병렬프로그램의 경합조건을 수행 중에 효율적으로 탐지하기 위한 레이블링 기법

박 소 희[†] · 우 종 정[††] · 배 종 민[†††] · 전 용 기[††††]

요 약

병렬프로그램에서 경합 조건은 비결정적인 수행 결과를 초래하므로 디버깅을 위해 반드시 탐지되어야 한다. 이러한 경합을 수행 중에 탐지하는 기존의 기법들은 병행성 정보 생성 시에 공유 자료구조를 사용하여 심각한 병목현상을 발생시키거나, 병행성 정보 비교 시에 내포병렬성의 정도에 의존하는 비효율적인 시간 복잡도를 가진다. 본 논문에서는 개별 자료구조를 사용함으로써 병목현상을 제거하여 병행성 정보를 확장적으로 생성하며, 생성된 병행성 정보의 비교 시간을 상수적인 복잡도로 개선한 새로운 레이블링 기법을 제안한다. 그러므로 제안된 레이블링 기법의 확장성 및 효율성은 공유메모리와 메시지전달 프로그램뿐만 아니라 이를 혼합하여 사용하는 병렬프로그램에서도 효율적인 수행중 경합 탐지를 가능하게 한다.

A Labeling Scheme for Efficient On-the-fly Detection of Race Conditions in Parallel Programs

So-Hee Park[†] · Jong-Jung Woo[††] · Jong-Min Bae[†††] · Yong-Kee Jun[††††]

ABSTRACT

Race conditions, races in short, need to be detected for debugging parallel programs, because the races result in unintended non-deterministic executions. To detect the races in an execution of program, previous techniques use a centralized data structure which may incur serious bottleneck in generating concurrency information, or show inefficient time complexity which depends on the degree of nested parallelism in comparing any two of them. We propose a new labeling scheme in this paper, which is scalable in generating the concurrency information without bottleneck by using private data structure, and improves time complexity into constant in checking concurrency. The scalability and time efficiency therfore makes on-the-fly race detection efficient not only for programs with either shared-memory or message-passing, but also for programs with mixed model of the two.

키워드 : 경합조건(Race Condition), 수행중 탐지(On-the-fly Detection), 레이블링(Labeling), 확장성(Scalability), 효율성(Efficiency)

1. 서 론

공유메모리(shared-memory)[5, 13]와 메시지전달(message-passing)[8]을 혼합하여 사용하는 병렬 프로그램[2-4, 10, 16]은 논리적인 오류 외에도 경합조건(race conditions) 등의 병렬오류로 인한 비결정적인(non-deterministic) 수행을 초래하므로 기존의 순차프로그램보다 디버깅이 어렵다. 이러한 비결정적인 수행 결과의 주된 원인은 병렬로 수행되는 스레드들이 적절한 동기화 없이 공유메모리에 적어도 하나 이상의 쓰기사건으로 동시에 접근하거나, 혹은 스레드들 간의 메시지전달에 결정적인 순서화를 보장하지 않는 경우에 나타난다. 이러한 병렬프로그램 오류를 공유메모리 프로그램의 경우에 자료경합(data race)[1, 6]이라 하고, 메시지전달 프로그램의 경우에 메시지경합(message race)[8, 11]이라 한다. 경합 조건에 의한 프로그램의 비결정적인 수행은 프로그래머가 의도하지 않은 결과를 초래할 수 있으므로 병렬프로그램의 디버깅에 있어서 중요한 문제이다.

경합 조건을 탐지하기 위한 현실적인 기법으로, 프로그램 수행 중에 경합탐지에 필요한 정보만을 유지하는 수행중 경합탐지 기법(on-the-fly race detection)[1, 6, 11, 12]이 있다. 수행 중 경합탐지 기법은 수행 중에 경합을 탐지하여 보고하기 위한 경합탐지 프로토콜[6, 12]과 경합탐지 프로토콜을 적용시 병행한 사건간의 논리적인 병행성 여부를 판단하기 위한 병행성 정보 생성 기법[1, 6]으로 구성된다. 각 스레드들은 병행성 정보에 해당하는 유일한 값인 레이블(label)이 있어야만 수행사건 발생시에 스레드들간의 논리적 병행성을 검사하여 경합을 탐지하고 보고할 수 있다. 병행성 정보는

※ 본 연구는 정보통신부에서 지원하는 대학기초연구지원사업과 한국학술진흥재단에서 지원하는 신진연구인력 연구장려금으로 수행되었음.
† 준 회 원 : 경상대학교 대학원 컴퓨터과학과
†† 종신회원 : 성신여자대학교 컴퓨터정보학부 교수
††† 종신회원 : 경상대학교 컴퓨터과학과 교수
†††† 종신회원 : 경상대학교 컴퓨터과학과 교수, 컴퓨터·정보통신연구소 연구원(교신저자)
논문접수 : 2002년 7월 24일, 심사완료 : 2002년 12월 2일

수행중에 동적으로 생성 가능해야 하며, 어떠한 프로토콜을 적용하더라도 경합탐지가 가능해야 한다. 그러므로 수행중 경합탐지 기법의 효율성은 병행성 정보 생성효율성과 생성된 병행성 정보의 적용 효율성으로 크게 나눌 수 있으며, 각각 공간 및 시간복잡도로 나타낸다. 병행성 정보 생성효율성의 공간복잡도는 병행성 정보를 저장하기 위한 공간량이며, 시간복잡도는 병행성 정보를 생성하기 위한 시간이다. 생성된 병행성 정보의 적용 효율성의 공간복잡도는 경합을 탐지하기 위해 유지하게 되는 접근 역사와 메시지 역사에 대한 공간이며, 시간복잡도는 이전 수행사건과 현재 수행사건 사이의 논리적인 병행성을 결정하기 위한 시간이다. 수행 중에 경합을 탐지하기 위해 제안된 기존의 레이블링 기법들[1, 6]은 병행성정보 적용시에 효율적인 시간복잡도를 가지지만 병행성정보 생성시에 공유 자료구조를 사용하여 심각한 병목현상을 발생시키는 경우[6]이거나, 개별 자료구조를 사용하여 병행성 정보의 생성 시에 병목현상은 발생하지 않지만 병행성 정보의 적용시에 내포병렬성(nested parallelism)에 의존하는 비효율적인 시간복잡도를 가지는 경우[1]이다.

본 논문에서는 개별 자료구조를 사용함으로써 병목현상이 없이 병행성 정보를 확장적으로 생성하며, 생성된 병행성 정보의 비교 시간을 상수적인 복잡도로 개선하여 시간복잡도가 효율적인 DV(Distributed Vectors) 레이블링 기법을 제안한다. 제안된 레이블링 기법은 병행성 정보 생성시에 부모의 병행성 정보와 자신의 스레드 인덱스 값을 이용하므로 동시에 발생 가능한 모든 스레드들이 병행적으로 생성 가능하며, 병행성 정보 비교시에 하나의 벡터값에서 해당 식별자의 버전값을 찾아 비교하므로 상수적인 비교 시간을 가진다. 적용대상이 되는 프로그램은 공유메모리와 메시지전달 프로그램뿐만 아니라 이를 혼합하여 사용하는 병렬프로그램 즉, 프로그램 수행 중에 새로운 스레드를 생성하는 메시지전달 프로그램에서도 병행성 정보를 동적으로 생성함으로써 효율적인 수행중 경합 탐지가 가능하다. 실험대상이 되는 프로그램은 MPI(Message Passing Interface)[8]와 OpenMP(Open MultiProcessing)[5, 13]를 혼합하여 사용하는 프로그램이며, 자료경합을 탐지하기 위해서는 Dinning 프로토콜[6]을 사용하고, 메시지경합을 탐지하기 위해서는 Netzer 프로토콜[12]을 사용한다. 실험하기 위한 시스템 환경은 3개의 알파 프로세서 노드를 가지는 클러스터 시스템과 리눅스 레드햇 6.2로 구성하였다. 알고리즘의 구현은 C언어로 하였으며, 예제 프로그램은 MPI 프로그램에 OpenMP 디렉티브를 삽입하여 작성하였고, Omni OpenMP[15]로 컴파일하여 MPICH[9]로 실행하였다. 또한 병행성 정보 생성 시간만을 효과적으로 측정하기 위해서 각 노드마다 하나의 프로세서만 수행되도록 변형된(instrumented) 프로그램을 생성하여 실험하였다.

