OpenMP 프로그램을 위한 효율적 병행성 정보의 생성기법†

하옥균[1], 김선숙[2], 전용기[1]
[1]경상대학교 정보과학과, [2]경상대학교 항공특성화대학원
{jassmin, sskim, jun}@gnu.ac.kr

An Efficient Scheme for Creating Concurrency Information in OpenMP Programs

Ok-Kyoon Ha[1], Sun-Sook Kim[2], Yong-Kee Jun[1]
[1]Dept. of Informatics, Gyeongsang National University
[2]Specialized Graduate School for Aerospace Engineering, Gyeongsang National University

요 약

OpenMP 프로그램의 수행 중에 발생하는 자료 경합과 같은 병행성 오류는 디버깅을 위하여 반드시 탐지되어야만 한다. 그러나 이를 탐지하는 것은 어려운 일이다. 접근사건의 발생 후 관계를 기반으로하는 경합 탐지 기법은 프로그램의 수행 중에 발생하는 스레드의 병행성 정보를 식별하기위한 레이블을 생성하고, 생성된 스레드의 레이블을 기반으로 공유변수에 접근하는 사건을 접근역사를 통해 감시함으로써 경합을 탐지한다. 이러한 경합 탐지의 방법에서 레이블 생성을 위한 NR 레이블링 기법은 병행성 정보생성 시에 지역자료 구조를 사용함으로써 병목현상이 발생하지 않으며, 접근역사에 저장하는 레이블의 크기가 상수 값을 갖는 공간적 효율성을 제공한다. 또한 부모스레드의 정보역사를 정렬된 리스트 형태로 가져 병행성 정보 비교 시에 이진탐색이 가능하므로 시간적 효율성을 가지는 우수한 기법이다. 그러나, NR 레이블링은 레이블의 생성시에 부모스레드의 정보역사를 유지하기 위해서 내포 병렬성의 깊이에 의존하는 시간적 비용이 요구된다. 본 논문에서는 부모스레드의 정보역사 유지를 위해 상수적인 시간 및 공간적 복잡도를 갖도록 NR 레이블링 기법을 개선한다. 합성 프로그램을 이용하여 실험한 결과에서 개선된 기법은 최대 병렬성의 증가에 따라 레이블의 생성과 유지시 기존의 기법보다 평균 4.5배 빠르고, 레이블링을 위해 평균 3배 감소된 기억공간을 요구하며, 내포 병렬성에 의존적이지 않음을 보인다.

1. 서 론

OpenMP[1,2] 표준 API는 공유 메모리(Shared memory)를 기반으로 하는 병렬 프로그래밍 모델로써 표준 C/C++와 Fortran 77/90을 확장하여 병렬화하는 컴파일러 디렉티브(Compiler diretives)와 수행시간 라이브러리(Run- time libraries)들의 집합이다. 이러한 OpenMP 프로그램은 일반적인 순차 프로그램에 스레드 병렬화를 위한 디렉티브를 삽입하여 병렬로 수행되어지고, 동기화 디렉티브를 삽입함으로써 스레드의 동기화가 있는 병렬 수행이 가능하다. 본 논문에서 대상으로 하는 OpenMP 프로그램은 스레드 동기화를 위한 디렉티브를 사용하지 않고 스레드를 병렬로 수행하는 내포된 병렬 프로그램 모델로 분기(Fork)에 의해 병렬 스레드가 생성되고 합류(Join)에 의해 하나로 합쳐진다.

공유 메모리 기반의 병렬 프로그램[3](Parallel programs)인 OpenMP 프로그램에서 경합[4,5](Races)은 병행하여 수행되는 스레드에 의해 적절한 동기화 없이 적어도 하나 이상의 쓰기 접근으로 동일한 공유 메모리에 접근이 이루어질 때 발생하는 병행성 오류[6](Concurrency bugs)이다. 프로그램의 수행 중에 발생하는 경합은 의도하지 않는 비결정적 수행의 결과를 초래할 수 있기 때문에 디버깅을 위하여 반드시 탐지(Detection)되어야 한다. 경합을 탐지하는 것은 병행성을 갖는 스레드의 수행 구조에 대한 이해와 경합 조건(Race Condition)을 만족하기 위한 시간적 비용이 요구되기 때문에 어려운 일이다.

OpenMP와 같은 병렬 프로그램의 경합을 수행 중에 탐지(On-the-fly detection)하는 기법 중 접근 사건의 발생 후 관계[7](Happened-before relation)의 분석을 이용하는 방법은 프로그램의 수행 중에 발생하는 스레드의 생성 시 마다 식별자인 레이블을 부여하는 레이블링[8-13](Labeling)기법이 요구된다. 이때 레이블을 생성하고 유지하기 위한 시간 · 공간적 복잡도(Complexity)는 경합 탐지의 성능에 결정적인 역할을 한다. 따라서, 경합 탐지를 위한 비용의 절감은 레이블링 기법의 시간적 · 공간적 효율성 향상을 통해 이루어질 수 있다.

