# OpenMP 디렉티브 프로그램을 위한 자료경합 검증도구

## (A Verification Tool of Data Races in Programs with OpenMP Directives)

김 영 주 † 전 용 기 ††
(Young-Joo Kim) (Yong-Kee Jun)

**요 약** OpenMP 디렉티브를 가진 프로그램에서 경합은 프로그램의 의도하지 않은 비결정적 수행결과를 초래하기 때문에 디버깅을 위해서 반드시 탐지되어야 한다. 하지만 이를 위한 기존의 경합탐지 도구인 Intel사의 Thread Checker는 경합의 존재를 검증하지 못하고 경합을 탐지하는 비용이 크므로 비실용적이다. 본 논문에서는 OpenMP 프로그램의 특성 및 사용자 요구사항의 분석결과를 이용하여 최적의 기능과 성능으로 경합을 검증하는 웹 기반 도구를 제시한다. 그리고 합성 프로그램을 이용하여 실험한 결과로서 Thread Checker는 경합의 존재를 검증하지 못하고 경합탐지 시에 소요되는 시간의 증가비율은 총 접근사건수 $n$에 대해서 $O(n^2)$이지만 제안된 도구는 경합의 존재를 검증하고 소요되는 시간의 증가비율은 $O(n)$이므로 기능 및 성능적인 측면에서 실용적인 도구이다.

**키워드** : OpenMP 프로그램, 경합검증, Intel Thread Checker

***Abstract*** Races in programs with OpenMP directives must be detected for debugging, because they may cause unexpected result by non-deterministic executions. But, Thread Checker of Intel corporation, a well-known existing tool for detecting the races, is not practical because this tool does not verify the existence of races and is known that the cost for race detection is too big. This paper presents a web-based tool which verify the existence of races with an optimal functionality and performance using the results from the property analysis of OpenMP program as well as the user requirements. Our tool is proved to be practical in the aspect of functionality and performance by experiments using synthetic programs, because the suggested tool can verify the existence of race and shows $O(n)$ as the ratio of time consumption while Thread Checker can not verify the existence of race and shows $O(n^2)$ as the ratio, where $n$ is the number of total accesses.

**Key words** : OpenMP Programs, Race Verification, Intel Thread Checker

## 1. 서 론

병렬프로그래밍 API를 제공하는 OpenMP[1,2]는 1997년에 산업표준을 위하여 1997년 11월에 Fortran을 위한 OpenMP 1.0이 명세되었고, 현재의 Fortran와 C를 통합한 OpenMP 2.5는 2005년 5월에 명세되었다. OpenMP 프로그램과 같은 병렬 프로그램은 순차적으로 수행하는 프로그램보다 디버깅하는 것이 어렵다. 왜냐하면, 병렬 프로그램의 의도하지 않은 비결정적 수행으로 인해 경합[3]이라는 오류를 초래하기 때문이다. 경합은 적절한 동기화 없이 적어도 하나의 쓰기 접근사건으로 공유변수에 접근하는 병렬 스레드들의 집합에서 명령어들의 쌍이다. OpenMP 디렉티브(directives)를 가진 프로그램에서도 경합은 프로그램의 의도하지 않은 비결정적 수행결과를 초래하기 때문에 디버깅을 위해서 반드시 탐지되어야 한다. 이러한 경합을 탐지하기 위해서 사용되는 전통적인 디버깅 기법인 브레이크 포인트는 비효율적이다. 왜냐하면, 브레이크 포인트는 잘못된 수행으로 인해 실행 시간이나 순서가 변경될 수 있기 때문이다.

OpenMP 프로그램에서 발생하는 경합을 수행중에 탐지하는 기존 도구로는 Intel사의 Thread Checker[4-7]가 있다. Thread Checker는 프로그램의 수행과 분석을

† 정 회 원 : 한국정보통신대학교 IT공학부
yjkim@icu.ac.kr
†† 종신회원 : 경상대학교 정보과학과 교수
jun@gnu.ac.kr
논문접수 : 2007년 6월 13일
심사완료 : 2007년 8월 15일

동시에 진행하면서 경합을 탐지하는 projection 기법[7]을 사용한다. Thread Checker가 경합을 탐지하는 동작 순서는 다음과 같다. OpenMP 디렉티브로 작성된 프로그램이 컴파일 될 때 OpenMP 디렉티브와 공유변수에 대한 접근사건들을 전용 데이타베이스에 기록한다. 그리고 기록된 자료를 토대로 프로그램을 순차적으로 수행하면서 공유변수에 대한 접근사건들의 데이타 종속성을 검사하여 경합을 탐지한다. 데이타 종속성 검사는 경합탐지를 위해서 사용하는 것으로서 순차스레드에서 발생하는 접근사건들에 대한 병행성 여부를 검사하여 경합을 탐지한다. 그러나 이 도구는 내포된 스레드를 부모스레드와 순서적인 스레드로 간주하므로 경합의 존재를 검증하지 못할 뿐만 아니라 데이타 종속성을 검사하여 경합을 탐지하는 기법[7]을 사용하므로 소요되는 시간과 메모리의 비용이 크다고 알려져 있다.

본 연구에서는 OpenMP 프로그램의 특성 및 사용자 요구사항을 분석된 결과를 이용하여 최적의 기능과 성능으로 가진 선택된 엔진으로 경합의 존재를 검증하는 웹기반의 도구를 제시한다. 대상 프로그램은 C 언어를 기반으로 SISE(single input single execution)[8] 속성을 만족하는 OpenMP 프로그램이다. OpenMP 프로그램의 내포병렬성을 위해서는 OpenMP 디렉티브중에서 "#pragma omp parallel for"를 고려하고, 동기화를 위해서는 "#pragma omp critical"을 고려하여 분석된다. 그리고 사용자 요구사항은 경합탐지의 효율성을 시간과 공간적인 측면을 고려한다. 대상 프로그램의 특성과 사용자 요구사항을 기반으로 최적의 성능으로 경합존재를 검증할 수 있는 엔진을 구성한다. 여기서 경합검증을 위한 엔진은 레이블링 기법[8,9]과 프로토콜 기법[8]으로 구성되어 있다. 이러한 엔진은 사용자 컴퓨터에서 수행되지 않고 서버 컴퓨터에서 수행되며 사용자는 웹으로 탐지된 경합을 볼 수 있다.

합성 프로그램(synthetic programs)을 이용하여 제안된 도구와 기존 도구인 Thread Checker를 기능과 성능적인 측면에서 비교분석 하였다. 기능적인 측면에서, Thread Checker는 OpenMP 프로그램에서 경합의 존재를 검증하지 못하지만 제안된 도구는 경합검증이 가능하였다. 성능적인 측면에서, SPMD(single program multiple data)와 MPMD(multiple program multiple data) 병렬 프로그래밍 모델[10]을 기반으로 하는 합성 프로그램에서 경합을 탐지하는데 소요되는 시간의 증가비율은 싱글 프로세서와 멀티 프로세서 컴퓨터에서 측정하였다. 싱글 프로세서 컴퓨터에서 측정한 결과, MPMD 기반의 합성 프로그램에서 Thread Checker가 $O(n^2)$이고 제안된 도구는 $O(n)$이였다. 그리고 멀티 프로세서 컴퓨터에서 측정한 결과, 성능이 증가하여 Thread Checker는 $O(n)$에 가까웠고 제안된 도구는 소요시간 증가비율이 거의 없었다. 여기서 $n$은 총 접근사건수를 의미한다. 제안된 도구와 기존 도구의 기능을 실험하기 위한 환경으로는 Windows 계열의 운영체제가 탑재되어 있는 펜티엄-4에 512MB의 메모리를 가진 사용자 컴퓨터에 Intel C++ Compiler[11]와 VTune Performance Analyzer[12]가 필요하다. 그리고 제안된 도구와 기존 도구의 성능을 실험하기 위한 환경으로는 싱글 프로세서를 탑재한 컴퓨터와 멀티 프로세서를 탑재한 컴퓨터를 사용한다. 전자의 컴퓨터는 커널 버전이 2.2.x인 Redhat 리눅스가 설치되어 있는 펜티엄-III 프로세서에 256MB의 메모리를 가지고 있고, 후자의 컴퓨터는 커널 버전이 2.6인 Debian 리눅스가 설치되어 있는 64비트 Intel Xeon 듀얼 프로세서에 1GB의 메모리를 가지고 있다. 이들 컴퓨터는 Apache, Tomcat, Omni OpenMP Compiler[13], 그리고 Intel C/C++ Compiler가 필요하다.

