DOI: 10.3745/KIPSTA.2008.15-A.2.087

MPIRace-Check V 1.0: MPI 병렬 프로그램의 메시지경합 탐지를 위한 도구

박 미 영[†] · 정 상 화[††]

요 약

메시지전달 프로그램에서 발생하는 메시지경합은 프로그램의 비결정적 수행결과를 초래하므로 효과적인 디버깅을 위하여 탐지되어야 한다. 메시지경합을 탐지하는 기존의 도구는 임의의 메시지를 수신하는 모든 사건에서 경합이 발생한다고 보고한다. 그러나 메시지들이 전송되는 논리적인 통신채널이 서로 다르면 임의의 메시지를 수신하는 사건에서 경합이 발생하지 않을 수도 있으므로, 기존 도구의 부정확한 탐지정보는 프로그래머의 디버깅 작업을 더욱 어렵게 한다. 본 논문에서는 메시지 송수신 사건간의 병행성과 메시지들의 논리적 통신채널을 검사하여 보다 정확하게 메시지경합을 탐지하는 도구인 MPIRace-Check를 제안하다. 본 도구는 vector timestamp를 이용하여 프로그램 수행 중에 메시지를 전송하는 송수신 사건들간의 병행성을 검사하고, 메시지 부가정보를 이용하여 메시지들의 논리적인 통신채널이 동일한지를 검사하여 메시지경합을 탐지한다. 실험에서는 MPI_RTED와 벤치마크 프로그램을 이용하여 본 도구가 프로그램 수행 중에 효율적으로 모든 경합을 정확하게 탐지함을 보인다. 따라서 본 도구는 메시지경합을 정확하게 탐지하여 프로그래머의 디버깅 부담을 줄이고 신뢰성이 있는 병렬 프로그램의 개발을 가능하게 한다.

키워드 : 메시지전달 프로그램, 메시지경합, 디버깅, MPIRace-Check

MPIRace-Check V 1.0: A Tool for Detecting Message Races in MPI Parallel Programs

Mi-Young Park[†] · Sang-Hwa Chung[††]

ABSTRACT

Message races should be detected for debugging effectively message-passing programs because they can cause non-deterministic executions of a program. Previous tools for detecting message races report that message races occur in every receive operation which is expected to receive any messages. However message races might not occur in the receive operation if each of messages is transmitted through a different logical communication channel so that their incorrect detection makes it a difficult task for programmers to debug programs. In this paper we suggest a tool, MPIRace-Check, which can exactly detect message races by checking the concurrency between send/receive operations, and by inspecting the logical communication channels of the messages. To detect message races, this tool uses the vector timestamp to check if send and receive operations are concurrent during an execution of a program and it also uses the message envelop to inspect if the logical communication channels of transmitted messages are the same. In our experiment, we show that our tool can exactly detect message races with efficiency using MPI_RTED and a benchmark program. By detecting message races exactly, therefore, our tool enables programmers to develop reliable parallel programs reducing the burden of debugging.

Keyword : Message-passing Programs, Message Races, Debugging, MPIRace-Check

1. 서 론

메시지전달 프로그램(message-passing program)[6, 11, 12]은 분산된 메모리를 가진 병렬 컴퓨팅 환경에서 메시지의 전달로써 프로세스간의 통신을 수행하는 병렬 프로그램을 말한다. 프로그램 수행 중 프로세스들 간에 전송되는 메시지들은 네트워크의 통신지연이나 프로세스 스케줄링(scheduling)에 의해서 도착하는 순서가 달라질 수 있으며, 이러한 메시지들의 경합은 프로그램의 의도되지 않은 비결정적인 수행 결과를 초래하여, 병렬 프로그램의 정확성을 검증하고 재 수행을 이용한 디버깅 작업을 매우 어렵게 한다.

메시지경합(message race)[9, 10, 12]은 동일한 논리적 통신채널로 두 개 이상의 메시지들이 병행하게 전송될 때 발생한다. 성능향상을 위해서 프로그래머가 의도적으로 메시

※ 이 논문은 부산대학교 자유과제 학술연구비(2년)에 의하여 연구되었음
† 정 회 원 : 부산대학교 BK21 U-Port정보기술산학공동사업단 포닥연구원
†† 정 회 원 : 부산대학교 정보컴퓨터공학부 교수(교신저자)
논문접수 : 2007년 12월 11일, 심사완료 : 2008년 1월 26일

지경합을 가지는 프로그램을 작성하기도 하지만, 메시지들의 경합으로 인한 프로그램의 비결정적 수행(nondeterministic behavior)[6]은 기존의 재 수행을 이용한 디버깅 작업을 매우 어렵게 할 뿐만 아니라, 병렬 프로그램의 모든 수행에 대한 검증을 어렵게 한다. 그러므로 메시지전달 프로그램의 오류를 검사하고 신뢰성이 있는 프로그램의 개발을 위하여 프로그램의 비결정적 수행을 초래하는 메시지경합은 반드시 탐지되어야 한다.