프로그램의 수행 중 스레드 레이블을 생성하는 레이블링 기법들 중에서 NR 레이블링[11,14]은 내포 병렬성이 존재하는 병렬 프로그램 모델을 지원하고 레이블의 생성 시 지역자료 구조를 사용하여 병목현상이 발생하지 않으며, 접근역사에 저장하는 레이블의 크기가 상수 값을 갖는 공간적 효율성을 제공한다. 또한 부모스레드의 정보역사를 정렬된 리스트 형태로 가져 병행성 정보 비교 시에 이진탐색(Binary search)이 가능하므로 시간적 효율성을 가지는 우수한 기법이다. 그러나 NR 레이블링은 레이블의 생성시에 부모스레드의 정보역사를 유지하

† "본 연구는 지식경제부 및 정보통신산업진흥원의 대학 IT연구센터 지원사업의 연구결과로 수행되었음" (NIPA-2010-(C1090-1031-0007))

기 위해서 내포 루프의 깊이에 의존하는 시간 · 공간적 비용이 요구된다.

본 논문에서는 NR 레이블링을 위해 레이블의 생성과 유지 시에 이전 스레드의 부모 정보를 참조하는 방법으로 내포 깊이에 의존적이지 않고, 상수적인 시간 및 공간적 복잡도를 갖는 기법으로 개선한다. 합성 프로그램을 이용하여 실험한 결과에서 개선된 NR 레이블링은 레이블의 생성과 유지 시 기존의 기법보다 평균 4.5배 빠르고, 레이블링을 위해 요구되는 기억공간은 평균 3배 감소하며, 내포 깊이에 의존적이지 않음을 보인다.

2절에서는 OpenMP 프로그램에서의 자료경합과 병렬 스레드를 위한 레이블링 기법에 대해서 설명한다. 3절에서는 NR 레이블링과 제안하는 개선된 레이블링 기법 및 구현에 대해서 설명한다. 4절에서는 제안된 기법의 효율성 실험결과에 대해서 살펴보고, 마지막 5절에서는 결론과 향후과제를 제시한다.

2. 연구배경

본 절에서는 대상으로 하는 OpenMP 프로그램과 이 프로그램에서 발생할 수 있는 오류인 자료경합과 탐지를 위한 병행성 정보의 필요성에 대해서 설명한다.

2.1 병렬 프로그램에서의 자료경합

OpenMP 프로그램[1,2]은 표준 C/C++과 Fortran 77/90을 기반으로 작성된 공유메모리 병렬 프로그램을 위한 프로그램 모델로 POSIX 스레드나 MPI와는 달리 OpenMP 디렉티브(Directives)를 삽입하는 고수준 프로그래밍을 지원하여, 순차적 프로그램을 병렬화하기에 용이하므로 산업 표준으로 자리 잡고 있다. OpenMP에서 제공하는 컴파일러 디렉티브에는 병렬화 디렉티브, 작업 공유 디렉티브, 데이타 환경 디렉티브, 동기화 디렉티브 등이 있다. 병렬화 디렉티브는 새로운 스레드 그룹을 생성하는 것으로 'parallel'이 있고, 새로운 스레드의 생성 없이 기존의 스레드에 작업을 분할하여 수행하도록 하는 'do/for', 'section', 그리고 'single' 등이 있다. 데이터 환경을 위한 디렉티브는 데이타의 접근범위를 지정하는 'private'과 'shared'가 있으며, 동기화를 위한 디렉티브는 'critical'과 'barrier' 등이 있다. 따라서 결합된 병렬 디렉티브인 'parallel for', 'parallel section' 등에 의해 시작되는 병렬 구간에서 병렬 스레드들의 생성(Fork)이 이루어지고, 이들 병렬 구간이 끝나는 곳에서 병렬 스레드들의 합류(Join)가 이루어 진다.

병렬 프로그램의 수행은 방향성이 있는 비순환 그래프인 POEG (Partial Order Execution Graph)[10,15]으로 나타낼 수 있다. POEG에서의 정점(Vertex)은 병렬 스레드(T)의 생성과 합류를 나타내며, 정점들 사이의 간선(Arc)은 정점에서 생성된 스레드를 나타낸다. 그리고 각 스레드에 접근하는 읽기와 쓰기 사건을 원으로 표시하고, r과 w로 나타내어 접근사건의 유형을 구분한다. 여기서 사용되는 숫자는 특정 수행 시에 접근사건들의 발생 순서를 의미한다. 그림 1은 내포병렬성이 있는 프로그램 모델을 POEG으로 나타낸 것이다. POEG은 병렬

