*저자는 그림 및 표와 수식을 다시한번 확인하여 주시기 바랍니다.

단일 칩 다중 프로세서상에서 운영체제를 사용하지 않은 OpenMP 구현 및 주요 디렉티브 변환

(Implementation and Translation of Major OpenMP Directives for Chip Multiprocessor without using OS)

전 우 철 † 하 순 회 ††
(Woochul Jeun) (Soonhoi Ha)

요 약 단일 칩 다중 프로세서의 경우 표준화된 병렬 프로그래밍 방법이 없는데 OpenMP를 사용하면 병렬 프로그래밍이 쉬우므로 OpenMP는 단일 칩 다중 프로세서를 위한 매력적인 병렬 프로그래밍 모델이다. 그런데 단일 칩 다중 프로세서 시스템의 구조는 대상 응용 프로그램에 따라 다양할 수 있다. 따라서 각 시스템마다 다른 방식으로 OpenMP를 구현해야 할 필요가 있다. 본 논문에서는 운영체제를 사용하지 않는 단일 칩 다중 프로세서를 위한 OpenMP 구현과 주요 디렉티브의 효과적인 변환을 제안하여 특수한 하드웨어에 의존하지 않고 OpenMP 디렉티브의 추가적인 확장 없이 성능을 향상 시킬 수 있게 한다. 실험은 대상 플랫폼인 CT3400에서 수행하고 그 결과를 제시한다.

키워드 : OpenMP, OpenMP 구현, OpenMP 변환, 단일 칩 다중 프로세서, 동기화, 리덕션 절

Abstract OpenMP is an attractive parallel programming model for a chip multiprocessor because there is no standard parallel programming method for a chip multiprocessor and it is easy to write a parallel program in OpenMP. Then, chip multiprocessor systems can have various architectures according to target application programs. So, we need to implement OpenMP in different way for each system. In this paper, we propose the implementation and the effective translation of major OpenMP directives for a chip multiprocessor without using OS to improve the performance without using special hardware and without extending the OpenMP directives. We present the experimental results on our target platform CT3400.

Key words : OpenMP, OpenMP implementation, OpenMP translation, Chip multiprocessor, synchronization, reduction clause

1. 서 론

병렬 프로그래밍 모델은 크게 메시지 전달(message passing) 방식과 공유 주소 공간(shared address space) 방식으로 나눌 수 있다. 메시지 전달 방식은 프로세서마다 별도의 메모리 공간이 있는 구조를 가정하고 메시지 전달을 통해 프로세서간에 데이타를 교환한다. 이 방식은 프로그래머가 프로세서간에 교환할 데이타를 고려해서 프로그램을 최적화할 경우 좋은 성능을 얻을 수 있지만 프로그래밍이 어려운 특성이 있다. 메시지 전달 방식의 사실상의 표준(de facto standard)화된 인터페이스로는 MPI (Message Passing Interface)[1]가 있으며 물리적으로 분산 메모리(distributed memory) 구조를 가진 클러스터 등에서 많이 사용한다. 다음으로 공유 주소 공간 방식은 모든 프로세서가 공유하는 하나의 메모리 영역이 있는 구조를 가정하고 공유 메모리(shared memory) 영역의 접근을 통해서 프로세서간에 데이타를 교환한다. 이 방식은 메모리 일관성(memory consistency) 유지를 위한 메모리 일관성 모델을 필요로 하지만 프로그래밍이 쉽고 일반 프로그램을 병렬 프

· 본 연구는 두뇌한국 21 프로젝트, SystemIC 2010 프로젝트, 정보통신 선도기반기술개발사업, IT-SoC 핵심설계인력양성사업에 의해 지원되었으며, 이 연구를 위해 연구장비를 지원하고 공간을 제공한 서울대학교 컴퓨터연구소, 서울대학교 반도체공동연구소, 반도체설계 교육센터에 감사드립니다.

† 학생회원 : 서울대학교 전기컴퓨터공학부
wcjeun@iris.snu.ac.kr
†† 정 회 원 : 서울대학교 전기컴퓨터공학부 교수
sha@iris.snu.ac.kr

논문접수 : 2006년 6월 29일
심사완료 : 2007년 1월 23일

로그램으로 변환하기도 쉬운 특성이 있다. 공유 메모리 방식의 사실상의 표준화된 인터페이스로는 OpenMP[2]가 있으며 물리적으로 공유 메모리 구조를 가진 대칭형 다중처리 (SMP: symmetric multiprocessing)기계에서 주로 많이 사용한다[3,4]. 그리고 프로그래밍이 쉽다는 장점 때문에 대칭형 다중처리 기계 이외에도 클러스터와 시스템-온-칩을 비롯한 여러 병렬처리 플랫폼(platform)에서도 병렬 프로그래밍 모델로 OpenMP를 사용하려는 시도를 하고 있다[5-8].

OpenMP를 사용하기 위해서는 대상 플랫폼에서 실행 가능한 OpenMP 구현(implementation)이 있어야 하고 그 구현에 맞게 프로그램을 변환(translation)하는 OpenMP 변환기(translator)가 필요하다. OpenMP 인터페이스(interface)는 컴파일러 디렉티브(compiler directives)를 이용해서 프로그래머의 의도를 표현하는 명세방법(specification)에 대한 표준일 뿐이며 어떤 병렬처리 플랫폼에서 실제로 OpenMP 디렉티브를 어떻게 구현할 것인가는 정하고 있지 않다. 즉 플랫폼이 다른 경우 각 플랫폼에서 사용할 수 있는 API(Application Programming Interface)가 다르므로 플랫폼에 따라 OpenMP 구현이 달라지며 같은 병렬처리 플랫폼이라도 구현 방식에 따라 여러 가지 서로 다른 OpenMP 구현이 있을 수 있다. 따라서 OpenMP 프로그램을 특정한 OpenMP 구현에 맞게 변환하기 위해서는 해당 OpenMP 구현을 위한 OpenMP 변환기가 필요하다.

단일 칩 다중 프로세서(Chip Multiprocessor)의 경우 현재 표준화된 병렬 프로그래밍 방법이 없으므로 프로그래밍이 쉬운 OpenMP를 병렬 프로그래밍 모델로 사용하려는 연구들이 있었지만[7-9], 다른 병렬처리 플랫폼에 비해서 이러한 OpenMP 구현과 변환에 대한 연구가 아직 부족하다. 그런데 일반적으로 단일 칩 다중 프로세서의 경우 대상 응용 프로그램에 따라 운영체제 사용여부나 메모리 형태가 다양할 수 있다. 그리고 이렇게 다양한 플랫폼 구조에서 OpenMP 프로그램을 효과적으로 수행하려면 각각의 구조에 맞는 OpenMP 구현과 OpenMP 변환을 필요로 하지만 아직 각각의 구조에 따른 OpenMP구현과 변환에 대한 연구가 부족하다. 공유 메모리 구조에서 운영체제로 대칭형 다중처리 커널 리눅스를 사용하고 POSIX 쓰레드를 사용할 수 있는 플랫폼에서는 기존의 대칭형 다중처리 기계에서 사용하는 방식으로 OpenMP 구현과 변환을 쉽게 할 수 있다[8]. 하지만 그러한 운영체제를 사용하지 않고 쓰레드도 지원하지 않는 플랫폼에서 OpenMP를 사용하기 위해서는 기존의 대칭형 다중처리 기계에서 사용하는 방식과는 다른 OpenMP 구현과 변환을 필요로 한다.

본 논문에서는 운영체제를 사용하지 않는 단일 칩 다중 프로세서를 위한 효과적인 OpenMP 구현과 변환을 제안하고 대상 플랫폼인 CT3400 단일 칩 다중 프로세서 보드에 적용하여 그 성능을 분석한다. 특히 대상 플랫폼을 사용한 기존 연구들과[7,9] 달리 특수한 하드웨어에 의존하지 않고 OpenMP 디렉티브의 추가적인 확장 없이 성능을 향상 시킬 수 있는 OpenMP 구현과 주요 디렉티브의 변환을 제시하고 잘못 동작할 가능성이 있었던 기존 동기화 디렉티브 구현을 대체할 새로운 동기화 디렉티브 구현을 제안한다. 몇 가지 예제 프로그램을 변환하고 실행함으로써 제시한 OpenMP 구현과 변환이 유효한지 확인하고 제시한 변환 방법과 구현이 예제 프로그램의 성능을 향상시키는지 확인한다.

