거나, 메시지 수에 의존적인 추적파일의 분석을 요구하므로 비실용적이다.

본 연구에서는 추적파일을 생성하지 않으면서 단 한 번의 프로그램 수행으로 각 프로세스에서 발생하는 최초경합을 탐지하는 기법을 제시한다. 이를 위하여 본 논문에서는 최초경합에 관련된 사건들을 저장하는 메시지역사와 메시지역사를 이용하여 최초경합을 탐지하는 프로토콜를 제시한다. 메시지역사는 최초경합 탐지에 필요한 정보를 저장하기 위한 두 개의 배열 정보로 구성되고, 탐지 프로토콜은 메시지역사를 이용하여 각 프로세스에서 발생하는 최초경합을 탐지한다.

본 기법의 정확성을 보이기 위해서 표준 C언어와 MPI[6,14]에서 제공하는 MPI Profiling 인터페이스를 이용하여 본 기법을 구현하였다. 또한 mpptest[14]의 공개된 벤치마크 프로그램인 stress와 MPI_RTED[15](MPI Run Time Error Detection Test Suites)를 사용하여 본 기법이 각 프로그램에 존재하는 경합을 탐지함을 보인다.

2절에서는 메시지 경합과 그들 간의 영향관계에 대해서 설명하고, 이들을 탐지하기 위한 기존의 기법에 대해서 설명한다. 3절에서는 각 프로세스에서 발생한 최초경합을 탐기하기 위한 메지지 역사와 수행중 탐지기법을 제시한다. 그리고 4장에서는 벤치마크 프로그램을 이용한 실험을 통하여 본 기법의 정확성을 보이고 마지막 절에서 결론을 제시한다.

2. 연구 배경

본 절에서는 본 연구에서 대상으로 하는 프로그램 모델과 메시지 경합에 대해서 살펴보고, 경합들 간의 영향관계에 의한 영향받는 경합과 영향받지 않은 경합에 대해서 살펴본다. 또한 이러한 경합을 탐지하기 위한 기존 연구들을 보이고, 이들의 문제점을 살펴본다.

2.1 프로그램 모델

비동기적 메시지전달 프로그램[1-6]에서 프로세스간의 송수신은 논리적 채널 상에서 이루어지며, 각 송·수신 사건은 메시지를 송신하거나 수신하는 논리적 채널들을 명시한다. 이때, 하나 이상의 채널에서 메시지의 수신이 가능하면 수신사건은 임의의 채널을 비결정적으로 선택하여 하나의 메시지를 수신하게 된다. 그리고 임의의 채널로 송신된 메시지는 반드시 하나의 수신사건에 의해서 수신되며, 결과적으로 프로그램 수행 중에 송신된 모든 메시지들은 임의의 대응되는 수신사건에서 모두 수신된다고 가정한다.

메시지전달 프로그램의 수행은 수행 중에 발생하는 사건들의 집합과 그들 간의 순서관계(happened-before)[3,5]로 표현될 수 있다. 모든 수행에서 사건 a가 사건 b보다 항상 먼저 수행하는 경우 그들 간의 순서관계는 "(a 발생 후 b) 관계가 만족된다."고 하며, 이를 $a \rightarrow b$로 표기한다. 예를 들어 하나의 프로세스 내에서 순서적으로 발생하는 두 사건 $\{a, b\}$가 존재하는 경우에 $(a \rightarrow b \vee b \rightarrow a)$가 만족된다. 다른 프로세스 간에 메시지 전달을 위한 송수신사건 $\{s, r\}$이 존재하면, 메시지 송신사건 s와 대응되는 수신사건 r간에는 $s \rightarrow r$를 만족한다. 여기서 $\rightarrow$는 비반사적 추이적 폐쇄(irreflexive transitive closure) 관계를 의미하며, 임의의 세 사건 $\{a, b, c\}$가 존재하고 $(a \rightarrow b \wedge b \rightarrow c)$이 만족되면 $a \rightarrow c$가 만족된다. 임의의 두 사건 a와 b사이에서 (a 발생 후 b) 관계가 만족되지 않을 때는 그들 간의 관계를 $a \nrightarrow b$로 표기한다.

2.2 메시지 경합과 영향관계

메시지전달 프로그램에서 프로세스들 간에 전송되는 메시지들은 네트워크의 통신지연이나 프로세스 스케줄링에 의해서 도착하는 순서가 달라질 수 있으며, 이것은 프로그램의 비결정적인 수행 결과를 초래한다. 비록 성능향상을 위해서 일부 병렬 프로그램은 프로그래머가 의도적으로 경합을 가지도록 프로그램을 작성하기도 하지만, 다음과 같은 이유로 메시지 경합의 탐지는 디버깅 작업에서 매우 중요한다. 첫째 비결정적 수행[7]을 보이는 병렬 프로그램에서는 기존의 재수행을 통한 디버깅 작업이 거의 불가능하다. 둘째 병렬 프로그램의 모든 가능한 수행의 검증이 어렵다. 그러므로 메시지전달 프로그램의 오류를 검사하고 디버깅하기 위해서는 반드시 메시지들의 경합을 탐지해야 한다.

