MPI 병렬 프로그램의 순환 디버깅을 위한 인과관계 재실행

(Causal Replay for Cyclic Debugging of MPI Parallel Programs)

홍 철 의 † 김 영 준 †
(Chul-Eui Hong) (Yeong-Joon Kim)

요 약 메시지 전달 병렬 프로그램은 프로세스 사이의 메시지 경합에 의하여 실행의 비결정성이 발생하여 순차 프로그램에서 널리 사용되는 순환 디버깅 기법을 사용하기 어렵다. 본 논문은 MPI 병렬 프로그램에서 비결정적 실행에 영향을 미치는 메시지 전달 사건을 정의한 후, 기본실행에서의 사건의 발생 순서가 다음 재실행시 똑 같이 유지되도록 병행실행을 순차실행으로 변환하여 결정적 재실행을 보장함으로써 실행시 마다 같은 오류가 재현되도록 한다. 또한, MPI 병렬 프로그램의 디버깅을 보다 쉽게 하기 위하여 임의의 프로세스를 정지시켰을 때, 다른 모든 프로세스는 정지점 이전에 발생한 모든 사건을 반영하는 최초의 상태에 정지하게 하는 인과관계 정지점을 구현한다. 따라서 인과관계 재실행 기법을 이용하여 병렬 프로그램에서도 순차 프로그램 환경에서와 같이 순환 디버깅 기법을 사용할 수 있게 한다.

Abstract The cyclic debugging approach often fails for message passing parallel programs because they reveal non-deterministic characteristics due to message race conditions. This paper identifies the MPI events that affect non-deterministic executions, and then converts the concurrent execution to the sequential one that is controlled in order to make it equivalent to a reference execution by keeping their orders of events in two executions identical. This paper also presents an efficient algorithm for the causal distributed breakpoint, which is initiated by any sequential breakpoint in one process, and restores each process to the earliest state that reflects all events that happened causally before the sequential breakpoint. So a cyclic debugging approach can be used in debugging MPI parallel programs as like as in debugging sequential programming environments.

1. 서 론

디버깅(debugging)이란 프로그램을 분석하여 의심되는 오류를 찾아내고 수정하는 작업이다. 기존의 순차 프로그램을 위한 디버깅 방법의 대표적인 예로써 순환 디버깅(cyclic debugging)이 있다. 순환 디버깅이란, 프로그램이 실행되는 동안 반복적으로 실행을 정지시켜 프로그램의 상태를 조사하고, 계속 실행을 시키거나 새로이 다시 실행을 시키는 방법이다. 새로운 반복 실행 시 설정된 정지점들은 프로그래머로 하여금 오류의 부분에 좀더 가깝게 접근 하게 한다[1].

그러나, 동일한 입력에 대하여 항상 동일한 동작을 반복하는 순차 프로그램과는 다르게 병렬 프로그램의 동작은 전형적인 비결정적(non-deterministic) 특징을 나타낼 수도 있다. 즉, 동일한 입력 자료가 동일한 프로그램에 주어지더라도, 새로운 실행은 서로 다른 동작을 보일 수 있다. 따라서 프로그램의 오류가 불규칙적으로 발생하게 되므로 순환 디버깅은 의미가 없게 된다. 이러한 문제점을 극복하기 위하여 실행의 재실행(execution replay 또는 trace and replay)이라는 기법이 제안되었다.

비결정적 수행의 대표적인 경우는 메시지 경합 조건에 의해서 발생한다. 두개 이상의 메시지가 같은 목적지를 가지고 있고, 목적지에 도착하는 순서는 임의적 조건에 의해서 결정되어 진다고 하면, 이 메시지들은 경합 조건에 있다고 정의된다(그림 1).

† 정 회 원 : 상명대학교 정보통신학부 교수
hongch@sangmyung.ac.kr
yjkim@sangmyung.ac.kr
논문접수 : 2000년 9월 21일
심사완료 : 2001년 7월 19일

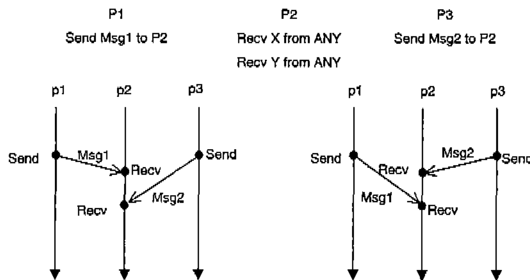

그림 1 통신에서의 경합 발생

정의 1: 프로그램의 실행 경로에 영향을 미치는 요소를 '**사건**'으로 정의하며, 실행 경로는 일련의 사건들의 순서로 나타낸다.

MPI 병렬 프로그램은 메시지 전달에 의해서만 정보를 교환하므로, 실행 경로에 영향을 미치는 사건은 메시지 전달 구문으로 구성된다. 또한, MPI 병렬 프로그램에서 메시지 경합은 결정적 재실행을 어렵게 하는 요인으로 작용한다. 이 경우, 처음의 기본실행과 다음의 재실행들이 똑같은 경로를 밟게 하기 위해서는 기본실행시 사건순서에 대한 정보를 수집하여 다음의 재실행시에 기본실행과 똑같은 사건이 순서적으로 발생하도록 조종한다.

결정적 재실행이 유지되는 병렬 프로그램의 오류를 정정하고자 할 때, 프로그래머는 순환 디버깅 기법의 정지점 기능을 이용하여 정지된 프로세스 상태에 영향을 미치는 인과관계의 사건들을 조사한다. 병렬 시스템에서의 인과관계 정지점은 사건의 부분 순서화에 기초하여 각 프로세스가 정지점 이전에 발생한 모든 사건들을 반영하는 최초의 상태에서 정지하게 한다. 반면에, 일반적인 병렬 정지점 기법은 각 프로세스는 정지점에 영향을 미치는 사건을 지나서 프로세스가 메시지 수신을 위하여 스스로 정지하는 상태에서 정지한다. 따라서 인과관계 정지점은 정지점에 영향을 미치는 사건들로만 구성되어 있으므로 디버깅에 유용하게 사용된다. 순차 실행에서는 실행 단위가 하나로 모든 사건들이 완전 순서화를 이루고 있으므로 인과관계가 자연히 이루어지나, 병렬 실행에서는 사건들이 부분 순서화를 이루고 있으므로 전체 프로세스에 대하여 인과관계에 대한 snapshot을 얻기가 어렵다.