논문의 구성은 살펴보면, 2절에서는 OpenMP 프로그램에서 발생하는 경합과 기존의 도구인 Intel Thread Checker에 대한 문제점을 지적하고, 이러한 문제점을 해결할 수 있는 실용적인 기법을 소개한다. 3절에서는 실용적인 경합탐지 기법을 이용하여 설계한 본 도구의 전체적인 구조와 구현된 도구의 내부구조를 사용자 인터페이스를 이용하여 설명한다. 4절에서는 합성 프로그램을 이용하여 Intel Thread Checker에 비해서 제안된 도구가 기능과 성능적인 측면에서 실용적임을 보인다. 마지막 절에서는 결론 및 향후과제를 제시한다.

## 2. 연구배경

이 절에서는 OpenMP 프로그램 모델에서 발생하는 경합에 대해서 설명하고, 이러한 경합을 탐지하는 기존 도구인 Intel사의 Thread Checker에 대한 문제점을 소개한다. OpenMP 프로그램에서 발생하는 경합을 예제 프로그램과 이를 표현하는 방향성이 있는 비순환 그래프인 POEG(partial order execution graph)으로 설명한다. 그리고 OpenMP 프로그램에서 경합을 탐지하기 위해서 설치되어야 하는 프로그램들과 그 프로그램들 중에서 경합탐지 도구인 Thread Checker에 대한 특징과 동작원리에 대해서 설명한다.

### 2.1 OpenMP 프로그램

산업 표준화를 위한 병렬프로그램 모델인 OpenMP [1,2]는 공유메모리를 사용하고 표준 C/C++와 Fortran 77/90을 확장하는 디렉티브와 라이브러리들의 집합이다. OpenMP는 디렉티브를 이용하여 순차적 프로그램을 쉽게 병렬화 할 수 있고, coarse-grain parallelism을 구현할 수 있는 orphan 디렉티브 개념을 제공하여 병렬프

로그램의 확장성을 높여준다. OpenMP에서 제공하는 디렉티브는 병렬화 디렉티브와 동기화 디렉티브가 있다. 병렬화 디렉티브는 새로운 스레드의 생성과 합류를 하는 "#pragma omp parallel for"와 "#pragma omp parallel sections" 등이 있고, 동기화 디렉티브는 스레드간의 수행 순서를 제어하는 "#pragma omp atomic", "#pragma omp barrier", "#pragma omp critical" 등이 있다. 또한 OpenMP는 런타임 수행환경을 제어할 수 있는 라이브러리와 환경변수들을 제공한다.

예를 들어, 그림 1 11번 줄에서 "#pragma omp parallel"에 의해서 스레드가 생성된다. 여기서는 2개의 스레드가 생성된다고 가정한다. 생성된 스레드들은 12번 줄의 "#pragma omp for private(i,y,z)"에 의해서 13번 줄의 for 문장부터 19번 줄의 "}"까지의 반복문 몸체에서 명시된 작업을 할당받는다. 반복문 내에서 첨자 변수인 *i*, *y*, *z*는 각 스레드에서 사용하는 private 변수이고, 정수형 변수인 *x*는 두 스레드에서 공유하는 shared 변수이다. 이 프로그램은 수행 중에 경합이 발생할 수 있으므로 그 이유를 그림 1을 통해서 설명한다. *x*, *y*, 그리고 *z* 변수의 초기값은 0으로 한다. 생성된 두 개의 스레드 중에서 첫 번째 스레드가 수행하는 15번 문장과 두 번째 스레드가 수행하는 18번 문장은 스레드가 병행성 관계에 있지만 "#pragrma omp critical(L1)"로 임계구역이 설정되므로 *x*에 대한 경합이 존재하지 않는다. 그렇지만 첫 번째 스레드가 수행하는 14번 문장에서 공유변수 *x*에 대해 읽기사건을 수행하고, 두 번째 스레드는 18번 문장에서 공유변수 *x*에 대해 쓰기접근사건을 수행한다. 두 스레드는 병행적 수행으로 인해 수행순서가 변경될 수 있다. 따라서 20번 문장에서 *x*의 변수 값은 100, 104, 204, 그리고 206 중에서 임의의 값 하나가 출력된다. 왜냐하면 두 스레드 사이에 적절한 동기화 없이 병렬 스레드들이 적어도 하나의 쓰기 접근사건들로 공유변수인 *x*에 접근하기 때문이다. 따라서 공유변수 *x*에 발생한 경합으로 인해서 비결정적 수행결과를 초래한다.

```
10: ...
11: #pragma omp parallel
12: #pragma omp for private(i,y,z)
13:       for (i=1 ; i<3 ; i++) {
14:         if(i==1) { y = x + 2;
15: #pragma omp critical(L1) { z = x + 2; x = y + z; }
16:         }
17:         else {
18: #pragma omp critical(L1) { x = 100; y = x + 1; }
19:         }}
20: printf("x value = %d \n", x);
21: ...
```

그림 1 OpenMP 프로그램

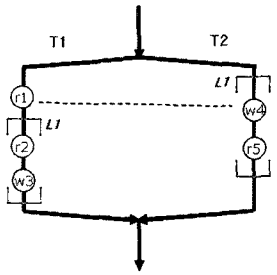



그림 2 공유변수 *x*에 대한 경합탐지

프로그램 수행을 표현하는 그래프인 POEG은 방향성이 있는 비순환 그래프로서 그림 2는 그림 1의 프로그램 수행을 나타낸 POEG이다. POEG에서 정점은 병렬 스레드의 생성과 합류지점을 나타내고 임의의 정점으로부터 시작되는 간선은 수행되는 스레드 블록을 의미한다. 그리고 각 간선에 *r*과 *w*는 임의의 공유변수에 접근하는 읽기 접근사건과 쓰기 접근사건을 나타내며, 접근사건들에 붙어있는 숫자는 임의의 발생순서를 의미한다. 그리고 ⊓기호와 ⊔기호로 지정된 영역은 록 변수인 *L1*의 임계구역을 의미한다. 그림 2에서 T1 스레드에 *r1*과 *T2* 스레드에 *w4*는 병행관계에 있고 *r1*이 임계구역으로 보호되어 있지 않기 때문에 *w4*와 경합이라는 것을 알 수 있다.

이런 경합을 탐지하기 위해서 사용되는 전통적인 디버깅 기법인 브레이크 포인트는 수행 시간을 간섭하여 잘못된 행위들이 사라질 수도 있기 때문에 비효율적이다. 그러므로 대부분의 병렬 프로그램의 디버거들은 경합을 효과적으로 탐지하는 것이 아니라 프로그래머의 능력에 의존한다. 이러한 경합을 효과적으로 탐지하기 위한 방법으로 수행중 경합탐지 기법이 있다. 이 기법은 프로그램 수행에서 나타날 수 있는 실제 경합들만을 탐지하고, 프로그램의 수행과 분석을 동시에 진행하므로 불필요한 수행 정보들은 삭제를 할 수 있기 때문에 기억공간에 있어서 효율성을 가진다. 현재 OpenMP 프로그램에서 발생하는 경합을 프로그램의 수행 중에 탐지하는 도구로는 Intel사의 Thread Checker가 있다.

### 2.2 Thread Checker

Thread Checker를 이용하기 위해서는 Intel사의 Threading Tools[4-6,11,12]가 필요하다. Threading Tools은 Intel C/C++ Compiler[11], VTune Performance Analyzer[12], 그리고 Intel Thread Checker[4-7]로 구성되어 있다. Intel C/C++ Compiler는 C/C++로 작성된 프로그램들을 컴파일할 수 있고 OpenMP 2.0을 지원하며 순차적 프로그램을 자동적으로 병렬화 하므로 멀티스레드 기반의 병렬프로그램을 개발하거나 최적화할 수 있다. VTune Performance Analyzer는 Intel 프로세서

상에서 최적의 성능으로 프로그램 튜닝을 할 수 있도록 도와주고, 수행중인 프로그램의 탐지된 오류들이나 성능 정보를 수집하거나 분석하여 텍스트 및 시각적으로 요약된 목록들을 사용자에게 보여준다. 그리고 Intel Thread Checker는 Threading Tools의 핵심 구성요소로써 병렬 프로그램에서 발생할 수 있는 오류를 원시프로그램에서 탐지할 수 있다. 여기서 탐지되는 오류들은 교착상태, 경합, 논리적 오류 등이다.