논문의 구성은 다음과 같다. 2장에서는 배경지식으로 연구에서 사용한 단일 칩 다중 프로세서 보드와 OpenMP에 대한 개략적인 소개를 한다. 3장에서는 세마포어는 지원하지만 대칭형 다중처리 커널 운영체제를 사용하지 않고 쓰레드도 지원하지 않는 단일 칩 다중 프로세서 플랫폼에서의 효과적인 OpenMP 구현과 변환을 주요 디렉티브를 중심으로 설명하고, 연구에서 사용한 단일 칩 다중 프로세서 보드인 CT3400을 위한 OpenMP구현과 OpenMP 변환을 제시한다. 4장에서는 CT3400을 위한 OpenMP 변환기를 설명하고 실험 결과를 제시한다. 5장에서는 관련연구를 소개하고 마지막 6장에서는 논문의 내용을 요약해서 결론을 내리고 앞으로 남은 과제들에 대해 소개한다.

2. 배 경

어떤 OpenMP구현과 변환을 제시하기 위해서는 특정한 병렬처리 플랫폼을 정하고 그 플랫폼에서 OpenMP 수행 모델에 따라 OpenMP 디렉티브를 어떻게 구현하고 변환할 지 결정해야 한다. OpenMP구현은 병렬처리 플랫폼에 따라 달라지며 OpenMP 변환은 OpenMP 구현에 따라 달라지기 때문이다.

2.1 CT3400 단일 칩 다중 프로세서 보드

본 논문에서 사용한 CT3400[10] 보드는 Cradle Technologies, Inc.에서 제작한 단일 칩 다중 프로세서 보드이다. 운영체제를 사용하지 않고 물리적으로 공유 메모리 구조를 가지며 하드웨어 세마포어를 제공하는 단일 칩 다중 프로세서 보드로 그림 1은 이 보드의 구조를 간략하게 표현한 것이다. 원래 여덟 개의 DSP를 갖고 있지만 DSP는 본 논문에서 다루고 있지 않은 부분이므로 논문과 관련 없는 부분들을 생략한 그림이다.

보드의 구성은 크게 점선으로 표시한 칩(chip)과 칩 바깥에 있는 256MB의 오프-칩(off-chip) 데이타 메모리로 나눌 수 있다. 칩은 쿼드(Quad)라는 기능 블록을 기본단위로 하며 쿼드들이 공유할 수 있는 전역 세마포

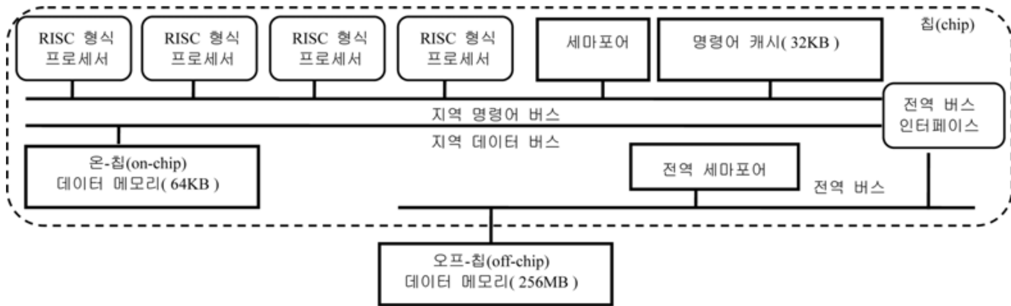

그림 1 간략하게 표현한 CT3400 단일 칩 다중 프로세서 보드의 구조

어를 갖고 있는데 하나의 쿼드는 네 개의 230MHz RISC 형식(RISC-like) 프로세서, 쿼드 내부에 있는 32개의 지역 세마포어, 32KB의 명령어 캐시, 64KB의 온-칩(on-chip) 데이타 메모리로 이루어진다. 온-칩 메모리와 오프-칩 메모리는 모든 프로세서들이 접근할 수 있는 물리적인 공유 메모리이다. 하지만 이 플랫폼에서는 변수를 선언할 때 별도의 키워드를 지정함으로써 변수가 존재할 메모리 위치를 온-칩 메모리와 오프-칩 메모리 중에서 선택할 수 있게 하고, 변수의 공유 여부를 지정할 수 있게 하고 있다. 키워드를 지정하지 않은 경우 변수는 오프-칩 메모리에 위치하며 해당 프로세서만 독점적으로 접근할 수는 단독(private)변수로 설정된다. 한가지 제약사항은 프로그램에서 전역변수로 선언한 변수만 공유변수(shared variable)로 지정할 수 있다는 점으로 프로세서간 통신도 이러한 공유변수를 통해서 이루어진다. 온-칩 메모리의 경우 일부 영역을 데이타 캐시(cache)로 지정할 수도 있지만 본 논문에서는 데이타 캐시를 사용하지 않도록 설정한다. 전역 세마포어는 64개를 사용할 수 있다. 프로세서를 위한 프로그래밍 언어로는 C언어를 사용할 수 있고 'gcc'호환 컴파일러인 'cragcc'[11]를 사용할 수 있다.

2.2 OpenMP 개요

OpenMP는 현재 C/C++언어와 FORTRAN언어 명세에 대해 정의하고 있으며 컴파일러 디렉티브(compiler directives)에 기반을 두고 있으며 병렬 프로그램을 쉽게 작성할 수 있는 장점이 있다. 그림 2는 0부터 9999까지의 합을 구하는 'for' 루프에 OpenMP 디렉티브를 추가해서 OpenMP 프로그램으로 만드는 C프로그램의 예로, 병렬적으로 실행할 부분에 OpenMP 디렉티브를 추가한 것이다.

다음으로 이렇게 OpenMP로 작성한 프로그램을 실행하기 위해서는 특정한 OpenMP 구현에 맞는 병렬 프로그램으로 변환하고 컴파일 해야 한다. OpenMP 변환이란 이러한 변환 작업을 말하며 OpenMP 변환을 수행하는 부분을 OpenMP 변환기(translator)라고 하고, OpenMP 변환기 부분과 컴파일 작업을 수행하는 부분이 합쳐진 형태를 OpenMP 컴파일러라고 한다. OpenMP 구현마다 그 구현에 맞는 병렬 프로그램을 생성하는 OpenMP 변환기가 필요하므로 여러 연구들이 다양한 플랫폼에서 OpenMP 변환에 대해서 다루고 있으며[3-8,12] 일부 연구의 경우 파서(parser)나 컴파일러 모듈을 갖다가 사용하는 방법을 이용함으로써 OpenMP 변환기 제작에 드는 시간과 노력을 줄이고 있다[3,6,8,12].

OpenMP는 포크/조인(fork/join) 모델을 수행 모델로 택하고 있다. 프로그램을 실행하면 OpenMP 디렉티브로 지정한 병렬처리 영역을 만날 때까지 하나의 주 쓰레드(master thread)로 수행하고 병렬처리 영역에서 주 쓰레드는 여러 개의 자식 쓰레드를 생성한 후 각 쓰레드별로 처리할 작업을 나누어 처리한다. 모든 쓰레드가 각각 할당된 작업을 마치고 병렬처리 영역의 끝에 도달하면 동기화(synchronization) 과정을 거친 뒤 모든 쓰레드는 다시 주 쓰레드로 합쳐진다. 그리고 주 쓰레드는

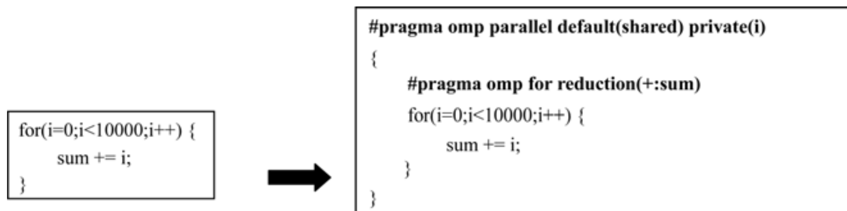

그림 2 0부터 9999까지 합을 구하는 for 루프에 OpenMP 디렉티브를 추가한 예

다음 병렬처리 영역을 만나면 위에서 언급한 과정을 반복하면서 프로그램의 끝까지 진행한다.

3. 운영체제를 사용하지 않는 공유 메모리 단일 칩 다중 프로세서를 위한 OpenMP 구현과 변환

OpenMP 구현과 변환은 대상 플랫폼에 종속적이므로 본 논문에서는 운영체제와 쓰레드 라이브러리 없이 OpenMP 구현을 해야 할 때 고려해야 하는 점들과 해결책을 주요 OpenMP 디렉티브를 중심으로 소개하고 앞에서 소개한 CT3400을 대상으로 실제 OpenMP 구현과 변환을 보인다.