메시지경합[8-11]은 동일한 채널로 두 개 이상의 메시지들이 그들 간의 도착순서가 보장되지 않고 전송될 수 있을 때 발생된다. 이러한 메시지 경합은 임의의 수신사건 r과 경합하는 메시지들의 집합 M으로 표현되며, $<r, M>$으로 표기한다. 이때 r은 M에 속한 메시지들 중에서 가장 먼저 도착하는 메시지의 수신사건이며, M에 속한 임의의 메시지 송신사건 s는 $s \nrightarrow r$ 또는 $r \nrightarrow s$를 만족한다. 그리고 송신사건 s가 송신한 메시지를 msg(s)로 나타낸다.

그림 1은 임의의 메시지전달 프로그램의 수행 중에 발생한 사건들 간의 부분적 순서관계[3]를 나타낸 것이

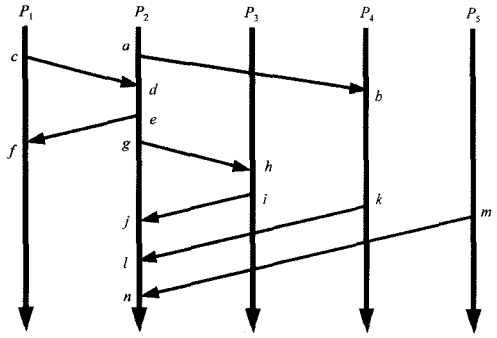

그림 1 메시지 경합

다. 그림에서 두 프로세스 P_3과 P_4가 메시지 $msg(i)$와 $msg(k)$를 프로세스 P_2로 송신한다. 이때 두 송신사건 i와 k는 P_2의 수신사건 j에 대해서 $\{ j \rightarrow i \vee j \rightarrow k \}$를 만족하지 않으므로 P_2의 수신사건 j에서 두 메시지 $\{msg(i), msg(k)\}$는 서로 경합한다. 또한 프로세스 P_5에서 송신된 메시지 $msg(m)$도 수신사건 j로 경합한다. 즉, 프로세스 P_2의 수신사건 j에서 발생한 경합은 경합하는 메시지들의 집합 $J = \{msg(i), msg(k), msg(m)\}$와 이들 메시지들 중 하나를 처음으로 수신하는 사건 j로 구성되므로 $<j, J>$로 나타낸다.

프로그램 수행 중에 발생하는 메시지경합 $<r, M>$은 하나의 수신사건과 서로 경합하는 두 개 이상의 메시지들로 구성되는데, 어떤 경합은 이전에 발생한 경합으로부터 영향을 받아 발생하기도 한다.

정의 1. 영향관계 두 개의 메시지경합 $<x, M>$과 $<y, N>$이 있을 때, $x \rightarrow y$를 만족하거나, N에 속하는 임의의 $msg(s)$에 대해서 $x \rightarrow s$를 만족하면, $<x, M> \rightarrow <y, N>$로 표기하며, 이들은 영향관계(affect-relation)에 있다고 한다. 그리고 $<y, N>$는 영향받은 경합이라 한다.

그림 1의 P_2에는 서로 다른 두 개의 경합이 존재한다. 하나는 수신사건 d에서 발생한 경합 $<d, \{msg(c), msg(k), msg(m)\}>$이고, 다른 하나는 수신사건 j에서 발생한 경합 $<j, \{msg(i), msg(k), msg(m)\}>$이다. 이들 간에는 $d \rightarrow j$를 만족하기 때문에 영향관계가 존재하며, 경합 $<j, \{msg(i), msg(k), msg(m)\}>$은 영향받은 경합이다. 두 경합 중에서 경합 $<d, \{msg(c), msg(k), msg(m)\}>$이 디버깅되면 영향받은 경합 $<j, \{msg(i), msg(k), msg(m)\}>$은 자연히 제거된다. 따라서 효과적인 디버깅을 위해서는 모든 경합을 탐지하는 것 보다는 영향받지 않은 경합만을 탐지하는 것이 효과적이다.

영향받지 않은 경합[8,10,12]을 효율적으로 탐지하는 것은 각 프로세스에서 처음으로 발생한 경합을 탐지하는 것이다. 왜냐하면 각 프로세스에서 처음으로 발생한 경합은 동일한 프로세스 내에서 발생된 다른 경합들로부터 영향을 받지 않은 경합임을 보장할 수 있기 때문이다. 즉, 그림 1의 P_2에서 발생한 최초경합은 $<d, \{msg(c), msg(k), msg(m)\}>$이다.

정의 2. 최초경합 임의의 한 프로세스 내에서 발생한 경합들의 집합을 R이라 할 때, R에 속한 임의의 경합 $<y, N>$에 대해서, $x \rightarrow y$를 만족하는 $<x, M>$가 R에 존재하지 않으면, $<y, N>$는 해당 프로세스에서 가장 먼저 발생한 경합으로써, 그 프로세스의 최초경합이라 한다.

2.3 기존 연구

메시지전달 프로그램에서 경합을 탐지하는 기법은 수행후 탐지기법[13], 수행중 탐지기법[8,11], 그리고 혼합 탐지기법[9,10,12]등으로 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 기억공간의 요구가 비현실적으로 크다. 수행후 탐지기법으로는 기존의 Tai가 연구한 기법[13]이 있다. 이 연구는 재귀적으로 호출하는 알고리즘을 이용하여 하나의 입력에 대한 메시지전달 프로그램의 수행 가능한 모든 인스턴스를 제공하여, 프로그램의 정확성을 판단한다. 그러나 재귀적 호출로 인하여 시간적·공간적 복잡도가 비현실적으로 크다.