본 논문의 구성을 보면, 2절에서 공유메모리와 메시지전달의 혼합된 병렬프로그램에 대해 설명하고, 3절에서는 제안된 DV 레이블링 기법의 알고리즘과 병행성 검사를 보인다. 4절에서는 제안된 DV 레이블링과 기존의 레이블링 기법 중에서 확장적이면서 효율적인 BD 레이블링[1]을 복잡도 및 실험결과를 통해 비교 분석하고, 5절에서 결론을 맺는다.

2. 연구 배경

본 절에서는 공유메모리와 메시지전달을 혼합하여 사용하는 병렬프로그램에 대해 소개하고, 이러한 프로그램 모델을 위한 수행중 경합탐지 기법에 대해 설명한다.

2.1 MPI/OpenMP 프로그램

메시지전달 프로그램 유형의 대표적인 예는 MPI[8]가 있고, 공유메모리 프로그램 유형의 대표적인 예는 산업 표준인 OpenMP[5, 13]가 있다. MPI는 분산메모리 환경을 기반으로 개발되어, 각 노드의 지역메모리 자료를 메시지 송·수신을 통하여 복사해야 하므로 지나치게 작은 단위의 병렬성을 가지는 프로그램 수행시에 통신 부담이 매우 큰 단점[2-4, 10, 16]이 있다. OpenMP는 구현이 쉽고, 작은 단위의 병렬성(fine grain parallelism)을 가지는 프로그램 수행시에 효율적이지만, 공유메모리 환경을 기반으로 개발되어 분산메모리 시스템에서 사용할 수 없는 단점[2]이 있다.

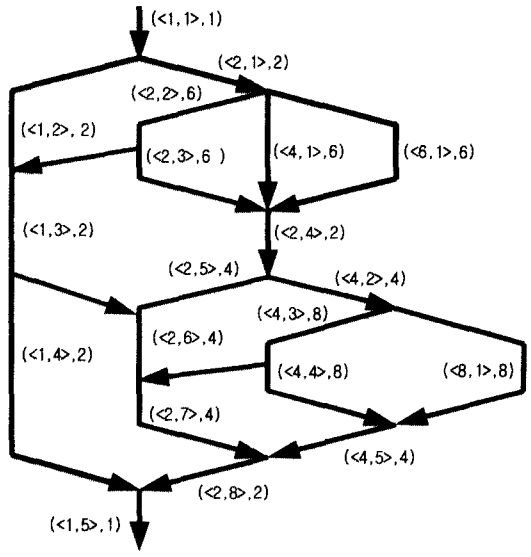
```
#define Comm MPI_Comm_World
int i1, i2, i3, i4, myid ;
MPI_Status st ;
MPI_Request rq ;
MPI_Init() ;
MPI_Comm_size ( Comm, &numprocs )
MPI_Comm_Rank ( Comm, &myid ) ;
#pragma omp parallel for
for ( i1 = 1 ; i1 <= 2 ; i1++ )
  if ( i1 == 1 ) {
    MPI_Recv ( MPI_Char, myid, 2, Comm, & st ) ;
    MPI_Isend ( MPI_Char, myid, 1, Comm, & rq ) ;
    for (flag = 0 ; !flag ; ) MPI_Test ( & rq, & flag, & st ) ; }
  else if ( i1 == 2 ) {
    # pragma omp parallel for
    for ( i2 = 1 ; i2 <= 3 ; i2++ ) {
      MPI_Isend ( MPI_Char, 0, 2, Comm, & rq ) ;
      for ( flag = 0 ; !flag ; ) MPI_Test ( &rq, &flag, &st ) ; }
    #pragma omp parallel for
    for ( i3 = 1 ; i3 <= 2 ; i3++ )
      if ( i3 == 1 ) {
        MPI_Recv ( MPI_Char, myid, 1, Comm, & st ) ;
        MPI_Recv ( MPI_Char, myid, 3, Comm, & st ) ; }
      else if ( i3 == 2 )
        #pragma omp parallel for
        for ( i4 = 1 ; i4 <= 2 ; i4++ )
          if ( i3 == 1 ){
          MPI_Isend ( MPI_Char, myid, 3, Comm, & rq ) ;
          for ( flag = 0 ; !flag ; ) MPI_Test(& rq, & flag, & st ) ; } }
MPI_Finalize ( ) ;
```

(그림 1) OpenMP/MPI의 혼합된 예제 프로그램

그러므로 최근에는 병렬프로그램의 수행 성능을 개선하기 위해서 SMP(Symmetric MultiProcessor) 클러스터에서 MPI

와 OpenMP를 혼합한 형태(hybrid mode)로 프로그래밍하는 것이 주된 경향이 되고 있다. 이러한 SMP 클러스터 시스템은 분산메모리를 사용하면서 각 노드 내에서는 공유메모리를 사용하게 된다. MPI/OpenMP의 혼합형 프로그램 모델은 MPI와 OpenMP로 각각 프로그래밍하여 수행한 것보다 효율적이며, 특히 MPI 프로그램이 부하 불균등을 보이거나, 작은 단위의 프로그램 수행으로 인해 확장성이 저하되거나, 자료의 복사 혹은 MPI 단위 프로세스 수의 제약으로 인해 메모리의 한계를 겪는 경우에 효율적인 장점[2-4, 10, 16]이 있다. (그림 1)은 MPI 프로그램에 OpenMP 디렉티브를 삽입하여 작성한 혼합형 예제 프로그램이다. 그림에서 MPI 프로그램은 MPI_Init()에 의해 시작되고 MPI_Finalize()에 의해 종료하게 된다. MPI 프로그램에 삽입된 OpenMP 디렉티브인 #pragma omp parallel for 영역은 스레드들을 병렬적으로 생성(fork)하고, 합류(join)하도록 하는 컴파일러 디렉티브이다. 각 스레드들 간의 메시지전달은 해당 메시지의 수행사건이 완료되어 다음의 메시지 수행이 가능하게 되었을 때만 다음 명령을 수행하게 되는 블럭킹(blocking) 타입과 해당 메시지의 수행사건이 발생된 것을 확인했을 때 곧바로 다음 명령을 수행하게 되는 넌블럭킹(non-blocking) 타입으로 구분된다. (그림 1)에서의 예제 프로그램과 같이 하나의 프로세스에 의해 수행되는 경우에, 송신 명령어로 블럭킹 타입인 MPI_Send()를 사용하면, 대응되는 수신 명령어가 수행완료되기 전인 경우이므로 해당 송신 메시지가 안전하게 수신완료되었는지를 확인할 수 없기 때문에 교착상태(deadlock)가 발생된다. 그러므로 송신 메시지가 송신버퍼에 버퍼링된 것만을 확인하고 곧바로 다음을 수행하게 하는 넌블럭킹 타입의 MPI_Isend() 함수를 사용해야 하며, 그 후에 버퍼링된 송신 메시지가 송신되었음을 확인하기 위해 MPI_Test() 함수를 사용한다. 수신 명령어로 블럭킹 타입인 MPI_Recv() 함수를 사용하면 해당 수신 메시지가 수신되지 않는 경우에 블럭킹되며, 문맥교환(context switching)을 통해서 대응되는 송신명령을 포함하는 수행을 계속하여 진행할 수 있다. 넌블럭킹 타입인 MPI_Irecv() 함수를 사용하면 MPI_Isend() 함수와 마찬가지로 MPI_Test() 함수를 사용하여 해당 수신 메시지가 수신버퍼에 수신된 것을 확인할 수 있으므로 순차 수행의 경우에는 MPI_Recv() 함수와 동일한 효과를 가진다. 본 예제에서는 MPI_Recv() 함수를 사용한다.