3.1 병렬처리 영역에서의 변수 변환

'parallel'과 'for' 등의 OpenMP 디렉티브 뒤에는 병렬처리 영역에서 사용하는 변수의 속성을 지정하는 용도로 절(clause)을 사용할 수 있으며 변수 속성에는 'shared'속성과 'private'속성이 있다.

- 'shared' 속성 변수: 병렬처리 영역 안에서 프로세서들이 논리적으로 공유하는 변수.
- 'private' 속성 변수: 병렬처리 영역 안에서 프로세서들이 논리적으로 공유하지 않는 독점적인 변수.

이러한 속성은 대상 플랫폼에서 해당 변수의 실제 선언이 공유변수인지 단독변수인지와 상관 없이

- 공유변수: 물리적으로 모든 프로세서들이 접근할 수 있는 변수.
- 단독변수: 물리적으로 하나의 프로세서만 독점적으로 접근할 수 있는 변수.

OpenMP 프로그램에서 지정한 속성을 나타낸다. 본 논문에서는 혼동을 막기 위해 OpenMP 속성으로 구분하는 경우는 'shared'변수와 'private'변수라는 용어를 사용하고, 실제 플랫폼에서의 선언에 따라 물리적으로 다른 프로세서들이 직접 접근할 수 있는지 여부를 나타내는 구분은 공유변수와 단독변수라는 용어를 사용한다. 따라서 OpenMP에서 속성을 'shared'변수로 지정했더라도 실제 선언은 공유변수가 아니라 단독변수로 했을 수 있다.

표 1은 OpenMP 변수 속성과 변수의 원래 선언에 따른 OpenMP 변환 방법을 정리한 것이다. 'private' 속성 변수는 변수의 원래 선언에 상관없이 병렬처리 영역 내부에서 같은 이름의 변수를 새로 선언하면 된다. 반면에 'shared' 속성 변수는 원래의 변수 선언에 따라 변환을 달리해야 한다. 특히 운영체제가 없는 단일 칩 다중 프로세서 플랫폼은 각각의 프로세서에 실행 이미지를 적재하고 실행시키는 구조이므로 일부 'shared'변수들은 단독변수일 가능성이 있고 이를 위한 일관성(consistency) 유지 방법을 필요로 한다.

CT3400에서는 전역변수를 공유변수로 선언할 수 있으므로 'shared'속성을 갖는 공유변수가 존재할 수 있다. 그런데 변수를 선언하는 방법은 메모리 할당방법에 따라 정적할당과 동적할당의 두 가지가 가능하며 각각의 경우 변수가 위치하는 메모리 영역은 프로그램에서의 전역 데이타(global data) 영역과 힙(heap) 영역으로 다르다. 정적할당은 'int data[10000]'와 같이 전역변수 선언을 하는 방법으로 배열은 전역 데이타 영역에 위치한다. 반면 동적할당은 'int *data'와 같이 전역 포인터 변수를 선언하고 나중에 'malloc()'과 같은 함수를 이용해서 메모리를 할당하는 방법이다. 이 경우에 포인터인 'data'는 전역 데이타 영역에 위치하지만 실제 데이타가 있는 위치는 'malloc()'으로 할당 받은 힙(heap) 영역에 위치한다.

CT3400에서는 'shared'속성 공유변수를 위해 정적할당을 사용하는 것이 동적할당을 사용하는 것보다 성능을 높일 수 있다. PC에서는 전역변수의 메모리 위치가 전역변수 영역인지 힙 영역인지에 따라 큰 성능 차이가 나지는 않는다. 하지만 CT3400에서 사용하는 컴파일러인 'cragcc'는 전역 데이타 영역에 위치한 전역변수를 효율적으로 처리하는 기능을 지원한다[11]. 물론 이러한 기능은 컴파일러의 동작에 따라 성능 향상 여부가 결정되므로 전역변수를 정적할당 한다고 모든 응용 프로그램에서 항상 성능 향상을 가져오는 것은 아니다. 하지만 전역변수를 동적할당하면 전역변수를 전역변수 영역이 아닌 힙 영역에 변수를 위치시키게 되므로 컴파일러가 성능을 향상시킬 수 있는 기회가 아예 없다. 또한 다른 플랫폼에서도 이 정보를 이용하는 컴파일러를 제작할 경우 성능을 향상시킬 수 있는 가능성을 제공한다.

다음으로 CT3400에서 지역변수는 단독변수이므로 'shared'변수 속성을 가진 지역변수라도 프로세서마다 동일한 이름의 단독변수가 있을 뿐이다. 각 프로세서는 'shared'변수의 복사본을 이 단독변수에 저장하고 유지한다. 따라서 OpenMP 변환은 각 프로세서가 갖고 있는 단독변수 값들의 일관성을 유지하게 하는 것이다. 공

표 1 OpenMP 변수 속성과 변수의 원래 선언에 따른 OpenMP 변환 방법

속성 \ 변수의 원래 선언	공유변수	단독변수
'shared' 속성 변수	- 별다른 변환이 필요하지 않고 정적할당과 동적할당만 선택하면 된다.	- 해당 변수의 일관성(consistency) 유지 방법이 필요하다.
'private' 속성 변수	- 병렬처리 영역 내부에서 같은 이름의 변수를 새로 선언하는 방법을 사용한다.	- 병렬처리 영역 내부에서 같은 이름의 변수를 새로 선언하는 방법을 사용한다.

유 메모리의 특정한 부분을 통신용으로 지정하고 그곳에 각 프로세서들이 값을 읽고 쓰는 방법으로 프로세서 사이에 값을 전달한다.

3.2 병렬처리 디렉티브

'parallel'디렉티브는 병렬처리 영역을 나타내며 병렬처리를 하고 싶은 코드 블록(code block)에 '#pragma omp parallel'과 같은 디렉티브를 추가하는 방법으로 사용한다. 쓰레드를 지원하는 대칭형 다중처리 커널 운영체제를 사용하는 플랫폼이라면 기존의 대칭형 다중처리 기계에서 OpenMP를 구현하는 것과 마찬가지로 쓰레드 API를 이용하면 된다. 하지만 대칭형 다중처리 커널 운영체제가 없고 쓰레드를 지원하지 않는 플랫폼의 경우 여러 프로세서에서 프로그램을 수행하기 위해서는 일반적으로 주 프로세서(master processor)가 실행 파일 이미지를 나머지 프로세서에 적재(load)하고 수행하도록 해야 한다. 이 때 프로그램 수행이 끝날 때까지 모든 프로세서들이 전역변수를 공유하기 위해서는 프로그램 수행을 시작할 때 모든 프로세서가 실행을 시작하고 프로그램이 끝날 때 모두 같이 종료해야 한다. 이러한 동작은 원래 OpenMP 수행 모델과는 다르므로 프로그램을 시작할 때 모든 프로세서들이 같이 실행을 시작하고 프로그램이 끝날 때 같이 끝나도록 병렬처리 디렉티브의 처리를 바꿔야 한다. 먼저 원래 응용 프로그램의 main 함수를 app_main처럼 보통의 함수 이름으로 바꾼다. 다음으로 주 프로세서가 다른 프로세서를 실행시키는 사전 작업 코드와 app_main을 호출하는 코드를 포함하는 새로운 main함수를 작성하는 방법을 사용한다. 다른 병렬처리 플랫폼에서도 해당 플랫폼에서의 초기화 작업을 응용 프로그램의 시작 전에 처리하기 위해서 이러한 방식을 사용한다[6,7,9,12]. 모든 프로세서들이 프로그램을 시작하면 주 프로세서만 프로그램을 수행하고 다른 프로세서들은 주 프로세서가 병렬처리 영역에 도착할 때까지 기다린다. 주 프로세서가 병렬처리 영역에 도착하면 모든 프로세서는 병렬처리 작업을 나눠서 수행하고 다시 주 프로세서가 다음 병렬처리 영역에 도착할 때까지 나머지 프로세서들은 대기한다[3,6,12].

'parallel'디렉티브를 OpenMP 변환할 때 병렬처리 영역은 별도의 함수로 분리해내고 원래 위치에서는 분리한 함수를 호출하도록 원래 OpenMP프로그램을 변환한다. 쓰레드를 지원하지 않는 플랫폼에서 이러한 구조가 직접적인 장점은 없지만 쓰레드를 지원하는 플랫폼으로 OpenMP 변환기를 확장하기 유리한 장점이 있다. 또한 기존의 OpenMP 변환기들은 쓰레드를 지원하는 플랫폼을 위한 것이 많은데 이러한 변환기를 재사용할 수 있으므로 새로운 OpenMP 변환기를 제작에 드는 노력과 수고를 줄일 수 있는 장점이 있다.