OpenMP 프로그램에서 발생하는 경합을 수행 중에 탐지하는 대표적 기법인 Thread Checker의 projection 기법[7]은 병렬 프로그램의 순차적 수행정보를 이용하여 데이타 의존성 여부를 수행 중에 검사한 후에 경합을 탐지한다. 이 기법은 OpenMP 디렉티브들로 구성된 relaxed sequential OpenMP 프로그램에만 적용될 수 있다. Thread Checker가 경합을 탐지하는 동작순서는 다음과 같다. OpenMP 2.0 명세서에 정의되어 있는 parallel region constructs, work-sharing constructs, synchronization constructs 등에 해당하는 OpenMP 디렉티브로 작성된 프로그램이 Intel C/C++ Compiler에 의해 컴파일 될 때 OpenMP 디렉티브와 공유변수에 대한 접근사건들을 전용 데이타베이스에 기록한다. 그리고 프로그램의 수행 시에 OpenMP 디렉티브들을 무시하고 순차적으로 프로그램을 수행하면서 OpenMP 디렉티브들을 만나면 전용 데이타베이스에 기록된 정보를 참조하여 공유변수에 대한 접근사건들의 데이타 종속성[14]을 검사한 후에 input data dependency[14]를 제외한 anti, flow, 그리고 output data dependency[14]를 경합으로 보고한다.

이러한 특징을 가지고 경합을 탐지하는 Thread Checker는 기능과 성능적인 측면에서 문제점을 가지고 있다. 기능적인 측면에서, Thread Checker는 동기화가 없고 내포 병렬성이 존재하지 않는 프로그램에 대해서만 경합존재에 대한 검증이 가능하고 이 이외의 프로그램 모델에서는 경합검증이 불가능하다. 예를 들어, 그림 2와 같이 동기화가 있고 내포병렬성이 존재하지 않는 프로그램 수행의 경우에 *r1*과 *w4*는 경합이지만 Thread Checker는 경합으로 보고하지 못한다. 왜냐하면, 순차적 수행으로 T1 스레드에서 *r2* 사건이 발생하면 *r1* 사건과 순서화 관계에 있다고 *r2* 사건에 의해 *r1*에 대한 정보를 삭제하기 때문에 T2 스레드에서 *w4* 사건이 발생하면 병행성 관계에 있는 접근사건은 T1 스레드에 있는 *r2* 사건과 *w3* 사건이지만 동일한 록 변수 *L1*으로 임계구역이 설정되어 있어서 경합으로 보고하지 못한다. 따라서 경합이 존재하고 있지만 경합을 보고하지 못하므로 Thread Checker는 경합의 존재를 검증하지 못한다.

두 번째 성능적인 측면에서, projection 기법[7]은 경합탐지 시에 소요되는 시간과 공간적 비용이 크다고 알려져 있다. 예를 들어, 동기화가 없고 내포 병렬성이 존재하지 않는 프로그램에서 최대병렬성이 32이고 하나의 공유변수에 대해서 총접근사건수를 1000개와 4000개를 가진 프로그램 A와 프로그램 B를 작성하여 Thread Checker를 이용하여 경합 시에 소요되는 시간과 공간을 측정하였다. 소요시간은 프로그램 A에서는 50초, 프로그램 B에서는 671초가 걸리고 증가비율은 13.4배나 된다. 소요메모리는 프로그램 A에서는 20MB, 프로그램 B에서는 33MB가 소요되고 증가비율은 1.7배이다. 소요파일은 프로그램 A에서는 0.7MB, 프로그램 B에서는 1.4MB가 소요되고 증가비율은 2배가 된다.

실험환경은 Windosws 계열의 운영체제가 탑재되어 있는 펜티엄-4에 512MB의 메모리를 가진 컴퓨터에서 경합탐지 시에 소요되는 시간과 메모리를 측정하였다. 여기서 소요메모리의 측정결과는 Thread Checker 자체에서 소요된 메모리라고 보기는 어렵다. 그러나 경합탐지 시에 소요된 시간은 비효율적인 성능을 보인다는 것을 알 수 있다.

### 2.3 실용적 경합탐지 기법

본 논문에서 경합검증을 위해서 기존에 연구된 레이블링 기법[8,9,15-18]과 경합탐지 프로토콜 기법[8,17,19,20]을 사용한다. 이들 기법들 중에서 제안된 도구에 적용된 레이블 기법은 EH[9]과 LC[8]이고 프로토콜 기법은 LC[8]이다.

레이블링 기법은 프로그램의 수행 중에 생성된 스레드에 논리적 병행성 정보를 생성하여 스레드마다 고유한 식별자를 부여하므로 생성되는 모든 다른 스레드들과 구별할 수 있는 유일성을 가진다. 스레드마다 생성되는 레이블 정보는 프로그램에서 발생하는 경합을 탐지하기 위해서 접근사건들 간의 논리적 병행성 관계를 검사할 때 사용된다. 이러한 레이블을 생성하는 기법은 확장성 여부에 따라서 두 가지로 분류된다. 첫 번째, 공유자료구조를 사용하여 레이블을 생성하는 기법[15]은 공유자료구조인 "dag"를 이용하면서 부모스레드의 고유한 식별자를 자식 스레드가 재활용하므로 스레드 레이블의 생성 시에 심각한 병목현상이 발생한다. 이러한 병목현상은 프로그램에서 발생하는 스레드에 대한 최대 병렬성의 증가에 의존적이다. 두 번째, 개별자료구조를 사용하여 레이블을 생성하는 기법[8,9,16-18]은 스레드마다 존재하는 개별 자료구조(private data structure)를 사용하여 논리적 병행성 정보를 생성하므로 최대 병렬성이 증가하더라도 병목현상이 발생하지 않는 확장성을 제공한다. 이 기법은 부모 스레드의 고유 식별자인 레이블 정보를 개별 자료구조를 사용하여 새로운 자식 스레드의 생성 시에 부모 스레드의 레이블 정보를 참조하여

자식 스레드의 새로운 레이블 정보를 생성한다.

경합탐지 프로토콜 기법은 병렬 프로그램에서 발생하는 경합을 탐지하기 위해서 스레드에서 발생하는 접근사건의 발생 시마다 공유자료 구조인 접근역사(access history)에 저장되어 있는 이전의 접근사건들과 병행성 관계를 검사하여 경합을 탐지한다. 접근역사는 프로그램 수행 중에 발생한 접근사건들 중에서 병행한 접근사건들로만 구성되어 있다. 이러한 프로토콜은 프로그램의 수행 중에 발생하는 경합들을 모두 탐지하지 못하지만 감시하는 공유변수에 대해서 적어도 하나의 경합은 탐지할 수 있는 경합검증 프로토콜[8,17]과 발생되는 경합들 중에서 가장 먼저 발생하는 경합을 탐지하는 최초경합 탐지 프로토콜[19,20]이 있다. 확장성 레이블링 기법과 경합검증 프로토콜 기법은 본 논문에서 제안한 경합탐지 도구인 RaceStand의 핵심 엔진으로 이용된다.

## 3. 경합검증 도구

이 절에서는 경합검증을 위한 기법을 이용하여 경합을 탐지하는 도구의 전체적인 구조와 서버-클라이언트 모델로 구현된 사용자 인터페이스를 이용하여 원격으로 경합을 탐지하는 방법에 대해서 설명한다. 제안된 도구에서 설계된 모듈과 그 모듈의 입력과 출력에 대해서 설명한다. 그리고 구현된 Web Interface로 경합을 탐지하는 각 단계에 대해서 자세히 설명한다.

