메시지전달 프로그램의 영향받지 않은 경합조건 탐지를 위한 경합상태 전이기법

(Race State Transition for Detecting Unaffected Race Conditions in Message-Passing Programs)

박 미 영 † 강 현 석 †† 전 용 기 ††
(Mi-Young Park) (Hyun-Syug Kang) (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 탐지된 경합의 상태를 프로그램의 수행 종료까지 수신하는 메시지들의 영향 여부에 따라 전이하는 새로운 기법을 제안한다. 본 기법은 경합을 탐지하고 그들간의 영향관계를 프로그램 종료까지 유지하므로, 영향받지 않은 경합만을 효율적으로 탐지한다.

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

Abstract Detecting unaffected race conditions is important to debugging message-passing programs effectively, because such a message race can affect other races to occur or not. The previous technique to detect efficiently unaffected races detects racing messages by halting at the receive event of the first race to occur in each process. However this technique does not guarantee that all of the detected races are unaffected, because halting such processes does disconnect some chain of affects-relations among those races. In this paper, we present a novel technique that manages the state of the detected race by examining if every received message is affected until the execution terminates. Our technique therefore guarantees to detect efficiently the unaffected races, because it maintains affects-relations of the races all along the execution of program.

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

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)[4-9]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[10,12]과 수행중(on-the-fly) 탐지기법[11,12]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(veri-

· 본 연구의 일부는 한국과학재단 목적기초연구(R05-2003-000-12345-0)의 지원으로 수행되었음
† 비 회 원 : 경상대학교 컴퓨터과학부
park@race.gnu.ac.kr
†† 종신회원 : 경상대학교 컴퓨터과학부 교수
컴퓨터정보통신연구소 연구원
hskang@gnu.ac.kr
jun@gnu.ac.kr
(Corresponding author임)
논문접수 : 2004년 2월 5일
심사완료 : 2006년 4월 26일

fication)하는 기법[11,9]과 영향받지 않은(unaffected) 경합을 탐지하는 기법[4,5,13,14]으로 나눌 수 있다.

영향받지 않은 경합을 가장 효율적으로 탐지하는 기존의 기법[14]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 중단하여 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 첫 수행에서 프로세스별로 가장 먼저 발생하는 경합의 위치 정보를 획득하고, 두 번째 수행에서는 그 위치 정보를 이용해서 탐지된 경합의 상태를 프로그램의 수행 종료까지 수신하는 메시지들의 영향 여부에 따라 전이하는 기법을 제안한다. 이 기법은 탐지된 경합의 상태 전이를 위해서 매 수신사건마다 다른 프로세스에게 송신할 영향관계 정보를 생성하고, 프로그램의 수행 종료까지 수신된 메시지의 영향 여부에 따라 탐지된 경합의 상태를 전이함으로써 영향받지 않은 경합만을 보고한다. 그러므로 본 기법은 경합을 탐지하고 그들간의 영향관계를 프로그램 종료까지 유지하므로, 프로세스별로 가장 먼저 발생하는 경합들 중에서 영향받지 않은 경합만을 효율적으로 탐지한다.

본 기법의 실험은 산업표준인 MPI(Message Passing Interface)[3]를 구현한 MPICH[15]를 네 개의 Alpha 프로세서 노드로 구성된 클러스터 시스템에 설치하여 수행하였다. 본 기법은 사용자 프로그램과의 투명성을 높이기 위해서 MPI Profiling Interface를 이용한 C 라이브러리로 구현되었으며, Mpptest[16] 등의 공개된 벤치마크 프로그램에 적용하여 정확성을 평가하였다. 실험 결과, 본 기법이 기존의 기법과는 달리 모든 영향받지 않은 경합을 탐지함을 확인하였다.