본 논문은 MPI 병렬 프로그램에서 재실행 및 인과관계 정지점을 구현하기 위하여 사건벡터를 이용한다[2].

정의 2: **프로세스 고유번호**는 MPI 병렬 프로그램에 참가하는 각 프로세스를 구별하기 위하여 프로세스당 다르게 부여되는 자연수로 정의한다.

정의 3: 정의 1에서 정의된 사건의 발생 순서를 표시하기 위하여 사건의 발생시마다 증가하는 카운터의 역할을 하는 **사건시간**을 각 프로세스당 부여한다.

정의 4: **사건벡터**는 프로세스 고유번호와 사건 발생시 마다 증가되는 논리적 사건시간의 쌍으로 정의된다.

메시지 전달시 사건벡터(프로세스 고유번호, 사건시간)를 함께 보내어 결정적 재실행을 보장한다. 또한, 재실행을 기반으로 하는 순환 디버깅 기법 사용시 정지점이 수행되는 프로세스를 시작으로 하여 기본실행의 사건순서에 따라 순차적으로 수행함으로써 인과관계 정지점을 보장한다.

2장에서는 병렬 프로그래밍 환경으로서 재실행 시스템에 대한 기존의 연구를 소개하고 본 논문에서 제안한 시스템과의 차이점 설명한다. 또한 MPI 시스템에서 제공하는 메시지 전달 병렬 프로그램의 사건 순서화 기법 및 분산 인과관계 기법에 대하여 소개한다. 3장에서는 MPI 병렬 프로그램에서의 재실행 시스템을 구현하기 위한 기법을 제안하고, 4장에서 제안된 재실행 시스템을 사용자가 보다 쉽게 디버깅하기 위하여 필요한 분산 인과관계 구현 알고리즘 및 자료구조를 자세히 설명한다.

2. 관련 연구 및 배경

Fromentine[3] 및 송후봉 등[4]은 병렬 프로그래밍 환경에서의 재실행 시스템의 정의 및 다양한 종류에 대한 구현 방법을 설명하였고, Claudio[5] 등은 PVM 라이브러리에 모니터링 코드를 삽입하여 병렬 프로그램의 재실행 시스템을 구현하였다. 또한, 최근에는 네트워크로 구성된 자바 병렬 프로그래밍 환경에서의 재실행 시스템 및 이를 이용한 디버거를 구현하였다[6, 7]. 그 이외에 현재 개발된 대표적인 병렬 디버깅 환경으로는 PVM 및 MPI 병렬 프로그램을 위한 p2d2[8] 및 Total View[9], nCUBE2 병렬처리 시스템용의 MAD[10]와 XPVM[11], XMPI[12] 등이 있다.

본 논문에서는 위에 열거한 기법과는 다르게 MPI 병렬 프로그램을 효율적으로 디버깅하기 위한 재실행 및 분산 인과관계 정지점을 동시에 구현하였다. 따라서, 본 장에서는 먼저 MPI 병렬 프로그램을 재실행하기 위하여 사건 순서화 기법을 설명하고, 분산 인과관계 정지점 기법의 구조를 간략히 설명한다.

2.1 MPI 병렬 프로그램에서의 사건 순서화

메시지 전달 기법은 분산 메모리 병렬 컴퓨터에서 광범위하게 사용되어 왔다. 메시지 전달 기법의 기본 개념은 잘 정의되어 있으나, 병렬 컴퓨터의 종류에 따라서

서로 다른 메시지 전달 기법이 사용되어 소프트웨어 이식성을 어렵게 하였으며 메시지 전달 기법의 발달이 이루어지지 않는 요인이 되었다. A Message-Passing Interface (MPI) 표준안은[13] 광범위한 사용자에게 유용한 핵심 라이브러리의 문법 및 의미를 정의하여 광범위한 컴퓨터에서 공통적인 메시지 전달 기법이 사용 가능하게 하기 위하여 개발되었다. MPI 주요 통신구문은 다음과 같다.

- nonblocking : 호출에서 지정된 버퍼와 같은 자원에 대하여 재사용이 허락되기 전에 구문이 반환된다.
- blocking : 구문이 반환된 후에 호출에서 지정된 자원을 재사용할 수 있다.
- collective : 프로세스 그룹내의 모든 프로세스가 구문을 실행한다.

MPI는 소스 프로세스 고유번호 및 메시지 태그에 근거해서 메시지를 선택하는 point-to-point 통신 기법을 제공한다. 메시지 수신시 소스와 태그는 wildcard로 표시될 수 있어서 소스와 태그에 관계 없이 메시지가 선택될 수 있으나, 송신시는 wildcard를 사용할 수 없다. 소스와 목적 프로세스는 그룹 안에서의 순서(rank)로 표시된다.

MPI는 여러 자료형이 혼합되며, 메모리상에 연속되어 있지 않은 메시지를 규정할 수 있는 기법을 제공한다. 사용자는 MPI 자료형 정의 루틴을 이용하여 일련의 자료형과 그 위치에 대한 offset으로 구성되는 사용자 정의 자료형을 규정할 수 있다. 일단 자료형이 규정되면, 그 자료형은 point-to-point 또는 collective 통신에 사용될 수 있다. 자료는 비연속적인 장소로부터 모아져서, 전송된 후 수신자 측의 비연속적인 장소에 저장된다. 구현에 따라서, 자료가 전송되기 전에 연속적인 버퍼에 패킹 되거나, 원래 있는 장소에서 직접 모아질 수 있다.

병렬 프로그램을 디버깅하려고 할 때 프로세스의 상태나 실행 중에 생성되는 사건들의 발생 순서를 정확하게 알 수 있는 방법이 없다고 한다면 오류의 위치를 국부화시키는데 많은 어려움이 따르게 된다. 분산 시스템에서는 전역 시간이 존재하지 않으며, 망 지연 등과 같은 요인으로 인하여 물리적으로 분산되어 있는 프로세스들 상에서 발생된 프로세스의 생성, 종료, 메시지의 송/수신 등과 같은 사건 발생에 대하여 물리적 시간을 사용할 수 없다.