메시지전달 프로그램의 개발을 위한 라이브러리로는 산업적으로 표준화되어 널리 사용되는 MPI(Message Passing Interface)[3, 11]가 있다. MPI 병렬 프로그램에서 메시지경합을 탐지하는 기존의 도구로는 MAD[7], MARMOT[4, 5], MPVisualizer[1] 등이 있다. 이들 도구들은 MPI로 작성된 병렬 프로그램의 수신사건에서 임의의 메시지를 수신하도록 하는 mpi_any_source가 사용되면 무조건 해당 수신사건에서 경합이 발생한다고 보고하거나, 전송되는 메시지들의 논리적인 통신채널을 검사하지 않기 때문에 탐지결과가 정확하지 않다. 또한 프로그래머가 MPI 프로그램의 병행성을 높이기 위해 의도적으로 mpi_any_source를 사용하기도 하며, 때로는 mpi_any_source가 사용되더라도 경합이 존재하지 않을 수도 있다. 따라서 기존 도구들의 탐지는 부정확하며, 이것은 프로그래머의 디버깅 작업을 더욱 어렵게 한다.

본 연구에서는 MPI로 작성된 병렬 프로그램에서 메시지경합을 정확하게 탐지하여 상세한 디버깅 정보를 제공하는 MPIRace-Check 도구를 제시한다. 본 도구는 vector timestamp[2, 8]와 메시지 부가정보(message envelop)를 이용하여 MPI 프로그램 수행 중에 발생하는 모든 경합을 정확하게 탐지한다. vector timestamp는 프로그램 수행 중에 메시지를 전송하는 송수신 사건들 간의 병행성을 검사하기 위해 사용되고, 메시지 부가정보는 병행하게 전송되는 메시지들이 동일한 논리적 통신채널로 전송되는지 검사하기 위해 사용된다. 또한 탐지된 메시지경합의 효과적인 디버깅을 위하여 경합에 참여한 프로세스, 메시지, 그리고 오류를 포함한 원시코드에 관한 상세한 정보 등을 사용자에게 보고한다.

본 도구는 표준 C언어와 MPI의 Profiling Interface[11]를 이용하여 사용자 라이브러리로 구현되었으며, 사용자는 프로그램 원시코드를 변경하지 않고 본 라이브러리를 연결하여 존재하는 메시지경합을 탐지할 수 있다. 본 도구는 MPI-1에서 제공하는 모든 일대일 통신(point-to-point communication) 함수에 적용될 수 있으며, 실험에서는 MPI_RTED (MPI Run Time Error Detection Test Suites)[13]를 이용하여 본 도구의 정확성을 평가하였고, 벤치마크 프로그램을 이용하여 본 도구의 효율성을 평가하였다. 실험에서 본 도구는 일대일 통신을 사용하는 모든 MPI_RTED 테스트 프로그램에서 존재하는 메시지경합을 탐지하였고, 본 도구로 인한 시간적 오버헤드는 최악의 경우가 35%이내였다.

이어지는 2절에서는 메시지전달 프로그램에서 발생하는 메시지경합의 개념과 이러한 경합을 탐지하는 기존 도구의 특징 및 문제점에 대해서 설명한다. 3절에서는 정확한 메시지경합 탐지를 위한 vector timestamp와 메시지 부가정보에 대해서 설명하고, 이들을 이용한 탐지 기법을 제시한다. 4절에서는 MPI_RTED 프로그램과 벤치마크 프로그램을 이용하여 본 도구의 정확성과 효율성을 평가하고, 마지막으로 5절에서 결론을 내린다.

2. 연구배경

2.1 프로그램 모델

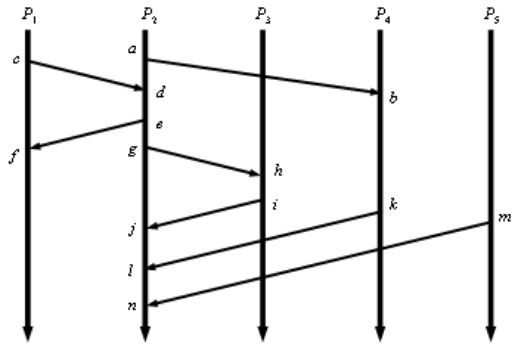
메시지전달 프로그램[6, 11, 12]의 수행은 수행 중에 발생하는 사건들의 집합과 그들 간의 순서관계(happened-before)[2, 8]로 표현될 수 있다. 모든 수행에서 사건 a가 사건 b보다 항상 먼저 수행하는 경우 그들 간의 순서관계는 "(a 발생 후 b) 관계가 만족된다."고 하며, 이를 $a \rightarrow b$로 표기한다. 예를 들어 하나의 프로세스 내에서 순서적으로 발생하는 두 사건 $\{a, b\}$가 존재하는 경우에 $(a \rightarrow b \vee b \rightarrow a)$가 만족된다. 다른 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.

메시지경합[9, 10, 12]은 동일한 논리적 통신채널로 두 개 이상의 메시지들이 병행하게 전송될 때 발생하며, 네트워크의 통신지연이나 프로세스 스케쥴링에 의해서 경합하는 메시지들의 도착순서가 달라진다. 이러한 메시지경합은 임의의 수신사건 r과 경합하는 메시지들의 집합 M으로 표현되며, $<r, M>$으로 표기한다. 이때 M에 속한 모든 메시지들은 r에 수신가능하며, M에 속한 임의의 메시지 송신사건 s는 $s \nrightarrow r$ 그리고 $r \nrightarrow s$를 만족한다. 그리고 송신사건 s가 송신한 메시지를 $msg(s)$로 나타낸다.