2절에서는 메시지전달 프로그램의 디버깅에서 영향받지 않은 메시지경합의 중요성을 소개하고, 이러한 경합을 탐지하기 위한 기존 기법들의 문제점을 살펴본다. 3절에서는 이러한 문제점을 해결하기 위해 메시지경합의 영향관계 정보를 생성하여 경합의 상태를 효율적으로 전이하는 새로운 경합탐지 알고리즘을 소개한다. 그리고 4절에서는 공용 벤치마크 프로그램을 이용하여 본 기법의 정확성을 평가한 실험 결과를 보인다. 마지막으로 5절에서는 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 효율적으로 탐지하는 기존 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[1,10,12]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 수신 여부를 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 그들간의 순서관계(happened-before)[17]로 표현될 수 있다. 모든 수행에서 사건 a가 사건 b보다 항상 먼저 수행하면, 그들간의 순서관계는 (a 발생후 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 수신사건 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[14,18]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[4-9]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_2와 P_4는 메시지 $msg(w_1)$와 $msg(w_2)$를 P_3으로 송신한다. 이때 두 송신사건 w_1과 w_2는 P_3의 수신사건 w에 대해서 $\{w \rightarrow w_1 \wedge w \rightarrow w_2\}$를 만족하지 않으므로, $msg(w_1)$과 $msg(w_2)$는 수신사건 w로 서로 경합한다. 따라서 P_3에 존재하는 경합은 경합하는 메시지들의 집합 $W = \{msg(w1), msg(w2)\}$와 이들 메시지를 수신 가능한 첫 수신사건 w로 구성되므로 $<w, W>$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 1(b)는 하나의 영향받지 않은 경합은 $<w, W>$, $W = \{msg(w_1), msg(w_2)\}$와 두 개의 영향받은 경합 $<x, X>$, $X = \{msg(x_1)\ msg(x_2)\}$, $<y, Y>$, $Y = \{msg(y_1), msg(y_2)\}$를 보이고 있다. 그리고 영향받은 메시지들은 여섯 개가 존재하면 점선으로 표시되어 있다. 여기서 $<x, X>$와 $<y, Y>$는 $<w, W>$로부터 영향받은 경합이다. 왜냐하면 $<x, X>$에 대해서 $w \rightarrow x_2$가 만족되고, $<y, Y>$에 대해서는 $w \rightarrow y$가 만족되기 때문이다. 영향받은 $<x, X>$와 $<y, Y>$의 발생은 $<w, W>$의 발생 결과에 의존적일 수 있으며, $<w, W>$을 디버깅하면 사라질 수도 있는 경합이다. 따라서 $<w, W>$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.

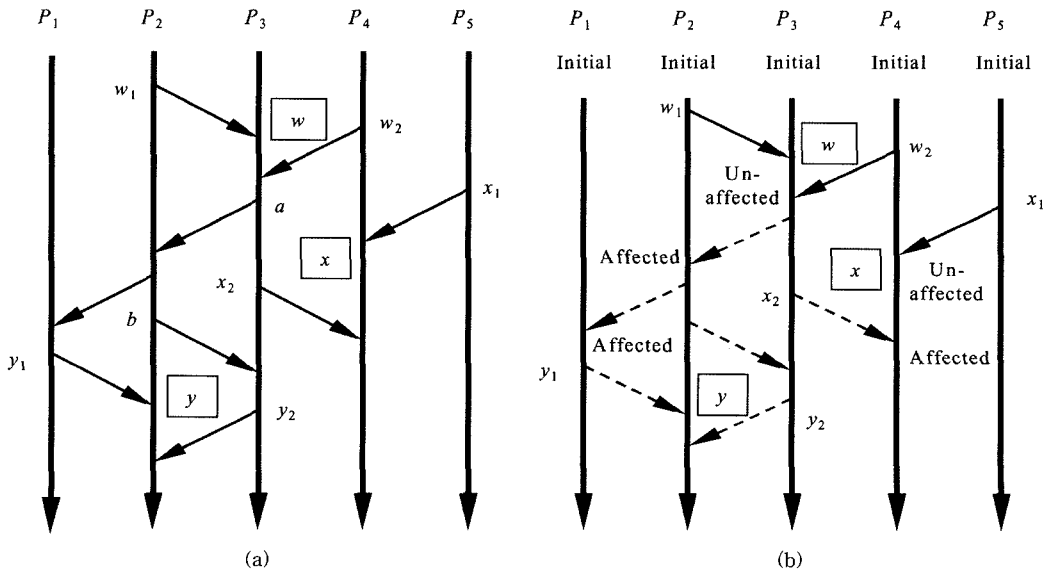

그림 1 메시지 경합

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 {<x, X>, <y, Y>}는 다른 지역적 최초경합 <w, W>로부터 영향을 받은 경합이다

2.2 관련 연구

메시지전달 프로그램에서 동적으로 경합을 탐지하는 기법은 탐지시점에 따라서 수행후(post-mortem) 탐지기법[10,12]과 수행중(on-the-fly) 탐지기법[9,11]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 수행 중에 재사용하므로 대규모의 병렬프로그램에서도 적용할 수 있는 현실적인 기법이다.