물리적 시간을 사용하여 사건을 순서화 시키는 문제점을 해결하기 위하여 Lamport[14]의 논리적 시간을 도입하였다. MPI 병렬 시스템 모델에서는 메시지 전달만을 이용하여 프로세스 사이에 통신이 이루어지며, 사건은 메시지의 송신(sending) 또는 수신(receipt)으로 정의 된다. 각 사건은 사건벡터(프로세스 고유번호, 사건시간)를 이용하여 다른 사건과 구별되며, 사건들은 happened before 관계를 이용하여 순서화 된다. →는 happened before 관계를 나타낸다.

a 와 b가 같은 프로세스내의 사건이고 a 가 b보다 먼저 발생하면, $a \rightarrow b$로 나타낸다.

- a 가 메시지의 송신 사건이고 b가 같은 메시지의 수신 사건이면, $a \rightarrow b$로 나타낸다.
- $a \rightarrow b$이고 $b \rightarrow c$이면, $a \rightarrow c$이다.
- $a \rightarrow b$가 아니고 동시에 $b \rightarrow a$가 아니면, 두 사건 a와 b를 동시적(*concurrent*)이라 한다.

정의 5: Happened before 관계에 있는 사건들은 인과관계를 가지고 실행 경로에 영향을 미치므로 **종속사건**으로 정의된다.

정의 6: 병렬 수행이 가능한 동시적 사건들은 실행 경로에 영향을 미치지 않으므로 **독립사건**으로 분류된다.

정의 5 및 6으로부터 순차 프로세스의 모든 사건들은 종속 사건으로 완전 순서화를 이루고 있으나, 병렬 프로세스는 사건의 발생이 동시적이므로 부분 순서화를 이룬다.

논리적 시간을 나타내기 위하여 본 논문은 Colin Fidge[15]의 타임 스탬프 기법을 수정하여 사용하였다.

초기화(Initialization)

프로세스 p_i가 처음 실행될 때, 이에 해당하는 사건시간 c_i는 0으로 설정된다.

- 시간 증가(Ticking)
  프로세스 p_i 상에서 하나의 사건 발생은 사건시간 c_i를 하나 증가시키며, 사건시간 c_i는 어떠한 경우에도 감소하지 않는다.
- 송신(Sending)
  프로세스 p_i가 메시지를 송신하고자 할 때, 프로세스 고유번호 i와 사건시간 c_i가 메시지에 첨가되어 송신된다.
- 수신(Receiving)
  프로세스 p_i가 프로세스 p_j로부터 메시지를 수신하였을 때, 수신 메시지에 첨가된 송신 프로세스 고유번호 i와 사건시간 c_i를 추출한다.

현재의 MPI 표준에서는 프로세스가 동적으로 생성되지 않으므로 프로세스 생성에 대한 규칙은 생략하였다. 동적 프로세스 생성이 허용되면, 이를 사건으로 규정하여 순서화 시키며 자식 프로세스는 부모 프로세스와 다른 프로세스 고유번호를 부여한다.

사건시간은 사건 발생시마다 증가하는 카운터의 역할

을 한다. 프로세스 고유번호는 MPI_Init에 의하여 초기화한 후에 MPI_COMM_WORLD 그룹에서 각 프로세스에 부여되는 순서(rank)로 주어진다. 예를 들어 n개의 프로세스가 계산에 참여했을 때, 프로세스 고유번호는 0 서부터 n-1로 주어진다.

(그림 2)는 재실행 시스템에서 사건 순서화의 예를 보여준다.

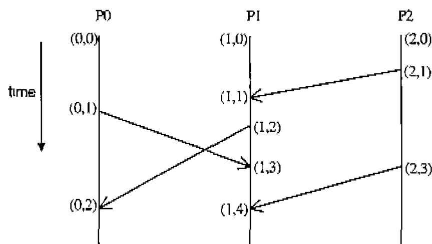

그림 2 메시지 전달 병렬 시스템에서의 사건 순서화

2.2 분산 인과관계 정지점

프로세스 상태는 실행 중 한 점(snapshot)에서의 해당 프로세스의 실행 상태를 말하며, 시스템 상태란 수행되는 모든 프로세스 상태의 집합을 말한다. 병렬처리에서 하나의 프로세스에서 발생하는 모든 사건은 완전 순서화를 이루고 있으므로 프로세스 상태는 해당 프로세스에서 이전에 발생한 모든 사건을 반영하고 있으나, 서로 다른 프로세스에서 발생하는 사건들은 부분 순서화를 이루고 있으므로 임의의 프로세스 상태가 다른 프로세스의 모든 사건들을 반영하지 못할 수 있다. 시스템 상태가 일치(consistent)하기 위해서는 임의의 프로세스 상태에 대하여 happened before 관계에 있는 모든 사건이 다른 모든 프로세스에서 발생하여야 한다. 예로서, 메시지 전달 병렬 프로그램에서는 수행된 모든 메시지 수신에 대하여 반드시 송신이 이루어져야 한다.

정지점 사건은 정지점을 실행한 프로세스에서 정지점 이전에 발생한 사건 중 가장 최근에 발생한 사건을 말한다. 인과 관계 정지점은 다음과 같은 시스템 상태로 정의된다[16].

1. 정지점 프로세스의 경우, 정지점을 설정할 때의 상태를 말한다.
2. 다른 프로세스의 경우, 정지점을 실행한 프로세스의 정지점 사건 이전의 모든 사건을 반영하는 최초의 프로세스 상태를 말한다.

어떠한 정지점 상태도 일치 시스템 상태를 나타내나, 인과관계 정지점은 정지점 사건에 영향을 미치는 사건으로만 구성되어 있으므로 순환 디버깅에 유용하다. 반면에, 일반적인 정지점 수행 방법은 정지점 프로세스의 경우 정지점 설정 상태에서 정지하나, 다른 프로세스의 경우 정지점 설정 후에도 계속 수행하여 보내지지 않은 메시지 수신 사건과 같은 수행이 불가능한 상태에서 정지하게 된다.