(그림 1)은 임의의 메시지전달 프로그램의 수행 중에 발생한 사건들 간의 부분적 순서관계를 나타낸 것이다. 그림에서 세로 화살표는 시간적 흐름에 따른 프로세스의 수행을 나타내고, 프로세스 간에 비스듬히 그려진 화살표는 메시지의 전송을 나타내며, 모든 메시지는 동일한 논리적 통신채널로 전송됨을 가정한다. 그림에서 두 프로세스 P_3과 P_4가 메시지 $msg(i)$와 $msg(k)$를 프로세스 P_2로 송신한다. 이때 송신사건 k는 P_2의 수신사건 j에 대해서 $\{ k \nrightarrow j \wedge j \nrightarrow k\}$를 만족하므로, 메시지 $msg(k)$는 P_2의 수신사건 j로 경합한다. 또한 프로세스 P_5에서 송신된 메시지 $msg(m)$도 수신사건 j로 경합한다. 따라서 프로세스 P_2의 수신사건 j에서 발생한 경합은 메시지들의 집합 $J = \{msg(k), msg(m)\}$와 이들 메시지들 중 하나를 처음으로 수신하는 사건 j로 구성되므로 $<j, J>$로 나타낼 수 있다.

2.2 기존 연구

MPI 병렬 프로그램에서 메시지경합을 탐지하는 대표적인

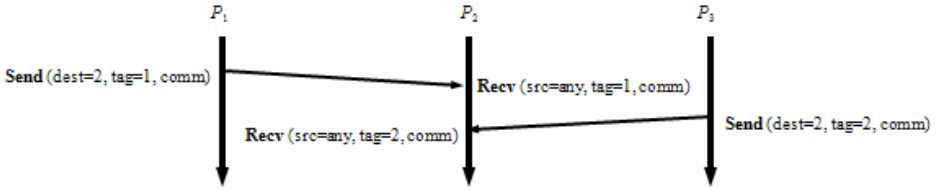


(그림 1) 메시지경합의 예

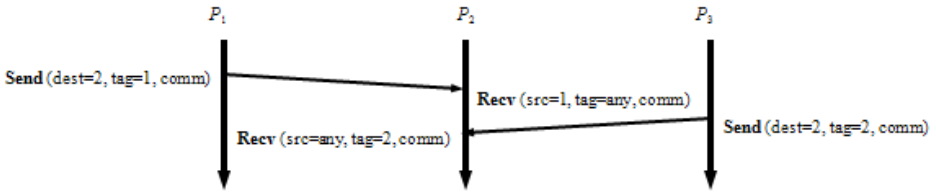
도구로는 MAD[7], MARMOT[4, 5], MPVisualizer[1] 등이 있다. MAD는 멀티 프로세스에 대한 중단점 기능과 변수 값 검사, 재수행 등과 같은 기능을 제공하고, MARMOT는 MPI[3, 11] 프로그램 수행 시 발생하는 전형적인 오류 탐지 기능 외에 데드락(deadlock) 탐지, 경합탐지(race detection) 기능을 제공한다. MPVisualizer는 MPI 프로그램 수행 시 발생하는 사건 정보를 추적파일로 만들어 재수행하는 기능을 제공한다. 또한 효과적인 디버깅을 위한 시각화를 제공할 뿐만 아니라 경합의 발생도 알려준다.

메시지경합을 탐지하는 측면에서 MARMOT은 MPI로 작성된 병렬 프로그램의 수신사건에서 mpi_any_source가 사용되면 무조건 해당 수신사건에서 경합이 발생한다고 보고한다. 그러나 MPI 프로그램의 병행성을 높이기 위해서 프로그래머가 의도적으로 mpi_any_source를 사용할 뿐만 아니라, mpi_any_source가 사용되더라도 경합이 존재하지 않을 수도 있다. 그 외 다른 도구들은 vector timestamp를 사용하여 병행성을 검사함으로써 MARMOT보다는 정확한 탐지를 제공하지만, 논리적 통신 채널의 동일 여부를 검사하지 않기 때문에 여전히 탐지가 정확하지 않다.

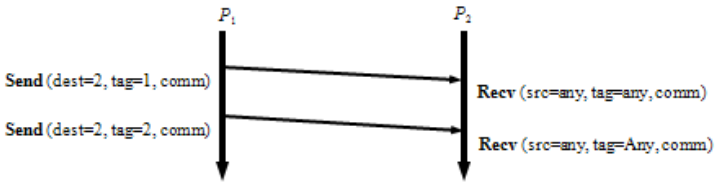
(그림 2)는 간단한 MPI 프로그램의 수행을 통해서 mpi_any_source를 사용한 수신사건에서 경합이 발생하지 않는 경우와 발생하는 경우를 보여준다. 그림에서 각 송수신사건의 파라매타(parameter)로는 메시지의 수신여부를 결정하는 메시지 부가정보(message envelop)만을 표현하였다. 메시지 부가정보는 소스(source), 목적지(destination), 태그(tag), 커뮤니케이터(communicator) 등 네 가지로 구성된다. 부가정보들 중에서 목적지, 태그, 커뮤니케이터는 메시지를 보내는 송신사건의 파라매타 값으로부터 알 수 있고, 소스는 메시지 송신자의 식별자로 알 수 있다. 메시지를 송신하