### 3.1 RaceStand의 설계

본 논문에서는 OpenMP 프로그램의 특성 및 사용자 요구사항의 분석된 결과를 이용하여 최적의 기능과 성능을 가진 선택된 엔진으로 경합을 검증하는 웹 기반 도구를 제시한다. 그림 3은 RaceStand의 전체적인 구조를 보인 것이다. 이 그림에서 사용자 인터페이스에 해당하는 Web Interface는 클라이언트로서 최적의 성능으로

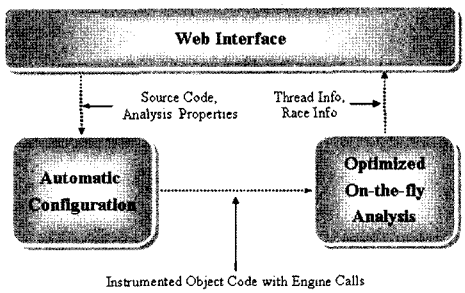



그림 3 제안된 도구인 RaceStand의 전체적 구조

경합을 탐지하기 위한 환경을 제공한다. 그리고 사용자 요구사항을 결정하여 변형된 프로그램을 생성하는 Automatic Configuration과 변형된 프로그램의 수행 중에 경합을 탐지하는 Optimized On-the-fly Analysis는 서버에 해당된다.

첫 번째 모듈인 Automatic Configuration은 Program Scanner, Engine Selector, 그리고 Source Instrumentor로 구성되어 있다. 그림 4는 Automatic Configuration의 세부모듈을 보인 것이다. Program Scanner는 Source Code와 Analysis Properties를 입력받는다. 여기서 Source Code는 OpenMP 프로그램이고 Analysis Properties는 사용자 요구사항이다. Program Scanner는 OpenMP 프로그램에 병렬화/동기화 디렉티브를 사용한 여부를 검사하면서 프로그램의 특성을 분석한다. 분석된 결과는 경합탐지를 위한 Pseudo 코드를 삽입한 변형된 OpenMP 프로그램을 생성한다. 그림 5는 그림 1에서 작성된 OpenMP 프로그램 모델을 Program Scanner에 의해서 Pseudo 코드가 삽입된 프로그램을 보인 것이며 그림 4에서 PS-Code에 해당된다. 감시코드가 삽입되는 지점은 //@pse LABEL과 //@pse CRITICAL 등을 위

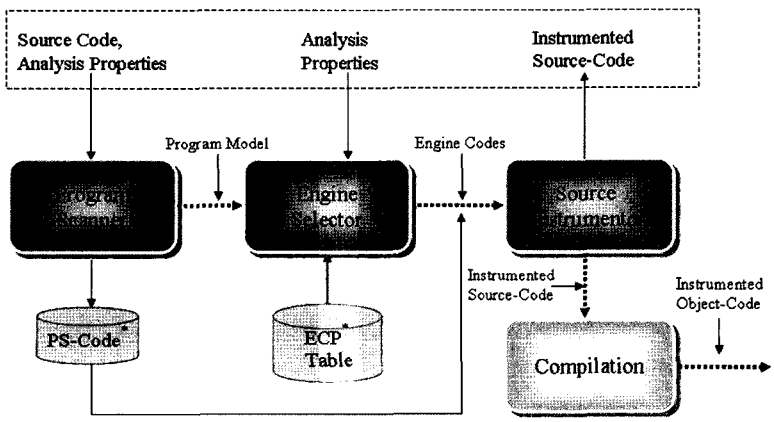



그림 4 Automatic Configuration의 세부모듈

```
10: ...
11: #pragma omp parallel
12: #pragma omp for private(i,y,z)
13:       for (i=1 ; i<3 ; i++) {
14: //@pse LABEL(i, 1, 3, 1)
15:         if(i==1) { y = x + 2;
16: //@pse READ(x)
17: #pragma omp critical(L1) {
18: //@pse CRITICAL(L1)
19:                    z = x + 2;
20: //@pse READ(x)
21:         ... }
22:         else {
23: #pragma omp critical(L1) {
24: //@pse CRITICAL(L1)
25:                    x = 100;
26: //@pse WRITE(temp)
27:         ...
28: //@pse CLOSE LABEL
29:       }
30: printf("x value = %d \n", x);
31: ...
```

그림 5 Pseudo OpenMP 프로그램

한 병렬화/동기화 디렉티브와 //@pse READ와 //@pse WRITE 등을 위한 공유변수의 접근사건이 있다. 그리고 Engine Selector는 분석된 프로그램의 정보와 사용자 요구사항에 따라 최적의 기능과 성능으로 경합을 탐지할 수 있는 경합탐지 엔진 정보를 생성한다. 그림 4에서 ECP(engine code property) Table은 최적의 기능과 성능을 선택할 수 있도록 프로그램 모델별로 경합탐지 엔진들이 분류되어 있다. 마지막으로 Source Instrumentor는 선택된 경합탐지 엔진 정보를 이용하여 수행 중에 경합을 탐지하는 감시엔진인 레이블링과 프로토콜 엔진을 Pseudo 코드가 삽입되어 있는 지점에 대체시킨다. 대체된 OpenMP 프로그램은 수행 가능한 감시엔진이 삽입되어 있는 변형된 프로그램으로써 Omni OpenMP Compiler[12]나 Intel C/C++ Compiler[11]를 이용하여 감시엔진이 삽입되어 있는 변형된 목적 프로그램(Instrumented Object Code with Engine Calls)을 생성한다.

이렇게 생성된 변형된 목적 프로그램을 사용자가 수행시키면 삽입되어 있는 감시엔진인 Optimized On-the-fly Analysis가 수행되어 경합여부를 검사하여 보고한다. Optimized On-the-fly Analysis는 3절에서 설명한 레이블링 엔진과 경합탐지 프로토콜 엔진의 수행으로 구성되어 있다. 레이블링 기법은 동기화 명령어와 내포병렬성이 있는 프로그램 모델에서 스레드에 논리적 병행성 정보를 확장적으로 생성할 수 있는 EH(English-Hebrew) 레이블링[9]과 LC(LockCover) 레이블링[8]을 이용한다. 경합탐지 프로토콜 기법[4]은 각 접근

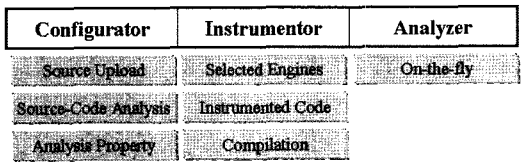



그림 6 사용자 웹 인터페이스

사건들에 대한 병행성 정보를 저장하는 형태를 네 가지의 접근역사로 분류하여 접근사건들을 저장 및 삭제한다. 이렇게 분류된 형태는 동기화 명령어를 포함하고 있는 읽기 접근사건과 쓰기 접근사건을 위한 접근역사들과 동기화 명령어를 포함하고 있지 않은 읽기 접근사건과 쓰기 접근사건을 위한 접근역사들로 구성되어 있다. LC 프로토콜은 이런 접근역사를 기반으로 프로그램 수행동안 공유변수에 대한 매 접근사건들을 감시하여 접근역사에 저장되어 있는 이전의 접근사건들과 병행성 관계를 검사한 후에 병행성 관계에 있는 것을 경합으로 보고 해당되는 접근역사에 유지한다. 또한 이 프로토콜은 접근역사의 기억공간을 줄이기 위해서 동기화 명령어를 포함하지 있지 않은 쓰기 접근사건에서 경합이 발생하면 경합보고 후에 네 가지 종류의 접근역사에 있는 모든 내용을 삭제하고 현재의 접근사건을 해당 접근역사에 유지한다.