수행중 탐지기법은 탐지되는 경합의 성격에 따라서 경합의 존재 여부를 검증하는 기법[9,11]과 영향받지 않은 경합을 탐지하는 기법[4,5,13,14]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[4,13]과 MtOt(Multi-threads-at-One-time) 기법[5,14]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[14] 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 {$w \in P_3$, $x \in P_4$, $y \in P_2$}를 찾기 위한 *cutoff* 정보로서 {null $\in P_3$, $w_2 \in P_4$, $b \in P_2$}를 추적파일에 기록한다. 그리고 두 번째 수행에서는 {w, x, y} 위치에서 해당 프로세스의 실행을 중단하고, 경합하는 메시지들을 수신버퍼에 저장한다. 이때 P_2의 수신버퍼는 P_3의 수행 중단으로 비어 있는 상태가 되고, P_3의 수신버퍼는 W = {$msg(w_1)$, $msg(w_2)$}, P_4의 수신버퍼는 X = {$msg(x_1)$, $msg(x_3)$} 등으로 구성된다. 결과적으로 2-Pass 기법은 <w, W>와 <x, X>를 영향받지 않은 경합으로 보고한다. 그러나 그림 1에서 알 수 있듯이 <x, X>는 <w, W>로부터 영향받은 상태의 경합이다. 이러한 잘못된 탐지는 P_3의 수행이 w 위치에서 중단됨으로써 $msg(x_2)$가 P_4로 송신되지 못하여 두 경합간의 영향관계가 반영되지 않았기 때문이다. 그러므로 기존의 2-Pass 기법은 탐지된 모든 경합이 영향받지 않은 경합임을 보장하지 못한다.

경합을 탐지하는 대표적인 도구로는 MAD, MARMOT, MPVisualizer[7,20,21] 등이 있다. MAD[21]는 멀티 프로세스에 대한 중단점 기능과 변수 값 검사, 재수행 등과 같은 기능을 제공하고, MARMOT[7]는 MPI 프로그램 수행 시 발생하는 전형적인 오류 탐지 기능 외에 데드락(deadlock) 탐지, 경합탐지(race detection)

기능을 제공한다. MPVisualizer[10]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_SOURCE가 사용되면 무조건 경합으로 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. 영향받지 않은 경합의 탐지

본 절에서는 기존의 기법과 동일한 복잡도를 가지면서 각 프로세스에서 수신되는 메시지들의 경합 여부와 메시지에 포함된 영향관계 정보를 검사하여 영향받지 않은 경합만을 탐지하는 새로운 기법을 제안한다. 먼저 프로세스별로 발생하는 지역적 최초경합을 효율적으로 탐지하는 2-Pass 기법을 소개하고, Pass-2 알고리즘을 확장하여 탐지된 경합이 영향받지 않은 경합인지를 판단하는 기법을 소개한다.

3.1 지역적 최초경합의 탐지

본 기법은 지역적 최초경합을 효율적으로 탐지하기 위해서 매 수신사건에서 2-Pass 알고리즘을 수행한다. Pass-1에서는 프로그램 수행 중에 현재 프로세스에서 발생한 지역적 최초경합의 위치정보를 탐지하고, Pass-2에서는 Pass-1에서 탐지된 위치정보를 이용하여 지역적 최초경합이 발생된 수신사건과 경합하는 메시지들을 탐지한다.

Pass-1 알고리즘은 그림 2에서 보인다. 현재의 수신사건이 지역적 최초경합에 참여하는지를 판단하기 위해서는 현재의 수신사건이 경합에 참여하는지 여부와 그 경합이 현재 프로세스에서 처음으로 발생한 것인지 여부를 판단해야 한다. 먼저 그림의 1행에서는 현재 수신사건이 메시지를 수신한 채널과 동일한 채널에서 메시지를 수신했던 이전 수신사건의 위치를 *prevRecv*에 저장한다. 이것은 이전 수신사건과 현재 수신된 메시지의 송신사건간의 병행성을 2행에서 검사하여 현재의 수신사건이 경합에 참여하는지 여부를 판단하기 위해서이다. 이때 순서관계를 검사하기 위한 병행성 정보로는 벡터 타임스탬프[18,19]를 이용한다.