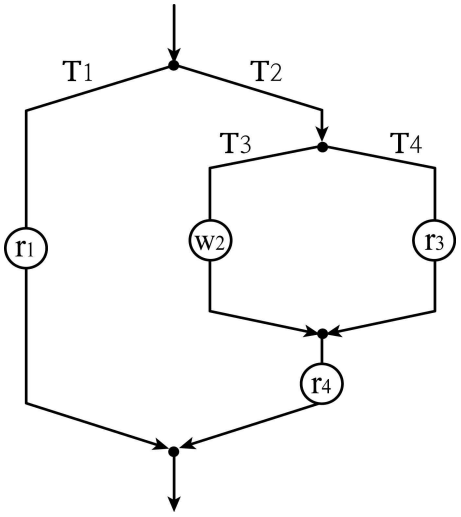

그림 1 내포 병렬성이 있는 프로그램 모델의 POEG

프로그램의 수행 시에 스레드들 간의 부분적 순서(Partial order) 관계를 나타낸다. 두 사건사이에 경로가 존재하면 두 사건이 순서화(Ordered)관계에 있고 경로가 존재하지 않으면 이 두 사건은 병행(Concurrent)관계에 있다. 그림에서 r1과 w2, w2와 r3은 그들 사이에 경로가 존재하지 않으므로 서로 병행관계이고, w2와 r4, r3과 r4는 그들 사이에 경로가 존재하므로 선행관계에 있다.

이렇게 병행관계에 있는 두 개의 접근사건이 적어도 하나의 쓰기 사건을 포함하고 하나의 공유변수에 대해서 적절한 동기화가 없이 나타날 때 발생하는 병행성 오류[6](Concurrency bugs)를 경합[4,5](Races)이라고 하며, ei-ej로 표시한다. 그림 1에서는 경합 {r1-w2, w2-r3}이 존재 한다. 이러한 경합은 프로그램 상에 의도하지 않은 비결정적(Non-deterministic)인 수행 결과를 초래하므로 OpenMP와 같은 병렬 프로그램의 디버깅을 위해서 반드시 탐지되어야 한다. 그러나 경합을 탐지하는 것은 병행성을 갖는 스레드의 수행 구조에 대한 이해와 경합 조건(Race condition)을 만족하기 위한 시간적 비용이 요구되기 때문에 어려운 일이다.

2.2 경합탐지를 위한 병행성 정보

병렬 프로그램의 경합을 수행 중에 탐지하는 기법은 수행 중에 발생한 각 스레드들에서 발생하는 공유변수에 대한 접근 사건들을 감시하고, 이를 공유 변수의 접근 역사[10,15](Access history)에 기록하여, 접근 사건이 발생할 때마다 접근 역사에 기록된 사건과 현재 발생한 사건을 Lamport의 발생 후 관계[7] (Happened-before relation)에 따라 순서 또는 병행 관계를 분석하여 경합의 여부를 판단한다.

이러한 경합 탐지기법에서 필요로하는 정보들은 공유변수와 이들에 관련된 접근 사건들에 대한 병행성 정보들이다. 접근 사건에 대한 병행성 정보는 접근 사건들이 발생한 스레드들의 논리적 병행관계를 결정하기 위해 필요하다. 논리적 병행성은 스레드들의 논리적 발생 순서

를 표시해주는 레이블(Label)에 의해 결정되는데, 이 레이블은 프로그램의 수행에서 유일한 값을 가지기 때문에 생성되는 모든 다른 스레드들과 구별할 수 있는 유일성을 가진다. 병렬 프로그램에서 발생하는 스레드의 논리적 병행성 정보를 유지하는 레이블을 생성하고 유지하는 기법을 레이블링(Labeling)이라고 하며, Task Recycling[8], English Herbew[9], Offset Span[10], Nest Region[11], Breadth Depth[12,14], Vector Clocks[13] 등 다양한 기법들이 소개되고 있다.

이들 레이블링 기법들이 갖는 공통점은 레이블의 생성과 유지를 위한 공간적 복잡도(Space complexity) 및 수행 시간 복잡도(Time complexity)를 모두 고려하여, 한 스레드의 레이블은 조상(Ancestor) 스레드의 정보를 유지해야 한다는 것이다. 이는 경합 탐지의 성능을 결정하는 주요한 요인이 되며, 경합 탐지를 위한 비용의 절감은 레이블을 생성하는 기법의 효율성을 증가시킴으로서 달성할 수 있기 때문에 보다 효율적인 레이블링 기법의 개발이 꾸준히 연구되어 오고 있다.

3. 효율적인 병행성 정보의 생성

본 절에서는 효율적인 경합탐지를 위한 레이블링 기법 중 OpenMP 프로그램과 같이 합류-분기 프로그램 모델에서 효과적인 NR 레이블링의 동작원리와 특성에 대해 설명하고, 기존의 기법이 스레드 연산에 의해 레이블의 생성과 유지시 시간·공간적 복잡도 측면에서 갖는 내포 깊이 만큼의 의존성을 극복하는 개선된 NR 레이블링을 제시한다.