병렬프로그램에서 수행중에 경합을 탐지하려면 프로그램 수행의 동적인 특성을 고려하여야 한다. 이를 위해 프로그램 수행 중에 발생하는 모든 수행사건들 간의 논리적인 순서관계를 판단하여야 한다. 이러한 논리적 발생 순서는 POEG (Partial Ordered Execution Graph)[6]이라는 방향성 비순환 그래프로 나타낼 수 있다. (그림 2)는 (그림 1)의 예제 프로그램에 대해 병행성 정보를 생성한 결과를 POEG으로 보여준다. 정점(vertex)은 병렬 스레드의 생성과 합류를 의미하며, 각 정점들로부터 출발하는 간선(arc)은 그 정점에서 생성된 스레드를 말한다. 스레드간의 간선은 동기화를 나타내며, 본 논문에서 구현하는 프로그램에서는 메시지전달에 의한 동기화를 나타낸다. 각 스레드에 대한 정보는 본 논문에서 제안하는 새로운 레이블링 기법의 스레드 식별자를 보인 것으로서, 3절에서 상세히 설명한다. POEG은 병렬프로그램을 수행할 때 스레드들 간의 부분적인 순서(partial order) 관계를 나타내기 때문에 병렬 스레드들간의 발생후(happened-before)[6] 관계를 알 수 있다. 두 스레드 간에 발생후 관계가 성립하면 두 스레드는 순서화된(ordered) 관계이지만, 발생후 관계가 성립하지 않는 두 스레드는 병행한(concurrent) 관계이다.



(그림 2) POEG과 DV 레이블링의 스레드 정보

2.2 수행중 경합탐지

프로그램의 비결정적인 수행 결과를 야기시키는 주된 원인인 경합은 자료경합[1, 6]과 메시지경합[8, 11]으로 구분한다. 자료경합은 병렬로 수행되는 스레드들이 적절한 동기화 없이 공유메모리에 적어도 하나 이상의 쓰기 사건으로 동시에 접근하는 경우에 발생하며, 메시지경합은 스레드들 간의 메시지전달에서 결정적인 순서화를 보장하지 않는 경우에 발생한다. 경합에 의한 프로그램의 비결정적인 수행은 프로그래머가 의도하지 않은 결과를 초래하므로 디버깅을 위해 반드시 탐지되어야 한다. 이러한 경합을 수행 중에 탐지하여 보고하기 위한 기법은 경합탐지 프로토콜과 프로토콜을 적용 시에 병행한 스레드간의 논리적인 병행성 여부를 판단하기 위한 병행성 정보 생성 기법으로 구성된다.

공유메모리와 메시지전달의 혼합된 프로그램에서 자료경합을 탐지하기 위한 프로토콜은 Dinning 프로토콜[6]이 있으며, 메시지경합을 탐지하기 위한 프로토콜은 Netzer 프로

토콜[12]이 있다. Dinning 프로토콜은 동기화가 있는 공유 메모리 프로그램에서 자료경합을 탐지하기 위한 프로토콜로서, 공유 자료구조인 접근역사에 병행한 읽기 접근사건들을 저장하였다가 쓰기 접근사건이 발생하면 접근역사(access history)에 저장된 읽기 접근사건들과 병행성 여부를 판단하여 보고한다. 이때, 접근역사는 최악의 경우 최대 병렬성 만큼의 읽기 접근사건들을 유지하지만 쓰기 접근 사건은 하나만을 유지하게 된다. Netzer 프로토콜은 분산메모리 프로그램에서 메시지경합을 탐지하기 위한 프로토콜로서, 각 노드마다 지역 자료구조인 메시지역사(message history)를 가지며 이전 수신사건의 병행성 정보를 저장하였다가 현재 수신사건에 대한 송신사건과의 병행성 여부를 판단하여 보고한다. 이때, 메시지역사는 다음에 발생될 수신사건 시에 메시지경합을 탐지하기 위해 현재 수신사건의 병행성 정보만을 저장하게 된다. 이러한 경합탐지 프로토콜을 수행하기 위해 각 스레드들은 유일한 값인 병행성 정보가 있어야 하며, 수행 중에 동적인 생성이 가능해야 한다.

수행중 경합탐지 기법의 효율성은 병행성 정보 생성과 생성된 병행성 정보의 적용 측면으로 나눌 수 있다. 병행성 정보 생성효율성은 병행성 정보 생성시의 저장공간과 생성 시간에 대한 복잡도로 나타낼 수 있으며, 병행성 정보 적용 효율성은 경합을 탐지를 위해 유지하게 되는 저장공간과 수행사건들 간의 논리적인 병행성을 결정하기 위한 시간에 대한 복잡도로 나타낼 수 있다. 수행중 경합탐지를 위한 기존의 레이블링 기법들[1, 6]은 병행성 정보 적용시에 효율적인 시간복잡도를 가지지만 병행성 정보 생성시에 공유 자료구조를 사용하여 심각한 병목현상을 발생시키는 경우[6]이거나, 개별 자료구조를 사용하여 병행성 정보 생성 시에 병목현상은 발생하지 않지만 병행성 정보 적용시에 내포병렬성의 정도에 의존하는 비효율적인 시간복잡도를 가지는 경우[1]이다. 또한, 최근의 프로그래밍 경향인 MPI/OpenMP가 혼합된 프로그램 모델을 지원하지 못하여, 특정 프로그램 유형에만 적용 가능한 제한점을 가진다. 예를 들어, 타임스탬프(time-stamp)[7]를 이용하는 기법을 제외한 대부분의 레이블링 기법들은 공유메모리 모델인 OpenMP 프로그램에서만 적용 가능하다. 하지만 수행 중에 새로운 스레드를 생성하는 MPI-2[8]에서는 벡터 타임스탬프 기법도 사용할 수 없다. 즉, 기존의 레이블링 기법은 OpenMP와 MPI의 혼합형 프로그램을 지원하지 않기 때문에 공유메모리와 분산메모리의 각 모델 특성에 적합한 다른 기법을 사용해야 한다.

3. DV 레이블링

본 절에서는 공유메모리 기반의 OpenMP 프로그램과 메시지전달 기반의 MPI 프로그램에서도 모두 적용 가능하고, OpenMP/MPI의 혼합된 프로그램에서도 적용 가능한 새로운 DV 레이블링 기법을 제안한다.

```
1  DV_Init ( )
2    s := n ;
3    t := r + 1 ;
4    v := 1 ;
5    m := 1 ;
6    M := [1] ;
7  End DV_Init
```

(a) 초기화 알고리즘

```
1  DV_Fork ( )
2    s := sp×U ;
3    t := tp + (I - 1)×sp ;
4    if ( Mp[t] : = Ø ) then
5         v := 1 ;
6    else v := Mp[t] +1 ;
7    m := max { mp, t } ;
8    for i := 1 to m do
9         if ( i = t ) then
10             M[t] := v ;
11        else if ( i ≤ mp ) then
12             M[i] := Mp[i] ;
13        else M[i] := 0 ;
14   endfor
15 End DV_Fork
```

(b) 스레드 생성 알고리즘

```
1  DV_Join ( )
2    sc := sp ;
3    tc := tp ;
4    if t = tc then
5       vc := v + 1 ;
6    endif
7    if ( m ≤ mc ) then
8       k := 1 ;
9    else k := mc + 1 ;
10      mc := m ;
11   endif
12   for i := k to m do
13      if ( i = tc ) then
14         M[t] := vc ;
15      else Mc[i] := max { M[i], Mc[i] } ;
16   endfor
17 End DV_Join
```

(c) 스레드 합류 알고리즘

```
1  DV_Send ( )
2    v := v + 1 ;
3    M[t] := v ;
4  End DV_Send
```

(d) 메시지 송신 알고리즘

```
1  DV_Receive ( )
2    v := v + 1 ;
3    m := max { m, mr } ;
4    for i := 1 to m do
5       if ( i = t ) then
6          M[t] : = v ;
7       else M[i] := max {M[i], Mr[i]} ;
8    endfor
9  End DV_Receive
```

(e) 메시지 수신 알고리즘

(그림 3) DV 레이블링의 레이블 생성 알고리즘

3.1 레이블링 알고리즘

DV 레이블링 기법의 병행성 정보는 스레드 정보와 부가정보로 구성되며, 두 정보를 구분하는 구분자(delimiter)는 §로 나타낸다. 스레드 정보에는 접근역사 혹은 메시지역사에 저장되는 스레드 레이블인 $<t, v>$와 t를 구하기 위해 사용되는 s가 있다. 여기서, t는 각 스레드의 식별자를 의미하며, 새로 생성되는 각 스레드마다 고유한 정수값을 가진다. v는 각 스레드 식별자의 재사용에 대한 횟수인 스레드 버전(version)을 의미하며, 스레드의 생성과 합류, 그리고 메시지전달 시에 스레드 식별자가 재사용될 때마다 1씩 증가하게 된다. s는 동시에 발생 가능한 최대 병행 스레드 수인 스레드 공간(space)을 의미하며, 스레드 인덱스의 상한값(upper bound)으로 계산한다. 부가정보에는 이전에 수행한 스레드와의 병행성 여부를 검사하기 위한 버전벡터(version vector)인 M이 있다. M은 현재 시점까지 스레드가 알고 있는 모든 스레드들의 버전 값을 나타내며, 현재 시점까지 스레드가 알지 못하는 스레드들의 버전 값은 0이 된다.