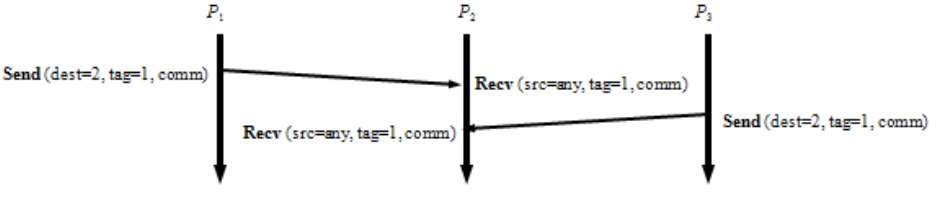
(a) 경합 없음: 서로 다른 tag 사용



(b) 경합 없음: 하나의 mpi_any_source



(c) 경합 없음: MPI의 non-overtaking 보장



(d) 경합 발생: 두 개 이상의 mpi_any_source와 동일 tag

(그림 2) mpi_any_source와 경합의 발생 관계

는 Send()에서는 파라매타 값으로 목적지, 태그, 커뮤니케이터만을 표현하였고, 메시지를 수신하는 Recv()에서는 소스, 태그, 커뮤니케이터를 표현하였다. 송신사건에서는 메시지를 송신할 때 해당하는 메시지 부가정보를 메시지에 첨부하여 보내고, 수신사건에서는 명시된 소스, 태그, 커뮤니케이터가 일치하는 부가정보를 가진 메시지만을 수신한다.

(그림 2)의 (a)에서 P_2의 두 수신사건은 메시지의 소스로 mpi_any_source를 사용함으로써 임의의 프로세스로부터 송신된 메시지를 받는다. 그러나 두 수신사건에서 사용하는 태그는 서로 다르기 때문에 첫 수신사건은 항상 P_1이 송신한 메시지를 수신하고, 두 번째 수신사건은 항상 P_3의 메시지를 수신한다. 따라서 두 수신사건에서 mpi_any_source를 사용하더라도 서로 다른 태그로 인하여 두 메시지들 간의 경합은 존재하지 않는다.

(그림 2)의 (b)에서 P_2의 첫 수신사건은 P_1의 메시지를 받겠다고 명시하였고, 두 번째 수신사건은 메시지의 소스로 mpi_any_source를 사용하였다. 그러나 첫 수신사건이 메시지의 소스로 1을 명시했기 때문에 항상 P_1의 메시지만을 수신할 것이며, 결과적으로 두 번째 수신사건은 P_3의 메시지만을 수신하게 된다. 따라서 예제 (b)에서도 경합이 존재하지 않는다.

예제 (c)에서는 두 수신사건 모두가 mpi_any_source와 mpi_any_tag를 사용하고 있다. 그러나 두 메시지 모두 항상 보내진 순서대로 P_2에 수신된다. 왜냐하면, MPI는 메시지들 간의 non-overtaking을 보장하는데, 이것은 한 프로세스에서 연속적으로 동일한 다른 프로세스로 메시지를 송신할 때 그들은 송신된 순서대로 목적지 프로세스에 수신됨을 말한다. 따라서 (c)예제에서 모든 수신사건이 메시지 소스로 mpi_any_source와 mpi_any_tag를 사용하지만 경합이 존재하지 않는다.

마지막으로 (그림 2)의 (d)는 경합이 발생하는 예를 보여준다. 예제에서 P_2의 두 수신사건은 모두 메시지 소스로 mpi_any_source를 사용하고, 태그도 동일한 값 1이 명시되었다. 따라서 P_2의 두 수신사건은 P_1 또는 P_3이 송신한 임의의 메시지를 수신할 수 있으며, 프로세서의 스케쥴링이나 네트워크의 지연에 따라 그들의 도착순서가 달라지고, 결과적으로 프로그램의 수행결과도 비결정적이다. 이와 같이 소개된 예제를 통해서 수신사건이 mpi_any_source를 사용하더라도 메시지경합이 존재하지 않을 수 있음을 알 수 있다.

3. 경합탐지 알고리즘

메시지경합은 두 개 이상의 메시지들이 “동일한 논리적인 통신채널”로 “병행”하게 전송될 때 발생한다. 여기서 논리적인 통신채널은 MPI에서 메시지의 부가정보를 의미하고 “병행”하게 전송된은 메시지들을 송수신하는 두 사건간의 “순서관계”가 성립되지 않음을 뜻한다. 따라서 프로그램 수행 중에 메시지경합을 정확히 탐지하기 위해서는 vector timestamp를 이용하여 송수신사건들 간에 순서관계가 존재하지 않는지를 검사하고, 메시지 부가정보를 이용하여 메시지들이 동일한 통신채널로 전송되는지도 검사해야 한다.

예를 들어 (그림 2)의 모든 예제에서 메시지를 송신하는 두 사건들은 모두 순서관계 없이 병행하게 수행되며, 동일한 수행 형태를 가지기 때문에 그들의 vector timestamp은 모두 동일하다. 그러나 앞서 설명했듯이 세 개의 예제 프로그램에서는 메시지경합이 존재하지 않는데, 그 이유는 메시지 부가정보 즉 논리적 통신채널이 서로 다르기 때문이다. 따라서 본 논문에서는 메시지 부가정보를 이용한 논리적 통신채널별로 vector timestamp를 관리하고 병행성을 검사하여 메시지경합을 탐지한다.