수행중 탐지기법은 프로그램의 수행을 감시하면서 경합의 발생여부를 검사한다. 이 기법은 프로그램에 경합이 존재한다면 하나 이상의 경합 탐지를 보장하며, 수행중에 사용되는 기억공간을 재사용하므로 대규모의 병렬프로그램에서도 적용할 수 있는 현실적인 기법이다. 그러나 이 기법은 영향받은 경합을 탐지할 수 있기 때문에 디버깅에는 효과적이지 않다. Damodaran이 제시한 OtOt 기법[8]은 수행 중에 최초경합을 탐지하지만, 한 번의 수행에서 하나의 프로세스만을 감시하여 경합을 탐지하므로, 적어도 프로세스의 수만큼 반복해서 수행해야 하는 문제점이 있다.

혼합 탐지기법[10,12,16]은 기존의 수행후 기법과 수행중 기법을 모두 사용하는 기법이다. 혼합 기법은 두 번의 프로그램 수행을 통해서 각 프로세스에서 발생한 최초경합을 탐지한다. 첫 수행에서는 매 수신사건마다 최초경합의 발생여부를 검사하여 그 위치정보를 추적파일에 기록하고, 두 번째 수행에서는 최초경합이 발생한 위치에서 프로세스의 수행을 중단하여 경합하는 메시지들을 탐지한다. 이 기법 역시 최초경합을 탐지하기 위해서 프로그램을 두 번 수행해야 하는 단점을 가지고 있다.

경합을 탐지하는 대표적인 도구[17-19]로는 MAD, MARMOT, 그리고 MPVisualizer 등이 있다. MAD [19]는 멀티 프로세스에 대한 중단점 기능과 변수 값 검사, 재수행 등과 같은 기능을 제공하고, MARMOT[18]는 MPI 프로그램 수행 시 발생하는 전형적인 오류 탐지 기능 외에 데드락(deadlock) 탐지, 경합 탐지(race detection) 기능을 제공한다. MPVisualizer[17]는 MPI 프로그램 수행 시 발생하는 사건 정보를 추적파일로 만들고 재수행하는 기능을 제공한다. 또한 효과적인 디버깅을 위해서 시각화 제공뿐만 아니라 경합의 발생도 알려준다.

그러나 위의 도구들은 MPI의 수신사건에서 MPI_ANY_SOURVCE가 사용되면 무조건 경합을 탐지한다. MPI에서 MPI_ANY_SOURCE는 프로그램의 병행성을 높이기 위해 프로그래머가 의도적으로 종종 사용되므로 MPI_ANY_SOURCE만으로 경합을 탐지하는 것은 정확한 탐지라 할 수 없다. 부정확한 경합의 탐지는 프로그램의 디버깅 작업을 어렵게 하므로, 보다 정확한 경합 탐지를 위한 새로운 기법이 요구된다.

3. 최초경합의 탐지

본 절에서는 단 한번의 수행으로 각 프로세스에서 발생하는 최초경합을 탐지하는 기법을 제안한다. 먼저 최초경합 탐지에 필요한 정보를 저장하는 메시지역사를 제시하고, 이 정보를 이용하여 최초경합을 탐지하는 수행중 탐지 알고리즘을 제시한다.

3.1 메시지역사

메시지역사는 두 개의 정보로 구성되며, 하나는 각 프로세스로 메시지를 보낸 마지막 송신사건 정보를 저장하며, 다른 하나는 마지막 송신사건 이후에 처음으로 매 시지를 수신하는 수신사건 정보를 저장한다. 송신사건 정보는 해당 프로세스와 다른 프로세스간의 순서관계를 판단하는데 사용되며, 수신사건 정보는 해당 프로세스에서 발생하는 최초경합의 후보 사건 정보이다.

두 정보를 관리하기 위한 메시지역사의 자료구조는 *LastSend*와 *FirstRecv*로 구성된다. 이들은 프로세스 수만큼의 크기를 가지는 배열구조로써, 각 배열의 원소는 각 프로세스에 일대일 대응되며 *localclock*을 저장하는 공간이다. 각 프로세스의 *localclock*은 정수 값을 가지며, 0부터 시작하여 해당 프로세스에서 송수신사건이 발생할 때마다 1씩 증가함으로써 한 프로세스 내에서 발생한 각 사건들을 식별하는데 사용된다.

*LastSend*는 해당 프로세스에서 다른 프로세스로 메시지를 송신한 마지막 사건의 *localclock*값을 가지며, *FirstRecv*는 *LastSend*에 저장된 송신사건 이후에 처음으로 메시지를 수신한 수신사건의 *localclock*을 가진다. 결과적으로 *FirstRecv*의 각 원소는 항상 *LastSend*의 각 원소보다 큰 값을 가진다.

그림 2는 프로그램 수행 중에 메시지역사를 생성 및 유지하는 알고리즘이다. 그림 2(a)는 메시지역사의 *FirstRecv*와 *LastSend*의 각 원소를 초기화하는 알고리즘이며, 수행 초기에 한번만 수행된다. 여기서 변수 *size*는 프로세스의 수를 나타내며, 모든 변수, *LastSend*, *FirstRecv*, 그리고 *localclock* 은 0으로 초기화된다.