DV 레이블링 기법은 병행성 정보를 생성하는 알고리즘과 병행성 정보의 병행성 여부를 검사하는 알고리즘으로 구성된다. 병행성 정보를 생성하는 알고리즘에는 초기화 알고리즘, 스레드 생성 알고리즘, 스레드 합류 알고리즘, 메시지전달 시의 송신·수신 알고리즘 등이 있다. (그림 3)의 병행성 정보 생성 알고리즘에서 사용되는 변수들의 아래첨자 p와 c는 각각 부모(parent)와 자식(child) 스레드 정보를 의미하며, r은 수신(receive) 되어진 스레드 정보를 의미한다.

〈표 1〉 DV 레이블링의 버전벡터

Thread-Info	Vector	Thread-Info	Vector
(<1, 1>, 1)	[1]	(<2, 6>, 4)	[3,6,0,1,0,1]
(<1, 2>, 2)	[2]	(<2, 7>, 4)	[3,7,0,3,0,1]
(<1, 3>, 2)	[3,2]	(<2, 8>, 2)	[3,8,0,5,0,1,0,1]
(<1, 4>, 2)	[4,2]	(<4, 1>, 6)	[1,1,0,1]
(<1, 5>, 2)	[5,8,0,5,0,1,0,1]	(<4, 2>, 4)	[1,4,0,2,0,1]
(<2, 1>, 2)	[1,1]	(<4, 3>, 8)	[1,4,0,3,0,1]
(<2, 2>, 6)	[1,2]	(<4, 4>, 8)	[1,4,0,4,0,1]
(<2, 3>, 6)	[1,3]	(<4, 5>, 4)	[1,4,0,5,0,1,0,1]
(<2, 4>, 2)	[1,4,0,1,0,1]	(<6, 1>, 6)	[1,1,0,0,0,1]
(<2, 5>, 4)	[1,5,0,1,0,1]	(<8, 1>, 8)	[1,4,0,2,0,1,0,1]

먼저, (그림 3)(a)는 병행성 정보를 생성하기 위해 사용되는 변수들을 초기화하는 알고리즘이다. s는 동시에 발생 가능한 최대 병행 스레드 수인 스레드 공간을 말하는데, 초기화 과정에서는 병렬로 수행되는 노드 수인 n을 입력받는다. t는 각 스레드의 식별자 의미하는데, 초기화 과정에서는 각 노드의 순위(rank)인 r을 입력받게 된다. 이때, 노드의 순위는 0부터 시작하므로 1 증가한 값을 가진다. v는 처음으로 사용되는 스레드 식별자를 가지게 되므로 1로 초기화한다. m은 현재 스레드의 버전벡터 M에 저장될 버전의 수를 의미하며, m의 값에 의해 M을 구하게 된다. 초기화 스레드에서 버전의 수가 1이므로 m은 1이 되고, 버전벡터 M은 스레드 식별자 t의 버전값 1만을 가지게 된다. 예를 들어, (그림 2)에서 초기화 스레드의 스레드 정보는 (<1, 1>, 1)이고, 부가정보인 버전벡터는 <표 1>에서와 같이 [1]이다.

(그림 3)(b)는 스레드 생성(fork) 명령 직후에 수행하여 병행성 정보를 만드는 스레드 생성 알고리즘이다. 즉, 스레드 생성 알고리즘 수행시 부모의 병행성 정보를 이용하여 현재 스레드의 병행성 정보를 생성하는 것이다. U는 스레드 인덱스의 상한값을 의미하며, I는 해당 스레드의 인덱스를 의미한다. 현재의 s값을 구하기 위해 s_p에 U를 곱하게 된다. s_p을 이용하여 현재 스레드 식별자 t를 구하고, 부모 스레드의 버전벡터 M_p에 t에 해당하는 버전 값이 없는 새로운 스레드 식별자인 경우는 버전 값 v를 1로 초기화한다. 만약 현재 스레드 식별자 t의 버전 값이 M_p에 있다면, M_p에서 해당 스레드 버전 값을 1 증가시킨다. m은 현재 스레드 식별자 t와 부모 스레드의 버전 수 m_p와 비교하여 최대값을 가지게 된다. 현재 스레드의 버전벡터 M은 m값만큼 각 스레드 식별자에 대한 버전 값들을 가지게 되는데, 먼저 m_p값만큼 M_p값에서 버전 값들을 가져오고 나머지는 0으로 채우게 되며, 해당 스레드 식별자의 버전 값을 현재 스레드 버전 값으로 갱신한다. 예를 들어, (그림 2)에서 스레드 생성 예는 (<1, 2>, 2)와 (<2, 1>, 2)의 스레드들이다. 스레드 인덱스의 상한값 U = 2이므로, s값 2는 이전 s_p인 1에다 현재 스레드 인덱스의 상한값 2를 곱하여 구한 것이다. 여기서, (<1, 2>, 2)의 스레드 식별자 t는 3번 수행문의 계산에 의해 1이 되고, 스레드 식별자가 재사용되었으므로 v는 1 증가한다. (<2, 1>, 2)의 스레드 식별자 t는 3번 수행문의 계산에 의해 새로운 스레드 식별자 2가 되었으므로 v는 초기값 1이 된다. 스레드 (<1, 2>, 2)의 버전벡터 M을 구하기 위해 m은 현재 스레드 식별자 t = 1과 부모 스레드의 버전 수 m_p = 1을 비교하여 최대값을 가지게 되므로 m = 1이 된다. 버전벡터 M은 m = 1만큼 가지되, 먼저 M_p값에서 m_p = 1 만큼 버전 값을 가져오고, 스레드 식별자 t = 1에 대한 현재 스레드 버전값 2로 갱신하므로 버전벡터 M은 [2]를 가지게 된다. 마찬가지로, 스레드 (<2, 1>, 2)의 버전벡터 M을 구하기 위해 m은 현재 스레드 식별자 t = 2와 부모 스레드의 버전 수 m_p = 1을 비교하여 최대값을 가지게 되므로 m = 2가 된다. 버전벡터 M은 M_p값에서 m_p = 1 만큼 버전 값을 가져오고, 스레드 식별자 t = 2에 대한 현재 스레드 버전값 1로 갱신하므로 버전벡터 M은 [1, 1]을 가지게 된다. 그러므로 버전벡터 생성 예는 <표 1>에서와 같이 스레드 (<1, 2>, 2)는 [2]이고, 스레드 (<2, 1>, 2)는 [1, 1]이다.