(그림 3)은 논리적 통신채널별로 송수신사건마다 갱신되는 vector timestamp와 병행성을 검사하여 메시지경합을 보고하는 알고리즘을 보인다. 알고리즘에서 *timestamp*는 프로세스 수만큼의 크기를 가지는 배열구조로서 각 송수신사건에서의 vector timestamp을 나타내며, *channel*은 메시지 부가정보로 표현된 논리적인 통신채널을 나타낸다. (그림 3)의 (a)는 vector timestamp을 위한 초기화 알고리즘으로 프로그램 수행 시 단 한번 수행된다. 초기화 알고리즘에서 *size*는 프로그램 수행에 참여하는 프로세스의 개수이며, *localclock*은 한 프로세스 내에서 발생하는 송수신 사건을 식별하는 변수로서, 매 송수신 사건마다 1씩 증가한다. 또한 0으로 초기화되는 각각의 *timestamp*, *prerecv*, *sender*은 *size* 크기의 배열 변수로서, *timestamp*은 각 프로세스가 수행 중인 가장 최근의 송수신 사건에 대한 vector timestamp를 나타내고, *prerecv*는 각 프로세스의 *timestamp* 이전에 발생한 수신사건에 대한 vector timestamp을 나타낸다. 마지막으로 *sender*는 각 프로세스가 수행 중인 수신사건에 대응되는 송신사건의 vector timestamp를 나타낸다.

(그림 3)의 (b)는 매 송신사건에서 수행되는 알고리즘으로써, *pid*는 현재 송신사건을 수행하는 프로세스 자신을 가리킨다. 메시지를 송신하는 사건이 발생할 때마다 프로세스는 자신의 *localclock*을 1씩 증가시키고, *timestamp*에서 자신의 *pid*에 대응되는 항목을 새로운 *localclock*으로 갱신한다. 그리고 갱신된 *timestamp*는 메시지와 함께 송신된다. 이렇게 송신된 메시지는 수신사건에서 사용자의 메시지와 vector timestamp로 분리되고, 추출된 vector timestamp는 (그림 3)의 (c)와 (d) 알고리즘에서 송신자의 vector timestamp인 *sender*로 사용된다.

(그림 3)의 (c)는 매 수신사건에서 수행되는 알고리즘으로써, 먼저 해당 수신사건에서 경합이 발생하였는지를 검사하기 위해 (그림 3)의 (d) 알고리즘을 호출한다. 병행성 검사를 위한 **CheckConcurrency** 알고리즘은 이전 수신사건(*prerecv*)의 vector timestamp와 수신된 메시지에서 추출한 송신사건의 vector timestamp(*sender*)에서 자신의 *pid*에 해당하는 값을 비교한다. 이때 이전 수신사건(*prerecv*)값이 크면 송신사건이 송신한 메시지는 현재 수신사건 뿐만 아니라 이전 수신사건으로도 수신될 수 있음을 의미하므로 메시지경합을 보고한다.

```
00: TimestampInit()
01: localclock := 0
02: for i from 0 to size do
03:     timestamp[channel][i] := 0
04:     prerecv[channel][i] := 0
05:     sender[channel][i] := 0
06: end for
```

(a) 초기화 함수

```
00: TimestampInSend()
01: localclock := localclock + 1
02: timestamp[channel][pid] := localclock
```

(b) 송신사건에서 수행

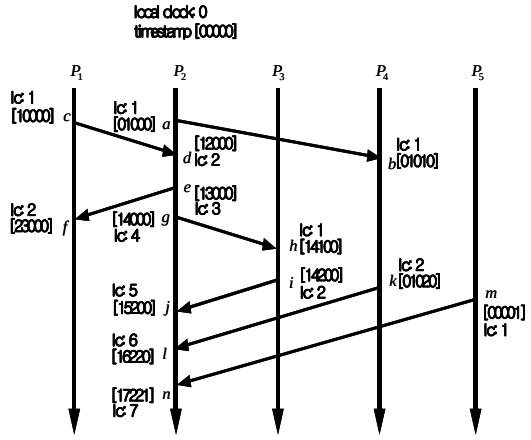
```
00: TimestampInRecv()
01: call CheckConcurrency()
02: for i from 1 to size do
03:     timestamp[channel][i] := max(timestamp[channel][i],
                                     sender[channel][i])
04: end for
05: localclock := localclock + 1
06: timestamp[channel][pid] := localclock
07: prerecv := timestamp
```

(c) 수신사건에서 수행

```
00: CheckConcurrency()
01: if prerecv[channel][pid] > sender[channel][pid]
02:     report a message race
03: end if
```

(d) 병행성 검사

(그림 3) vector timestamp 알고리즘



(그림 4) vector timestamp의 적용 예

알고리즘 (c)의 첫 라인에서 경합 여부를 검사한 후, 2행부터 4행 사이에서는 송신사건(*sender*)의 vector timestamp와 비교하여 큰 값을 취함으로써 자신의 *timestamp*을 갱신한다. 그리고 자신의 *localclock*을 1씩 증가시켜, *timestamp*의 *pid*에 해당하는 항목을 *localclock*으로 갱신한다. 마지막으로 다음 수신사건을 위해서 현재 수신사건 정보를 *prerecv*에 저장한다.