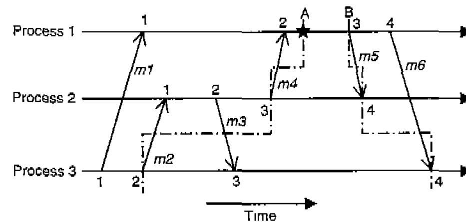

그림 3 프로세스 1의 정지점 사건에 대한 인과관계 정지점

(그림 3)는 메시지 *m1*에서 *m6*가 3개의 프로세스 사이에서 전달된 상태를 보여준다. 정지점은 프로세스 1에서 '★'로 표시된 부분에서 발생한다. **A**와 **B**선 사이의 어느 상태도 일치 시스템 상태를 나타낸다. 이중에서 인과 관계 정지점은 일치 시스템 상태 중 최초의 상태인 **A**선에 걸쳐 있는 각 프로세스의 상태로 표시된다. **B**선을 시스템 상태로 가정하면, 프로세스 3은 메시지 *m3*를 받은 후이다. 그 결과 프로세스 3 의 상태가 변화되어 메시지 *m2*의 전송에 대한 정보가 변화될 수 있다. 메시지 *m2*는 프로세스 1 의 정지점 상태에 대하여 인과 효과를 가지고 있는 사건이므로 이에 대한 정보가 프로세스 상태에 나타내져야 프로그래머는 손쉽게 오류를 정정할 수 있다.

3. MPI 병렬 프로그램에서의 재실행

MPI 병렬 프로그램을 디버깅하는 과정에서 프로그램의 재실행에 필요한 실행 경로를 정하기 위해서는 기준점들이 있어야 한다. 병렬 프로그램의 실행에서 경로의 구성은 사건을 기반으로 이루어진다. 사건의 정의는 각 프로세스들 간의 정보 교환 과정을 중심으로 이루어진다. MPI 관련형 사건 중 point-to-point 통신에 관련된 사건만이 실행 경로에 영향을 미친다. 모든 사건은 실행 경로에 영향을 미치는 종속적인 사건과 실행 경로와는 무관한 독립적 사건으로 분류된다.

- 종속 사건
  - MPI_Send, MPI_Recv
  - MPI_Bsend, MPI_Ssend, MPI_Rsend, MPI_Isend

- MPI_Irecv, MPI_Ibsend, MPI_Issend, MPI_Irsend
- MPI_Wait, MPI_Test
- MPI_Waitany, MPI_Testany
- MPI_Waitall, MPI_Testall
- MPI_Waitsome, MPI_Testsome
- MPI_Iprobe, MPI_Probe
- MPI_Start, MPI_Startall
- MPI_Sendrecv, MPI_Sendrecv_replace

• 독립 사건

종속적 사건으로 정의된 이외의 모든 MPI 구문은 독립적 사건으로 정의된다.

프로그램을 처음 실행(기본실행)시 재실행에 필요한 부분 순서를 정의하기 위하여 최소한의 사건 정보를 수집한다. MPI의 경우 독립 사건은 실행 경로에 영향을 미치지 않으므로 다른 프로세스와 관계가 있는 종속 사건에 대해서만 사건벡터를 기록한다. 사건벡터는 프로세스 고유번호와 사건 발생시마다 증가되는 논리적 시간의 쌍으로 정의된다

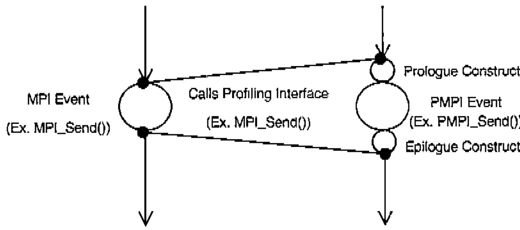

그림 4 prolog와 epilog

(그림 4)와 같이 MPI profiling 인터페이스를 이용하여 종속 사건의 앞뒤에 각각 prolog와 epilog 구문을 첨가한다. prolog 구문은 사건 발생 직전의 시점을 나타내며, epilog 구문은 사건 발생 직후의 시점을 나타낸다. 이러한 prolog와 epilog 구문은 사건 수집과 등가 재실행을 위한 hook으로 사용된다.

기본실행시 prolog는 사건시간을 증가시킨다. 또한, 메시지 송신시 사용자 정의 자료 구조를 이용하여 사건벡터와 메시지를 조합하며, 수신측에서는 같은 사용자 정의 자료구조를 이용하여 사건벡터와 메시지를 수신한다. epilog 구문에서 수신자는 전달된 자료로부터 메시지와 송신자 사건벡터를 분리하여 사건 정보를 각 프로세스마다 할당된 history 파일에 저장한다. 그러나, 메시지 송신시 수신 노드와 테그에 대하여 wildcard를 사용하지 않으므로 송신자는 epilog 구문에서는 사건 정보를 기록하지 않는다.

사건벡터는 송신되는 메시지에 사용자 정의 자료 구조를 이용하여 첨가된 후 송신된다. 이는 메시지와 사건벡터를 따로 송신할 경우 발생할 수 있는 다른 메시지와의 간섭 효과를 배제하기 위해서 이다. blocking 및 non-blocking 통신이 함께 사용될 때 메시지 도착 순서가 송신 순서로 보장되지 않으며, non-blocking 수신의 경우 수신되는 메시지에 해당하는 사건벡터를 언제 받을지 알 수 없으므로, 다른 메시지의 사건벡터 수신과 구별이 되지 않는다. 수신자는 송신시와 같은 사용자 정의 자료형을 정의하여 메시지를 수신함으로써 메시지 수신 후 송신자 사건벡터와 메시지를 쉽게 분리할 수 있다.