CT3400의 경우도 모든 프로세서가 프로그램을 시작하기 위해서는 주 프로세서(master processor)가 실행 파일 이미지를 나머지 프로세서에 적재(load)하고 수행하도록 해야 한다. 그림 3은 CT3400을 위한 'parallel'디렉티브의 변환 예로 '#pragma omp parallel'로 지정한 블록 대신에 들어가는 코드를 간략하게 표현한 것이다. 위쪽은 'parallel'디렉티브가 있었던 코드 블록을 변환한 것이고 아래쪽은 병렬처리 영역을 분리해낸 함수이다. 'parallel'디렉티브가 있던 곳에서는 먼저 병렬처리 영역에서 접근하는 'shared'변수인 'sum'에 대한 정보를 해당 프로세서가 독점적으로 접근하는 온-칩 메모리에 주소로 넘기고, 병렬처리 영역을 수행하는 함수인 'thread_

['parallel'디렉티브 대체 구문]

```
thread_shared_0   _t_shared;            // 프로세서 안에서의 'shared'변수 구조체 선언
_t_shared._sh_sum = &sum;                // 'shared'변수 정보 저장
parallelize(thread_main_0, &_t_shared);  // 병렬처리영역 호출 및 정보 전달
```

독점적인 온-칩 메모리

[병렬처리 영역 수행 함수]

```
void thread_main_0(void *args) {          //  'shared' 변수 정보 획득
    thread_shared_0   *_t_shared = (thread_shared_0 *) data;
    node_shared_0   _n_shared;        // 프로세서들 사이의 'shared'변수 구조체 선언
    if( 주 프로세서인 경우  )
    { _n_shared.sum = _t_shared->_sh_sum; } // 주 프로세서가 'shared'변수 정보 저장
    bcast(&_n_shared, sizeof(node_shared_0));   // 프로세서간 'shared'변수 구조체 전달
    if( 주 프로세서가 아니라면 )
    { _t_shared->_sh_sum = _n_shared.sum; } // 프로세서간 'shared'변수 정보 획득
}
```

그림 3 CT3400을 위한 'parallel'디렉티브 변환의 예

main_0'는 온-칩 메모리로부터 정보를 읽어와서 수행한다. 이 때 'shared'변수 'sum'이 원래 단독변수로 선언되었다면 병렬처리 영역에 도착하기 전까지 프로그램을 수행한 것은 주 프로세서이므로 유효한 'shared'변수 값을 갖고 있는 것은 주 프로세서뿐이다. 따라서 다른 프로세서들이 프로그램을 제대로 수행하기 위해서는 주 프로세서로부터 유효한 'shared'변수 값을 넘겨받아야 하는데 이러한 작업은 주 프로세서가 다른 프로세서들에게 방송(broadcast)하는 것과 비슷한 동작을 하므로 그림 3에서는 'bcast'라는 이름의 함수로 추상화해서 표시했다. 이 과정이 끝나면 각 프로세서는 독점적인 온-칩 메모리에 유효한 'shared'변수 값을 갖게 되므로 원래의 병렬처리 영역을 수행할 수 있게 된다.

3.3 동기화 디렉티브

'critical', 'atomic', 'barrier' 디렉티브는 동기화와 연관 있는 작업을 수행하는 디렉티브로 '#pragma omp critical', '#pragma omp atomic', '#pragma omp barrier'와 같은 형식으로 사용한다. 'critical' 디렉티브를 붙인 코드 블록은 동시에 두 개 이상의 프로세서가 해당 코드 블록에 접근하는 것을 막고 특정 시점에는 오직 하나의 프로세서만 해당 코드 블록을 수행하도록 한다. 'atomic' 디렉티브는 'critical' 디렉티브와 동일하지만 코드 블록을 대상으로 하지 않고 특정한 형태의 연산 문(statement)만 사용할 수 있다. 'barrier' 디렉티브는 연관되는 코드 블록이나 연산 문(statement)은 없으며 모든 프로세서가 해당 디렉티브가 있는 지점에 도달할 때까지 기다리게 함으로써 프로세서 사이의 동기(synchronization)를 맞추는 역할을 한다. 이러한 동기화 작업은 'for'디렉티브나 'parallel'디렉티브의 마지막에도 묵시적으로 수행하는 것으로 OpenMP 명세는 규정하고 있다.

POSIX쓰레드를 지원하는 대칭형 다중처리 커널 운영체제를 사용하는 플랫폼의 경우 쓰레드 라이브러리에서 지원하는 함수들을 이용해서 동기화 디렉티브를 쉽게 구현할 수 있다. 하지만 쓰레드를 지원하는 대칭형 다중처리 커널 운영체제를 사용하지 않는 플랫폼의 경우라면 일반적으로 하드웨어 세마포어를 지원해서 시스템 내의 상호 배제적인 접근 문제를 해결한다. 'critical' 디렉티브와 'atomic' 디렉티브는 세마포어를 락(lock) 대신에 사용하는 것만으로 쉽게 구현이 가능하다. 하지만 'barrier' 디렉티브는 프로세서 사이에 신호(signal)를 주고 받는 API가 없으므로 다르게 구현해야 한다. 기본적으로는 배리어에 도착한 프로세서들의 숫자를 세는 공유변수를 사용하고 프로세서들이 배리어에 도착할 때마다 그 공유변수를 1씩 증가시켜서 모든 프로세서가 배리어에 도착했는지 확인한다. 그런데 각 프로세서들이 공유변수 값을 읽거나 쓸 때 모두 세마포어를 이용해서 상호 배제적인 접근을 한다면 그 부하가 매우 클 것이므로 공유변수 값을 수정하는 경우에만 세마포어를 통해서 접근하고 값을 읽는 경우에는 자유롭게 접근하도록 구현한다.

CT3400에서도 칩에 있는 하드웨어 세마포어를 사용해서 동기화 디렉티브를 구현한다. 배리어 구현은 앞에서 설명한 기본적인 배리어 알고리즘을 따르는데 그림 4(a)는 CT3400을 이용한 기존 연구[7,9]의 배리어(barrier) 구현이고 그림 4(b) 본 논문에서 제안하는 배리어 구현이다. 양쪽에서 공통적으로 'PES'는 전체 프로세서의 개수를 의미하며 'my_peid'는 프로세서마다 부여한 고유한 번호로 n개의 프로세서가 있을 경우 0부터 n-1까지의 번호를 갖는다. 'done_pe'는 현재까지 배리어 영역에 도착한 프로세서 숫자를 세는 공유변수다.

기존 연구에서 제시한 배리어 구현은 배리어가 끝날 때마다 'done_pe' 값을 초기화함으로써 하나의 공유 변수로 여러 배리어 연산에 사용할 수 있도록 했다. 하지만 그림 4(a)의 구현은 배리어를 제일 마지막으로 수행하는 프로세서의 'my_peid' 값이 0이 아닌 경우 문제가 발생한다. 마지막 프로세서가 락을 잡고 'done_pe++'를 수행하기 전까지 나머지 프로세서는 'while(done_pe < PES)' 반복 문 때문에 대기한다. 그리고 마지막 프로세서가 'done_pe++'를 수행하면 마지막 프로세서가 락을 놓기 전에 다른 프로세서들은 'while' 루프를 빠져 나올 수 있고 그 중 0번 프로세서는 'if'문에서 볼 수 있는 것처럼 'done_pe'를 초기화 하는 문장을 실행할 수 있다.