그림 2(b)는 메시지역사의 *LastSend*를 갱신하는 알고리즘이며 매 송신사건마다 수행된다. 변수 *dest*는 메시지의 목적지, 즉 메시지를 수신하는 프로세스를 나타낸다. 알고리즘에서 먼저 *localclock*이 1 만큼 증가하고, *LastSend*의 목적지 프로세스(*dest*)에 해당하는 원소를 송신사건의 *localclock*으로 갱신됨으로써, 다른 프로세스로 메시지를 가장 마지막에 송신한 사건의 정보를 가진다.

그림 2(c)는 *FirstRecv*를 갱신하는 알고리즘이며, 매 수신사건 마다 수행된다. 이 알고리즘에서도 먼저 *localclock*이 1 만큼 증가한다. 그리고 3행에서는 *FirstRecv*와 *LastSend*의 각 원소들을 비교하여, *LastSend*의 원소가 크거나 동일한 경우에만 *FirstRecv*의 해당 원소를 해당 수신사건의 *localclock*으로 갱신한다. 이렇게 함으로써, *FirstRecv*는 *LastSend*의 각 원소에 해당하는 마지막 송신사건 이후에 발생한 최초의 수신사건 정보를 가지며, 큰 값을 유지한다.

그림 3은 그림 1의 예제에서 P_2의 메시지역사에 대한 변화 과정을 보인 것이다. 그림에서와 같이 프로그램 시작 시점에서는 해당 프로세스의 메시지역사는 '0'으로 초기화된다. P_2의 송신사건 a가 P_4에 메시지를 전송할

```
0   CheckHistoryInit()
1   for i from 1 to size do
2       localclock := 0
3       LastSend[i] := 0
4       FirstRecv[i] := 0
5   endfor
                        (a)
0   CheckHistorySend()
1    localclock :=  localclock + 1
2    LastSend[dest] := localclock
                        (b)
0   CheckHistoryRecv()
1   localclock :=  localclock + 1
2   for source from 1 to size do
3       if LastSend[source]  >=  FirstRecv[source] then
4           FirstRecv[source] := localclock
5       endif
5   endfor
                        (c)
```

그림 2 메시지역사 생성 알고리즘

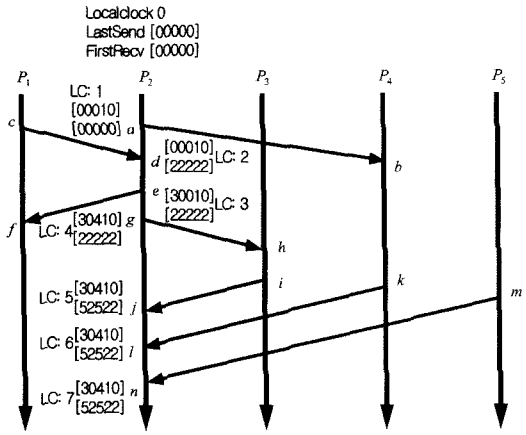

그림 3 P_2에서의 메시지역사 변화 과정

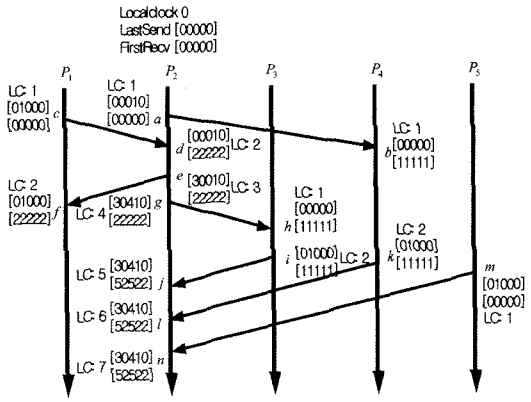

그림 4 모든 프로세스에서의 메시지역사의 변화 과정

때 프로세스 4에 대응되는 *LastSend*의 네 번째 원소는 현재의 *localclock* '1'로 갱신된다. 이는 프로세스 4로 메시지를 가장 마지막에 송신한 사건은 1번째 사건임을 나타낸다.

수신사건 *d*가 발생하면 *FirstRecv*의 각 원소 값과 *LastSend*의 원소를 비교하는데, 이때 *LastSend*의 각 원소가 *FirstRecv*의 각 원소보다 크거나 동일한 경우 *FirstRecv*의 각 원소는 현재의 *localclock*인 '2'로 갱신된다. 즉 *FirstRecv*는 <22222>로 갱신된다. 이렇게 하여 *FirstRecv*는 항상 *LastSend* 이후에 처음으로 메시지를 수신한 사건 정보를 저장하며, *LastSend* 보다 큰 값을 가지게 된다.

송신사건 *e*가 프로세스 P_1으로 메시지를 보내면, 프로세스 1에 대응되는 *LastSend*의 첫 번째 원소는 현재의 *localclock* '3'으로 변경된다. 수신사건 *j*에서는 *FirstRecv*의 각 원소와 대응되는 *LastSend*의 각 원소들을 비교하여 *LastSend* 원소가 *FirstRecv*의 원소보다 크거나 동일하면 현재의 *localclock*인 '5'로 *FirstRecv*의 원소를 갱신한다. 따라서 *FirstRecv*의 첫 번째와 세 번째 원소는 각각 '5'로 갱신되고, *FirstRecv*은 <52522>가 된다. 수신사건 *l*에서는 *LastSend*의 각 원소보다 작거나 동일한 *FirstRecv*의 각 원소 값이 존재하지 않으므로, *FirstRecv*은 갱신되지 않는다.