두 번째로, 송신사건이 보내온 벡터 타임스탬프에서 현재 프로세스에 대한 정보인 *Send*[*me*]와 처음으로 발생된 경합의 위치정보 *cutoff*를 비교하여 현재 탐지된 경합이 현재 프로세스에서 처음으로 발생한 것인지 여부를 판단한다. 만일 *Send*[*me*]가 *cutoff*보다 먼저 수행되었으면 탐지된 경합이 현재까지 프로세스에서 처음으로 발생된 경합인 경우이다. 이러한 경우에는 3행과 4행에서 *cutoff*를 *Send*[*me*] 값으로 수정하고, 처음으로 발생된 경합의 채널 정보 *firstChan*을 현재 수신된 채널 *thisChan* 값으로 수정한다. 예를 들면, 그림 1에서 <y, Y>에 대한 *cutoff*는 $msg(y_2)$를 수신한 사건이 수행될 때 P_2에서 y_2와 순서화된 가장 최근의 송신사건인 b의 위치이다.

마지막으로, 6-8행에서는 현재의 수신사건과 관련된 모든 채널 수만큼의 항목을 가진 *PrevBuf*를 현재 수신사건인 *Recv*[*me*] 값으로 갱신한다. 이것은 다음 수신사건에서는 현재 수신사건이 경합을 탐지하기 위해서 참고해야 하는 이전 수신사건 *prevRecv*가 되기 때문이다.

그림 3은 Pass-1에서 생성된 *cutoff*와 *firstChan*를 이용하는 Pass-2 알고리즘을 보인다. 이 알고리즘은 지역적 최초경합이 발생된 수신사건과 경합하는 메시지들을 탐지하고, 마지막에는 탐지된 경합의 상태를 전이하는 알고리즘을 호출한다. 그림의 1-2행에서는 지역적 최초경합의 위치정보 *cutoff*와 현재 수신 사건의 위치간의

```
00: CheckReceivePass1(Send)
01: prevRecv := PrevBuf[thisChan];
02: if (prevRecv ↛ Send) ∧ (Send[me] → cutoff) then
03:     cutoff := Send[me];
04:     firstChan := thisChan;
05: endif
06: for all i in Channels do
07:     PrevBuf[i] = Recv[me];
08: endfor
```

그림 2 Pass-1 알고리즘

```
00: CheckReceivePass2(Send, recv, Msg, cutoff, firstChan)
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 (firsRecv = ¬null ∧ firstRecv ↛ Send) then
09:         racingMsg := racingMsg ∪ Msg;
10:         racing := true;
11: endif
12: state := CheckRace(state, Msg[affecting], racing);
```

그림 3 Pass-2 알고리즘

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 *firstRecv*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.

그림 3의 8행에서는 *firstRecv*의 존재를 검사하고 수신한 메시지가 *firstRecv*로 경합하는 메시지인지를 결정한다. 이때 *firstRecv*가 null이 아니거나 메시지를 송신한 사건과 순서관계가 아니면, 지역적 최초경합의 첫 수신사건이 존재하고 수신된 메시지는 지역적 최초경합의 첫 수신사건으로 경합하는 메시지이다. 그러므로 8행의 조건이 만족되면 9행에서 수신된 메시지를 *racingMsg*로 수집하고, 수신된 메시지가 경합한다는 사실을 활용하기 위해서 *racing*을 참으로 설정한다. 12행에서는 메시지에 첨부된 영향관계 정보인 *Msg*(*affecting*), 현재의 상태 정보 *state*, 수신한 메시지의 경합 여부인 *racing* 등을 그림 4의 상태전이 알고리즘으로 전달하여 호출함으로써 탐지된 경합의 상태를 전이한다.

3.2 경합의 상태 전이

Pass-2 알고리즘을 적용하여 프로그램 수행이 시작되면, 각 프로세스는 경합이 발생되지 않은 초기적 상태이다. 이 상태에서 매 수신사건이 수행될 때마다 경합의 다음 상태를 결정하는데, 이때 중요한 두 가지 요소는 수신된 메시지에 포함된 영향관계 정보와 수신된 메시지의 경합 여부이다.