(그림 4)는 (그림 1)의 수행에서 각 송수신사건에 대한 vector timestamp을 보여주고 있다. 각 송수신 사건에서 *localclock*은 *lc*로 표현되었고, 해당 사건에서의 vector timestamp은 '[]'안에 표현되었다. 프로세스 P_2의 송신사건 *a*에서 **TimestampInSend**는 P_2의 *localclock*을 1 증가시키고, *timestamp*의 두 번째 항목을 현재의 *localclock*으로 갱신함으로써 *timestamp*는 [01000]이 된다. 이때 갱신된 [01000]은 메시지와 함께 송신된다. 프로세스 P_4의 수신사건 *b*에서는 **TimestampInRecv**가 호출되어 병행성을 검사하고, 송신자(*sender*)의 vector timetamp와 자신의 *timestamp*을 비교하여 큰 값으로 갱신함으로써, *timestamp*은 [01000]이 된다. 그리고 자신의 *localclock*을 1증가 시키고, P_4에 대응되는 네 번째 항목을 *localclock*으로 갱신함으로써 [01010]이 된다. 이러한 방식으로 각 송수신 사건에서 vector timestamp을 갱신한다.

이와 같이 송수신사건에서 vector timestamp를 갱신함으로써 각 송수신 사건은 고유한 *timestamp*을 가지며, 이것은 메시지전달 병렬 프로그램에서 수행된 사건들 간의 부분적 순서관계를 나타낸다. 이를 역이용하면 송수신 사건들 간의 병행적 관계를 알 수 있기 때문에 병행하게 송신되어 경합하는 메시지들을 탐지할 수 있다. 예를 들어 (그림 4)에서 P_3의 수신사건 *h*의 *timestamp*인 [141000]은 P_3의 첫 번째 사건이 P_1의 첫 번째 사건과 P_2의 4번째 사건과 순서관계에 있음을 나타낸다. 이를 다르게 해석하면 P_3의 *h*는 P_1의 2번째 사건과는 순서적이지 않으며, 만약에 P_1의 2번째 사건이 메시지를 P_3로 송신하면, 이것은 P_2의 *h*에서 다른 메시지와 경합함을 의미한다.

메시지경합을 탐지하기 위해서 각 수신사건에서는 **CheckConcurrency()** 알고리즘을 호출하여 병행성을 검사한다. 각 수신사건에서 이전 수신사건(*prerecv*)과 현재 메시지를 송신한 사건(*sender*)을 비교하여, 이전 수신사건이 큰 값을 가지면 이전 수신사건과 현재 수신사건으로 보내진 두 메시지가 서로 경합한다고 보고한다.

예를 들어 프로세스 P_2의 수신사건 *j*에서 **CheckConcurrency()**를 호출했을 때 이전 수신사건(*prerecv*)은 P_2의 수신사건 *d*이고, 수신사건 *j*에 대응되는 송신사건은 P_3의 *i*이다. 두 사건 *d*와 *i*의 *timestamp*은 [12000]와 [14200]이며, 이들의 두 번째 (수신사건 *j*는 P_2에서 발생) 항목을 비교하면, 이전 수신사건이 작기 때문에 경합이 보고 되지 않는다(2<4). 다음으로 P_2의 수신사건 '*l*'에서 병행성을 검사하면, 이전 수신사건 *j*의 *timestamp*은 [15200]이고, 현재 수신사건으로 메시지를 보낸 송신사건 *k*의 *timestamp*는 [01020]이다. 이들 사건 *j*와 *k*의 *timestamp*의 두 번째 항목을 비교하면(5>1), 이전 수신사건이 크므로 메시지경합을 보고한다.

4. 탐지도구의 구현 및 실험

본 연구에서는 MPI[3, 11]으로 작성된 병렬 프로그램에서

```
00:   MPI_Send(buf, count, datatype, dest, tag, comm)
01:   {
02:     TimestampInSend();
03:     MPI_Pack(timestamp, size, MPI_INT, buffer, buffersize, pos, comm);
04:     MPI_Pack(buf, count, datatype, buffer, buffersize, pos, comm);
05:     PMPI_Send6(buffer, pos, MPI_PACKED, dest, tag, comm);
06:   }
```

(a)

```
00:   MPI_Recv(buf, count, datatype, source, tag, comm, status)
01:   {
02:     PMPI_Recv(buffer, buffersize, MPI_PACKED, source, tag, comm, status);
03:     MPI_Unpack(buffer, buffersize, pos, sender, size, MPI_INT, comm);
04:     MPI_Unpack(buffer, buffersize, pos, buf, count, datatype, comm);
05:     TimestampInRecv();
06:   }
```

(b)

(그림 5) MPI Profiling Interface

메시지경합을 정확하게 탐지하여 상세한 디버깅 정보를 제공하는 MPIRace-Check 도구를 구현하였다. 본 도구는 표준 C언어와 MPI의 Profiling Interface를 이용하여 사용자 라이브러리로 구현되었으며, 사용자는 프로그램 원시소스를 변경하지 않고 본 라이브러리를 연결하여 존재하는 메시지경합을 탐지할 수 있다. MPI Profiling Interface는 일반 사용자가 MPI의 함수 호출에 개입하여 별도의 작업을 수행하도록 허락하는 인터페이스로서, 모든 MPI 함수들이 또 다른 이름으로 호출이 가능하다. 즉 모든 형태의 MPI_xxx은 또 다른 이름의 PMPI_xxx로의 호출로 대체 가능하며, 이를 이용하여 사용자는 그들 자신만의 MPI_xxx를 구현할 수 있다.