서버에 있는 Automatic Configuration과 Optimized On-the-fly Analysis 모듈은 그림 6에 있는 웹 기반의 사용자 인터페이스에 의해서 동작된다. 사용자가 클라이언트에서 서버에 있는 모듈들을 동작시킬 수 있는 환경을 제공하기 위해서 Java Script와 JSP(Java Server Pages)를 이용한다. Configurator는 세 가지의 하위 메뉴들로 구성되어 있다. 이 하위메뉴들은 프로그램을 업로드 할 수 있으며 업로드 된 프로그램을 분석하고 사용자 요구사항을 설정할 수 있도록 되어있다. Configurator의 세 가지 메뉴들은 그림 4의 Automatic Configuration 모듈에서 Program Scanner에 해당된다. Instrumentor는 세 가지의 하위 메뉴들로 구성되어 있다. 이 하위메뉴들은 분석된 프로그램과 사용자 요구사항을 이용하여 경합탐지를 위한 엔진을 선택하고 그 엔진들을 원시프로그램에 삽입하여 변형된 프로그램을 생성한 후에 컴파일하여 변형된 목적 프로그램을 생성하게 한다. Instrumentor의 세 가지 메뉴 중에서 두 가지 메뉴는 그림 4의 Automatic Configuration 모듈에서 Engine Selector와 Source Instrumentor에 해당된다. compilation은 경합탐지를 위해서 변형된 OpenMP 프로그램을 컴파일하기 위해서 Omini OpenMP Compiler나 Intel C/C++ Compiler를 사용한다. Analyzer는 한 가지의 하위 메뉴로 구성되어 있다. 이 메뉴는 프로그램

의 수행 중에 발생하는 경합을 탐지하기 위한 것으로 그림 3의 Optimized On-the-fly Analysis 모듈이 동작되어 경합을 탐지한다. Optimized On-the-fly Analysis의 출력 값인 Thread Info와 Race Info는 수행 중에 탐지된 경합정보이다. 이 정보는 Text 방식으로 클라이언트 화면에 출력된다.

### 3.2 RaceStand의 구현

RaceStand는 실용적인 경합탐지 기법을 적용한 서버-클라이언트 구조를 가지고 있다. 서버에 구현되어 있는 "Automatic Configuration" 모듈은 웹과 연결되어 구동되어야 하므로 Java Script와 JSP로 구현하였다. 그리고 "Optimized On-the-fly Analysis" 모듈은 경합탐지를 위해서 변형된 목적 프로그램이 수행되면서 런타임 시에 이 모듈을 참조하므로 C 언어를 이용한 동적 라이브러리 형태로 구현되어 있다. 별도의 프로그램 설치 없이도 경합탐지를 가능하도록 하기 위해서 웹 페이지로 Web Interface 구축하였다.

Web Interface에서 경합탐지를 하는 과정을 살펴보자. 그림 7은 프로그래머가 경합을 탐지하고자 하는 OpenMP 프로그램을 업로드 하는 화면이고 하나 이상의 소스코드를 업로드할 수 있는 기능도 있다. 이렇게 업로드 된 프로그램들은 자동으로 분석된다. 그림 8은 업로드 된 프로그램에 대한 모델 분석이 된 결과를 보인 것이다. 그 결과는 내포 병렬성이 존재하며 임계구역을 가진 프로그램 모델이라는 것을 알 수 있다. 그림 9는 프로그램의 분석된 결과를 이용하여 사용자가 확장성과 효율성 측면을 고려하여 경합을 탐지할 수 있는 환경을 보인 것이다. 하지만 확장성과 효율성은 고려할 수 있는 환경만 제공된 상태이다. 그림 10은 그림 9의 사용자 요구사항을 참조하여 경합탐지를 위해서 사용될 엔진이 선택된 것을 보인 것이다. 선택된 결과는 레이블링 엔진을 위해서 NuRu86[9]와 DiSc91[8]이고 경합탐지를 위한 엔진을 위해서 DiSc91[8]이다. 이렇게 선택된 경합탐지 엔진을 이용하여 사용자는 그림 10에 있는 "Next(Instrument)" 버튼을 클릭하면 그림 11처럼 경합탐지를 위한 감시엔진 코드가 삽입되어 변형된 OpenMP 프로그램이 생성된다. 프로그램을 감시하는 엔진에 해당하는 라이브러리들을 살펴보자. EHAddLock_Label_EH_()는 임계구역을 감시하는 엔진, EHAddChild_Label_EH_()는 생성되는 스레드마다 레이블 정보를 생성하는 엔진, LCCheckCSReader_()와 LCCheckCSWriter_()는 임계구역 내에서 발생한 공유변수의 접근사건을 감시하는 엔진, 그리고 LCCheckReader_()와 LCCheckWriter_()는 임계구역 외부에서 발생한 공유변수의 접근사건을 감시하는 엔진이다. 이 변형된 프로그램을 실행 가능한 프로그램으로 변형하기 위해서 그림 11의 하단에 있는 "Next(Compile)" 버튼을 클릭하면 Omni OpenMP Compiler에 의해 컴파일 되면서 감시엔진이 동적으로 링크되어 변형된 목적 프로그램이 생성되고, 사용자가 그 프로그램을 실행시키게 되면 프로그램의 수행 중에 경합을 탐지하게 된다. 그림 12는 경합탐지 과정을 마친 후에 보고된 경합정보들을 보인 것이다. 이렇게 보도된 경합정보들은 스레드의 논리적 병행성 정보와 수행 중에 탐지된 경합들이다. RaceStand는 클라이언트 컴퓨터에 병렬프로그램을 분석하는 프로세스, 감시코드를 삽입하는 프로세스, 병렬 프로그램을 컴파일하는 컴파일러 등

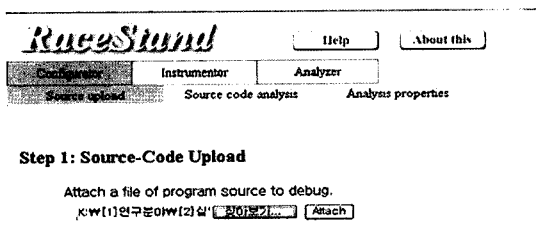



그림 7 OpenMP 프로그램의 업로드그림

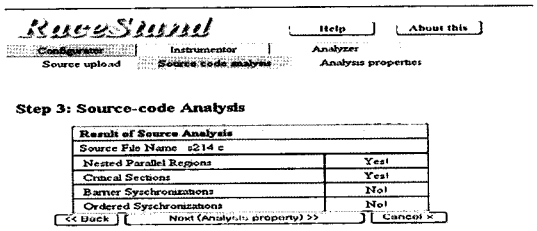



그림 8 프로그램 모델 분석그림

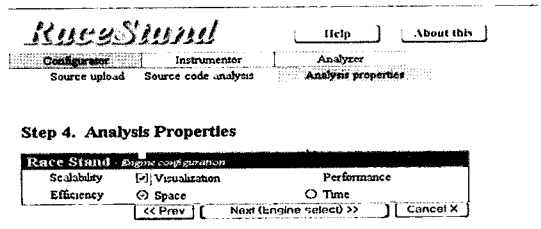



그림 9 사용자 요구사항그림

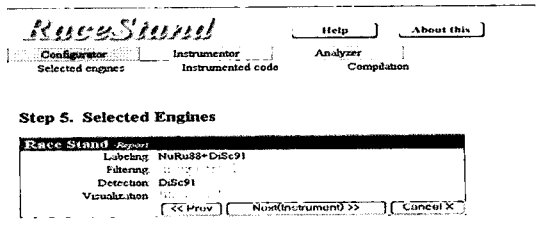
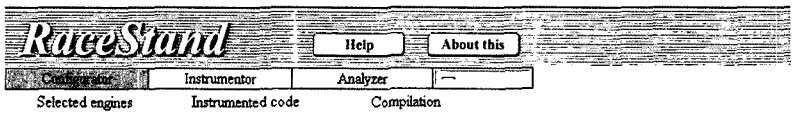



그림 10 경합탐지를 위한 엔진 정보그림




**Step 6. Instrumented Source-code**

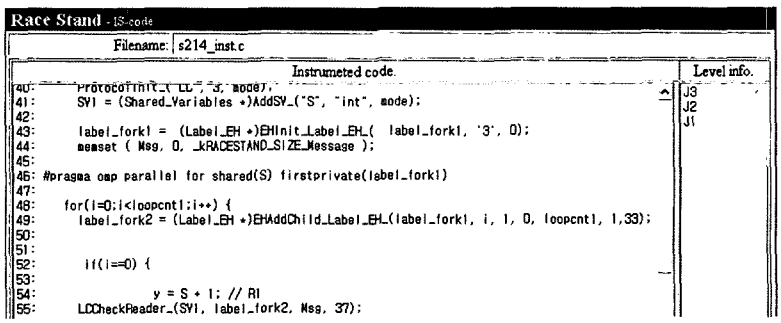
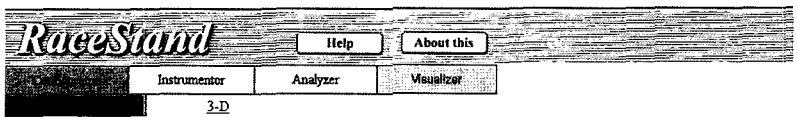