(a) 기존 연구의 배리어 구현

```
semaphore_lock(Sem.p);
done_pe++;
semaphoer_unlock(Sem.p);
while(done_pe<(PES))
      _pe_delay(1);
if(my_peid == 0)
      done_pe = 0;
```

(b) 제안하는 배리어 구현

```
semaphore_lock(Sem.p);
phase = (phase + 1) % 2;
if(done_pe[phase]+1 == PES)
      done_pe[(phase+1)%2] = 0;
done_pe[phase]++;
semaphoer_unlock(Sem.p);
while(done_pe[phase]<(PES))
      _pe_delay(1);
```

그림 4 CT3400에서의 배리어 디렉티브 구현과 변환의 예

병렬처리 플랫폼에서 프로세서간 명령어를 수행하는 순서는 보장할 수 없으므로 마지막 프로세서가 락을 놓고 'while(done_pe<PES)'를 평가하기 전에 'my_peid'가 0인 프로세서가 'done_pe'의 초기화 작업을 수행하는 것도 가능하다. 이렇게 되면 다른 프로세서들은 프로그램 수행을 계속 할 수 있지만 마지막으로 'done_pe'를 증가시켰던 프로세서는 'while'문을 못 빠져 나온다. 이런 상황이 항상 발생하지는 않지만 발생할 가능성이 있고 다중 프로세서의 동작은 비결정적(non-deterministic)이므로 문제가 된다. 문제의 원인은 마지막 프로세서가 'done_pe'를 읽기 전에 초기화 작업을 수행한 것이므로 그런 상황이 일어나지 않도록 그림 4(b)와 같은 구현을 우리가 제시한다. 먼저 각 프로세서는 0과 1의 값만을 번갈아 가질 수 있는 단독변수 'phase'를 이용해서 연속된 배리어들을 구분한다. 각 프로세서는 락을 잡을 때마다 현재 'phase'에 속해있는 'done_pe'를 증가시키고 제일 마지막 프로세서가 배리어에 도착하면 미리 다음 'phase'의 배리어에서 사용할 'done_pe'의 값을 0으로 초기화한다. 이 때 아직 현재 'phase'에 있는 'done_pe'를 증가시키지 않은 상황이므로 나머지 프로세서들은 다음 'phase'로 넘어가지 않는다. 그리고 이 마지막 프로세서가 현재 'phase'의 'done_pe'를 증가 시키면 모든 프로세서가 'while'문을 넘어 가면서 다음 'phase'의 배리어까지 진행한다. 이런 구현에서는 마지막 프로세서가 'while'문을 수행하기 전에 다른 프로세서들이 다음 배리어에 도착하더라도 'phase'가 다르므로 각각 올바르게 동작하는 것을 보장할 수 있다.

3.4 작업 분배 디렉티브

'for' 디렉티브는 작업 분배(work sharing) 디렉티브의 하나로 'for' 루프에 붙여서 지정한 'for' 루프의 반복회수(iteration)를 여러 프로세서에게 나눠주는 일을 한다. 이 때 'reduction'절로 지정한 변수는 프로세서마다 해당 변수에 대한 리덕션(reduction) 연산을 수행하고 최종적으로 각 프로세서의 변수 값을 하나로 합친다. 'for' 디렉티브에서 수행할 부분을 프로세서들에게 분배하는 작업은 운영체제 사용여부나 플랫폼에 상관없이 루프의 초기값과 종료 조건을 받아서 각 프로세서가 수행할 반복 회수에 따라 새로운 초기값과 종료 조건을 설정하는 방법으로 구현이 가능하다.

쓰레드를 지원하는 대칭형 다중처리 커널 운영체제를 사용하는 플랫폼에서 리덕션 연산을 처리하는 방법은 락을 잡고 공유변수인 리덕션 변수에 접근하는 것이다. 하지만 그러한 운영체제를 사용하지 않는 플랫폼에서는 리덕션 변수가 공유변수가 아닌 단독변수인 경우 다른 방법을 필요로 한다. 한 가지 방법은 리덕션 변수의 값을 대신 저장할 공유변수를 지정하는 방법으로 주로 대칭형 다중처리 기계에서 쓰레드를 사용할 때와 비슷한 방식이고 기존 연구[7,9]의 리덕션 구현과도 개념적으로 비슷하다. 각 프로세서는 락을 잡고 해당 공유변수에 리덕션 연산을 수행하고, 모든 프로세서들이 리덕션 연산을 마치면 해당 공유변수의 값을 각 프로세서의 독점적인 리덕션 단독변수에 복사해줌으로써 리덕션 변수 값의 일관성을 지킨다. 하지만 이 방식의 단점은 한 프로세서가 락을 잡고 리덕션 연산을 하는 동안 다른 프로세서들은 락을 잡기 위해서 기다려야 하므로 프로세서들의 수행이 직렬화 된다는 점이다. 다른 방법은 공유메모리에 프로세서마다 값을 저장할 위치를 지정하는 방식으로 두 단계로 구성된다. 우선 모든 프로세서는 각자의 리덕션 단독변수 값을 지정된 위치에 락을 잡지 않고 저장한 다음 배리어 연산을 수행해서 모든 프로세서가 값을 저장한 것을 확인한다. 다음으로 각 프로세서들이 공유 메모리로부터 다른 프로세서들의 리덕션 변수 값을 읽어서 최종적인 리덕션 결과값을 구한다. 이 방식은 주로 클러스터와 같은 분산메모리 플랫폼에서 메시지 전달 방식을 사용하는 것과 비슷한 방식으로 임시 저장 위치에 접근할 때에는 락을 잡을 필요가 없으므로 락을 잡는 시간이 최소화되고 리덕션 연산을 병렬적으로 수행할 수 있다는 장점이 있다.

그림 5는 그림 2의 예제 프로그램에서 '#pragma omp for'로 둘러싼 부분을 CT3400에 맞게 번역한 것으로 덧셈에 대해서 'sum'변수의 리덕션 연산 처리를 포함하고 있다. 먼저 'for' 루프의 반복 회수를 프로세서에 분배하면 각 프로세서가 수행하는 'for' 루프의 초기값과 종료조건은 서로 다를 것이므로 변수를 새로 선언해서 사용한다. 현재는 'static schedule'만 지원하므로 원래 'for' 루프의 초기값과 종료 조건을 균등하게 나눠준다. 또 리덕션 연산의 경우 초기값을 사용하고 나중에 병렬처리 영역에서 수정한 결과값을 병렬처리 영역이 끝난 다음에도 유지해야 하므로 이를 위한 변수들도 선언하고 초기화한다. 각 프로세서는 원래 'for' 디렉티브로 둘러쌌던 코드를 '_t_i'와 '_t_n'에 따라 나눠서 수행하고 모든 프로세서들이 동기를 맞추도록 배리어 함수를 수행하는데 만약 'for' 디렉티브에 리덕션 절이 있었다면 배리어 함수 이전에 리덕션 연산을 처리한다. 각 프로세서마다 리덕션 변수 정보 구조체인 'reduction_0'형식의 변수 '_t_red'에 현재 프로세서가 'for' 루프를 수행한 결과를 저장하고 'reduce'함수를 호출해서 리덕션 처리를 통해 하나의 최종 결과값으로 만든다. 여기서 'reduce_0'함수는 덧셈이나 곱셈 같은 리덕션 연산을 수행하는 함수다.

리덕션 처리를 하는 'reduce'함수는 앞에서 소개한 두 가지 방식으로 구현이 가능하다. 그림 6은 리덕션 공유변수를 사용하는 방식의 'reduce'함수 코드이다. 우선 각

```
int _t_i, _t_n, i;  // 루프 초기값, 종료 조건, 루프 변수
int *_rdl_sum = &(*(_t_shared->_sh_sum)); // 리덕션 결과를 돌려주기 위한 변수
int _rdf_sum = (*(_t_shared->_sh_sum));  // 리덕션 변수의 초기값
int sum = 0;
int _rd_sum = 0;
// _t_i , _t_n에 현재 프로세서를 위한 루프 변수 저장
for ( i = _t_i ; i < _t_n ; i++ ) {   // 실제 for 루프 수행
     sum += i;
}
_rd_sum = 0;               // 리덕션 임시 변수 초기화
_rd_sum = _rd_sum + sum;      // 현재 프로세서의 계산 결과 저장
{  reduction_0 _t_red;
    _t_red._rd_sum = _rd_sum;    // 프로세서간 'shared'변수에 결과 저장
   reduce(&_t_red, sizeof(reduction_0), reduce_0);  // 프로세서간 리덕션 수행
   *_rdl_sum = _rdf_sum + _t_red._rd_sum;    // 리덕션 결과를 돌려준다.
}
barrier();
```

그림 5 CT3400을 위한 'for' 디렉티브와 'reduction'절의 구현과 변환

```
void reduce(void *buf, int size, parade_reduce_function function) {
semaphore_lock(Sem.p);          // 각 프로세서는 락(lock)을 잡는다
phase = (phase + 1) % 2;      // 동기화를 위한 변수 처리
if( done_pe[phase] + 1 == PES )
     done_pe[(phase + 1) % 2] = 0;
if( done_pe[phase] == 0)   // 첫 번째 프로세서는 리덕션 공유변수 초기화
      memcpy(reduction_buffer, buf, size);
 else function(reduction_buffer, buf); // 나머지는 공유변수에 리덕션 연산 수행
 done_pe[phase]++;
 semaphore_unlock(Sem.p);       // 락을 푼다
 while(done_pe[phase] < PES)   // 동기화: 모든 프로세서가 끝날 때를 기다림
      _pe_delay(1);
 memcpy(buf, reduction_buffer, size); // 최종 리덕션 결과를 각 프로세서로 복사
}
```

그림 6 하나의 리덕션 공유변수를 사용하는 방식에서의 'reduce'함수

프로세서는 세마포어를 이용해서 락을 잡고 리덕션 결과값을 저장할 리덕션 공유 변수 'reduction_buffer'값을 독점적으로 갱신한다. 제일 처음 락을 잡은 프로세서는 자신의 리덕션 결과값을 리덕션 공유변수 값에 저장해서 초기화 하고, 그 외의 프로세서들은 락을 잡으면 리덕션 공유변수에 자기 결과값을 'function'함수를 이용해서 적용한다. 모든 프로세서가 이 작업을 마치면 리덕션 공유변수는 최종적인 리덕션 연산의 결과값을 갖게 되므로 각 프로세서는 최종 결과값을 자기가 갖고 있는 지역 변수에 복사해서 사용할 수 있게 한다.