그림 4는 모든 프로세스에서의 *LastSend*와 *FirstRecv*를 보여주고 있다. 이와 같이, 메시지역사는 각 프로세스마다 존재하며, 수행 중에 다른 프로세스와는 무관하게 해당 프로세스가 독립적으로 메시지역사 정보를 관리하므로, 메시지역사 정보의 생성 및 유지를 위한 프로세스간의 통신 오버헤드는 존재하지 않는다.

3.2 탐지 알고리즘

본 기법은 vector timestamp[3,5]와 메시지역사의 *FirstRecv*를 이용하여 프로그램 수행 중에 각 프로세스에서 발생한 최초경합을 탐지한다. 먼저 vector timestamp를 이용하여 경합의 발생 여부를 검사하고, 메시지역사를 이용하여 발생한 경합이 해당 프로세스의 최초경합인지를 판단한다.

그림 5는 최초경합을 탐지하는 알고리즘으로서, 각 프로세스의 매 수신사건마다 수행하여 최초경합의 발생 여부를 판단한다. 알고리즘의 4행에서는 이전 수신사건인 *preRecv*와 현재의 송신사건 *Send*간의 순서관계를 검사한다. 그들 간에 $prevRecv \nrightarrow Send$ 관계가 만족되면, 현재의 메시지는 *prevRecv* 수신사건으로 경합함을 의미한다.

또한 *firstRace*와 현재 수신사건의 메시지역사 *FirstRecv*[*source*]를 비교하여 *FirstRecv*[*source*]가 *firstRace*보다 작은지를 검사한다. 여기서 *firstRace*는 최초경합의 위치 정보인 *localclock*을 저장하는 변수이며, 수행 초기에 최대 정수 값으로 초기화된다. *source*는 현재 수신사건으로 메시지를 송신한 프로세스를 나타낸다. 메시지역사의 *FirstRecv*는 해당 프로세스에서 발생한 최초경합의 후보사건들로 구성되므로, 프로세스 P_i의 *FirstRecv*[*j*]는 프로세스 P_j가 프로세스 P_i로 송신한 메

```
0  CheckFirstRace()
1    prevRecv = the previous receive event
2    Send = a event that sent a Message to this receive event
3    Recv = this receive event
4    if(prevRecv ↛ Send and FirstRecv[source] < firstRace) then
5      firstRace := FirstRecv[source]
6      firstChan := thisChan
7      RacingMessage := Message
8    elseif (prevRecv ↛ Send and FirstRecv[source] = firstRace) then
9      RacingMessage := RacingMessage ∪ Message
10   endif
11   prevRecv := Recv
```

그림 5 수행중 최초경합 탐지 알고리즘

시지가 경합하는 P_1의 최초 수신사건 정보를 기억한다.

따라서, 4행의 조건에서 메시지역사의 *FirstRecv*[*source*]의 값이 *firstRace*보다 작으면 해당 프로세스에서 최초경합이 탐지되었음을 의미하므로, *firstRace*를 비롯하여 최초경합에 대한 정보가 5행과 7행 사이에서 갱신된다. 수행 초기에 *firstRace*는 최대 정수 값으로 초기화되므로 가장 먼저 탐지된 경합의 위치정보가 *firstRace*에 기억되며, 이후에 *firstRace*보다 먼저 발생한 경합이 탐지될 때 마다 해당 *firstRace*는 갱신된다. 따라서 프로그램 수행이 종료된 시점에서의 *firstRace*는 해당 프로세스에서 발생한 최초경합의 수신사건을 기억한다. 이 외에도 경합하는 메시지가 전송된 채널의 정보는 *firstChan*에 저장되고, 경합하는 메시지는 *RacingMessage*에 저장된다.

8행과 같이 메시지역사의 *FirstRecv*[*source*]의 값이 *firstRace*의 값과 일치하는 경우는 현재의 메시지가 이미 탐지된 최초경합의 *firstRace*를 향해 경합함을 의미하므로, 경합하는 메시지만 *RacingMessage*에 저장한다. 알고리즘의 11행에서는 다음 수신사건을 위해서 현재의 수신사건 정보를 *prevRecv*에 저장한다.

그림 3의 예제에 본 탐지 알고리즘을 적용하면 다음과 같다. 먼저 수신사건 l에서 이전에 발생한 수신사건 j와 현재의 송신사건 k가 병행하므로 l에서 경합이 탐지된다. 이때 기존의 기법들은 모두 최초경합이 수신사건 l에서 발생하였다고 보고한다. 그러나 메시지역사 정보를 이용하는 본 기법에서는 $msg(k)$가 수신사건 d에서 경합한다고 보고한다. 즉, P_4가 메시지 $msg(k)$를 송신했으며, 수신사건 l의 *FirstRecv*에서 P_4에 대응하는 원소의 값은 2이다. 이것은 P_4로부터 수신된 메시지 $msg(k)$가 P_2의 두 번째 사건 즉, 수신사건 d에서 경합함을 알려준다. 앞서 2.2절에서 언급했듯이, 그림 4를 보면 $msg(k)$가 수신사건 d로 경합함을 쉽게 확인 할 수 있다. 따라서 수신사건 l에서 본 기법은 해당 경합의 위치정보 2를 *firstRace*에 저장된다.