예를 들어, 메시지 송신 구문인, int MPI_Send(void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)에 대한 사건 포획 자료 구조는 다음과 같다.

```
/*Define data types and their length for the event
  record and data. An event record consists of two
  MPI_INT typed data */
  Cap_length[0] = 2; Cap_type[0] = MPI_INT;
  Cap_length[1] = count; Cap_type[1] = datatype;
/*Absolute addresses are made for an event record
  and data. Cap_vector is for an event record and
  Cap_displ is its displacement relative to address
  0 */
  MPI_Address(Cap_vector, Cap_displ);
  MPI_Address(buf, Cap_displ+1);
/*Define a newly derived data type, Cap_newtype,
  for a data and its event record. */
  MPI_Type_struct(2, Cap_length, Cap_displ, Cap_type,
  Cap_newtype);
/*A new data type, Cap_newtype, is committed and
  can be used in send communications later. */
  MPI_Type_commit(Cap_newtype);
/*Send a message with a newly derived data type. */
  MPI_Send(MPI_BOTTOM, 1, Cap_newtype, dest,
  tag, comm);
/*Release the new data type */
  MPI_Type_free(&Cap_newtype);
```

프로세스 고유번호와 사건시간의 쌍으로 정의되는 사건벡터를 메시지에 첨가하여 송신하는 경우 원래 송신되는 메시지의 크기보다 사건벡터에 해당하는 2개의 MPI_INT 만큼의 크기가 더해져서 status에 저장된다. 이는 MPI_Get_count 및 MPI_Get_elements의 사용을 불가능하게 한다. 그러므로, 수신 후 status 구조의

count 항목에서 2개의 MPI_INT에 해당하는 바이트 수를 빼준다.

non-blocking 송신 및 수신을 위하여 새로운 자료구조가 정의된다. 먼저, 사건벡터가 메시지에 첨가되어 송신되기 때문에, 송신될 때의 사건벡터가 안전한 장소에 저장되어 다음 명령에 의하여 갱신되지 말아야 한다. 수신자 측에서는 송신자로부터 전송되어 온 메시지 및 사건벡터를 사건 파일에 저장한다. 또한, 수신자 측에서만 통신 구문을 비교하여 수신 메시지를 선택하므로 수신시에만 사건 정보를 기록한다. 다음은 *non-blocking* 통신에 대한 자료 구조이다.

```
struct Cap_nonblock_struc {
    int persistent_flag;
    int *coming_event_record_p;
    int event_record[2];
    MPI_Datatype newtype;
    MPI_Request *request;
    struct Cap_nonblock_struc *next;
} *Cap_nonblock_head;
```

persistent_flag 멤버는 persistent 통신구문 발생 여부에 대한 플래그이다. Coming_event_record_p 멤버는 통신할 메시지에 첨가될 사건벡터의 포인터를 나타내며, event_record는 non-blocking 통신이 발생하는 시점의 사건벡터를 나타낸다. newtype는 사용자 정의 자료 구조를 나타내며, next는 연결 리스트의 다음 노드 포인터이다. non-blocking 통신에서는 송신자와 수신자 모두 Cap_nonblock_struc 자료 구조를 이용하여 새로이 할당된 메모리 영역에서 통신을 수행하며, MPI_Wait나 MPI_Test와 같은 통신 종료를 확인하기 위한 구문은 request 멤버를 확인하여 연결 리스트로부터 해당되는 사건 정보를 찾아 history 파일에 기록한다.

같은 인수에 대한 통신이 루프 안에서 반복적으로 시행될 때 persistent communication requests가 사용된다. 이 경우 통신 인수 리스트를 처음에 한번 바인딩한 후 송/수신에 대한 시행과 종료를 반복적으로 사용함으로써 통신 메커니즘을 최적화시킬 수 있다. persistent communication requests는 MPI_Send_init나 MPI_Recv_init로 이루어 지며, MPI_Start에 의하여 시행된 후, MPI_Wait등에 의하여 종료가 확인되고, MPI_Request_free에 의하여 바인딩이 해체된다. MPI_Start는 바인딩시의 request 인수에 의하여 시행되므로 사건 기록 및 재실행을 위해서는 바인딩시에 사용된 통신 인수를 안전한 장소에 보관하여 사건벡터를 첨가하거나 인출할 수 있는 별도의 연결 리스트를 마련한다.

등가 재실행시는 각 프로세스마다 할당된 history 파일에 저장되어 있는 사건 정보를 수신 시간 순으로 정렬한다. 일반적으로 파일의 크기가 크므로 외부 합병 정렬을 사용하였다. 다음, 송신시 원래의 태그 대신 사건시간을 태그로 사용하며, 수신자 또한 원래의 태그 대신 파일에 저장되어 있는 송신 사건시간 태그로 사용하여 초기 실행과 등가의 재실행을 순차처리 한다. 순차처리의 순서는 분산 인과관계 정지점 수행의 예(그림 6)에 표시되어 있다. 변화된 메시지가 다음 명령에 사용될 수 있기 때문에 수신자는 수신하자 마자 원래의 태그로 status.MPI_TAG를 변화시킨다. 이러한 기법은 Netzer [17]에서 수신 메시지를 위한 버퍼 사용시 발생하는 메모리 할당 및 탐색에 대한 오버헤드를 없앤다.

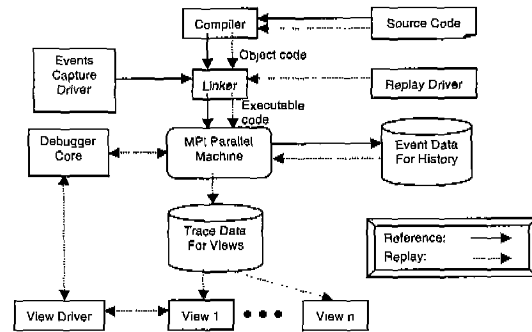

그림 5 재실행 과정

(그림 5)는 기본실행과 이에 대한 재실행의 절차를 보여준다.

4. MPI 재실행 시스템에서의 분산 인과 관계 정지점 구현

재실행이 보장되는 MPI 병렬 프로그램에서 사용자가 보다 쉽게 디버깅하기 위하여 분산 인과관계 정지점을 구현하는 알고리즘은 다음과 같다.

1. 프로세스 시작: MPI_Init
   - 정지점 프로세스만이 수행되고 그 이외의 프로세스는 시작점에서 대기한다.
2. 메시지 수신: MPI_Recv, MPI_Wait 등
   - 모든 수신 프로세스는 송신 프로세스에게 수신하려고 하는 메시지에 해당하는 인과관계 메시지를 송신 사건 시간을 첨가하여 송신 프로세스에게 전송한다.
3. 메시지 송신: MPI_Send 등
   - 정지점 프로세스의 경우 다른 행동을 취하지 않

고 단지 메시지를 송신한다.