먼저, 수신된 메시지의 영향관계 정보는 임의의 지역적 최초경합이 다른 프로세스의 경합으로부터 영향을 받는 경우를 반영하기 위해서 중요하다. 예를 들어, 초기 상태에서 경합이 발생하기 전에 영향받은 메시지를 수신하면 이후에 발생되는 지역적 최초경합은 영향받은 상태의 경합이 되고, 영향받지 않은 상태의 경합도 이후에 영향받은 메시지를 수신하면 영향받은 상태의 경합이 된다.

두 번째, 메시지의 경합 여부는 임의 경합 $<r, M>$의 M을 결정하기 위해서 중요하다. 왜냐하면, $<r, M>$은 하나 이상의 영향받은 메시지가 M에 존재하는 경우에 영향받은 상태의 경합이 되기 때문이다. 예를 들어, 수신된 메시지가 영향받은 메시지이라 하더라도 r에 수신되기 위해서 경합하는 메시지가 아니라면, $<r, M>$의 상태는 전이되지 않는다. 왜냐하면 그 메시지는 경합 $<r, M>$의 발생 여부에는 영향을 미칠 수 없기 때문이다. 그러므로 매 수신사건마다 경합의 상태를 전이하기 위해서는 메시지에 첨부된 영향관계 정보뿐만 아니라 수신된 메시지의 경합 여부의 검사도 중요하다.

각 프로세스에서 발생한 경합이 가지는 상태는 초기 상태인 Initial, 영향받지 않은 상태인 Unaffected, 영향받은 상태인 Affected 등 세 가지이다. 프로그램 수행 시 모든 프로세스는 Initial 상태에서 전이를 시작한다. Initial 상태 이후로는 매 수신사건의 수행시에 경합 여부와 메시지의 영향여부에 의해서 경합의 다음 상태가 결정된다. Initial 상태에서 경합이 발생한 첫 수신사건이 수신한 메시지가 영향받은 메시지가 아니면 Unaffected 상태가 된다. 반면에 Initial 상태에서 경합의 발생 여부와 상관없이 영향받은 메시지를 수신하면 이후에 발생되는 지역적 최초경합은 영향받은 상태가 되므로 Affected 상태로 전이된다. Initial 상태에서 수신한 메시지가 영향받은 메시지가 아니거나 경합이 발생하지 않은 경우는 상태 전이는 이루어지지 않는다.

Unaffected 상태에서는 지역적 최초경합의 수신사건으로 경합하는 메시지가 영향받은 메시지인 경우에만 Affected 상태로 전이가 이루어지고 그렇치 않으면 Unaffected 상태로 남아 있다. 왜냐하면 Unaffected 상태의 $<r, M>$은 영향받은 메시지가 M에 속하는 경우에만 영향받은 상태가 되기 때문이다. Affected 상태에서는 어떤 경우에도 상태변화 없이 Affected 상태로 남아 있게 된다. 왜냐하면 한번 영향받은 상태의 경합은 영향받지 않은 상태의 경합으로 되돌아갈 수 없기 때문이다.

그림 4는 상태 전이를 위한 알고리즘이다. 알고리즘에서 *racing*은 수신된 메시지의 경합 여부를 나타내는 논리 변수이고, *affected*는 수신된 메시지에 첨부된 영향관계 정보를 나타내는 논리변수이다.

알고리즘의 (1-6)행에서는 Initial 상태에서 Unaffected 상태 또는 Affected 상태로의 전이가 이루어질 수 있는 경우를 보인다. 이때 Initial 상태에서 ($\neg$*affected* $\wedge$ *racing*)을 만족하면 Unaffected 상태로 전이하고, Initial 상태에서 *affected*를 만족하면 Affected 상태로 전이한다. 왜냐하면 Initial 상태는 $\neg$(*affected* $\vee$ *racing*)을 만족하는 상태이므로, 이 조건이 만족되지 않는 경우에만 상태 전이가 이루어지기 때문이다. 7행에서는 Unaffected 상태에서 Affected 상태로의 전이를 결

```
00: CheckRace(state, affected, racing)
01:   if (state = Initial ∧ ¬affected ∧ racing) then
02:       return Unaffected;
03:   endif
04:   if (state = Initial ∧ affected) then
05:       return Affected;
06:   endif
07:   if (state = Unaffected ∧ affected ∧ racing) then
08:       return Affected;
09:   endif
```

그림 4 경합의 상태 전이 알고리즘

정하는 것으로, (*affected* ∧ *racing*)을 만족하면 Affected 상태로 전이하고, 그 외의 조건에서는 상태 변화 없이 Unaffected 상태가 된다.