3.1 NR 레이블링

병렬 스레드의 논리적 병행성 정보인 레이블을 생성하고 유지하는 기법들 중에서 NR 레이블링[11,14]은 내포 병렬성이 존재하는 프로그램 모델을 지원하고 레이블의 생성 시 지역자료 구조를 사용하여 병행성 정보를 생성하므로 최대 병렬성(Maximum parallelism)에 비례하는 병목현상이 발생하지 않으며, 접근역사에 저장하는 레이블의 크기가 상수 값을 갖는 공간적 효율성을 제공한다. 또한 한 스레드가 갖는 부모 스레드의 정보역사를 정렬된 리스트 형태로 가져 병행성 정보 비교 시에 이진 탐색(Binary search)이 가능하므로 시간적 효율성을 가지는 우수한 기법이다.

NR 레이블링에서의 레이블 생성, 유지, 그리고 병행성 비교 방법에 대해서 살펴보면, 레이블의 생성 시 지역자료 구조를 각 스레드가 내포할 수 있는 정수로 분할하여 활용한다. 이때 레이블이 가질 수 있는 정수의 최대값은 시스템이 표현할 수 있는 최대의 정수값으로 가정한다. 최초 발생한 스레드로부터 현재 스레드까지의 합류(Join)된 회수를 카운트한 λ와 내포 영역(Nest region)의 시작 값 α, 내포 영역의 끝 값 β를 사용하여 $[\lambda, <\alpha,\beta>]$의 형태로 단일 방향 영역 OR(One-way region)을 구성하고, 각 레이블에 부모 스레드의 정보역사 리스트인 OH와 그 길이를 나타내는 ξ를 추가적으로

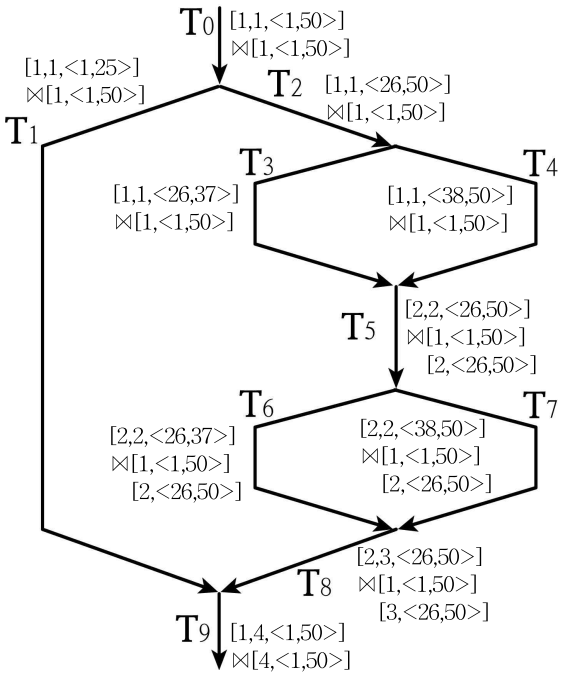

그림 2 다중 방향 루프 모델에서의 NR 레이블링의 예

구성하여 $[\xi, OR] \bowtie OH$ 로 재표현 함으로써 병렬 스레드의 병행성을 유지한다.

그림 2는 다중 방향 루프(Multi-way loop) 프로그램 모델을 대상으로 NR 레이블링을 적용한 예를 보인 것으로, 최대 정수 값을 50으로 설정하였다. 최초의 스레드 T_0는 <1, 50>의 내포 영역을 갖고, 최초로 합류된 스레드이므로 λ_0는 1이 되어 OR_0은 [1,<1,50>]이 되며, 합류된 스레드이므로 자신의 OR정보를 OH에 등록한다. 이때 OH에 동일한 OR이 존재하면 리스트에서 λ만 변경한다. 최종적으로 T_0 스레드를 위한 레이블은 [1, 1,<1,50>]⋈[1,<1,50>]이 된다. T_1과 T_2는 초기 스레드 T_0에서 두 개로 분기된(forked) 자식 스레드(Child-thread)이므로 내포 영역으로 <1,25>와 <26,50>을 각각 할당 받는다. 분기된 스레드는 부모 스레드로부터 λ, ξ, 그리고 OH 리스트를 복사하여 생성한다.

NR 레이블은 비교하는 스레드의 내포 영역이 겹쳐지면 순서관계이며, 겹쳐지지 않으면 병행관계임을 쉽게 알 수 있는 기법이다. 임의의 T_i 스레드와 T_j 스레드가 존재할 때, 각 스레드는 $<\alpha_i, \beta_i>$와 $<\alpha_j, \beta_j>$의 내포 영역을 가진다. 이때 $\alpha_i < \beta_j \wedge \alpha_j > \beta_i$의 조건을 만족하면 병행관계이고 $T_i \parallel T_j$로 표기하며, $\alpha_i \leq \beta_j \wedge \alpha_j \leq \beta_i$의 조건을 만족하면 두 스레드는 병행하지 않아 순서관계이고 $i < j$ 일 때 $T_i \rightarrow T_j$ 로 표기한다.