그림 11 변형된 OpenMP 프로그램그림




**Step 7: Race Report**

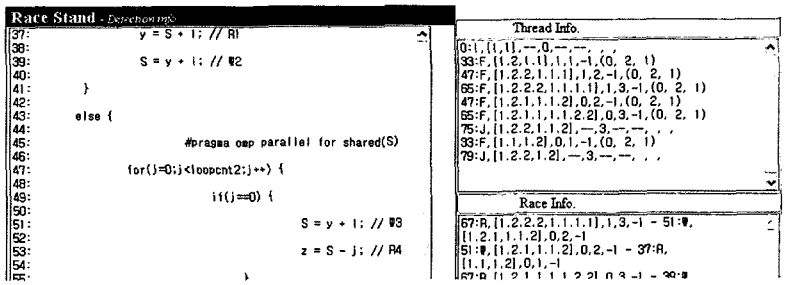



그림 12 수행중에 탐지된 경합정보

이 존재하지 하더라도 네트워크로 연결되어 있는 컴퓨터라면 언제 어디서든지 병렬 프로그램에 대한 경합검증 작업을 할 수 있다.

## 4. 실험 및 분석

이 절에서는 합성 프로그램을 이용하여 Thread Checker와 RaceStand를 기능적 측면과 성능적인 측면에서 비교분석하였다. 먼저, 기능과 성능을 효과적으로 측정하기 위한 합성프로그램 모델을 설명한다. 그리고 이러한 합성프로그램을 이용하여 Thread Checker와 RaceStand를 기능적인 측면과 성능적인 측면에서 분석하여 Thread Checker가 경합의 존재를 검증을 못할 뿐만 아니라, 비효율적인 이유를 설명한다.

### 4.1 합성 프로그램

스레드 기반의 프로그램 모델[10]에는 parallel computing 프로그램과 activity management 프로그램이 있다. parallel computing 프로그램은 단일 계산 작업을 여러 개의 병렬작업으로 나누어져 처리하며 이들 병렬 작업들은 동일한 종류의 데이타 구조나 변수들을 가지고 있다. 그리고 activity management 프로그램은 각 스레드에서 수행되는 작업들은 부모스레드와 독립적으로 수행되며 다른 종류의 데이타 구조들을 가지고 있다. 이러한 두 가지 프로그래밍 모델을 기반으로 작성한 합성 프로그램으로 Thread Checker의 기능과 성능을 분석한다. 기능분석을 위해서는 스레드마다 존재하는 접근 사건들에 대한 병행성 여부를 판단할 수 있어야 하므로

접근사건들이 다양한 지점에서 발생하는 명시적인 접근사건들이 필요하다. 또한 성능분석을 위해서도 접근사건들과 최대병렬성 등이 도구의 효율성에 의존적인지 여부를 파악해야 하므로 접근사건들이 다양한 지점에서 발생해야하고 독립적으로 수행하는 스레드들이 필요하다. 따라서 parallel computing 프로그램보다 activity management 프로그램 모델을 기반으로 하는 합성 프로그램을 작성하는 것이 효과적이다.

activity management 프로그램 모델을 적용한 합성 프로그램으로 도구의 기능을 분석하기 위해서 임계구역, 내포깊이, 최대병렬성을 조합하여 합성 프로그램을 작성하였다. 왜냐하면, 이 세 가지 요소는 병렬 프로그램 작성 시에 필요한 요소이기 때문이다. 작성된 프로그램으로 경합검증과 최초경합 여부를 분석하여 경합탐지 기능을 파악할 수 있고, 접근사건들의 수행중 유지정책을 알 수 있으므로 도구의 경합탐지 원리도 알 수 있다. 그리고 도구의 성능을 측정하기 위해서 최대병렬성, 내포깊이, 총 접근사건수, 스레드당 접근사건 수를 조합하여 합성 프로그램을 작성하였다. 이러한 프로그램으로 경합탐지 시에 소요되는 시간과 소요되는 메모리 및 파일공간을 측정하여 탐지성능을 분석하여 최대병렬성, 내포깊이, 그리고 접근사건수와의 의존성 관계를 파악할 수 있다.

### 4.2 기능적 측면

표 1은 동기화를 가진 비내포 병렬프로그램의 특성을 가진 합성 프로그램으로 Thread Checker와 Racestand를 이용하여 경합을 탐지한 예를 보인 것이다. F-102"에서 "F"는 탐지기능 분석을 위한 합성 프로그램을 의미하며 "1"은 임계구역을 설정하는 록 변수의 수를 의미하고, "0" 내포깊이를 의미하므로 내포병렬성이 없다는 의미이며, "2"는 최대병렬성을 의미한다. 그리고 [ ] 기호는 임계구역을 나타낸 것이다. 그리고 r은 읽기 접근사건이고 w는 쓰기 접근사건이며 접근사건들에 있는 번호는 발생순서를 의미한다. 예를 들어, "F-102"의 첫 번째 합성프로그램은 두 개의 스레드가 생성되어 하나의 스레드에 두 개의 읽기 접근사건이 발생하고 또 다른 스레드에 쓰기 접근사건이 발생하며 임계구역으로 보호되고 있다. 표에서 첫 번째와 다섯 번째 합성 프로그램에는 경합이 존재하지만 Thread Checker는 경합탐지를 하지 못하는 반면에 RaceStand는 경합을 탐지한다. Thread Checker가 경합을 탐지하지 못하는 이유는

표 1 동기화 비내포 병렬 프로그램의 경합탐지

| | F-102 | | Thread Checker | 제안된 도구 |
|---|---|---|---|---|
| 1 | r1 [r2] | [w3] | - | w3-r1 |
| 2 | [r1] | [w2] w3 | r1-w3 | w3-r1 |
| 3 | [r1] [w2] | [r3] | - | - |
| 4 | [w1] r2 | [r3] | - | - |
| 5 | w1 [w2] | [r3] | - | r3-w1 |

*r2*에 의해서 *r1*이 삭제되고 *w2*에 의해서 *w1*이 삭제되는 유형이 실험적으로 관측되었기 때문이다. 또한 내포병렬성이 존재하는 프로그램에서도 경합검증을 하지 못한다. 왜냐하면 내포된 두 스레드에서 병행관계에 있는 읽기접근사건과 쓰기접근사건이 경합으로 보고되지 않기 때문이다. 따라서 Thread Checker는 동기화가 없고 비내포 병렬프로그램에 대해서만 경합검증이 가능하지만 RaceStand는 동기화와 내포병렬성을 가진 프로그램 모델에 대해서 모두 경합검증이 가능하다.

이러한 실험을 통해서 Thread Checker가 수행중 경합탐지 시에 접근사건들을 어떻게 유지하는지 알 수 있었다. 현재 읽기접근사건이 이전의 읽기접근사건과 순서화 관계에 있으면 이전의 읽기접근사건을 삭제하고 현재의 접근사건들을 유지하고, 병행성 관계에 있으면 현재 읽기접근사건을 삭제하고 이전의 읽기접근사건을 유지한다. 그리고 쓰기접근사건은 항상 이전 쓰기접근사건을 삭제하고 현재 쓰기접근사건을 저장한다.