본 연구에서는 일대일 통신에 관련된 모든 MPI_xxx 내에 vector timestamp[2, 8]을 생성하고 유지하는 알고리즘과 탐지 알고리즘을 삽입하여 MPI-1의 모든 통신 함수를 재구현하였다. (그림 5)는 재구현된 MPI 함수들 중에서 **MPI_Send()**와 **MPI_Recv()**를 보여주고 있다. 사용자 프로그램이 **MPI_Send()**를 호출하면 먼저 재구현된 (그림 5)(a)의 **MPI_Send()**가 호출된다. 재구현된 **MPI_Send()**는 메시지를 송신하기 전에 **TimestampInSend()**를 호출하여 *timestamp*을 갱신한다. 그리고 **MPI_Pack()**을 호출하여 *timestamp*와 사용자의 메시지(*buf*)를 하나의 자료구조로 연결한 후 **PMPI_Send()**을 통하여 메시지 송신한다.

이와 유사하게, 사용자 프로그램에서 **MPI_Recv()**를 호출하면 재구현된 (그림 5)(b)의 **MPI_Recv()**가 호출되고, **PMPI_Recv()**를 통해서 메시지를 수신한다. 다음에 **MPI_Unpack()**을 이용하여 사용자 메시지(*buf*)와 *timestamp*을 분리한 후, **TimetampInRecv()**을 호출함으로써 경합을 탐지하고 자신의 *timestamp*을 갱신하다. 이때 메시지경합이 탐지되면, gdb을 호출하여 탐지된 경합에 관련된 상세한 정보를 추출하여 사용자에게 보고한다. 이와 같이 MPI Profiling Interface을 이용하여 사용자가 프로그램을 수정하지 않고 본 도구를 이용하여 메시지경합을 탐지할 수 있도록 하였다.

본 연구는 MPICH[3, 11]가 설치된 분산 시스템에서 본 도구의 정확성과 효율성을 실험하였다. 정확성 평가를 위한 실험에서는 MPI_RTED (MPI Run Time Error Detection Test Suites)[13]를 이용하여 본 도구가 모든 테스트 프로그램에서 메시지경합을 탐지하는지를 실험하였다. MPI_RTED는 MPI 프로그램의 오류탐지 도구들을 평가하기 위해서 Iowa State University의 High Performance Computing Group에서 개발한 테스트 프로그램 패키지이다. MPI_RTED에서는 MPI 프로그램에서 발생 가능한 런타임 오류들이 유형별로 분류되어 있으며, 각 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 개의 언어로 작성되었으며, 본 실험에서는 MPI-1의 통신함수들로만 구성되어 메시지경합을 가진 테스트 프로그램을 사용하였다.

구현된 라이브러리의 이름은 MPIRace-check이며, MPI 프로그램을 컴파일하는 명령은 다음과 같다.

>> mpicc -g -o *목적파일이름* *소스프로그램이름*.c -lMPIRace-check

MPI_Profiling Interface내에서 gdb를 호출하기 위해서는 컴파일 옵션 -g를 사용해야 하며, 컴파일 후 프로그램을 수행하는 명령은 다음과 같다.

>> mpirun -np 5 *목적파일이름*

옵션 -np 다음에는 수행될 프로세스 개수를 명시하는데, 위 예제에서는 5개의 프로세스가 해당 프로그램을 수행한다.

<표 1>은 본 도구를 MPI_RTED에 적용했을 때 메시지경합의 탐지여부를 보여주고 있다. 첫 번째 칸은 실험에 사용된 테스트 프로그램의 이름이며, 두 번째 칸은 해당 테스트 프로그램에서 사용된 MPI 통신 함수를 나타낸다. 그리고 마지막 칸은 본 도구의 탐지 여부를 보여준다. 표에서와 같이 본 기법이 모든 테스트 프로그램에서 메시지경합을 탐지함을 확인하였다.

```
WARNING(RaceCondition):
The message which was sent at 'l' from 'P_2' is racing toward 'l' receive event in 'P_l'
(the current channel is -2-l)  at c_B_l_l_a_Ml.c:76
>>          MPI_Recv(&recvbuf_2, l, MPI_INT, MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, &status);
```

(그림 6) 에러 메시지의 예

〈표 1〉 MPI__RTED에서의 메시지경합 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

(그림 6)은 테스트 프로그램들 중 c_B_1_1_a_M1.c에서 본 도구가 메시지경합을 탐지했을 때 보이는 에러 메시지이다. 탐지된 메시지경합에 대한 에러 메시지에서는 경합이 발생한 프로세스, 경합하는 메시지를 송신한 프로세스, 통신채널, 프로그램 이름, 경합이 발생한 원시코드의 위치정보, 그리고 원시코드 정보를 제공한다. (그림 6)의 탐지결과를 살펴보면, P_2의 *localclock*이 1인 송신사건이 보낸 메시지가 P_1의 *localclock*이 1인 수신사건에서 경합함을 보인다. 또한 경합이 발생한 수신사건의 위치가 원시코드 상에서 76 라인임을 보여준다.