예를 들어 그림 2에서 T_1 스레드와 T_3 스레드의 병행성 관계를 비교하기 위해서는 λ_1과 λ_3을 비교하여 OR_1과 OH_3을 대상으로 할 것인지, OR_1과 OR_3를 대상으로 할 것인지를 결정한다. $\lambda_1 = \lambda_3 = 1$로 동일한 부모 스레드 정보를 갖고 있기 때문에 단순 OR간의 비교를 통해 1<37∧25<26으로 병행관계 조건을 만족하여 $T_1 \parallel T_3$

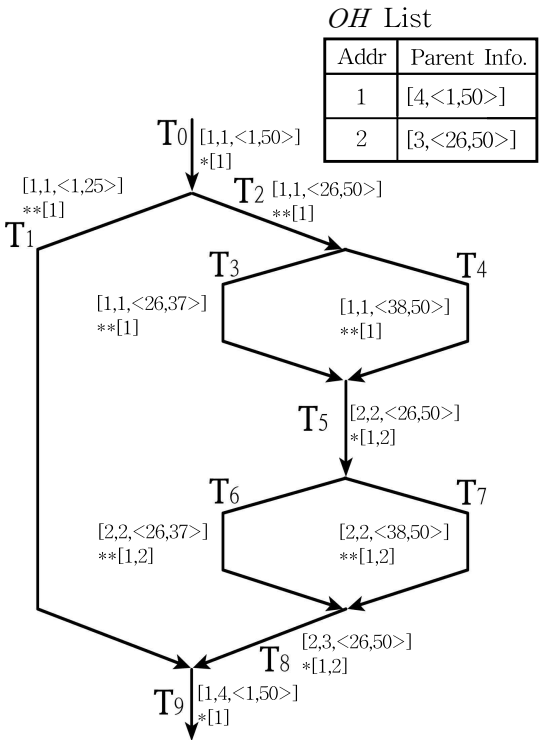

그림 3 개선된 NR 레이블링의 예

임을 알수 있다. 그리고 T_3 스레드와 T_7 스레드의 병행성 관계 비교에서 $\lambda_3 < \lambda_7$로 서로 다른 부모 스레드 정보를 보유하고 있기 때문에 T_7 스레드의 OH_7에서 가장 최근 부모 스레드 정보인 OR_5을 사용하여 비교한다. 따라서 $26 \leq 50 \wedge 26 \leq 37$을 만족하므로 T_7은 T_3과 병행하지 않고 발생 후 관계에 의해 $T_3 \rightarrow T_7$임을 알 수 있다.

3.2 개선된 NR 레이블링

앞서 살펴본 바와 같이 기존 NR 레이블링은 내포 병렬성이 존재하는 분기-합류(fork-join) 프로그램 모델에서 효과적인 기법이다. 그러나 레이블의 매 생성 시마다 부모 스레드의 정보역사 리스트인 *OH*를 재생성하고 유지하기 때문에 내포 깊이에 비례하는 시간 및 공간적 복잡도를 요구한다. 본 절에서 기존 NR 레이블링의 개선은 *OH*를 대상으로 스레드의 생성을 위한 분기와 합류 연산이 발생할 때, 이전에 발생한 스레드의 부모 정보를 참조만하고 재생성하지 않게함으로써 내포 깊이에 의존적이지 않고, 상수적인 시간 및 공간적 복잡도를 제공한다.

NR 레이블링의 개선을 위해 스레드의 *OR*은 기존의 방식을 그대로 따르고, *OH*의 자료구조와 유지정책을 변경한다. 먼저 생성된 스레드마다 생성하고 유지하던 *OH*를 모든 스레드를 위한 하나의 *OH*만 생성하고 리스트로 유지하게 설계한다. 그리고 각 스레드별로 자신의 *OH*를 리스트로부터 선별적으로 참조하기 위한 지시자(Pointer)를 추가한다. 그림 3은 제시된 방법을 이용하여 그림 2의 NR 레이블링을 개선한 예를 보인다.

개선된 기법에서는 합류 연산에 의해 생성된 스레드는 자신의 *OR*을 *OH* 리스트에 추가하고 그림 3과 같이 '*' 기호로 표시한다. 이때, 지시자가 참조하는 위치의 내포 영역 값이 같으면 λ만 갱신한다. 그림 3의 합류된 스레드 T_0, T_5, 그리고 T_9에서 T_0는 *OH*의 첫 번째 위치에 자신의 *OR*인 [1,<1,50>]을 추가하고 지시자는 그 위치를 참조한다. 그리고 T_5가 발생하게 되면, 현재의 지시자가 참조하는 *OR*의 내포 영역이 다르므로 *OH*에 자신의 *OR*[2,<26,50>]을 추가하고 지시자를 2개의 부모 정보를 참조하도록 갱신한다. 마지막으로, T_9의 경우 현재의 지시자가 참조하는 *OR*의 내포 영역이 <1,50>으로 동일하므로, *OH* 리스트의 λ값을 1에서 4로 갱신하고 지시자는 갱신하지 않는다.