### 4.3 성능적 측면

도구의 효율성을 분석하기 위해서 두 종류의 합성 프로그램을 개발하였다. 첫 번째는 스레드당 접근사건수에 따른 효율성 측정을 위한 프로그램이고 두 번째는 최대병렬성에 따른 효율성 측정을 위한 프로그램이다. 첫 번째 합성 프로그램은 스레드당 접근사건수를 100, 200, 300, 400으로 분류하고 최대병렬성을 2의 지수승인 2, 4, 8, 16, 32로 분류한다. 두 번째 합성 프로그램은 총 접근사건 수를 1000, 4000으로 분류하고 최대병렬성을 2의 지수승인 2, 4, 8, 16, 32로 분류한다. 표 2의 실험결과는 Thread Checker에서 합성 프로그램의 수행시간에 대한 경합탐지 수행시간의 증가비율을 나타낸 것이

표 2 Thread Checker의 실험결과

| | 접근사건수 | | 최대병렬성 | | | |
|---|---|---|---|---|---|---|
| | 1000 | 4000 | 2 | 4 | 8 | 16 |
| 소요시간 | 391배 | 4296배 | 170배 | 642배 | 1952배 | 4752배 |
| 소요메모리 | 20배 | 27배 | 10배 | 14배 | 18배 | 31배 |
| 소요파일 | 7배 | 12배 | 6배 | 7배 | 9배 | 13배 |

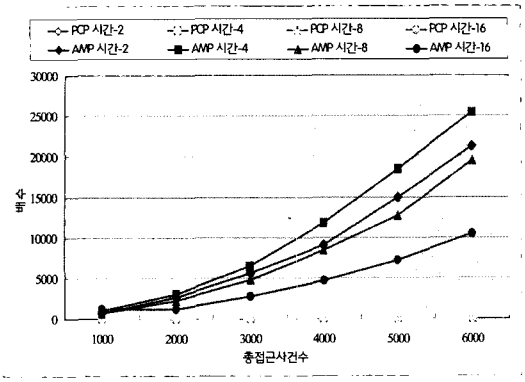



그림 13 소요시간 측정결과: PCP vs AMP

다. 표 2에서 알 수 있는 것은 소요메모리와 소요파일공간과는 달리 소요시간에 대한 증가비율은 접근사건수가 4000인 경우에 평균 4296배 증가되었고, 최대병렬성이 16인 경우에 평균 4752배 증가되었기 때문에 효율성에 가장 큰 영향을 미치는 요인은 스레드당 접근사건 수와 최대병렬성이라고 알 수 있다. 이 두 요인에서 공통적인 것은 총접근사건수이다. 지금까지의 효율성 분석결과는 activity management 프로그램 모델[10]에 대해서 분석한 결과이다. parallel computing 프로그램 모델[10]을 이용한 합성 프로그램에서는 경합탐지 시에 소요되는 시간과 공간에 대해서 변화율이 거의 없다. 왜냐하면 프로그램 수행 중에 발생하는 접근사건들에 대해서 하나의 접근사건으로 인식하기 때문이다. 즉 대부분의 경우 parallel computing 프로그램 모델은 반복문 구조로 되어있기 때문이다. 그림 13은 parallel computing 프로그램과 activity management 프로그램 모델에서 총접근사건수의 증가에 따른 측정결과이다. 여기서 parallel computing 프로그램을 PCP라고 하며, activity management 프로그램을 AMP라고 한다. 실험환경은 싱글 프로세서를 탑재한 컴퓨터이다. PCP와 AMP 프로그램 모델에서의 실험 분석을 요약하면, 소요시간에서는 AMP는 PCP보다 최대 2039.8배 증가되었고, 소요메모리에서 AMP는 PCP보다 최대 6.4배 증가되었으며, 소요파일에서 AMP는 PCP보다 3.5배 증가되었다는 것을 알 수 있었다. 그리고 AMP 프로그램 모델에서 소요시간이 가장 민감하며 소요시간의 증가비율 $m$은 총접근사건수 $n$의 $O(n^2)$로서, $m = 0.0007n^2 + 0.3592n - 193.281$이였다. 왜냐하면 경합탐지를 위해서 프로그램 내에 존재하는 모든 접근사건들을 순차적으로 수행하면서 병행성 여부를 검사하여 경합을 탐지하기 때문이다.

일반적으로 Thread Checker는 경합탐지 시에 소요되는 시간이 비효율적으로 알려져 있다. 따라서 싱글 프로세서와 멀티 프로세서를 탑재한 컴퓨터에서 Thread Checker와 RaceStand를 소요시간 관점에서 비교분석하였다. 표 3은 최대병렬성을 2, 4, 8, 16으로 하고 각 최대병렬성마다 총접근사건수를 2000, 3000, 4000인 합성 프로그램을 이용하여 경합탐지 시에 소요된 시간을 측정한 결과이다. 예를 들어, 싱글 프로세서에서 최대병렬성 2에서 총 접근사건 수 4000일 때 RaceStand는 1.03초가 소요되고, Thread Checker는 272초가 소요된다. 그리고 최대병렬성 16에서 총 접근사건 수가 4000일 때 RaceStand는 1.05초가 소요되고, Thread Checker는 928초가 소요된다. RaceStand는 최대병렬성과 총 접근사건 수가 증가되더라도 소요시간의 변화율이 0.02초 차이가 보이지만 Thread Checker는 소요시간의 변화율이 무려 656초 차이를 보인다. 그림 14는 표 3의 실험결과 중에서 싱글 프로세서 실험한 결과를 참고하여 RaceStand와 Thread Checker간의 소요시간의 증가비율을 나타낸 것이다. X축은 최대병렬성과 총접근사건수를 의미하며 최대병렬성이 2, 4, 8, 16이고 최대병렬성마다 총접근사건 수가 2000, 3000, 4000으로 되어 있다. Y축은 소요시간의 증가비율은 나타낸 것이다. 예를 들어, 최대병렬성 2에서 총 접근사건 수가 2000일 때 RaceStand

표 3 싱글 프로세서/멀티 프로세서에서 Thread Checker와 RaceStand의 소요시간

| 최대 병렬성 | 접근사건수 | 총접근사건수 | 싱글 프로세서: 소요시간 (Sec) | | 멀티 프로세서: 소요시간 (Sec) | |
|---|---|---|---|---|---|---|
| | | | RaceStand | Thread Checker | RaceStand | Thread Checker |
| 2 | 1000 | 2000 | 0.71 | 74 | 0.1047 | 8.1523 |
| | 1500 | 3000 | 0.87 | 158 | 0.1253 | 8.2460 |
| | 2000 | 4000 | 1.03 | 272 | 0.1730 | 8.3453 |
| 4 | 500 | 2000 | 0.76 | 154 | 0.0720 | 11.5240 |
| | 750 | 3000 | 0.87 | 334 | 0.0927 | 13.7840 |
| | 1000 | 4000 | 1 | 591 | 0.1117 | 16.3877 |
| 8 | 250 | 2000 | 0.72 | 211 | 0.0553 | 13.3627 |
| | 375 | 3000 | 0.89 | 460 | 0.0593 | 15.7660 |
| | 500 | 4000 | 0.96 | 802 | 0.0710 | 15.9343 |
| 16 | 125 | 2000 | 0.71 | 239 | 0.0417 | 14.2973 |
| | 188 | 3008 | 0.89 | 548 | 0.0480 | 15.6130 |
| | 250 | 4000 | 1.05 | 928 | 0.0513 | 15.7810 |

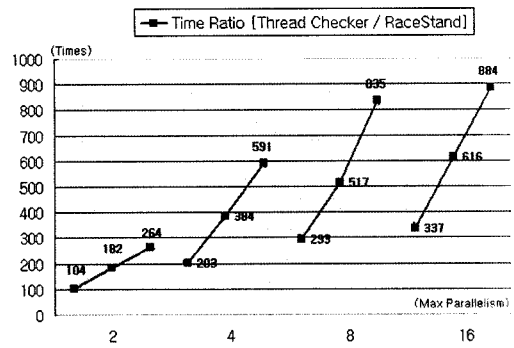



그림 14 RaceStand/Thread Checker의 시간 증가비율: 싱글 프로세서

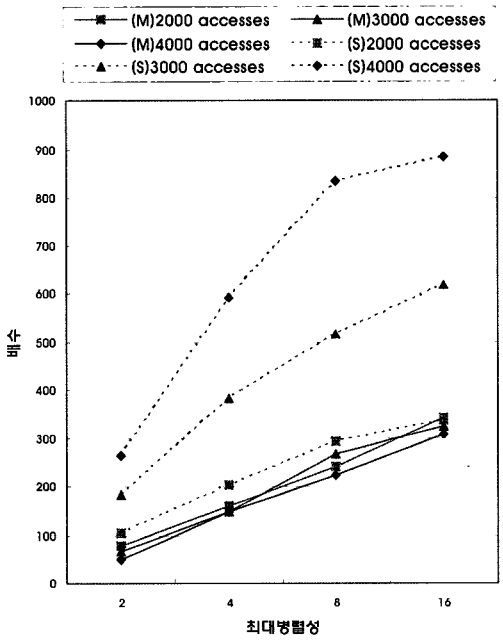



그림 15 싱글프로세서와 멀티프로세서에서 RaceStand / Thread Checker의 시간 증가비율

보다 Thread Checker가 경합탐지 시에 소요되는 시간이 104배이고, 최대병렬성 16에서 총 접근사건수 4000일 때의 증가비율은 884배이다. 그림 14에서 알 수 있는 것은 많은 수의 접근사건과 병렬성을 가지고 있다고 하더라도 경합탐지를 위해서 소요되는 시간은 Thread Checker보다 RaceStand가 실용적임을 알 수 있다.