(그림 3)(c)는 스레드 합류(join) 명령 직전에 수행하여 병

행성 정보를 만드는 스레드 합류 알고리즘이다. 즉, 스레드 합류 알고리즘 수행시 부모의 병행성 정보를 이용하여 자식 스레드의 병행성 정보를 생성하는 것이다. 현재 스레드들의 합류로 인해 자식 스레드는 스레드 생성 이전의 스레드로 복귀하므로 s_c는 부모 스레드의 s_p를 그대로 가져오고, 자식 스레드 식별자 t_c도 부모 스레드의 t_p를 그대로 가져온다. 자식 스레드 식별자의 버전 값인 v_c는 현재 스레드 식별자 t와 t_c가 같을 경우에만 현재 스레드의 버전 값인 v를 1 증가시킨다. 자식 스레드의 버전벡터 M_c는 m이 m_c보다 작거나 같다면 현재 스레드 식별자의 버전벡터를 그대로 가지고, 현재 스래드의 식별자에 대해서 v값만을 갱신하며, 나머지 버전벡터는 최대값으로 유지한다. 하지만, m이 m_c보다 크다면 현재 시점에서 생성된 M_c는 그대로 두고 m_c 이후의 스레드 식별 자에 대한 버전벡터 값만을 최대값으로 유지한다. 예를 들어, (그림 2)에서 스레드 합류의 예는 (<2, 4>, 2)의 스레드이다. s_c는 부모의 s_p = 2를 그대로 가지게 되고, 스레드 식별자 t_c도 부모의 t_p = 2를 그대도 가진다. 자식 스레드 식별자의 버전 값인 v_c는 현재 스레드들 (<2, 3>, 6)와 (<4, 1>, 6) 그리고 (<6, 1>, 6) 중에서 현재 스래드 t와 t_c = 2가 같은 경우에만 v값을 1 증가시키므로, 해당 스레드 식별자 t = 2의 버전 값 v = 3을 1증가시킨 4를 가진다. 자식 스레드 (<2, 4>, 2)의 버전벡터 M_c는 현재 스레드가 (<6, 1>, 6)이면 m = 6이 되고, m_c는 2 또는 4가 되므로 m값이 m_c보다 크기 때문에 현재 시점에서 생성된 버전벡터 M_c는 그대로 두고, m_c이후의 스레드 식별자에 대한 버전벡터 값만을 최대값으로 유지한다. 그러므로, 스레드 (<2, 4>, 2)의 버전벡터 생성 예는 <표 1>에서와 같이 [1, 4, 0, 1, 0, 1]이다.

(그림 3)(d)는 메시지 전달시의 송신 알고리즘으로서, 송신 메시지 이후의 병행성 정보를 생성한다. 송신메시지 이후에는 현재 스레드 식별자의 버전을 1증가시킨 후 버전벡터 M에서 해당 스레드의 버전을 갱신한다. 예들 들어, (그림 2)에서 송신메시지 이후의 스레드 생성 예는 (<2, 3>, 6)의 스레드이다. 이전 스레드 (<2, 2>, 6)에서 버전 값만이 1 증가됨을 알 수 있다. 버전벡터 역시 해당 스레드 식별자의 버전 값만이 1 증가됨을 알 수 있다. 그러므로, 스레드 (<2, 3>, 6)의 버전벡터 생성 예는 <표 1>에서와 같이 [1, 3]이다.

(그림 3)(e)는 메시지전달 시의 수신 알고리즘으로서, 수신메시지 이후의 병행성 정보를 생성한다. 메시지를 송신한 스레드로부터 버전벡터인 m_r과 M_r을 전달받게 된다. 현재 스레드 식별자의 버전을 1 증가시킨다. 버전벡터 M은 전달된 m_r과 현재 m을 비교하여 최대값 만큼의 버전 값들을 생성하게 되는데, 전달된 M_r과 메시지를 수신한 스레드의 버전벡터인 M을 비교하여 최대값들을 유지한다. 예를 들어, (그림 2)에서 수신메시지 이후의 스레드 생성 예는 (<1, 3>, 2)의 스레드이다. 현재 스레드에서 버전 값만 1증가됨을 알 수 있다. 여기서, 버전벡터 M을 구하기 위해 m은 m_r = 2와 현재 m = 2 중에서 최대값을 가지게 되므로 m = 2가 된다. 버전벡터 M은 m = 2 만큼의 버전 값들을 가지게 되는데, 메시지 송신 시에 전달된 버전 벡터과 비교하여 최대값들을 유지하고 스레드 식별자 t = 1에 대한 현재 스레드 버전값 3으로 갱신한다. 그러므로, 스레드 (<1, 3>, 2)의 버전벡터 생성 예는 <표 1>에서와 같이 [3, 2]이다.

3.2 병행성 검사

(그림 4)는 병행성 정보의 병행성 여부를 검사하는 알고리즘이다. 사용되는 변수들의 아래첨자 x와 y는 임의의 스레드를 의미한다. 임의의 스레드 x에 대한 병행성 정보를 L_x라 하고, 임의의 다른 스레드 y에 대한 병행성 정보를 L_y라 하자. L_x의 스레드 버전값 v_x와 L_y의 버전벡터 M_y 중에서 L_x의 스레드 식별자에 해당하는 버전값 $M_y[t_x]$를 비교하게 된다. 이때, v_x가 $M_y[t_x]$ 보다 작거나 같으면 순서화된 관계로 결정하게 되고, 그렇지 않으면 병행한 관계로 결정하게 된다. 왜냐하면, v_x가 $M_y[t_x]$보다 작거나 같다는 것은 스레드 y가 스레드 x 보다 이후에 수행되어 M_y의 스레드 버전값들이 갱신되었음을 의미하기 때문이다.

```
1  DV_Ordered ( Lx, Ly )
2    if ( My[tx] ≠ ∅ ) and ( My[tx] ≥ vx ) then
3         return true ;
4    else return false ;
5    endif
6  End DV_Ordered
```

(그림 4) DV 레이블링의 병행성 검사 알고리즘

(그림 2)의 POEG에서 보여진 두 스레드 (<2, 2>, 6)과 (<4, 5>, 4)는 발생후 관계가 성립하므로 순서화된 관계이다. (그림 2)와 <표 1>를 참고하여 병행성 검사의 과정을 살펴보자. 스레드 (<2, 2>, 6)의 병행성 정보를 L_x라 하고, 스래드 (<4, 5>, 4)의 병행성 정보를 L_y라 할 때, 각각의 병행성 정보는 다음과 같다.

L_x : (<2, 2>, 6) § [1, 2]
L_y : (<4, 5>, 4) § [1, 4, 0, 5, 0, 1, 0, 1]

예를 들어, t_x = 2에 대한 버전값 v_x = 2와 L_y의 버전벡터 중에서 t_x = 2에 해당하는 위치의 버전값 $M_y[2]$ = 4를 비교한다. 이때, $M_y[2]$가 v_x 보다 크므로 순서화 된 관계로 결정하게 된다. 다른 예로서, (그림 2)의 POEG에서 보여진 두 스레드 (<2, 5>, 4)와 (<4, 4>, 8)는 발생후 관계가 성립하지 않으므로 병행한 관계이다. (그림 2)와 <표 1>를 참고하여 병행성 검사의 과정을 살펴보자. 스레드 (<2, 5>, 4)의 병행성 정보를 L_x라 하고, 스레드 (<4, 4>, 8)의 병행성 정보를 L_y라 할 때, 각각의 병행성 정보는 다음과 같다.

L_x : (<2, 5>, 4) § [1, 5, 0, 1, 0, 1]
L_y : (<4, 4>, 8) § [1, 4, 0, 4, 0, 1]

예를 들어, $t_x = 2$에 대한 버전값 $v_x = 5$와 L_y의 버전벡터 중에서 $t_x = 2$에 해당하는 위치의 버전값 $M_y[2] = 4$를 비교한다. 이때, $M_y[2]$가 v_x보다 작으므로 병행한 관계로 결정하게 된다. 병행한 관계의 경우는 그 역의 비교에서도 성립되어야 한다. 그 역을 살펴보면, $t_y = 4$에 대한 버전값 $v_y = 4$와 L_x의 버전벡터 중에서 $t_y = 4$에 해당하는 위치의 버전값 $M_x[4] = 1$을 비교한다. 이때, $M_x[4]$가 v_y보다 작으므로 병행한 관계로 결정하게 된다.