수신사건 n이 발생하면 *firstRace*는 갱신되지 않고, 수신된 메시지만 *RacingMessage*에 포함된다. 왜냐하면 메시지 $msg(m)$는 P_5로부터 수신되었고, 현재 메시지역사 *FirstRecv*의 다섯 번째 원소는 현재의 *firstRace*와 동일한 2이기 때문이다. 프로그램이 종료되는 시점의 *firstRace*의 위치는 2이며, *RacingMessage*는 {$msg(k)$, $msg(m)$}이다. 따라서 본 기법은 한 번의 수행으로 P_2에서 발생한 최초경합 <d, {$msg(k)$, $msg(m)$}>을 정확히 탐지함을 알 수 있다.

4. 실 험

본 연구는 MPICH[6]가 설치된 분산 시스템에서 본 기법의 정확성을 실험하였다. 이를 위하여 표준 C언어와 MPI[6,14]에서 제공하는 Profiling Interface를 이용하여, 사용자 프로그램을 수정하지 않고 본 기법을 사용할 수 있도록 구현하였다. 본 기법이 최초경합을 정확히 탐지함을 보이기 위해서 C언어와 MPI 라이브러리로 작성된 mpptest[14]의 공개된 벤치마크 프로그램인 stress와 MPI_RTED(MPI Run Time Error Detection Test Suites)[15]를 사용하였다.

mpptest는 Argonne National Laboratory에서 MPI의 기본적인 통신 함수들의 성능을 평가하기 위해서 개발한 벤치마크 프로그램이며, 여기에 속하는 Stress는 프로세스간의 point-to-point 통신 성능을 평가한다. Stress에서 사용된 MPI_Recv나 MPI_Irecv에서는 임의의 메시지를 도착하는 순서대로 수신하기 위해서 MPI_ANY_SOURCE를 사용하고 있으나, 본 기법을 적용한 결과 메시지 경합은 탐지되지 않았다. 경합이 탐지되지 않은 이유를 분석한 결과, stress에서는 각 MPI_Recv와 MPI_Irecv에서 서로 다른 tag을 사용함으로써 메시지들이 결정적으로 도착하였다. 따라서 stress에서 메시지 경합을 유발하기 위해서, 본 실험에서는 모든 수신사건에서 서로 다른 tag가 명시된 파라메타를 MPI_ANY_TAG로 수정하였다. 표 1은 수정 전후의 stress 코드를 보여 주고 있다.

표 2는 수정된 stress에 본 기법을 적용한 결과이다. P_0의 최초경합은 네 번째 수신 사건에서 발생하였으며, 세 개의 메시지들이 경합함을 보인다. P_1의 최초경합은 다섯 번째 사건에서 발생하며, 세 개의 메시지들이 경합한다. 마찬가지로 P_2와 P_3의 최초경합은 각 프로세스의 첫 번째 사건에서 발생하며, 각각 2개와 3개의 메시지로 경합한다.

다음은 MPI_RTED에 본 기법은 적용한 결과이다.

표 1 수정 전후의 stress 코드

	stress 코드
수정 전	MPI_Recv(buffer, bufmsize, MPI_BYTE, MPI_ANY_SOURCE, *tag,* MPI_COMM_WORLD, &status);
수정 후	MPI_Recv(buffer, bufmsize, MPI_BYTE, MPI_ANY_SOURCE, ***MPI_ANY_TAG,*** MPI_COMM_WORLD, &status);

표 2 stress에서의 최초경합의 탐지

pid	The First Racing Receive Event	The Number of Racing Messages
0	4	3
1	5	3
2	1	2
3	1	3

MPI_RTED(MPI Run-Time Error Detection)는 Iowa State University의 High Performance Computing Group에서 개발된 테스트 프로그램 패키지이며, MPI 프로그램의 오류탐지 도구들을 평가하기 위해 개발되었다. MPI_RTED는 MPI 프로그램에서 발생 가능한 런타임 오류 유형별로 분류되어 있으며, 각 오류 유형별로 적게는 수십 개 많게는 수백 개의 해당 오류를 가진 테스트 프로그램들로 구성된다. 각 테스트 프로그램은 Fortran, C/C++에 적용될 수 있도록 동일한 테스트에 대해서 세 개로 언어로 작성되었으며, 본 실험에서는 메시지 경합 오류 섹션의 point-to-point 함수 들로 구성된 C 언어 테스트 프로그램을 사용하였다. 실험에 사용된 각 프로그램은 하나의 메시지 경합을 가지도록 구현되었다.

실험의 결과는 표 3과 같다. 표 3에서 첫 번째 칸은 실험에 사용된 테스트 프로그램이 이름이며, 두 번째 칸은 해당 테스트 프로그램에서 사용된 MPI 통신 함수를 나타낸다. 그리고 마지막 칸은 본기법의 탐지 여부를 보여준다. 표에서와 같이 본 기법이 해당 프로그램에서 메시지 경합을 탐지함을 확인하였다. (MPI_RTED 중에서 경합 오류를 가진 테스트 프로그램의 소스코드를 부록에서 일부분 소개하며, 모든 테스트 프로그램은 http://rted.public.iastate.edu/에서 다운로드 할 수 있다.)