다음으로 그림 7은 각 프로세서마다 공유 메모리에 임시 저장 위치를 두고 메시지 큐(message queue)처럼 사용하는 방식의 'reduce'함수 코드이다. 'reduction_buffer'라는 임시 저장 위치로 'char'형식의 4KB공간을 프로세서 개수만큼 공유 메모리에 할당하고 프로세서 번호를 이용해서 각 프로세서의 저장 위치에 접근하도록 했다. 각 프로세서는 'memcpy'를 이용해서 자신의 리덕션 결과값을 임시 저장 위치에 복사한다. 모든 프로세서가 복사를 마쳤는지 배리어 연산을 이용해서 확인한 후 임시 저장 위치에서 다른 프로세서들의 리덕션 결과값들을 읽어 최종 리덕션 결과값을 계산한다.

4. 실험 및 결과

제안한 OpenMP변환 및 구현을 OpenMP 변환기를 이용해서 예제 프로그램을 변환하고 단일 칩 다중 프로세서 보드인 CT3400에서 실행했다. 계산 작업과 프로세서 개수를 늘려가면서 제시한 OpenMP 구현과 변환 방법의 성능을 분석했고 각 OpenMP 디렉티브의 부하(overhead)가 얼마나 되는지 벤치마크를 이용해서 확인했다.

4.1 실험 환경 및 방법

본 논문에서 제시한 OpenMP 구현과 변환을 실험하기 위해서는 CT3400을 위한 OpenMP 변환기가 필요한

```
void reduce(void *buf, int size, parade_reduce_function function) {
int i;
memcpy(reduction_buffer[my_peid], buf, size); // 자기 결과 값을 임시 저장 위치에 복사
semaphore_lock(Sem.p);           // 각 프로세서는 락(lock)을 잡는다
phase = (phase + 1) % 2;      // 동기화를 위한 변수 처리
if( done_pe[phase] + 1 == PES )
      done_pe[(phase + 1) % 2] = 0;
 done_pe[phase]++;
 semaphore_unlock(Sem.p);        // 락을 푼다
 while(done_pe[phase] < PES)   // 동기화: 모든 프로세서가 끝날 때를 기다림
      _pe_delay(1);
for (i = 0 ; i < PES ; i++ )   // 모든 프로세서에 대해서
     if( PES != i )   // 자신을 제외한 나머지 프로세서의
          function(buf, reduction_buffer[i]); // 결과 값들을 자기의 결과 값을 저장한
}                                          //  단독변수에 리덕션 연산한다.
```

그림 7 임시 저장 위치를 메시지 큐처럼 사용하는 방식에서의 'reduce' 함수

데 OpenMP 프로그램을 입력으로 받고 앞 장에서 소개한 OpenMP 구현과 변환에 맞춰 CT3400 플랫폼에서 제공하는 세마포어 API 등을 이용한 병렬 프로그램으로 변환하는 작업을 수행한다. Omni C 컴파일러[5]에서 제공하는 C-front프로그램은 C프로그램을 자체적인 파스 트리(parse tree)로 변환하고 그 파스 트리로부터 C프로그램을 생성하는 코드 생성기 부분을 갖고 있으므로 파서와 코드 생성기의 용도로 사용할 수 있다. 원래의 C-front는 입력으로 받은 C파일을 그대로 출력하는 정도만 구현되어 있지만 우리는 우리가 원하는 코드를 생성하도록 이 프로그램을 수정해서 사용함으로써 변환기 제작에 드는 노력과 수고를 줄였다.

다음으로 Inspector는 Cradle Technologies, Inc.에서 제작한 CT3400보드의 시뮬레이터 프로그램으로 사이클 단위의 정확도(cycle accurate)를 갖는다[7,9]. 보드에서 실험해도 되지만 성능 측정을 사이클 단위로 정확하게 할 수 있고 디버깅 등 각종 편의를 위해서 기존 연구들과 마찬가지로 시뮬레이터에서 실험하였다.

실험에 사용한 예제는 앞에서 소개한 0에서 9999까지 더하는 프로그램을 수정한 프로그램과 CT3400 개발 도구에 들어있는 N차 정방 행렬의 곱셈 프로그램이다. 단일 칩 다중 프로세서에서 OpenMP를 사용하는 경우 아직 표준화된 벤치마크 프로그램이 없고 연구도 부족하므로 대칭형 다중처리 기계나 클러스터를 위한 벤치마크 프로그램을 사용하거나 행렬 곱셈 프로그램을 사용하는 경우가 있다[7,9]. 개발사에서 제공하는 행렬 곱셈 프로그램은 N차 정방 행렬 A와 B를 곱해서 행렬 C에 저장하는 간단한 프로그램이만 N의 크기를 조정함으로써 계산량을 조절하기 쉽고 기존 연구[7,9]에서 사용한 프로그램이므로 성능 비교가 가능하다는 장점이 있다. 프로세서 한 개를 사용하는 행렬 곱셈 프로그램과 프로세서 여러 개를 사용하도록 손으로 작성한 행렬 곱셈 프로그램을 제공하는데 프로세서 한 개를 사용하는 프로그램에 OpenMP 디렉티브를 추가해서 실험 대상 프로그램으로 사용했다. 행렬 곱셈은 리덕션 연산을 사용하지 않으므로 'reduction'을 사용하는 예제 프로그램으로는 그림 2의 0에서 9999까지 더하는 프로그램을 사용했다. 다만 전역변수의 영향도 같이 실험할 수 있도록 'int data[10000]'이나 'int *data'처럼 전역배열을 선언하고 'sum += i'대신에 'sum += data[i]'를 사용하도록 수정했다.

EPCC[13]는 여러 플랫폼에서 OpenMP 디렉티브의 부하를 측정하는 벤치마크로 단일 칩 다중 프로세서에서 디렉티브의 부하를 측정하는 벤치마크가 없으므로 시간 측정단위를 초 단위에서 사이클 단위로 수정해서 사용했다. 기존의 배리어 구현은 기존 논문[7,9]에 나온 코드를 그대로 구현하고 비교했다.

4.2 실험결과

표 2는 CT3400에서 24×24 행렬 곱셈을 수행한 결과이다. 행렬 곱셈은 전역변수를 정적 할당한 효과를 본

표 2 24×24 행렬 곱셈의 수행 시간 (단위: 사이클)

직렬 (오프-칩) (정적할당)	병렬 (오프-칩) (정적할당)	OpenMP (오프-칩)		OpenMP (온-칩)	
		동적할당	정적할당	동적할당	정적할당
3,664,513	3,653,761	5,221,225	3,674,336	1,081,775	847,907
해당 없음	1,827,537	2,622,901	1,845,050	549,432	431,722
해당 없음	914,127	1,320,474	933,549	286,282	226,760

경우로 모든 실험에서 프로세서 개수 증가에 따라 프로세서 개수에 반비례하게 수행 시간이 줄어 든다.

'직렬'과 '병렬'은 OpenMP를 사용하지 않고 손으로 작성한 프로그램으로 CT3400 개발 도구에서 제공한 프로그램이다. 우리가 제작한 OpenMP 변환기를 이용해서 변환한 프로그램은 'OpenMP'로 표기한다. 곱셈에 사용하는 행렬들은 공유 메모리 중에서 온-칩 메모리에 두는지 오프-칩 메모리에 두는지에 따라 '온-칩'과 '오프-칩'으로 구분한다. 또한 앞에서 설명한 것과 같이 전역변수의 메모리를 할당방법에 따라 '정적할당'과 '동적할당'으로 표기한다. 'OpenMP(정적할당,오프-칩)'의 경우 손으로 작성한 '직렬'이나 '병렬'과 비슷한 성능을 보이지만 'OpenMP(동적할당,오프-칩)'의 경우 '직렬'이나 'OpenMP(정적할당,오프-칩)'에 비해서 약 42% 정도의 시간이 더 걸리는 것을 볼 수 있다. 표에는 없지만 '직렬'에서 전역변수를 동적할당하면 5,207,562 사이클이 걸려서 OpenMP의 '동적할당,오프-칩'과 비슷한 성능 저하를 확인할 수 있었다. 즉 전역변수의 동적할당은 정적할당에 비해서 성능 저하를 가져올 수 있는데 이것은 앞에서 설명한대로 'cragcc' 컴파일러가 전역 데이타 영역에 위치한 변수들을 효율적으로 처리할 수 있기 때문에 나타난 현상이다.

