병렬 프로그래밍 언어 CC++

변석우, 우영춘, 지동해
컴퓨터.소프트웨어기술연구소
한국전자통신연구원

요 약

향후 다중 프로세서와 공유 메모리 구조를 갖는 고성능 컴퓨터가 점차 널리 사용될 것으로 예상한다. 따라서 병렬 프로그래밍에 대한 요구가 증대될 것이며, 그 적용 대상 또한 수치 계산 분야에 국한되지 않고 널리 확장될 것이라고 보여진다. 이런 상황에서 '제어 병렬성'에 기반한 병렬 언어 CC+에 대해서 주목할 필요가 있다고 생각한다. 본 고에서는 CC++의 주요 병렬 구문들에 대해서 소개하고, 병렬 프로그래밍의 향후 전망에 대하여 논의한다.

제 1 절 배 경

갈수록 병렬 구조를 갖는 컴퓨터가 일반화되고 있다. 최근 Intel에서는 4개의 CPU를 장착한 Dechutes 보드를 발표하였으며, 올 해 안에 이 보드를 이용한 4-way와 8-way 구조의 병렬 컴퓨터가 시장에 등장할 것으로 예상한다. 또한 대부분의 대형 컴퓨터 업체들은 캐쉬 코히어런스 기술을 이용한 cc-Numa 구조의 컴퓨터들을 계속 발표하고 있다 [정박윤98]. 그동안 대형 병렬 머신의 대부분은 노드사이의 메모리를 공유할 수 없는 NORMA (NO Remote Memory Aceess) 형태의 구조로 구성되어 있었으나, 업계의 동향을 고려할 때 가까운 장래에 PC 서버로부터 고성능 대형 머신에 이르기까지 시스템 구조의 주류는 논리적으로 단일 주소 공간을 제공하는 SMP(Symmetric Multi-Processing)가 될 것으로 예상한다. 프로그램 개발자들에게 이것은 매우 반가운 전망이다. SMP 구조에 기반하여 개발된 프로그램들은 하드웨어 시스템의 사양에 큰 영향없이 여러 병렬 머신에 쉽게 탑재될 수 있을 것이다.

공유 메모리 구조의 병렬 컴퓨터에서는 쓰레드 프로그래밍 기법이 적합하다. 이미 Unix 운영 체제에서는 쓰레드 프로그래밍 환경을 제공하고 있으며, 대부분의 프로그래밍 언어들이 이 환경을 이용하여 프로그래밍 할 수 있는 환경을 제공하고 있다. 그러나 병렬 프로그래밍을 운영체제의 쓰레드 환경에 의존하기보다는 병렬성을 적합하게 표현할 수 있는 병렬 프로그래밍 언어의 필요성이 제시되고 있다. 즉, 병렬성을 프로그래밍의 '환경'이 아닌 정식 구문으로서 정의하는 것이 필요하다고 생각한다. 이런 방식을 사용함으로써 다음과 같은 효과를 얻을 수 있다.

- 프로그램의 병렬성을 쓰레드보다도 더 높은 수준에서 표현함으로써 하부 구조의 구체적인 사항들을 기술할 필요가 없으며, 하부 구조의 변화에 상관없이 프로그래머에게 일관된 호환성을 제공할 수 있다.

- 프로그램 언어의 구문들은 컴파일러 과정에서 의미 분석을 거치게 되며, 이때 여러 검증 및 변환 기법을 적용할 수 있다. 예를 들어, 하드웨어에 장착된 CPU의 수는 상황에 따라 변화할 수 있으므로, 병렬 프로그램의 단위 크기 (granulity) 또한 상황에 따라 적절히 바뀌는 것이 바람직하다. 사용가능한 CPU 수를 고려하여 컴파일 시에 프로그램을 그에 최적화된 단위 크기를 갖도록 변형할 수 있다. 또한 제한적이기는 하지만 병렬 프로그램의 결정성을 컴파일 시에 검증할 수 있다.

- 순차 프로그래밍보다 병렬 프로그래밍에서의 프로그램 정확성(correctness)은 훨씬 더

어려워지게 된다. 병렬 알고리즘을 정확하고 간결하게 표현할 수 있는 논리적 구문은 프로그램의 논리적 추론과 검증을 위해서 필수적이다.