본 기법은 수행 중에 프로세스별로 가장 먼저 발생한 경합을 탐지하는 기법으로써, 기존의 기법이 두 번의 프로그램 수행을 요구하는 반면에 본 기법은 단 한번의 수행으로 최초경합을 탐지한다. 따라서 경합 탐지하는 데 소요되는 시간은 본 기법이 더 효율적임으로 알 수 있다.

표 3 MPI_RTED에서의 최초경합 탐지

프로그램 명	MPI 함수	탐지여부
c_B_1_1_a_M1.c c_B_1_2_a_M1.c	MPI_RECV MPI_RECV	탐지
c_B_1_1_b_M1.c c_B_1_2_b_M1.c	MPI_SENDRECV MPI_SENDRECV	탐지
c_B_1_1_c_M1.c c_B_1_2_c_M1.c	MPI_SENDRECV_REPLACE MPI_SENDRECV_REPLACE	탐지
c_B_1_1_d_M1.c c_B_1_2_d_M1.c	MPI_IRECV MPI_IRECV	탐지
c_B_1_1_e_M1.c c_B_1_2_e_M1.c	MPI_RECV MPI_SENDRECV	탐지
c_B_1_1_f_M1.c c_B_1_2_f_M1.c	MPI_RECV MPI_SENDRECV_REPLACE	탐지
c_B_1_1_g_M1.c c_B_1_2_g_M1.c	MPI_RECV MPI_IRECV	탐지
c_B_1_1_h_M1.c c_B_1_2_h_M1.c	MPI_SENDRECV MPI_SENDRECV_REPLACE	탐지
c_B_1_1_i_M1.c c_B_1_2_i_M1.c	MPI_SENDRECV MPI_IRECV	탐지
c_B_1_1_j_M1.c c_B_1_2_j_M1.c	MPI_SENDRECV_REPLACE MPI_IRECV	탐지

5. 결 론

본 논문에서는 메시지전달 프로그램에서 발생하는 최초경합을 추적파일의 분석 없이 단 한번의 수행으로 탐지하는 기법을 제시하였다. 또한 벤치마크 프로그램을 이용한 실험에서, 본 기법이 한번의 수행으로 각 프로세스에서 최초경합을 탐지함을 확인하였다. 즉 본 기법은 각 프로세스에서 발생하는 최초경합을 탐지하여 영향받은 경합의 탐지를 줄임으로써 불필요한 정보로 인한 디버깅 부담을 줄이는데 중요한 역할을 한다.

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, April 1994.

[2] 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, Oct. 1995.

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

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

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

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

[7] Kranzluller, D., and M. Schulz, "Notes on Nondeterminism in Message Passing Programs," *9th European PVM/MPI Users' Group Conf.*, Lecture Notes in Computer Science, 2474: 357-367, Springer-Verlag, Sept. 2002.

[8] 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.

[9] Kilgore, R., and C. Chase, "Re-execution of Distributed Programs to Detect Bugs Hidden by Racing Messages," *30th Annual Hawaii Int'l. Conf. on System Sciences*, Vol. 1, pp. 423-432, Jan. 1997.

[10] Netzer, R. H. B., T. W. Brennan, and S. K. Damodaran-Kamal, "Debugging Race Conditions in Message-Passing Programs," *SIGMETRICS Symp. on Parallel and Distributed Tools*, pp. 31-40, ACM, May 1996.
[11] 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.
[12] Park, M., and Y. Jun, "Detecting Unaffected Race Conditions in Message-Passing Programs," *11th European PVM/MPI User's Group Meeting*, Lecture Notes in Computer Science, 3241: 268-276, Springer-Verlag, Sept. 2004.
[13] Tai, K. C., "Race Analysis of Traces of Asynchronous Message-Passing Programs," *17th Int'l. Conf. Distributed Computing Systems*, pp. 261-268, IEEE, May 1997.
[14] Gropp, W., and E. Lusk, "Reproducible Measurements of MPI Performance Characteristics," *6th European PVM/MPI Users' Group Conf.*, Lecture Notes in Computer Science, 1697: 11-18, Springer-Verlag, Sept. 1999.
[15] HPC Group, MPI Run Time Error Detection Test Suites: http://rted.public.iastate.edu/MPI/, Iowa State University, USA, 2006.
[16] Park, Mi-Young, and Yong-Kee Jun, "Detecting Unaffected Message Races in Parallel Programs," Proc. of the 1st Int'l Conf. on Grid and Pervasive Computing (GPC), Lecture Notes in Computer Science, 3947: 187-196, Springer-Verlag, Taichung, Taiwan, May 2006.
[17] Claudio, A.P., J.D. Cunha, and M.B. Carmo, "Monitoring and Debugging Message Passing Applications with MPVisualizer," 8th Euromicro Workshop on Parallel and Distributed Processing, pp. 376-382, IEEE, Jan. 2000.
[18] Krammer, B., M.S. Muller, and M.M. Resch, "MPI Application Development Using the Analysis Tool MARMOT," 4th International Conference on Computational Science, Lecture Notes in Computer Science, 3038: 464-471, Springer-Verlag, june 2004.
[19] Kranzlmuller D., C. Schaubschlager, and J. Volkert, "A Brief Overview of the MAD Debugging Activities," 4th International Workshop on Automated Debugging (AADEBUG 2000), Aug. 2000.