'OpenMP (동적할당,온-칩)'과 'OpenMP (정적할당,온-칩)'은 둘 다 'OpenMP(동적할당,오프-칩)'과 'OpenMP (정적할당,오프-칩)'보다 성능이 훨씬 좋고 '정적할당'이 '동적할당'보다 성능이 좋다. 즉 전역변수를 정적으로 할당한 것은 해당 변수를 물리적으로 더 빠른 공유 메모리에 할당해서 성능 향상을 얻는 것이 아니라 같은 공유 메모리를 사용하면서도 전역 데이타 영역에 위치한 변수를 컴파일러가 효율적으로 처리할 수 있게 함으로써 성능 향상을 가능케 하는 것이다.

아래의 표 3, 표 4, 표 5, 표 6, 표 7은 각각 행렬의 크기를 4, 8, 16, 32, 64로 늘려가면서 행렬 곱셈 수행 시간을 측정한 결과다. 이 실험들은 모두 전역변수를 오프-칩 메모리에 두고 실험한 결과다. 물론 온-칩 메모리를 사용하는 것이 오프-칩 메모리를 사용하는 것에 비해서 성능이 훨씬 좋지만[7,9], 온-칩 메모리의 크기는 한계가 있고 여기서는 온-칩 메모리와 오프-칩 메모리

표 3 4×4 행렬 곱셈의 수행 시간 (단위: 사이클)

프로세서	직렬	병렬	OpenMP, 동적할당	OpenMP, 정적할당
1개	20,082	20,836	38,291	41,314
2개	해당 없음	11,270	31,120	32,756
4개	해당 없음	6,437	25,413	29,233

표 4 8×8 행렬 곱셈의 수행 시간 (단위: 사이클)

프로세서	직렬	병렬	OpenMP, 동적할당	OpenMP, 정적할당
1개	143,378	144,552	208,399	164,922
2개	해당 없음	73,128	116,330	92,578
4개	해당 없음	37,367	68,174	56,983

표 5 16×16 행렬 곱셈의 수행 시간 (단위: 사이클)

프로세서	직렬	병렬	OpenMP, 동적할당	OpenMP, 정적할당
1개	1,096,818	1,101,424	1,564,136	1,118,376
2개	해당 없음	551,564	794,121	566,935
4개	해당 없음	276,286	409,248	294,333

표 6 32×32 행렬 곱셈의 수행 시간 (단위: 사이클)

프로세서	직렬	병렬	OpenMP, 동적할당	OpenMP, 정적할당
1개	8,628,210	8,627,900	12,373,765	8,650,047
2개	해당 없음	4,314,852	6,198,865	4,332,926
4개	해당 없음	2,157,514	3,108,500	2,177,484

표 7 64×64 행렬 곱셈의 수행 시간 (단위: 사이클)

프로세서	직렬	병렬	OpenMP, 동적할당	OpenMP, 정적할당
1개	68,324,530	68,321,276	98,731,357	68,346,647
2개	해당 없음	34,161,540	49,375,560	34,181,382
4개	해당 없음	17,080,974	24,698,132	17,101868

표 8 주요 OpenMP 디렉티브의 부하(overhead) (단위: 사이클)

	프로세서 1개	프로세서 2개	프로세서 4개
기존 구현의 'barrier'	1,136	1,698	2,848
새로운 'barrier' 구현	1,404	2,128	3,581
'parallel' 디렉티브	1,645	5,094	8,109
'for' 디렉티브	7,340	8,933	12,039
'reduction'절, '공유변수'	1,713	8,790	14,028
'reduction'절, '메시지'	1,713	7,805	12,631

의 성능 비교가 초점이 아니기 때문이다.

계산량이 너무 작아서 계산 시간이 OpenMP 디렉티브를 수행하는 부하(overhead)와 비슷한 4×4에서의 OpenMP 실험을 제외하고는 모든 실험에서 프로세서 개수에 거의 반비례하게 수행 시간이 줄어드는 것을 볼 수 있었고 '정적할당'이 '동적할당'에 비해서 약 21%~31% 정도 성능이 좋으며 '병렬'과 비교해서는 약 0.04%~52% 정도의 성능 차이를 보였다.

표 8은 EPCC 벤치마크를 이용해서 주요 OpenMP 디렉티브의 부하를 측정한 결과이다. 행렬 곱셈 프로그램에서는 'barrier'가 2회 'parallel', 'for' 디렉티브는 각 1회씩 수행되는데 이 표에서 볼 수 있듯이 'parallel'과 'for'의 부하를 더하고 'barrier' 2회의 부하를 더하면 약 20,000 사이클 정도의 부하가 있음을 알 수 있다. OpenMP를 수행하기 위해서 발생한 이 20,000사이클 정도의 부하는 모든 실험에서 '병렬'과 '정적할당' 사이에 있는 약 20,000사이클 정도의 차이의 원인이며 행렬의 크기나 프로세서의 개수에 상관 없이 항상 일정하다. 이 부하가 거의 상수 값이므로 행렬의 크기가 커질수록 전체 수행 시간에서 부하가 차지하는 비중이 줄어들게 된다. 따라서 '동적할당'에 대한 '정적할당'의 성능 향상 정도가 커지고 '병렬'과의 차이는 감소한다. 다음으로 기존 'barrier' 구현과 논문에서 제시한 'barrier' 구현을 구분하여 측정했다. 새로운 'barrier'구현은 성능 개선이 아니라 문제를 해결하기 위해 필요한 것으로 기존의 잘못된 구현보다 200~700사이클 정도의 추가적인 부하가 필요하지만 다른 디렉티브에 비해서는 부하가 작은 것을 볼 수 있다. 마지막으로 'reduction' 절의 두 가지 방식을 구현하고 실험한 결과 임시 저장 위치를 메시지 큐처럼 사용하는 방식('메시지')이 하나의 리덕션 공유변수를 사용하는 방식('공유변수')보다 약 10%정도 적은 부하를 보였다. 프로세서 2개에서는 약 1000사이클, 프로세서 4개에서는 약 1400사이클 정도 감소했다.

표 9는 0에서 9999까지 더하는 프로그램을 전역변수를 이용하도록 수정한 프로그램의 실험결과다. 전역배열에 0부터 9999까지의 값을 저장하는 초기화 작업은 직렬로 수행하고 덧셈만 병렬적으로 수행하는 구조로 되어있다. 직렬 수행 부분인 350,000사이클 정도는 프로세서 개수에 상관없이 고정적이고 병렬 수행 부분만 프로세서 개수의 증가에 따라 반비례하게 줄어든다. 전역변수의 메모리 할당방법에 따라 '정적할당'과 '동적할당'으로 표현하고 리덕션 구현 방법에 따라 '공유변수'와 '메시지'로 표현했다. 정적할당의 경우 오히려 동적할당에 비해서 2%정도 느린데 OpenMP 변환기는 전역변수를 정적으로 할당함으로써 성능 향상을 위한 힌트를 컴파일러에게 줬지만 컴파일러가 성능 향상을 하지 못했기 때문이다. 다음으로 프로세서가 2개나 4개인 경우 '메시지' 방식의 리덕션 구현이 '공유변수' 방식의 리덕션 구현보다 1,000~2,700사이클 정도 수행시간이 줄었다. 이 프로그램은 'reduction' 절을 한 번만 사용하기 때문에 리덕션 구현 방식을 바꿈으로써 얻는 이익이 있기는 하지만 크게 나타나지 않았다.