- 그 이외의 프로세스는 인과관계 메시지의 사건 시간이 프로세스의 현재 사건 시간보다 클 때는 계속 수행하며, 같을 때는 메시지를 송신한 후 수행을 중지한다.

위의 인과관계 메시지 전송시 첨가되는 정보는 다음과 같다.

- 정수형의 송신 사건시간
- 인과관계 메시지에게 고유한 태그를 부여하기 위하여 MPI_TAG_UB를 사용한다.

또한, 인과관계 메시지를 수신하는 의사코드는 다음과 같다.

```
while ((인과관계 송신 사건 시간  현재 사건 시간)
        and !(현재 프로세스가 정지점 프로세스))
    MPI_Recv( )를 이용하여 인과관계 메시지 수신;
```

(그림 6)은 정지점 수행시 인과관계를 나타낸다. 사건은 (프로세스 고유번호, 사건시간)의 쌍으로 표시되며, 실선 화살표는 메시지 전달을 나타내며, 대시 화살표는 메시지 전달에 대한 인과관계 메시지 전달을 표시하고, 점선 화살표는 인과관계 정지점 구현시 순차적으로 수행되는 순서를 나타낸다.

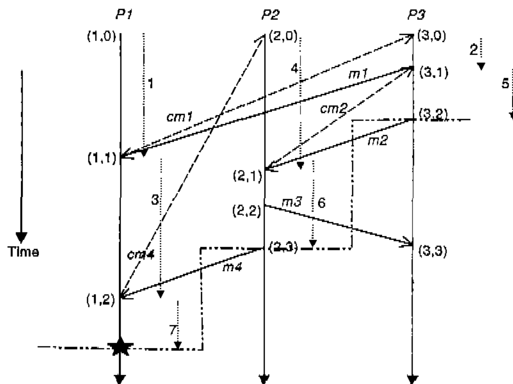

그림 6 인과관계 정지점 수행의 예

(그림 6)의 수행 순서를 자세히 살펴보면 먼저 정지점 프로세스인 *P1*만이 수행을 시작하고 그 이외의 프로세스인 *P2*와 *P3*는 시작 사건 지점에서 멈춰 있는다. 다음 정지점 프로세스 *P1*은 수신 사건 (1,1) 지점에서 메시지 *m1*에 대한 인과관계 메시지 *cm1*을 송신 사건 시간 1을 첨가하여 송신 프로세스인 *P3*에게 보낸 후 수행을 멈춘다. 반면에, *P3*는 시작 사건 (3,0) 지점에서 인과관계 메시지를 수신한 후 수행을 시작하여 *P1*에게 메시지 *m1*을 송신한 후 멈춘다. 이는 *cm1*에 포함된 송신 사건 시간이 1이므로 사건 시간 (3,1) 지점까지만 수행하기 때문이다.

*m1*메시지를 수신한 *P1*은 다시 수행을 시작하여 다음 수신 사건 (1,2) 지점에서 메시지 *m4*에 대한 인과관계 메시지 *cm4*를 송신 프로세스 P2에게 송신 사건 시간 3을 첨가하여 보낸 후 멈춘다.

이와 같이 메시지 수신 각각에 대하여 인과관계 메시지를 송신함으로써 인과관계 정지점을 구현한다. 여기서 *P2*의 *m3* 메시지 송신의 경우 *P1*으로부터 수신한 인과관계 메시지 *cm4*에 첨가된 사건 시간이 3이므로 송신 후 계속 수행하여 *m4* 메시지 송신 후 멈춘다.

5. 결 론

열 전달 및 Bitonic-Merge 정렬을 본 논문에서 구현한 인과관계 재실행 시스템에서 수행하여 재실행 및 분산 인과관계 정지점이 올바르게 수행되는 지를 조사하였다. (그림 7)은 Bitonic-Merge 정렬을 예를 보여준다. 병렬 프로그램은 그 자체에 메시지 경합에 의하여 비결정성을 가진다. 비결정성이란 동일한 입력 자료가 동일한 프로그램에 주어지더라도 각 실행 시마다 서로 다른 출력을 발생하는 것을 말한다. 이러한 비결정성은 순차 프로그램 디버깅시 유용한 순환 디버깅 기법을 병렬 프로그램에서는 사용할 수 없게 한다.

또한, 병렬 프로그램 환경에서 병렬 프로세스 중 하나의 프로세스가 정지점에서 멈추면 다른 프로세스는 실행이 계속 진행되어 정지점에 영향을 미치는 정보가 변화된다. 따라서, 정지점에 영향을 미치는 모든 사건을 반영한 상태에서 다른 프로세스가 정지하게 하는 인과관계 정지점의 구현은 디버깅에 유용하다. 즉, 정지점 프로세스를 제외한 모든 다른 프로세스는 정지점 사건 이전의 모든 사건을 반영하는 일치 상태 중 최초의 상태에 정지하여 프로그래머에게 디버깅에 유용한 정보를 제공한다.

본 논문에서는 MPI 병렬 프로그램에서 결정적 실행을 보장하는 재실행 및 인과관계 정지점 기능을 구현하여 프로그래머가 순차 프로그램 디버깅에서 사용하는 순환 디버깅 기법을 병렬 프로그램 디버깅에서도 자연스럽게 사용할 수 있도록 하였다.