다음으로 분기 연산에 의해 생성된 스레드는 분기되기 이전 스레드의 지시자를 참조하고, 이를 위해 '**'기호를 사용하여 구분한다. 그림 3에서 분기된 스레드 T_2는 T_0 스레드의 지시자를 참조하여 부모 스레드 정보인 [1,<1,50>]을 유지한다. T_3, T_4 스레드 역시 T_2 스레드의 참조하는 지시자를 참조하여 동일한 부모 스레드를 유지한다. 따라서 개선된 NR 레이블링은 스레드의 합류와 분기와 같은 레이블의 생성 시 부모 스레드 정보의 생성과 유지하기위해 지시자의 참조만을 사용하기 때문에 시간 · 공간적으로 $O(1)$의 복잡도를 갖는다.

개선된 NR 레이블링은 라이브러리 형태의 레이블링 엔진으로 C언어를 사용하여 구현하고, 분기를 위한 eNR_Forker 모듈, 합류를 위한 eNR_Joiner 모듈로 구성한다. eNR_Forker는 OpenMP 프로그램의 "#pragma parallel for"와 같은 디렉티브에 의해서 생성되는 스레드에 레이블 정보를 생성하고 유지하는 역할을 담당하며, eNR_Joiner는 병렬 스레드의 합류 시 레이블 정보의 생성과 삭제 그리고 유지를 담당한다.

4. 실험

본 절에서는 합성 프로그램의 집합을 이용하여 개선된 기법의 정확성을 실험하고, 효율성을 기존의 기법과 비교 분석한다. 실험결과 개선된 기법은 레이블의 생성 시 정확하게 스레드의 병행성 정보를 생성하고 유지하며, 기존의 기법에 비해 효율적으로 개선되어 내포 깊이에 의존적이지 않음을 보인다.

4.1 실험 설계

개선된 NR 레이블링의 레이블의 생성 및 유지와 효율성을 비교 실험하기 위해 OpenMP를 사용하는 합성 프로그램(Synthetic program)을 설계하고 작성한다. 정확성을 실험하기위한 합성 프로그램은 내포 병렬성과 다중 방향성의 유무에 따라 내포 깊이, 다중 방향성의 차수, 그리고 최대 병렬성에 변화를 주어 다양한 프로그램 수행 모델을 반영할 수 있게 작성한다. 효율성 비교를 위해서는 내포 병렬성과 다중 방향성의 유무에 따라 최대 병렬성을 변화시키면서 작성한다.

실험을 위한 환경은 2.6 커널의 리눅스 운영체제하의

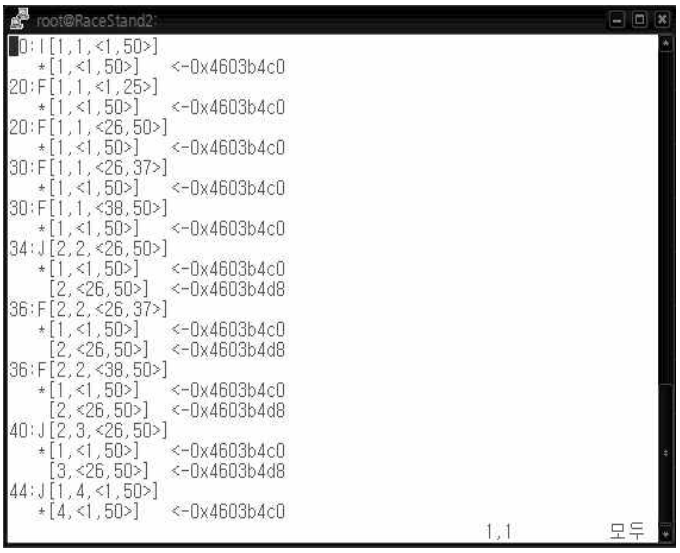

그림 4 개선된 NR 레이블링의 정확성 실험결과

64bit Intel Xeon Quad-Core 2 CPU, 8GB 메모리 시스템에 OpenMP 3.0[2]을 지원하는 gcc 4.3.3 Compiler를 설치하였으며 두 레이블링을 위해 구현한 라이브러리를 각각 설치하였다. 각 기법은 동일한 시스템 환경 하에서 실험하며 효율성 실험은 각 합성 프로그램을 10번씩 실행하여 측정된 시간과 메모리 소모를 대상으로 평균하여 비교한다. 이때 시간 측정은 리눅스의 'time' 프로세스를 백그라운드로 사용하고 system 시간과 user 시간의 합을 대상으로 한다.

4.2 결과 분석

설계된 합성 프로그램을 이용하여 개선된 NR 레이블링의 정확성을 분석하고, 기존의 기법과 개선된 기법의 효율성을 비교한다.