그리고 표 3의 멀티프로세서 상에서 실험하여 분석한 결과를 살펴본다. 합성프로그램은 싱글 프로세서 상에서 실험한 합성프로그램과 동일하고 실험결과는 Linux 기반의 Intel 64비트 듀얼 프로세서(4개의 CPU로 인식)에서 수행하여 측정한 것이다. 표 3에서 최대병렬성이 16이고 스레드당 접근사건수가 125일 때 Thread Checker에서 소요된 시간은 14.2973초이고 RaceStand에서 0.0417초이다. 따라서 최대병렬성 2에서 소요시간은 Thread Checker에 비해서 RaceStand는 평균 63.9배 감소, 최대병렬성 4에서는 평균 151.8배 감소, 최대병렬성 8에서는 평균 244배 감소, 그리고 최대병렬성 16에서는 평균 325.3배 감소되었다. 그림 15는 싱글 프로세서와 멀티프로세서 상에서 Thread Checker와 RaceStand의 경합탐지 시에 소요된 시간에 대한 증가비율을 그래프로 나타낸 것이다. (M) 기호는 멀티프로세서라는 의미이고, (S) 기호는 싱글 프로세서라는 의미이다. 여기서 싱글 프로세서 상에서 수행한 결과는 총접근사건수의 증가 시에 시간의 증가비율이 크지만 멀티프로세서 기반에서 수행한 결과는 증가비율이 거의 동일한 형태로 나타나 있다.

## 5. 결 론

OpenMP 프로그램에서 발생하는 경합을 탐지하는 도구인 Thread Checker는 protection 기법을 사용한다. 이 도구는 경합의 존재를 경합하지 못하고 특히, 경합을 탐지하는 비용이 크기 때문에 비실용적이다. 본 논문에서 제안한 도구인 RaceStand는 기능과 성능적인 측면에서 기존의 도구들보다 실용적인 도구이다. 왜냐하면 논리적 병행성 정보를 생성하기 위해서 확장적 레이블링 기법을 사용하여 생성된 모든 스레드들에 대해서 논리적 병행성 여부를 판단할 수 있고 경합에 참여할 수 있는 병행한 접근사건들만을 공유자료구조인 access history에 저장하여 현재 발생한 접근사건들과 경합여부를 검사하므로 경합이 존재하다면 적어도 하나의 경합을 탐지할 수 있는 경합검증 프로토콜을 사용했다. 또한 총 접근사건 수가 증가하더라도 경합을 탐지하는데 소요되는 시간이 Thread Checker보다 효율적임을 알 수 있었기 때문이다.

현재 감시엔진을 삽입하는 Source Instrumemtor는 C 언어를 기반으로 하는 OpenMP 프로그램 모델에서만 한정적으로 적용되며 OpenMP 병렬화 디렉티브 중에서 "#pragma omp parallel for"만을 고려하고 동기화 디렉티브 중에서 "#pragma omp critical"과 묵시적 동기화에 해당하는 join만을 고려하고 있다. 그리고 내부적 비결정성이 존재하지 않는 프로그램에 대해서만 RaceStand는 경합의 존재를 검증할 수 있다. 현재 지속적으로 Racestand의 세부모듈을 수정하고 있으며, RaceStand의 핵심엔진에 해당하는 실용적인 경합탐지 기법도 보다 효율적인 기법으로 변경할 것이다. 향후과제로는 탐지된 경합을 시각적으로 표현해서 사용자에게 제공함으

로서 효과적인 디버깅을 할 수 있는 환경을 제공하는 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 Programs Interface*, Version 2.5, May 2005.

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

[4] Intel Corp., *Getting Started with the Intel Thread Checker*, 2200 Mission College Blvd., Santa Clara, CA 95052-8119, USA, 2004.

[5] Intel Corp., *Intel Thread Checker for Windows 3.0 Release Notes*, 2200 Mission College Blvd., Santa Clara, CA 95052-8119, USA, 2005.

[6] Intel Corp., *Threading Methodology: Principle and Practices*, Version 2.0, 2200 Mission College Blvd., Santa Clara, CA 95052-8119, USA, 2003.

[7] Petersen, P., and S. Shah, "OpenMP Support in the Intel Thread Checker," *Proc. of the Int'l Workshop on OpenMP Application and Tools* (WOMPAT), Berlin Heidelberg, Lecture Notes in Computer Science, 2716: 1-12, Springer-Verlag, 2003.

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

[9] Nudler, I., and L. Rudolph, "Tools for the Efficient Development of Efficient Parallel Programs," *In 1st Israeli Conference on Computer System Engineering*, 1986.

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

[11] Intel Corp., *Getting Started with the Intel C++ Compiler 9.0 for Windows*, 2200 Mission College Blvd., Santa Clara, CA 95052-8119, USA, 2004.

[12] Intel Corp., *VTune(TM) Performance Analyzer 8.0 Release Notes*, 2200 Mission College Blvd., Santa Clara, CA 95052-8119, USA, 2006.

[13] Kusano, K., S. Satoh, and M. Sato, "Performance Evaluation of the Omni OpenMP Compiler," *3rd Int'l Symp. on High Performance Computing*, pp. 403-414, Springer-Verlag, 2000.

[14] Banerjee, U., B. Bliss, Z. Ma, and P. Petersen, "A Theory of Data Race Detection," *Proc. of Workshop on Parallel and Distributed Systems: Testing and Debugging* (PADTAD), pp. 69-78, ACM, Portland, USA, July 2006.

[15] Dinning, A., and E. Schonberg, "An Empirical Comparison of Monitoring Algorithms for Access Anomaly Detection," *2nd Symp. on Principles and Practice of Parallel Programming*, pp. 1-10, ACM, March 1990.

[16] 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.

[17] 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.

[18] Park, S., M. Park, and Y. Jun, "A Comparision of Scalable Labeling Schemes for Detecting Races in OpenMP Programs," *Int'l Workshop on OpenMP Applications and Tools* (Wompat), pp. 66-80, West lafayette, Indiana, July 2001.

[19] Kim, J., and Y. Jun, "Scalable On-the-fly Detection of the First Races in Parallel Programs," *Proc. of the 12nd Int'l Conf. on Supercomputing* (ICS), pp. 345-352, ACM, Melbourne, Australia, July 1998.

[20] Park, H., and Y. Jun, "Two-Pass On-the-fly Detection of the First Races in Shared-Memory Parallel Programs," *Proc. of the 2nd Symp. on Parallel and Distributed Tools* (SPDT), ACM, Welches, Oregon, August 1998.



김 영 주

1999년 경상대학교 컴퓨터과학과 졸업(학사). 2001년 경상대학교 컴퓨터과학과 졸업(석사). 2007년 경상대학교 컴퓨터과학과 졸업(박사). 관심분야는 병렬/분산 프로그램 디버깅 및 시각화, 유비쿼터스 컴퓨팅, 시스템 소프트웨어, 리눅스 클러스터링

전 용 기

정보과학회논문지 : 시스템 및 이론
제 34 권 제 7 호 참조