제안된 DV 레이블링 기법의 병행성 검사 알고리즘은 공유메모리 프로그램에서의 자료 경합탐지 프로토콜과 메시지전달 프로그램에서의 메시지 경합탐지 프로토콜에 모두 적용 가능하다. 공유메모리를 위한 프로토콜의 경우에는 이전 접근사건과 현재 접근사건 사이의 논리적인 병행성 여부를 결정하고, 다음에 발생될 수행사건과의 병행성을 검사하기 위해 접근역사를 유지한다. 그리고 메시지전달을 위한 프로토콜의 경우에는 이전 수신사건과 현재 수신사건에 대응되는 송신사건과의 병행성 여부를 결정하고, 다음에 발생될 수행사건과의 병행성을 검사하기 위해 메시지 역사를 유지하게 된다.

4. 분석 및 실험

본 절에서는 제안된 DV 레이블링 기법과 기존의 레이블링 기법 중에서 가장 확장적이면서 효율적인 BD 레이블링 기법을 이론 및 실험으로 비교 분석한다.

4.1 BD 레이블링 기법

BD 레이블링 기법[1, 14]은 동기화가 있는 프로그램에 적용할 수 있으므로 공유메모리와 메시지전달의 혼합형 프로그램에 확장 가능하고, DV 레이블링 기법과 마찬가지로 개별 자료구조를 사용하여 병행성 정보 생성 시에 병목현상이 발생하지 않는 확장성을 가진다. 그러므로 기존의 레이블링 기법 중에서 가장 확장적이고 효율적인 BD 레이블링 기법과 제안된 DV 레이블링 기법의 효율성을 비교 분석하는 것은 필요하다.

BD 레이블링 기법의 병행성 정보도 DV 레이블링기법과 같이 스레드 정보와 부가정보로 구성되며, 두 정보를 구분하는 구분자는 §로 나타낸다. 스레드 정보는 접근역사 혹은 메시지역사에 저장되는 스레드 레이블 (b, d)로서, b는 고유한 스레드 식별자를 의미하고, d는 병행성 정보 생성 시에 증가하는 타임스탬프 값을 의미한다. 부가정보는 이전에 수행한 스레드와의 병행성 여부를 검사하기 위한 정보의 집합인 S로 나타낸다. S는 (t_i, T_{i+1}, d_i)로 구성된 노드들이 각 송신 스레드들로부터 수신 메시지전달이 있을 경우에 내포수준마다 트리 형태로 추가된다. 사용되는 변수들의 아래첨자 i는 각 노드가 생성된 내포수준을 의미하고, t_i는 현재 스레드의 인덱스 값, T_{i+1}는 현재 스레드에서 생성되는 다음 내포수준에서의 전체 스레드의 수, d_i는 현재 스레드의 타임스탬프 값을 의미한다. T는 현재 스레드가 알 수 없으므로 0의 값을 가지게 되며, 현재 스레드가 새로운 스레드들을 생성하는 시점에서 T값을 생성하게 된다. 따라서 BD 레이블링 기법의 경우에는 임의 스레드의 병행성 정보뿐만 아니라, 임의 스레드 생성시점에 대한 병행성 정보도 생성하는 것이 필요하다.

병행성 여부 검사를 위해 임의의 스레드 x에 대한 병행성 정보를 L_x라 하고, 임의의 다른 스레드 y에 대한 병행성 정보를 L_y라 하자. BD 레이블링 기법은 DV 레이블링 기법과 마찬가지로 L_x의 스레드 정보와 L_y의 병행성 정보를 직접 비교하지 않고, L_y의 부가정보와 비교하여 병행성 여부를 검사하게 된다. 이때, L_x의 스레드 식별자 b를 이용하여 L_x의 첫 번째 내포깊이의 t_1을 구하고, L_x의 t_1과 L_y의 부가정보 중에서 첫 번째 t_1을 비교하게 되는데, 부가정보가 트리 형태로 구성되어 있으므로 분기되는 시점에서의 부가정보들 중에서 L_x의 t_1과 L_y의 t_1가 같은 노드를 선택하여 비교를 시작하게 된다. 만약, 서브 트리에서 분기되는 경우에도 L_x의 t_i와 L_y의 t_i가 같은 노드를 찾아 계속 비교하게 된다. 비교 시에 L_x의 스레드 정보에서 타임스탬프 d와 L_y의 첫 번째 부가정보 중에서 d_1과 비교하는데, L_y의 d_1이 L_x의 d보다 크거나 같으면 순서적 관계로 결정하고, 그렇지 않다면 L_x의 두 번째 내포깊이의 t_2를 구하고 L_y의 t_2와 일치하지 않으면 병행한 관계로 결정하고, 일치한다면 위와 동일한 방법으로 반복하여 비교한다. 이러한 병행성 여부의 검사는 부가정보가 내포깊이만큼의 항목들로서 구성되어 있으므로 최악의 경우에 내포깊이만큼 반복하여 비교된다.

4.2 이론적 비교

<표 2>는 DV 레이블링 기법과 BD 레이블링 기법에 대해 병행성 정보의 생성효율성과 병행성 정보의 적용효율성을 최악의 경우(worst-case)의 공간 및 시간복잡도로 나타낸 것이다. 병행성 정보 생성효율성에서의 Space는 병행성 정보를 저장하기 위한 공간복잡도를 의미하고, Time은 각 스레드들의 병행성 정보 생성시에 소요되는 시간복잡도를 의미한다. 병행성 정보 적용효율성에서의 Space는 병행성 여부를 결정하기 위해 병행성 정보와 비교하게 되는 스레드 레이블의 공간복잡도를 의미하고, Time은 병행성 여부를 결정하는데 소요되는 시간복잡도를 의미한다. 이러한 효율성에 영향을 주는 인수들은 프로그램의 최대 병렬성(T)과 내포깊이(N) 등이다.

〈표 2〉 DV와 BD 레이블링의 최악의 경우 복잡도

Labeling	Concurrency Info. Generation		Concurrency Info. Application	
	Space	Time	Space	Time
DV	$O(T^2)$	$O(T)$	$O(1)$	$O(1)$
BD	$O(T^2)$	$O(T)$	$O(1)$	$O(N)$

먼저, 병행성 정보 생성효율성 측면에서 공간복잡도의 경우를 보자. DV 레이블링 기법은 병행성 정보 $(<t, v>, s) \S M$이 최악의 경우에 $O(T)$개 만큼 생성되며, 각 스레드들이 다른 스레드들과의 송·수신한 메시지전달을 모두 고려한 M의 저장공간을 $O(T)$만큼 필요로 하므로, 전체적으로 최악의 경우에 $O(T^2)$의 공간복잡도를 가진다. BD 레이블링 기법도 DV 레이블링 기법과 마찬가지로 병행성 정보 $(b, d) \S S$가 최악의 경우에 $O(T)$개 만큼 생성되며, 각 스레드들이 다른 스레드들과의 송·수신한 메시지전달을 모두 고려한 S의 저장공간을 $O(T)$만큼 필요로 하므로 전체적으로 최악의 경우에 $O(T^2)$의 공간복잡도를 가진다. 시간복잡도의 경우를 보자. DV 레이블링 기법은 하나의 스레드가 병행성 정보를 생성할 때, 다른 스레드들과의 송·수신한 메시지전달을 모두 고려한 M을 생성하는데 소요되는 시간이 최악의 경우에 $O(T)$의 복잡도를 가진다. BD 레이블링 기법도 DV 레이블링 기법과 마찬가지로 다른 스레드들로부터 송·수신한 메시지전달을 고려한 S를 생성하는데 소요되는 시간은 최악의 경우에 $O(T)$만큼 필요하다.