본 기법을 그림 1(a)에 적용했을 때 각 프로세스에서 발생한 지역적 최초경합과 영향받은 메시지들은 그림 1(b)에서 보인다. 그림에서 점선으로 표시된 메시지들은 영향 받은 메시지들을 나타내고 있다. 프로그램이 수행되면 각 프로세스는 Initial 상태로 시작한다. P_3의 경우에는 수신사건 w에서 지역적 최초경합 $<w, W>$가 발생하고 $msg(w_1)$이 영향받지 않은 메시지이므로, 경합 상태는 Initial 상태에서 Unaffected 상태로 전이한다. P_4의 경우에도 수신사건 x에서 지역적 최초경합 $<x, X>$가 발생하고 $msg(x_1)$이 영향받지 않은 메시지이므로, 경합 상태는 Initial 상태에서 Unaffected 상태로 전이한다. 그러나 그 후에 $msg(x_2) \in X$의 수신에서 경합 $<w, W>$로부터 영향받은 메시지를 수신하므로, Affected 상태로 전이한다. P_2의 경우는 지역적 최초경합 $<y, Y>$가 발생하기 전에 $<w, W>$로부터 영향받은 메시지 $msg(a)$를 수신하므로, 경합 상태는 Initial 상태에서 Affected 상태로 전이한다. 결과적으로 그림 2에 존재하는 세 개의 지역적 최초경합들 중에서 P_3에서 발생한 $<w, W>$만이 영향받지 않은 경합이다.

4. 실 험

실험을 위한 분산 시스템을 구성하기 위해서, Compaq사의 Alpha 노드로 구성된 클러스터 시스템에 커널 버전이 2.2.14-6인 Linux RedHat 6.2를 설치하였다. 일반적인 비동기적(asynchronous) 메시지전달을 제공하는 병렬프로그램 라이브러리 모델을 위해서 산업 표준화된 MPI(Message Passing Interface)[8]를 채택하였으며, 실험 시스템에는 MPI의 모든 표준 라이브러리를 지원하는 MPICH[2]를 설치하였다. 본 기법은 표준 C 언어와 MPI에서 제공하는 Profiling Interface를 이용하여 사용자 프로그램을 수정하지 않고 투명하게 적용할 수 있도록 구현하였다.

MPI Profiling Interface는 일반 사용자가 MPI의 함수 호출에 개입하여 별도의 작업을 수행하도록 허락하는 인터페이스로서, 모든 MPI 함수 호출이 다른 이름으로 호출 가능하도록 한것이다. MPI_xxx 형태의 모든 호출은 PMPI_xxx로의 호출로 대체 가능하며, 이것을 사용해서 사용자는 그들 자신만의 MPI_xxx를 실험할 수 있다. 따라서 본 실험에서는 일대일 통신에 관련된 MPI_xxx 내에 본 기법을 추가한 뒤 대응되는 PMPI_xxx를 호출함으로써, 사용자가 의도한 MPI_xxx 수행시에 경합 탐지도 추가적으로 수행할 수 있도록 구현하였다. 본 실험을 위해서 수정된 MPI 함수들로는 MPI_Comm_size(), MPI_Comm_rank(), MPI_Send(), MPI_Isend(), MPI_Recv(), MPI_Irecv(), MPI_Finalize() 등이 있다.

본 실험에 사용된 공용 벤치마크 프로그램으로는 C언어로 작성된 MPI 프로그램으로서 mpptest[8]의 stress를 선택하였다. 왜냐하면 stress 벤치마크 프로그램을 수행했을 때 각 프로세스에는 1,248개의 송신사건과 1,248개의 수신사건이 하나의 논리적인 채널 상에서 발생하였으며 메시지를 수신할 때 MPI_ANY_SOURCE를 사용하므로, 경합이 존재할 가능성이 높기 때문이다. 그러나 본 기법을 stress 프로그램에 적용한 결과는 경합이 존재하지 않는 것으로 확인되었다. 그 원인은 MPI_ANY_SOURCE가 있는 수신사건이 수신하는 메시지는 서로 다른 tag를 사용함으로써 모든 메시지가 결정적으로 수신되었던 것이다. 본 실험에서는 본 기법과 기존 기법의 정확성을 비교 평가하기 위해서 Stress에서 수행되는 수신사건의 tag를 MPI_ANY_TAG로 수정하여 고의로 경합이 발생하도록 하였다.