이를 위하여 MPI에서 사용되는 통신 구문을 실행 경로에 영향을 미치는 종속 사건과 실행 경로에 영향을 미치지 않는 독립 사건으로 구분하여 종속 사건에 대하여 사용자 정의 자료구조를 이용하여 사건 순서화를 이루었다. 또한, blocking 통신뿐만이 아니라 non-blocking 통

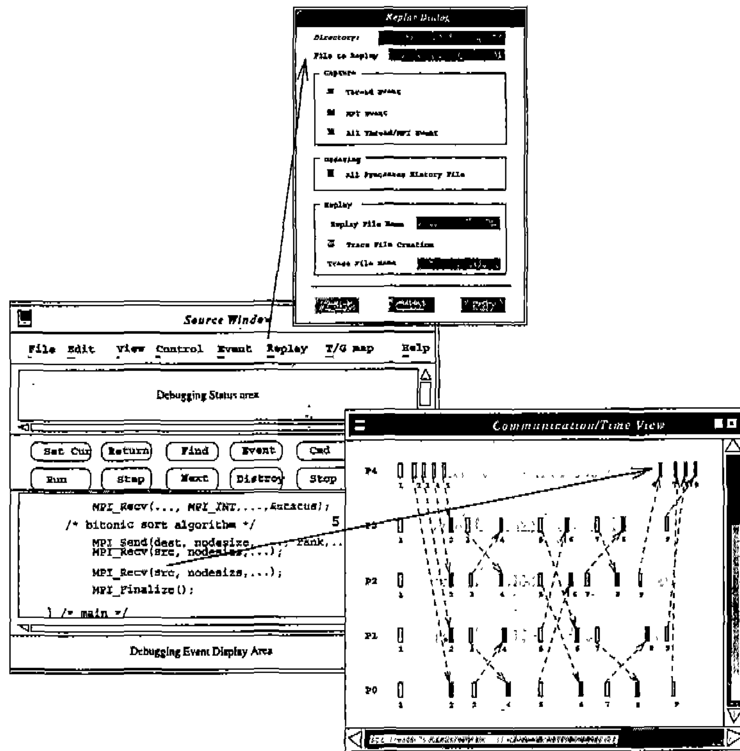

그림 7 MPI 재실행 시스템의 실례

신에 대하여서도 재실행을 가능하게 하였으며, 정보 저장의 크기를 최소화하기 위하여 사건 기반형 재실행 기법을 구현하였으며 재실행시 통신 버퍼가 필요 없도록 설계하였다. 인과관계 정지점 알고리즘은 정지점이 설정되는 프로세스의 모든 사건에 대하여 다른 프로세스의 사건들이 항상 인과관계를 만족하도록 순차적으로 수행되므로 인과관계 설정에 대한 오버헤드가 전혀 없다.

참 고 문 헌

[1] C. McDowell and D. Helmbold, "Debugging Concurrent Programs," *ACM Computing Surveys*, 21(4), pp.593-622, Dec. 1989.

[2] L. Gunaseelan and R. J. LeBlanc, Jr., "Debugging Objects and Threads in a Shared Memory System," *Proceedings of the USENIX Symp. on Experience with Distributed and Multiprocessor Systems*, pp.175-194, Sep. 1993.

[3] E. Fromentine, N. Plouzeau, and M. Raynal, "Replaying Distributed Execution," *2nd Intl. Workshop on Automated and Algorithmic Debugging*, pp.1-18, May 1995.

[4] 송후봉, 유재우, 백의현, "병행 프로그램 디버깅을 위한 결정적 재실행 시스템", *정보과학회 논문지(B)*, 제24권, 제2호, pp.218-230, 1997. 2.

[5] A. Claudio, J. Cunha, and M. Carmo, Monitoring and Debugging Message Passing Application with MPVisualizer, *Proceedings of the 8th Euromicro Workshop on Parallel and Distributed Processing*, pp.376-382, Jan. 2000.

[6] R. Konuru, H. Srinivasan, and J. Choi, "Deterministic Replay of Distributed Java Applications," *Proceedings of the 14th International Parallel and Distributed Processing Symposium*, pp.219-227, May 2000.

[7] 최동순, 김남훈, 김명호, "분산환경에서 병렬 프로그램 재실행을 위한 자바 디버거", *정보과학회 춘계학술발표회*, 2000.

[8] Hood, R, "The p2d2 Poject: Building a Portable Distributed Debugger," *Proceedings of SPDT'96*, May 1996.

[9] http://www.tc.cornell.edu/Parallel.Tools

[10] Kranzlmuller, D., Hugel, R., Volkert, J., "MAD-

Atop Down Approach to Parallel Program Debugging," *Proceedings of the HPCN'99*, April 1999.

[11] J. Kohl, G. Geist, "XPVM 1.0 User's Guide," *Oak Ridge National Laboratory, Oak Ridge, Tennessee*, http://netlib.uow.edu.au/pvm3/xpvm

[12] MPI and PVM User's Guide, http://www.techpubs.sgi.com/library

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

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

[15] C. Fidge, "Logical Time in Distributed Computing Systems," *IEEE Computer*, pp.28-33, Aug. 1991

[16] J. Fowler, W. Zwaenepoel, "Causal Distributed Breakpoints," *Proceedings of the 10th International Conference on Distributed Computing Systems*, pp.134-141, 1990.

[17] R. Netzer and B. Miller, "Optimal Tracing and Replay for Debugging Message-Passing Parallel Programs," *Proceedings of the International Conference on Supercomputing*, pp.502-511, Nov. 1992.