병행성 정보의 적용효율성 측면에서 공간복잡도의 경우를 보자. DV 레이블링 기법은 임의의 두 스레드에 대한 병행성 정보 비교 시에, 현재 스레드는 생성된 병행성 정보 중에서 부가정보인 M이 필요하며, 이전 스레드는 병행성 정보 중에서 상수 크기의 스레드 레이블 $<t, v>$만을 필요로 하므로 병행성 여부를 결정하기 위해 유지되는 이전 스레드에 대한 공간복잡도는 $O(1)$이 된다. BD 레이블링 기법도 DV 레이블링 기법에서와 마찬가지로 임의의 두 스레드에 대한 병행성 정보 비교시에, 현재 스레드는 생성된 병행성 정보 중에서 부가정보인 S가 필요하며, 이전 스레드는 병행성 정보 중에서 상수 크기의 스레드 레이블 $<b, d>$만을 필요로 하므로 병행성 여부를 결정하기 위해 유지되는 이전 스레드에 대한 공간복잡도는 $O(1)$이 된다. 시간복잡도의 경우를 보자. DV 레이블링 기법은 임의의 두 스레드를 L_x, L_y라 할 때, L_x의 스레드 버전값 v_x와 L_y의 버전벡터 M_y 중에서 L_x의 스레드 식별자에 해당하는 버전값 $M_y[t_x]$를 비교하여 병행성 여부를 검사할 수 있기 때문에 $O(1)$이 된다. 하지만 BD 레이블링 기법은 임의의 두 스레드를 L_x, L_y라 할 때, L_x의 타임스탬프값 d와 L_y의 부가정보 집합 S_y의 타임스탬프값 d를 비교하여 병행성 여부를 검사하게 된다. 이때 S_y가 트리 형태로 구성되어 있으므로 루트(root) 노드 이후의 해당 노드를 찾아서 비교하게 되며, 최악의 경우에 단말(terminal) 노드까지 비교하게 된다. 루트 노드에서 단말 노드까지의 트리의 수준은 N에 의존적이므로 병행성 여부를 검사하기 위한 시간복잡도는 $O(N)$이 된다.

이러한 비교결과를 요약하면, DV 레이블링 기법과 BD 레이블링 기법에 대한 병행성 정보 생성효율성 측면에서 공간복잡도와 시간복잡도는 동일하고, 병행성 정보의 적용효율성 측면에서 공간복잡도는 동일하나 시간복잡도는 BD 레이블링 기법보다 DV 레이블링 기법이 더 효율적이다. 본 연구에서는 DV 레이블링 기법의 시간적 적용효율성을 입증하기 위해서 추가적으로 실험적 비교를 수행하였다.

4.3 실험적 비교

실험을 위한 시스템 환경은 3개의 알파 프로세서 노드를 가지는 클러스터 시스템을 리눅스 레드햇 6.2운영체제로 구성하였다. 대상 프로그램은 MPI 프로그램에 OpenMP 디렉티브를 삽입하여 작성하였고, Omni OpenMP[15]로 컴파일하여 MPICH[9]로 실행하였다. 대상 프로그램에 대한 실험은 C언어로 구현된 함수를 삽입한 변형된(instrumented) 프로그램으로 생성하여 실험하였다.

〈표 3〉 커널 벤치마크 프로그램의 실험 기준

# of threads forked	Nesting Depth	Pro. Version	# of threads per node	Total # of threads	Uninstr. Exec. (U)
5	2	${}_5P_2$	25	75	0.010
	3	${}_5P_3$	125	375	0.013
	4	${}_5P_4$	625	1,875	0.015
10	2	${}_{10}P_2$	100	300	0.014
	3	${}_{10}P_3$	1,000	3,000	0.015
	4	${}_{10}P_4$	10,000	30,000	0.018

<표 3>은 본 실험을 위해서 개발된 커널 벤치마크 프로그램의 특성들을 보인다. 커널 벤치마크 프로그램은 각 내포수준마다 생성되는 스레드의 수를 각각 5, 10인 두 가지 유형이 있고, 각 유형에 대해 내포깊이가 2, 3, 4인 단일방향(one-way) 내포병렬성을 가진 프로그램들이다. 각 스레드 내에 접근사건의 수는 10으로 하였으며, 읽기 접근사건과 쓰기 접근사건의 비율은 각각 50%로 하였다. 메시지전달은 하나의 노드가 각 내포 수준마다 한 스레드에서 다른 두 노드의 같은 내포 수준의 한 스레드에 대해 각각 한번씩 송·수신하도록 하였다. 프로그램의 버전은 각 내포수준마다 생성되는 스레드 수와 내포깊이로 명명한 것이다. 전체 스레드 수는 전체 노드에서 생성되는 스레드의 수이다. 프로그램 수행시간은 프로그램 변형 이전의 프로그램을 수행한 시간이다.

<표 4>는 각 커널 벤치마크 프로그램에 대한 DV레이블링 기법과 BD 레이블링 기법의 수행시간을 측정한 결과이다. 측정된 단위시간은 초(second)이며, 블럭킹 시간을 제외한 순수한 프로그램을 실행 시간만을 측정하기 위하여

〈표 4〉 커널 벤치마크 프로그램의 실험 결과(단위 : 초)

Pro. Version	Thread Labeling (L)		Race Detection Protocol						Instrumented Exec. (I)			
			Dinning (P_d)		Netzer (P_m)		Total (P)		Partial (I_d)		Total (I)	
	DV	BD	DV	BD	DV	BD	DV	BD	DV	BD	DV	BD
${}_5P_2$	0.089	0.006	0.010	0.059	0.014	0.011	0.024	0.070	0.109	0.075	0.123	0.086
${}_5P_3$	0.177	0.008	0.274	2.045	0.007	0.217	0.281	2.262	0.464	2.066	0.471	2.283
${}_5P_4$	0.547	0.012	6.595	55.341	0.038	1.079	6.976	56.955	7.158	55.900	7.538	56.983
${}_{10}P_2$	0.119	0.008	0.127	0.892	0.008	0.022	0.135	0.914	0.260	0.914	0.268	0.936
${}_{10}P_3$	0.708	0.015	13.786	95.150	0.177	0.622	13.963	95.772	14.509	95.180	14.686	95.802
${}_{10}P_4$	29.424	0.025	1399.226	-	18.893	-	1418.119	-	1428.668	-	1447.561	-

clock() 함수를 사용하였다. 제외된 시간은 각 프로세스의 스케쥴링에 의한 블럭킹 시간과, 메시지 전달시에 표준모드 송·수신 명령어인 MPI_Send()와 MPI_Recv() 함수의 블럭킹 시간이다.

<표 4>에서 전체 수행시간(I)은 커널 벤치마크 프로그램의 수행시간(U), 병행성 정보 생성시간(L), 경합을 탐지하기 위해 프로토콜 적용시간(P)을 모두 포함한 시간이다. 이러한 시간들의 측정을 위해 각 노드마다 하나의 프로세스로 수행되게 하였다. L은 프로토콜 함수를 제외하고 병행성 정보 생성 함수만을 삽입하여 측정할 수 있으며, P는 병행성 정보가 생성되어야만 프로토콜을 적용할 수 있으므로 I에서 L과 U를 뺀 값 $P = (I - L - U)$으로 구할 수 있다. 자료경합을 탐지하기 위한 Dinning 프로토콜의 적용시간(P_d)은 Netzer 프로토콜 함수를 제외하고 수행한 수행시간(I_d)에서 L과 U를 뺀 값 $P_d = (I_d - L - U)$으로 구할 수 있다. 메시지경합을 탐지하기 위한 Nezter 프로토콜의 적용시간(P_m)은 자료경합과 메시지경합에 대한 프로토콜 적용시간을 모두 의미하는 P에서 P_d을 뺀 값 $P_m = (P - P_d)$으로 구할 수 있다. 혹은 Dinning 프로토콜 함수를 제외하고 수행한 수행시간(I_m)에서 L과 U를 뺀 값 $P_m = (I_m - L - U)$으로도 구할 수 있다.

비교 분석한 결과, DV 레이블링 기법이 BD 레이블링 기법보다 내포깊이와 스레드의 수가 증가할수록 효율적임을 알 수 있다. 프로그램 버전 ${}_5P_3$의 경우를 예를 들어보자. L은 DV 레이블링 기법이 BD 레이블링 기법에 비해 수행시간이 많은데, L과 P를 모두 포함한 I에 있어서 DV 레이블링 기법이 BD 레이블링 기법보다 약 4.8배 정도 효율적이다. 특히, P에 있어서 DV 레이블링 기법이 BD 레이블링 기법보다 약 8.0배 정도 효율적이다. 이는 한 수행사건에 대한 병행성 검사의 시간복잡도가 DV 레이블링 기법의 경우에 $O(1)$이고 BD 레이블링 기법의 경우에 $O(N)$이기 때문이다. 만약, 동일한 프로그램 버전에 있어서 한 스레드 내에 접근사건 혹은 메시지전달 사건이 증가한다면, 매 수행사건에 대한 병행성 검사 횟수가 증가하므로 BD 레이블링 기법이 DV 레이블링 기법보다 병행성 검사에 따른 시간이 상대적으로 증가하게 될 것이다. <표 4>에서 프로그램 버전 ${}_{10}P_4$의 경우에, DV 레이블링 기법의 I는 블럭킹 시간을 포함하여 약 4시간 정도 소요되었으나, BD 레이블링 기법의 I는 20시간 이상 수행하였음에도 불구하고 수행 결과를 얻을 수 없었다.