개선된 기법의 정확성 실험에서 합성 프로그램이 요구하는 레이블을 정확하게 생성하였으며, 참조를 통한 OH의 구성은 오류 없이 이루어졌다. 그림 4는 그림 3에서 보인 프로그램의 수행 중에 생성된 스레드 레이블의 로그를 보인다. 그림 3의 T_5 스레드는 해당 프로그램의 34번째 라인에 의해 생성된 합류된 스레드임을 "J" 기호로 표시하고, 지시자에 의해 정확한 OH를 유지한다. 이후, T_6, T_7 스레드는 프로그램에서 36번째 라인에 의해 분기된 스레드임을 "F" 기호로 표시하고, T_5 스레드에 의해 생성된 OH의 정보를 오류 없이 유지하고 있다. 그리고, T_8 스레드는 프로그램에서 40번째 라인에서 합류된 스레드로 합류 카운터가 증가하고 T_5 스레드의 정보에 자신의 합류 카운터만 증가 시켜 유지하고 있다.

그림 5는 두 기법의 효율성을 비교하기 위해 작성된 합성 프로그램 중 내포 병렬성의 차수가 3($N=3$)이고 다중 방향성의 차수가 2인 프로그램을 이용하여 최대 병렬성을 10^4에서부터 10^6까지 증가시키면서 실험한 결과를 보인다. 실험을 통해 개선된 기법은 기존의 기법보다 평균 4.5배 빠름을 알수 있다. 그림에서 첫 번째 점선은

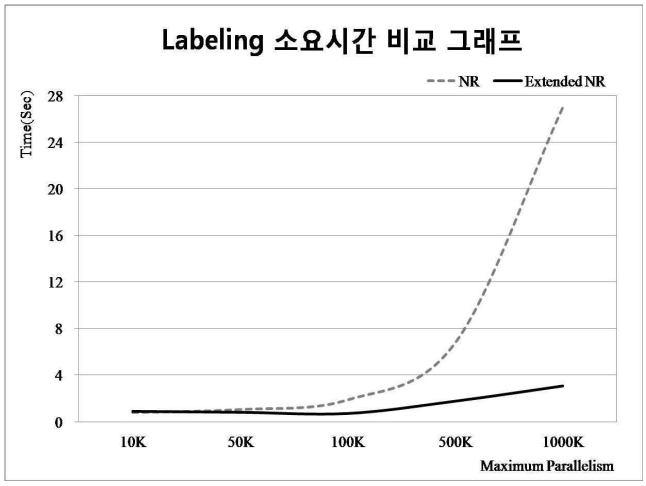

그림 5 최대병렬성 변화에 따른 레이블링 소요시간

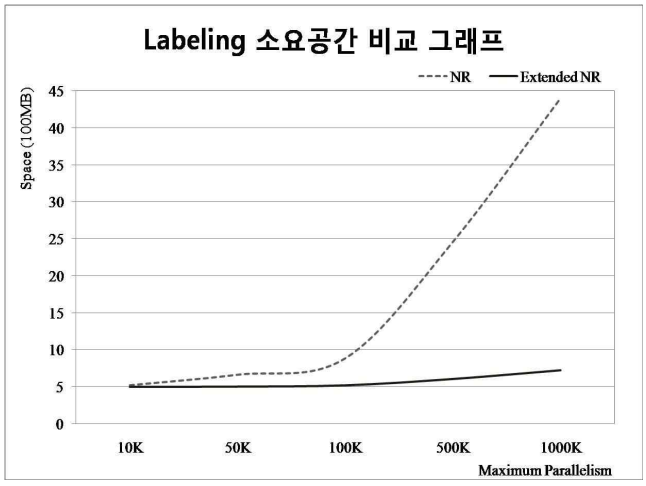

그림 6 레이블링을 위한 소요 메모리 비교

NR 레이블링의 결과이고, 두 번째 실선은 개선된 기법의 결과를 나타낸다.

결과에서 두 기법은 최대 병렬성이 작을수록 수행 시간의 차이가 나지 않지만 최대 병렬성이 100,000일 때부터 급격한 차이를 보인다. 이는 기존의 기법이 스레드의 생성을 위해 내포 깊이 N에 의존하는 $O(N)$의 복잡도를 갖지만, 유지하기 위해서는 최대 병렬성 T에 비례해야 하므로 $O(TN)$의 복잡도를 갖기 때문이다. 그러나 개선된 NR 레이블링은 스레드의 생성과 유지시 내포 깊이에 의존적이지 않기 때문에 생성 시 $O(1)$, 유지 시 $O(T)$의 복잡도를 보인다.

그림 6은 그림 5의 실험을 위해 소요된 메모리의 비교 결과를 보인다. 그림에서 첫 번째 점선은 NR 레이블링의 측정 결과이며, 아래의 실선은 개선된 NR 레이블링의 측정 결과이다. 실험에서 개선된 기법은 기존의 기법보다 평균 3배 감소된 기억 공간을 요구한다.