이와 같이 경합이 탐지될 때마다 메시지경합이 발생한 위치와 경합에 참여한 프로세스 등의 상세한 정보를 프로그래머에게 알려줌으로써, 디버깅 작업을 보다 쉽게 시작할 수 있다. 예를 들어 프로그래머가 c_B_1_1_aM1.c의 76라인으로 곧바로 가서 mpi_any_source를 특정 프로세스로 지정하든지, 아니면 mpi_any_tag를 변경하여 메시지들이 결정적으로 도착하도록 수정할 수 있다. 또는 P_2가 보내는 메시지가 P_1의 수신사건으로 보내지지 않도록 수정하여 경합을 제거할 수도 있다. 그리고 이러한 메시지경합의 탐지는 경합하는 메시지들의 도착 순서를 임의로 조절하여 결정적 재수행을 이용한 디버깅 작업을 가능하게 할 수 있다.

본 도구의 효율성 평가를 위한 실험에서는 벤치마크 프로그램을 개발하여 MPI_Wtime()로 시간적 오버헤드를 측정하였다. 개발된 벤치마크 프로그램은 오직 MPI_Send()와 MPI_Recv() 통신 함수로만 구성되어 있고, 프로그램을 실행할 때 통신 함수의 개수를 파라메타로 전달할 수 있도록 하였다. 이 프로그램에서 순위(rank)가 0인 프로세스는 임의의 모든 메시지들을 수신하고, 다른 각각의 프로세서는 순위가 0인 프로세스에게 파라메타로 전달되는 개수만큼 메시지를 전달한다.

<표 2>는 벤치마크 프로그램에 본 도구를 적용하지 않았을 때와 적용 했을 때의 시간을 측정하여 초단위로 나타낸 것이다. 예를 들어 송수신 사건의 개수를 10000으로 했을 때 수행 시간이 0.168초였다면, 본 도구를 적용했을 때의 시간은 0.212초로서, 26%만큼의 시간적 오버헤드가 발생하였다. 송수신 사건이 10000000인 최악의 경우에도 시간적 오버헤드는 35%만 증가함으로써, 본 도구가 효율적으로 메시지경합을 탐지함을 알 수 있다.

〈표 2〉 시간적 오버헤드 측정

# of Send/Recv	Original Run Time (s)	Monitored Run Time (s)	Slowdown
10000	0.168	0.212	26%
100000	1.673	2.234	34%
1000000	16.399	22.034	34%
10000000	164.471	221.736	35%

5. 결 론

본 연구에서는 메시지 송수신 사건들 간의 병행성과 메시지들의 논리적 통신채널을 검사하여 메시지경합을 정확하게 탐지하여 상세한 디버깅 정보를 제공하는 MPIRace-Check를 제안하였다. 본 도구에서는 vector timestamp를 이용하여 프로그램 수행 중에 송수신 사건들 간의 병행성을 검사하고, 메시지 부가정보를 이용하여 메시지들의 논리적인 통신채널이 동일한지를 검사하여 메시지경합을 탐지한다. MPIRace-Check는 표준 C언어와 MPI의 Profiling Interface를 이용하여 사용자 라이브러리로 구현되었으며, 사용자는 프로그램 원시코드를 변경하지 않고 본 라이브러리를 연결하여 존재하는 메시지경합을 탐지할 수 있다. 본 도구의 정

확성과 효율성을 평가하는 실험에서는 일대일 통신 함수를 이용한 모든 MPI_RTED 프로그램에서 메시지경합이 탐지됨을 보였고, 벤치마크 프로그램에서 최악의 경우에도 시간적 오버헤드는 35%만 증가함을 보였다. 따라서 본 도구는 MPI 프로그램에서 메시지경합을 효율적으로 정확히 탐지함으로써, 프로그래머의 디버깅 부담을 줄이고 신뢰성이 있는 MPI 병렬 프로그램의 개발을 가능케 한다.

참 고 문 헌

[1] 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.

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

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

[4] Krammer, B., K. Bidmon, M.S. Muller, and M.M. Resch, "MARMOT: An MPI Analysis and Checking Tool," *In proceedings of PARCO'03*, 13: 493-500, Elsevier, Sept., 2003.

[5] 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.

[6] 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.

[7] 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.

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

[9] 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.

[10] 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'06)*, Lecture Notes in Computer Science, 3947: 187-196, Springer-Verlag, May, 2006.

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

[12] 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.

[13] HPC Group, MPI Run Time Error Detection Test Suites: http://rted.public.iastate.edu/MPI/, Iowa State University, USA, 2006.

박 미 영

e-mail : openmp@computer.org

1999년 동서대학교 컴퓨터공학과(학사)

2001년 경상대학교 컴퓨터과학과(석사)

2005년 경상대학교 컴퓨터과학과(박사).

2006년 Iowa State University 포닥연구원

2007년 전남대학교 BK21 유비쿼터스정보가전사업단 포닥연구원

2008년 부산대학교 BK21 U-Port정보기술산학공동사업단 포닥연구원

관심분야 : 운영체제, 병렬 컴퓨팅 시스템, Grid 및 유비쿼터스 컴퓨팅

정 상 화

e-mail : shchung@pusan.ac.kr

1985년 서울대학교 전기공학과(학사)

1988년 Iowa State Univ. 컴퓨터공학과(석사)

1993년 Univ. of Southern California 컴퓨터공학과(박사)

1993년~1994년 Univ. of Central Florida 컴퓨터공학과 조교수

1994년~현 재 부산대학교 컴퓨터공학과 교수, 컴퓨터 및 정보통신연구소 연구원

2002년~2003년 Oregon State Univ. 컴퓨터공학과 초빙교수

관심분야 : 클러스터 시스템, 병렬처리, TOE, RDMA, RFID