5. 결 론

수행중 경합탐지 기법의 효율성은 병행성 정보 생성효율성과 생성된 병행성 정보의 적용효율성으로 크게 나눌 수 있으며, 각각 공간 및 시간복잡도로 나타낸다. 수행 중에 경합을 탐지하기 위해 제안된 기존의 기법들은 병행성 정보 생성시에 공유 자료구조를 사용하여 심각한 병목현상을 발생시키거나, 병행성 정보 비교 시에 내포병렬성의 정도에 의존하는 비효율적인 시간복잡도를 가진다.

본 논문에서는 공유메모리와 메시지전달을 가지는 병렬프로그램에서 병행성 정보 생성시에 개별 자료구조를 사용함으로써 병목현상 없이 병행성 정보를 확장적으로 생성하며, 생성된 병행성 정보의 비교 시간을 상수적인 복잡도로 개선하여 시간복잡도가 효율적인 DV 레이블링 기법을 제안하였다. 제안된 레이블링 기법은 병행성 정보 생성 시에 부모의 병행성 정보와 자신의 스레드 인덱스 값을 이용하므로 동시에 발생 가능한 모든 스레드들이 병행적으로 생성 가능하며, 병행성 정보 비교시에 임의 스레드에 대한 버전값과 다른 스레드의 버전 벡터에서 임의 스레드의 식별자에 대응되는 버전 값을 찾아 비교하므로 상수적인 비교 시간을 가진다. 적용대상이 되는 프로그램은 공유메모리와 메시지전달 프로그램뿐만 아니라 이를 혼합하여 사용하는 프로그램에서도 병행성 정보를 동적으로 생성함으로써 효율적인 수행중 경합 탐지가 가능하다. 향후 DV 레이블링 기법의 병행성 정보 생성에 따른 시간 부담을 줄이는 연구가 필요하다.

참 고 문 헌

[1] Audenaert, K., "Maintaining Concurrency Information for On-the-fly Data Race Detection," *Int'l Conf. on Parallel Computing : Fundamentals, Applications and New Directions*, Bonn, Germany, pp.319-326, Sept., 1997, Advances in Parallel Computing, Elsevier, North-Holland, Amsterdam, Vol.12, 1998.

[2] Mark, B. and L. Smith, "Programming Next Generation HPC Systems : the Mixed-mode Model," *Next Generation HPC Systems and the Grid*, The Univ. of Edinburgh Conf. and Training Centre, Edinburgh, U.K., Sept., 2001.
[3] Cappello, F. and D. Etiemble, "MPI versus MPI+ OpenMP on the IBM SP for the NAS Benchmarks," *Supercomputing*, Dallas, TX, 2000.
[4] Caubet, J., J. Gimenez, J. Labarta, L. DeRose and J. Vetter, "A Dynamic Tracing Mechanism for Performance Analysis of OpenMP Applications," *Int'l Workshop on OpenMP Applications and Tools*, LNCS, 2104, West Lafayette, Indiana, pp.53-67, July, 2001.
[5] Dagum, L. and R. Menon, "OpenMP : an Industry-Standard API for Shared-Memory Programming," *Computational Science and Engineering*, 5(1), IEEE, 46-55, Jan.-March, 1998.
[6] Dinning, A. and E. Schonberg, "An Empirical Comparison of Monitoring Algorithms for Access Anomaly Detection," *2nd Symp. on Principles and Practice of Parallel Programming*, ACM, pp.1-10, March, 1990.
[7] Fidge, C. J., "Partial Orders for Parallel Debugging," *1st Workshop on Parallel and Distributed Debugging*, ACM, pp.183-194, May, 1988.
[8] Gropp, W., E. Lusk and A. Skjellum, Using MPI : Portable Parallel Programming with the Message-Passing Interface, *The MIT Press*, 2nd Edition, 1999.
[9] Gropp, W. and E. Lusk, User's Guide for mpich, a Potable Implementation of MPI Version 1.2.2, *Univ. of Chicago Press*, 1996.
[10] Hoeflinger, J., B. Kuhn, W. Nagel, P. Petersen, H. Rajic, S. Shah, J. Vetter, M. Voss and R. Woo, "An Integrated Performance Visualizer for MPI/OpenMP Programs," *Workshop on OpenMP Applications and Tools*, LNCS, 2104, West Lafayette, Indiana, pp.40-52, July, 2001.
[11] Neyman, M., M. Bukowski and P. Kuzora, "Efficient Replay of PVM Programs," *6-th European PVM/MPI User's Group Meeting*, Barcelona, Spain, pp.83-90, Sept., 1999.
[12] Netzer, R. H. B. and Miller, B. P., "Optimal Tracing and Replay for Debugging Message-Passing Parallel Program," *The J. of Supercomputing*, 8(4), pp.371-388, Oct., 1994.
[13] OpenMP Architecture Review Board, OpenMP Fortran Application Program Interface, Version 2.0, Nov., 2000.
[14] Park, S, M. Park and Y. Jun, "A Comparison of Scalable Labeling Schemes for Detecting Races in OpenMP Programs," *Int'l Workshop on OpenMP Applications and Tools*, LNCS, 2104, West Lafayette, Indiana, pp.68-80, July, 2001.
[15] Sato, M., S. Satoh, M. Kusano and Y. Tanaka, "Design of OpenMP Compiler for an SMP Cluster," Sweden, Sept., 1999.
[16] Smith, L. and J. M. Bull, "Development of Mixed Mode MPI/ OpenMP Applications," *Int'l Workshop on OpenMP Applications and Tools*, San Diego, July, 2000.

박 소 희
e-mail : shpark@race.gsnu.ac.kr
1989년 경상대학교 전자계산학과 졸업 (학사)
1996년 경남대학교 대학원 컴퓨터공학과 졸업(공학석사)
2002년 경상대학교 대학원 컴퓨터과학과 졸업(공학박사)
관심분야 : 분산 및 병렬처리, 운영체제

우 종 정
e-mail : jwoo@cs.sungshin.ac.kr
1982년 경북대학교 컴퓨터공학과 졸업 (학사)
1982~1988년 산업연구원 책임연구원
1988~1993년 Univ. of Texas at Austin 전기컴퓨터공학과 졸업 석사 및 박사
1998~1999년 Univ. of Texas at Austin 전기컴퓨터공학과 방문교수
1993년~현재 성신여자대학교 컴퓨터정보학부 교수
관심분야 : 컴퓨터구조, 멀티미디어시스템, 전자상거래, 원격교육, CAD

배 종 민
e-mail : jmbae@base.gsnu.ac.kr
1980년 서울대학교 사범대학 수학교육과 (학사)
1983년 서울대학교 대학원 계산통계학과 (석사)
1995년 서울대학교 대학원 계산통계학과 (박사)
1982년~1984년 한국전자통신연구소 연구원
1997년~1998년 Virginia Tech. 객원연구원
1984년~현재 경상대학교 전임강사, 조교수, 부교수, 교수
관심분야 : XML 데이터베이스통합, 정보검색, 디지털 라이브러리, 데이터마이닝

전 용 기
e-mail : jun@nongae.gsnu.ac.kr
1980년 경북대학교 컴퓨터공학과 졸업 (학사)
1982년 서울대학교 컴퓨터공학부 졸업 (석사)
1993년 서울대학교 컴퓨터공학부 졸업 (박사)
1982년~1985년 한국전자통신연구소 연구원
1985년~현재 경상대학교 컴퓨터과학과 교수, 컴퓨터·정보통신연구소 연구원
관심분야 : 운영체제, 분산 및 병렬처리, 프로그래밍 환경