5. 관련 연구

표 10은 단일 칩 다중 프로세서에서 OpenMP를 사용하기 위한 기존 연구들과 본 논문의 비교이다. Yoshihiko et al.[8]은 프로세서 두 개가 있는 단일 칩 다중 프로세서 보드에 대칭형 다중처리를 지원하는 리눅스 커널을 올리고 POSIX 쓰레드를 이용해서 OpenMP를 사용할 수 있게 했다. Omni 컴파일러를 재사용해서 OpenMP변환기를 제작했는데 내장형(embedded) 시스템에서 전력 소비를 줄이는 최적화를 고려했지만 일반적인 대칭형 다중처리 기계와 동일한 환경에서의 OpenMP 구현과 변환이므로 운영체제가 없는 상황을

표 9 전역변수를 이용해서 0부터 9999까지의 합을 구하는 OpenMP 프로그램의 수행 시간 (단위: 사이클)

프로세서	동적 할당, 공유변수	정적 할당, 공유변수	동적 할당, 메시지	정적 할당, 메시지
1개	1,678,840	1,706,669	1,678,740	1,707,169
2개	1,014,402	1,028,935	1,013,134	1,027,983
4개	680,179	694,081	677,409	692,457

표 10 단일 칩 다중 프로세서에서 OpenMP를 사용하기 위한 연구들의 비교

	본 논문	Feng Liu et al.	Yoshihiko et al.
사용한 보드 (제조사 / 모델)	Cradle Technologies, Inc. CT3400	Cradle Technologies, Inc. CT3400	Renesas Technology. M3T-M32700UT
운영체제	사용하지 않음	사용하지 않음	Linux M32/R
쓰레드 라이브러리	지원하지 않음	지원하지 않음	POSIX 쓰레드
OpenMP 디렉티브 확장 여부	확장하지 않음	임의적인 확장	확장하지 않음
특수한 하드웨어	사용하지 않음	성능 향상을 위해 사용	사용하지 않음

위한 연구에는 도움이 되지 않는다.

Feng Liu et al.[7][9]는 우리가 연구에 사용한 CT3400 보드에서 DSP를 고려하여 OpenMP를 구현했다. 이들은 프로세서 이외에 DSP나 MTE(memory transfer engine)와 같이 보드에 있는 특수한 하드웨어를 이용하는 OpenMP 변환이나 OpenMP 디렉티브의 확장에 대해서 주로 제시하고 있다. 반면에 일반적인 단일 칩 다중 프로세서에서 OpenMP 디렉티브를 확장하지 않고 사용할 수 있는 OpenMP 변환과 구현에 대한 부분은 부족하다. 우리의 연구는 운영체제를 사용하지 않는 플랫폼에서 기존의 배리어 구현에 있는 문제 가능성을 없애고, 특수한 하드웨어나 OpenMP 디렉티브의 임의적인 확장 없이 프로그램의 성능을 향상시킬 수 있는 기법을 제시했다는 점에서 이들과 다르다.

6. 결 론

이 논문에서는 쓰레드를 지원하는 대칭형 다중처리 커널 운영체제를 사용하지 않고 물리적인 공유 메모리와 하드웨어 세마포어를 갖고 있는 단일 칩 다중 프로세서 보드에서의 OpenMP 구현과 효과적인 변환을 제시하고 있다. 특히 우리가 제안한 OpenMP 구현과 변환을 바탕으로 CT3400 단일 칩 다중 프로세서 보드를 위한 OpenMP구현과 OpenMP 변환기를 제작하고 단일 칩 다중 프로세서 보드 시뮬레이터에서 실험을 수행함으로써 주요 OpenMP 디렉티브가 제대로 동작하는 것을 확인했다.

먼저 공유 메모리에 전역변수를 선언할 때 정적으로 메모리를 할당하도록 OpenMP 변환을 함으로써 컴파일러에게 해당 변수가 전역 변수라는 정보를 알려주도록 했다. 이 정보를 이용할 수 있는 컴파일러는 계산 작업의 크기가 OpenMP 수행을 위한 부하보다 어느 정도 클 때 특정한 하드웨어나 OpenMP 디렉티브의 확장 없이도 성능을 향상시킬 수 있다. OpenMP 변환기가 준 정보를 이용해서 컴파일러가 효율적으로 동작한 프로그램의 경우 21%~31%정도의 성능 향상을 보였고, 효과를 보지 못한 프로그램의 경우에도 2%정도의 성능 저하를 겪을 뿐이라는 것을 실험을 통해서 확인했다. 또한 제시한 두 가지 리덕션 구현 방식 중에서는 임시 저장 위치를 메시지 큐처럼 사용하는 방식이 하나의 리덕션 공유변수를 사용하는 방식보다 좋다는 것을 벤치마크 프로그램을 통해서 확인했다. 마지막으로 잘못 동작할 가능성이 있었던 기존 동기화 디렉티브 구현을 대체할 새로운 동기화 디렉티브 구현을 제안하였다.

남은 과제는 번역기가 OpenMP 2.0[14]을 지원할 수 있도록 하는 것과 물리적으로 분산 메모리 구조를 갖는 단일 칩 다중 프로세서를 위한 OpenMP 구현과 변환에 대한 연구를 수행하는 것이다.

참 고 문 헌

[1] Message Passing Interface Forum, "MPI: A message-passing interface standard," International Journal of Supercomputer Applications and High Performance Computing, Vol.8, No.3/4, pp.159-416, 1994.

[2] OpenMP Architecture Review Board, "OpenMP C and C++ Application Program Interface," http://www.openmp.org, Version 1.0, Oct. 1998.

[3] Christian Brunschen and Mats Brorsson, "OdinMP/CCp - A portable implementation of OpenMP for C," (EWOMP'99) In the proceeding of first European Workshop on OpenMP, Lund, Sweden, Sept. 1999, pp.21-26.

[4] Vassilios V. Dimakopoulos and Elias Leontiadis, "A portable C compiler for OpenMP V.2.0," EWOMP 2003, In the proceeding of 5th European Workshop on OpenMP, Aachen, Germany, Sept. 2003, pp.5-11.

[5] Mitsuhisa Sato, Shigehisa Satoh, Kazuhiro Kusano, and Yoshio Tanaka, "Design of OpenMP Compiler for an SMP Cluster," EWOMP'99, In the proceeding of 1st European Workshop on OpenMP, Lund, Sweden, Sept. 1999, pp.32-39.

[6] Yang-Suk Kee, Jin-Soo Kim, and Soonhoi Ha, "ParADE: An OpenMP Programming Environment for SMP Cluster Systems," ACM/IEEE Supercomputing (SC'03), Nov 12-15, 2003.

[7] Feng Liu and Vipin Chaudhary, "A Practical OpenMP Compiler for System on Chips," WOMPAT 2003, LNCS 2716, pp. 54-68, 2003.

[8] Yoshihiko Hotta, Mitsuhisa Sato, Yoshihiro Nakajima, Yoshinori Ojima, "OpenMP Implemen-

tation and Performance on Embedded Renesas M32R Chip Multiprocessor," EWOMP 2004, Stockholm, Sweden, Oct. 2004, pp. 37-42.
[9] Feng Liu and Vipin Chudhary, "Extending OpenMP for Heterogeneous Chip Multiprocessors," Proceedings of the 2003 International Conference on Parallel Processing, Kaohsiung, Taiwan, Oct. 2003, pp.161-
[10] Cradle Technologies, Inc., CT3400 Multi-core DSP datasheet, http://www.cradle.com, 2004.
[11] Cradle Technologies,Inc., CRAGCC Compiler Addendum, http://www.cradle.com, 2004.
[12] 강신육, "SMP 클러스터를 위한 OpenMP Translator 구현", 서울대학교 대학원 전기컴퓨터공학부 공학석사 학위논문, 2003.
[13] EPCC OpenMP Microbenchmarks 1.0, http://www.epcc.ed.ac.uk/research/openmpbench, 1999.
[14] OpenMP Architecture Review Board, "OpenMP C and C++ Application Program Interface," http://www.openmp.org, Version 2.0, Mar. 2002.

전 우 철

1999년 2월 서울대학교 컴퓨터공학과 학사. 2001년 2월 서울대학교 전기컴퓨터공학부 석사. ~~2001년 3월~현재 서울대학교 전기컴퓨터공학부 박사과정~~ 관심분야는 병렬처리, 리눅스 클러스터링, 병렬 프로그래밍, OpenMP, 다중 프로세서 시스템-온-칩 프로그래밍 환경

하 순 회

1985년 2월 서울대학교 전자공학과 학사 1987년 2월 서울대학교 전자공학과 석사 1992년 미국 UCB 전기컴퓨터공학과 박사. 1993년~1994년 현대전자 근무. 1994년~현재 서울대학교 컴퓨터공학부 교수 관심분야는 하드웨어-소프트웨어 통합설계, 내장형 시스템을 위한 설계 방법론, MPSoC 내장형 소프트웨어 설계