표 1은 수정된 Stress에서 두 기법이 탐지한 경합을 보인다. 각 프로세스에서 경합이 발생한 수신사건의 위치는 두 기법이 동일하게 탐지하지만, 탐지된 영향받지 않은 경합의 수와 경합하는 메시지의 수에서 차이를 보이고 있다. Netzer는 수정된 프로그램에서 두 개의 영향받지 않은 경합을 탐지하였으며, 0번과 1번 프로세스의 경합은 다른 경합과 서로 영향받은 경합으로 탐지하였다. 반면에 본 기법은 수정된 프로그램의 2번 프로세스에서 하나의 영향받지 않은 경합만을 탐지하였다. 그리고 경합하는 메시지의 수에 있어서는 Netzer가 탐지한 메시지의 수가 적다. 이것은 Netzer 기법이 경합을 탐지하면서 해당 프로세스의 수행을 중단하여 경합이 발생한 이후의 메시지가 송신되지 않기 때문이다. 그러므로 Netzer 기법에서 보고한 경합은 본 기법에 비해서 정확하지 못하다.

표 1 수정된 stress의 경합

pid	location of first racing receive	Netzer's Report		Our Report	
		locally-first races	# of racing messages	locally-first races	# of racing messages
0	4	Affected	1	Affected	3
1	5	Affected	0	Affected	3
2	1	Unaffected	2	Unaffected	2
3	1	Unaffected	2	Affected	3

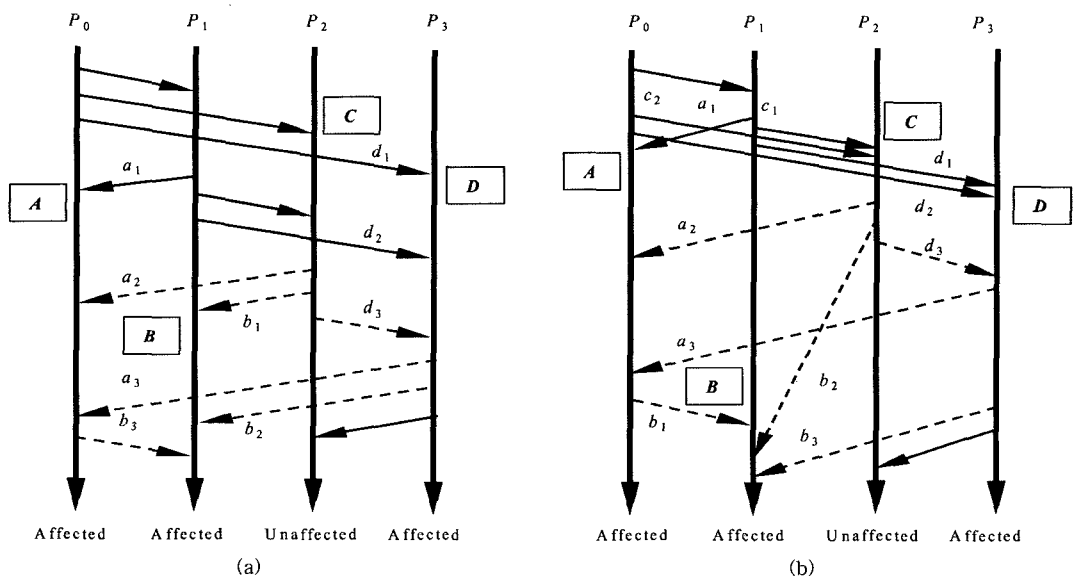

그림 5 메시지전달 패턴

두 기법이 탐지한 경합들간의 영향관계를 확인하기 위해서, 원래의 프로그램과 수정된 프로그램에서의 수행 내용을 추적파일에 저장하여 분석하였다. 원래의 프로그램에는 MPI_ANY_ SOURCE를 가진 수신사건들이 유일한 메시지 tag를 기다리므로 모든 메시지가 결정적으로 전달된다. 경합이 존재하도록 수정된 프로그램의 수행에서는 그림 5의 (a)와 같이 원래 프로그램과 동일한 메시지 패턴이 나타나는 경우가 대부분이었으며, 그림 5의 (b)와 같은 메시지 전송 패턴은 Stress를 100번 반복 수행해서 한두 번 나타난 결과이다.