부록 1 종속 사건에 대한 사건 정보 기록

함수	사건 번호	사건이력 기록	비고
MPI_Init	1	None	사건벡터 초기화 인과관계 수신자 생성
MPI_Finalize	2	None	인과관계 수신자 소멸
MPI_Get_count, MPI_Get_element	3	None	사건벡터를 빼줌
MPI_Send (B, S, R)	4	송신 사건시간, 송신 사건 번호, 수신자	인과관계 메시지 수신
MPI_Isend (b, s, r)	5	(통신 종료시 기록) 송신 사건시간, 송신 사건 번호, 수신자, 수신 사건시간, 태그, 종료 사건, 종료 시간	인과관계 메시지는 종료시 수신하므로, 인과관계 정지 시간을 메시지 종료시까지 연장함
MPI_Recv	6	수신 사건시간, 수신 사건 번호, 송신자, 송신 사건시간, 태그	수신자에게 인과관계 메시지 전송
MPI_Irecv	7	(통신 종료시 기록) 수신 사건시간, 수신 사건 번호, 송신자, 송신 사건시간, 태그, 종료 사건, 종료 시간	None
MPI_Wait(*)	8	사건시간, 사건 번호, 송신 플래그, 송신자, 송신 사건시간, 수신자, 수신 사건시간	수신 사건에 대해서는 관계되는 송신자에게 인과관계 메시지를 송신하고, 송신사건에 대해서는 인과관계 수신을 대기함
MPI_Test	9	플래그 값이 참일 때, 사건시간, 사건 번호, flag 값, 송신 플래그, 송신자, 송신 사건시간, 수신자, 수신 사건시간 기록 inactive한 request에 대해서는 사건시간, 사건 번호, flag 값을 적은 후 나머지는 -1은 기록 플래그 값이 거짓이면, 사건시간, 사건 번호, flag 값은 기록	flag 값이 true일 때, MPI_Wait와 동일하게 수행함
MPI_Waitany	10	사건시간, 사건 번호, 인덱스 값, 송신 플래그, 송신자, 송신 사건시간, 수신자, 수신 사건시간	종료된 인덱스에 대하여는 MPI_Wait와 동일하게 수행함
MPI_Testany	11	플래그가 참일 때 : 사건시간, 사건 번호, 플래그, 인덱스 값, 송신 플래그, 송신자, 송신 사건시간, 수신자, 수신 사건시간 플래그가 거짓일 때 : 사건시간, 사건 번호, 플래그 (active한 요구가 종료시 요구 값을 MPI_REQUEST_NULL로 한 후 인덱스 값을 반환한다. inac- tive한 요구에 대해서는 플래그 값을 거짓으로 인덱스 값은 MPI_ UNDEFINED로 기록한다)	flag 값이 true일 때 종료된 인덱스에 대하여는 MPI_Wait와 동일하게 수행함
MPI_Waitall	12	종료된 모든 사건에 대하여 사건시간, 사건 번호, 인덱스 값, 송신 플래그, 송신자, 송신 사건시간, 수신자, 수신 사건시간 기록 inactive한 request에 대해서는 사건시간, 사건 번호, 인덱스 값은 적은 후 나머지는 -1을 기록	종료된 인덱스에 대하여는 MPI_Wait와 동일하게 수행함
MPI_Testall	13	플래그가 참일 때 : (active한 요구가 종료시) 사건 시간, 사건 번호, flag 값, 인덱스 값, 송신 플래그, 송신자, 송신 사건시간, 수신자, 수신 사건시간을 기록 (inactive한 요구에 대해서는) 사건 시간, 사건 번호, flag 값, 인덱스 값은 기록하고 나머지는 1을 기록 플래그가 거짓일 때 : 사건 시간, 사건 번호, 플래그	종료된 모든 인덱스에 대하여는 MPI_Wait와 동일하게 수행함
MPI_Waitsome	14	사건 시간, 사건 번호, 종료된 요구 개수를 먼저 적고, 각각의 종료된 요구에 대하여, active한 요구가 종료시 사건 시간, 사건 번호, 인덱스 값, 송신 플래그, 송신자, 송신 사건시간, 수신자, 수신 사건시간 기록하며, inactive한 요구에 대해서는 사건 시간, 사건 번호, 인덱스 값을 적은 후 나머지는 1을 기록한다.	종료된 모든 인덱스에 대하여는 MPI_Wait와 동일하게 수행함

함수	사건번호	사건이력 기록	비고
MPI_Testsome	15	사건 시간, 사건 번호, 종료된 요구 개수를 먼저 적고, 각각의 종료된 요구에 대하여, active한 요구가 종료시 사건 시간, 사건 번호, 인덱스 값, 송신 플래그, 송신자, 송신 사건시간, 수신자, 수신 사건시간 기록하며, inactive한 요구에 대해서는 사건시간, 사건 번호, 인덱스 값을 적은 후 나머지는 1을 기록한다. (종료시 요구가 없을 시는 요구 개수=0은 즉시 반환한다.)	종료된 모든 인덱스에 대하여는 MPI_Wait와 동일하게 수행함
MPI_Iprobe	16	플래그가 참일 때 : 사건시간, 사건 번호, 플래그 값, status 구조의 송신자, 태그, count 값을 기록한다.	None
MPI_Probe	17	사건시간, 사건 번호, status 구조의 송신자, 태그, count 값은 기록한다.	None
MPI_Start	18	송신 사건일 경우 : 송신 사건시간, 송신 사건 번호, 송신 플래그, 태그, 수신자, 수신 사건시간, 종료 사건, 종료 사건시간 수신 사건일 경우 : 수신 사건시간, 수신 사건 번호, 송신 플래그, 태그, 송신자, 송신 사건시간, 종료 사건, 종료 사건시간	MPI_Isend와 MPI_Irecv에서와 같이 수행 (송신 통신일 경우에 종료 구문에서 인과관계 수신을 대기함)
MPI_Startall	None	count 개수 만큼의 MPI_Start로 나누어서 수행	재실행시 MPI_Startall은 count 개수 만큼의 MPI_Start로 나누어서 수행한다. MPI_Isend와 MPI_Irecv에서와 같이 수행
MPI_Sendrecv	19	사건시간, 사건 번호, 수신자(destination), 송신자, 송신 사건시간, 태그	송신자에 대해서는 인과 관계 메시지를 송신하고, 메시지를 송신한 후 인과 관계 메시지를 대기함
MPI_Sendrecv_replace	20	사건시간, 사건 번호, 수신자(destination), 송신자, 송신 사건시간, 태그	송신자에 대해서는 인과 관계 메시지를 송신하고, 메시지를 송신한 후 인과 관계 메시지를 대기함
MPI_Request_free	21	사건시간, 사건 번호	큐에 있는 요구를 없애면서 종료 사건 시와 같은 행동을 함

홍 철 의

1985년 한양대학교 금속공학과 졸업(학사). 1989년 미국 New Jersey Institute of Technology 전산학(석사). 1992년 미국 Univ. of Missouri-Rolla 전산학(박사). 1992년 ~ 1997년 한국전자통신연구원 선임연구원. 1997년 ~ 현재 상명대학교 정보통신학부 조교수. 관심분야는 분산 및 병렬 알고리즘, 최적화 기법

김 영 준

1984년 고려대학교 산업공학과 졸업(학사). 1990년 미국 Univ. of Houston 전산학(석사). 1996년 미국 Univ. of Houston 전산학(박사). 1997년 ~ 현재 상명대학교 정보통신학부 조교수. 관심분야는 분산처리, 기계학습, 전문가 시스템