부록: MPI_RTED에서 메시지경합 오류를 가진 테스트 프로그램의 소스코드

```
/*
!*******************************************
! Copyright (c) 2006 Iowa State University, Glenn
  Luecke, James Coyle,
! James Hoekstra, Marina Kraeva, Mi-Young Park,
  Olga Taborskaia,
! Andre Wehe, and Ying Xu, All rights reserved.
! Licensed under the Educational Community
  License version 1.0.
! See the full agreement at http://rted.public.iastate.edu/ .
!*******************************************
!
!   Name of the test: c_B_1_1_a_M1.c
!
!   Summary:  Racing Messages in MPI_Recv Routine
!
!   Version of MPI:  Conforms to MPI 1.1, does
                     not require MPI 2
!                    implementation
!
!   Test description: In process 1, two MPI_Recvs
    are called with
!                    MPI_ANY_SOURCE and
                     MPI_ANY_TAG and
!                    their values are nondeterministic
!
!   Support files:     not needed
!
!   Env. requirements: not needed
!
!   Keywords:          MPI_Recv, MPI_ANY_SOURCE,
                       MPI_ANY_TAG,
!                      Point-to-Point Communication
!
!   Last modified:     02/22/06
!
!   Programmer:        Mi-Young Park
!*******************************************/
#include <mpi.h>
#include <stdlib.h>
#include <stdio.h>

#define EXITCODE_BAD 9
#define EXITCODE_OK 1

#define N            4

int main(argc, argv)
int argc;
char **argv;
{
```

```
  MPI_Status status;
  int rank, size;
  int sendbuf, recvbuf_1, recvbuf_2;

  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &size);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);

  if (size < N) {
  printf("ERROR: this test requires at least %d
  processes\n", N);
  MPI_Finalize();
  exit(EXITCODE_BAD);
  }

  if (size > N && rank == 0) {
  printf("This program is designed to execute with
  four processes.\n");
  printf("More than four processes will be ignored.\n");
}

if (rank == 0) {
  sendbuf = rank;
  MPI_Send(&sendbuf, 1, MPI_INT, 1, rank, MPI_
  COMM_WORLD);
}
if (rank == 2) {
  sendbuf = rank;
  MPI_Send(&sendbuf, 1, MPI_INT, 1, rank, MPI_
  COMM_WORLD);
}

if (rank == 1) {
  MPI_Recv(&recvbuf_1, 1, MPI_INT, MPI_ANY_
  SOURCE, MPI_ANY_TAG,
            MPI_COMM_WORLD, &status);
  MPI_Recv(&recvbuf_2, 1, MPI_INT, MPI_ANY_
  SOURCE, MPI_ANY_TAG,
            MPI_COMM_WORLD, &status);

  printf("\nName of the test: c_B_1_1_a_M1.c\n");
  printf("Racing Messages in MPI_Recv Routine.\n");
  printf("recvbuf_1 and recvbuf_2 are nondeterministic.
  \n");
  printf("<< Process %d: recvbuf_1=%d and recvbuf_2
  =%d >>\n", rank,
            recvbuf_1, recvbuf_2);
  }

  MPI_Finalize();

  return(EXITCODE_OK);
}
```

박 미 영

1999년 동서대학교 컴퓨터공학과 졸업(학사). 2001년 경상대학교 컴퓨터과학과 졸업(석사). 2005년 경상대학교 컴퓨터과학과 졸업(박사). 2006년 Iowa State University 포닥 연구원. 2007년 현재 전남대학교 BK21 유비쿼터스정보가전사업단 포닥연구원. 관심분야는 운영체제, 병렬 컴퓨팅 시스템, Grid 및 유비쿼터스 컴퓨팅

강 문 혜

2001년 경상대학교 컴퓨터과학과 졸업(학사). 2003년 경상대학교 대학원 컴퓨터과학과 졸업(석사). 2005년 경상대학교 대학원 컴퓨터과학과 박사과정. 관심분야는 운영체제, 분산병렬처리, 시스템 소프트웨어

전 용 기

1980년 경북대학교 컴퓨터공학과 졸업(학사). 1982년 서울대학교 컴퓨터공학부 졸업(석사). 1993년 서울대학교 컴퓨터공학부 졸업(박사). 1982년~1985년 한국전자통신연구소 연구원. 1995년~1996년 캘리포니아 주립대(UCSC) 컴퓨터과학과 연구원. 1985년~현재 경상대학교 정보과학과 교수. 컴퓨터·정보통신연구소 연구원. 관심분야는 분산병렬처리, 내장형시스템, 시스템소프트웨어

박 혁 로

1987년 서울대학교 전산학과(학사). 1989년 한국과학기술원 전산학과(공학석사) 1997년 한국과학기술원 전산학과(공학박사). 1994년~1996년 연구개발정보센터 연구원. 1997년~1998년 연구개발정보센터 선임연구원. 1999년~2002년 전남대학교 전산학과 조교수. 2002년 University of Maryland UMIACS Post Doc. 2002년~현재 전남대학교 전자컴퓨터공학부 부교수. 관심분야는 자연어처리, 정보검색, 텍스트마이닝