그림에서 *A, B, C, D* 등의 문자는 수행 중에 발생한 지역적 최초경합의 수신사건 위치이며, 점선으로 표시된 메시지는 영향받은 메시지이다. 예를 들어 프로세스 P_0에서 발생한 최초경합의 수신사건은 *A* 위치이고, 이 수신사건으로 경합하는 메시지들은 $\{a_1, a_2, a_3\}$인데, 이들 중에서 $\{a_2, a_3\}$은 다른 경합으로부터 영향받은 메시지들이다. 이 두 그림에서는 프로세스 P_2에서 발생한 경합만이 영향받지 않은 경합이고, 그 밖의 다른 프로세스에서 발생한 경합은 영향받은 경합임을 알 수 있다. 이 실험을 통해서 Netzer의 기법이 프로세스 P_3에서 발생한 영향받은 경합을 영향받지 않은 경합으로 잘못 보고한 것을 확인할 수 있으므로, 본 기법이 Netzer의 기법보다 정확성이 높은 최초경합 탐지기법임이 입증된다.

5. 결 론

본 연구에서는 프로세스별로 탐지된 지역적 최초경합들간의 영향관계에 따라 탐지된 경합의 상태를 전이함으로써, 영향받지 않은 경합만을 보고하는 기법을 제안하였다. 본 기법의 정확성은 공용 벤치마크 프로그램을 이용하여 평가하였다. 본 기법과 기존의 기법간의 정확성에 대한 평가 실험에서는 본 기법만이 정확하게 영향받지 않은 경합을 모두 보고함을 확인하였다.

본 기법은 Pass-1을 수행하지 않고 프로그래머가 $\{cutoff, firstChan\}$을 임의로 설정함으로써 Pass-1로부터 독립되어 사용될 수 있다. 이렇게 함으로써 프로그래머가 의도한 경합이 아닌 의도하지 않은 경합들에 대해서만 영향받은 경합과 영향받지 않은 경합을 구분하여 탐지할 수 있다. 또한 *cutoff* 값을 점진적으로 증가시켜 탐지된 경합을 디버깅함으로써 각 프로세스별 존재하는 모든 경합들을 탐지할 수 있다. 그러므로 본 기법은 매

시지전달 프로그램의 효과적인 디버깅 도구의 개발을 가능하게 한다. 향후과제로서 탐지된 경합들의 상태를 다양하게 시각화할 수 있다.

참 고 문 헌

[1] Cypher, R., and E. Leu, "The Semantics of Blocking and Nonblocking Send and Receive Primitives," *8th IEEE 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* (HICSS), Vol. 1, pp. 423-432, Jan. 1997.

[6] Kranzlmüller, D., *Event Graph Analysis for Debugging Massively Parallel Programs*, Ph.D. Dissertation, Joh. Kepler University Linz, Austria, Sept. 2000.

[7] Krammer, B., M.S. Müller, and M.M. Resch, "MPI Application Development Using the Analysis Tool MARMOT," *4th Int'l Conf. on Computational Science, Lecture Notes in Computer Science*, 3038:464-471, Springer-Verlag, june 2004.

[8] Kranzlmüller, 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.

[9] 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.

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

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

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

[13] 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.

[14] 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* (SPDT), ACM, May 1996.

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

[16] Gropp, W. and E. L. Lusk, "Reproducible Measurements of MPI Performance Characteristics," *6th European PVM/MPI Users' Group Conf.*, Barcelona, Spain, *Lecture Notes in Computer Science*, 1697: 11-18, Springer-Verlag, Sept. 1999.

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

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

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

[20] 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.

[21] Kranzlmüller, D., C. Schaubschlager, and J. Volkert, "Brief Overview of the MAD Debugging Activities," *4th International Workshop on Automated Debugging* (AADEBUG 2000), Aug. 2000.

박 미 영

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

강 현 석

1981년 동국대학교 전자계산학과 졸업
1981년 서울대학교 전산학과 졸업(석사)
1986년 서울대학교 전산학과 졸업(박사)
1981년~1984년 한국전자통신연구원
1984년~1993년 전북대학교 부교수. 1993년~현재 경상대학교 컴퓨터과학과 교수
관심분야는 MPEG-7, 내장형 데이타베이스

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 컴퓨터과학과 교수
컴퓨터·정보통신연구소 연구원. 관심분야는 분산병렬처리, 내장형시스템, 시스템소프트웨어