기존의 기법은 최대 병렬성의 증가에 따라 요구되는 기억공간이 급격하게 증가하며, 개선된 기법은 최대 병렬성이 증가하더라도 요구되는 기억공간이의 변화가 작다. 이는 기존의 기법이 레이블의 생성시 요구되는

$O(N)$의 공간적 복잡도와 레이블의 유지 시 최대 병렬성의 증가에 따라 내포 깊이에 의존된 $O(TN)$의 공간적 복잡도를 갖는 반면 개선된 기법은 내포병렬성에 의존적이지 않고 레이블의 생성 시 $O(1)$, 유지 시 $O(T)$의 공간적 복잡도를 요구하기 때문이다. 그림 5와 그림 6의 효율성 비교 실험통해 개선된 NR 레이블링 기법은 스레드의 생성과 유지 시에 내포 깊이에 의존적이지 않고, 상수적인 복잡도를 갖는 기법임을 알 수 있다.

5. 결론

OpenMP와 같은 병렬 프로그램에서 수행 중에 스레드의 병행성 정보를 생성하고 유지하는 기존의 기법은 프로그램에 존재하는 내포 루프의 깊이에 의존하는 시간 및 공간적 복잡도를 가진다. 본 논문에서는 기존의 기법을 개선하기 위해 레이블의 생성과 유지 시에 이전 스레드의 부모 정보를 참조하는 방법으로 내포 깊이에 의존적이지 않고, 상수적인 시간 및 공간적 복잡도를 갖는 기법으로 개선하였다.

개선된 기법은 레이블의 생성과 유지 시 기존의 기법보다 평균 4.5배 빠르고, 레이블링을 위해 평균 3배 감소된 기억공간을 요구하며, 내포 병렬성에 의존적이지 않는 효율적인 레이블링임을 보였다. 향후과제로는 개선된 기법이 스레드들간의 병행관계를 이진탐색을 통해 비교하는 것을 개선하여 비교시 상수적 복잡도를 갖는 기법으로 개발하는 것이다.

참 고 문 헌

[1] Dagum, L., and R. Menon, "OpenMP: An Industry-Standard API for Shared Memory Programming," *Computational Science and Engineering*, 5(1): 46-55, IEEE, January-March 1998.

[2] OpenMP Architecture Review Board, "OpenMP Application Program Interface," *www.open-mp.org*, version3.0, May 2008

[3] Rinard. M., "Analysis of Multithreaded Programs," *Int'l Static Analysis Symposium (SAS)*, Lecture Notes in Computer Science, 2126: 1-19, Springer-Verlag, July 2001.

[4] Netzer, R. H. B., and B. P. Miller, "What Are Race Conditions? Some Issues and Formaliza-tions," *Letters on Prog. Lang. and Systems*, 1(1): 74-88, ACM, March 1992.

[5] Bücker, H. M., A. Rasch, and A. Wolf, "A Class of OpenMP Applications Involving Nested Parallelism," *Pro. of the 19th ACM Symposium on Applied Computing (SAC)*, 1: 220-224, Nicosia, Cyprus, New York, ACM, March 2004.

[6] Farchi, E., Y. Nir, and S. Ur, "Concurrent Bug Patterns and How to Test Them", *Proceedings of the 17th international Symposium on Parallel and Distributed Processing (IPDPS)*, pp. 286.2, IEEE, April 2003.

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

[8] Fidge, C., "Logical Time in Distributed Computing Systems," *Computer*, 24(8): 28-33, August 1991.

[9] A. Dinning and E. Schonberg, "An Empirical Comparison of Monitoring Algorithms for Access Anomaly Detection", *In Proceedings of the second ACM SIGPLAN symposium on Principles & practice of parallel programming (PPoPP'90)*, pp.1-10, February 1990.

[10] Mellor-Crummey, J. M., "On-the-fly Detection of Data Races for Programs with Nested Fork-Join Parallelism," *Supercomputing*, pp. 24-33, ACM/IEEE, Nov. 1991.

[11] Jun, Y. and K. Koh, "On-the-fly Detection of Access Anomalies in Nested Parallel Loops," *3rd ACM/ONR Workshop on Parallel and Distributed Debugging*, pp. 107-117, ACM, May 1993.

[12] Audeneert, K., "Clock Trees: Logical Clocks for Programs with Nested Parallelism," *Transation on Software Engineering*, 23(10): 646-658, IEEE, October 1997.

[13] Baldoni, R. and M. Raynal, "Fundamentals of Distributed Computing: A Practical Tour of Vector Clock Systems," *IEEE Distributed Systems Online*, 3(2), February 2002.

[14] Park, S., M. Park, and Y. Jun, "A Comparison of Scalable Labeling Schemes for Detecting Races in OpenMP Programs," *Proc. of the Int'l Workshop on OpenMP Applications and Tools (Wompat)*, Lecture Notes in Computer Science, 2104: 68-80, Springer-Verlag, July 2001.

[15] Dinning, A., and E. Schonberg, "Detecting Access Anomalies in Programs with Critical Sections," *2nd Wrokshop on Parallel and Distributed Debugging*, pp. 85-96, ACM, May 1991.