CC++ (Compositional C++)은 Caltech의 Chandy 교수팀에 의해서 설계 구현되고 있는 병렬 언어이다 [CC++Web]. C++ 언어에 병렬성을 표현하기 위한 여러 병렬 구문들이 첨가되었다. Chandy 교수는 병렬 프로그래밍 분야의 권위자로서 Texas 대학의 Misra 교수와 함께 연구용 병렬 언어 UNITY를 고안함으로써 높은 평가를 받고 있다 [CM88]. CC++의 병렬 구문은 이와 같은 기반을 바탕으로 설계되었으리라고 본다. 이 병렬 구문들의 일부는 C++의 객체와 밀접하게 연관되어 있으나 나머지 대부분은 그렇지 않으므로 다른 프로그래밍 언어에서도 쉽게 응용될 수 있다.

한편, ParaC는 ETRI의 주전산기 IV 시스템의 병렬 프로그래밍 환경을 제공하기 위해서 개발되었다 [김진미97]. ETRI에서 개발된 주전산기 IV의 하드웨어 구조를 형성하는 기본 구성 요소는 노드와 네트워크 스위치이다. 한 노드는 4개의 Pentium Pro 프로세서와 공유 메모리로서 구성되어 있으며, 다량의 노드들이 네트워크 스위치로서 연결된 클러스터링 시스템이다. 즉, 한 노드내에서는 메모리가 공유되지만, 노드 사이에는 메모리가 공유되지 않는 클러스터링 구조를 갖는다. CC++는 병렬 프로그래밍을 위한 통신 매개체로서 쓰레드 사이에 공유되는 변수와 *logical processor object*를 이용한 메지시 패싱을 모두 지원하므로 주전산기 IV의 클러스터링 구조에 적합한 병렬 언어로 판단되어 주전산기 IV에 탑재되었다. 그러한 병렬 C++ 언어 외에 병렬 C에 대한 필요성이 제시됨에 따라, ETRI에서는 CC++의 병렬 구문과 유사한 병렬 구문들을 C 언어에 첨가한 ParaC를 개발하였다. C++와는 달리 C에서는 객체에 대한 개념이 없으므로 *logical processor object*와 같은 개념을 이용한 메시지 패싱을 이용할 수는 없었다. 대신 remote procedure call과 유사한 기능을 하는 원격 타스크 (remote task)의 기능이 구현되었다 [우영춘97]. 결과적으로 공유 메모리용 병렬 구문에 있어서 CC++와 ParaC는 매우 유사하지만, 분산 메모리의 경우에 두 언어는 매우 다르다.

본 고에서는 CC++의 병렬 구문들과 구현 방법에 대해서 소개하고, SMP 머신을 위한 병렬 프로그래밍 환경의 향후 전망에 대해서 논의한다.

제 2 절 CC++의 병렬 구문들

CC++는 C++에 병렬 구문을 첨가함으로써 다음과 같은 새로운 기능들을 갖게 되었다 [CK93] [CC++Tut].

- 제어의 병렬화 - *par, parfor, spawn*
- 동기화 및 통신 - *sync*
- 비결정성 (nondeterminism)
- 원자적 제어 흐름 (atomic control flow) - *atomic*
- heterogeinty - *global*

1 *par*

가장 기본적인 병렬 구문으로서, *par* 내에 있는 각 문장 혹은 블럭 (statement or block)에 대해서 하나의 쓰레드가 생성되어 이 쓰레드들이 interleaving방식으로 수행된다. 이때, 블럭에는 다시 병렬 구문 들이 나타날 수 있다. 즉, 병렬 구문은 nested될 수 있다. *par* 문에 의해서 생기는 쓰레드의 수는 블럭의 수와 같으며 이것들은 컴파일시에 정적으로 결정된다.

```
par {
  { a1(); a2(); a3() }    // Statement S1
  { b1(); b2(); b3() }    // Statement S2
}
```

위의 주어진 C++ 프로그램에서 *par* 블럭들은 다음과 같은 순서로서 수행될 가능성이 있다.

```
a1()  a2()  a3()  b1()  b2()  b3()
a1()  b1()  b2()  b3()  a2()  a3()
b1()  a1()  a2()  b2()  a3()  b3()
⋮                       ⋮
```

*par*문의 수행이 종료되는 시기는 *par* 문내에서 생성된 모든 쓰레드들이 수행을 완료하여 종료될 때이며, 그 중에 어떤 한 블럭이라도 종료되지 않을 때는 *par* 문은 종료되지 않는다. 한 *par* 안에 있는 블럭들 사이에서의 통신은 공유 변수들을 사용함으로써 가능하다.

2 *parfor*

*parfor*는 *par*와 같이 고정된 수의 쓰레드를 컴파일할 때 결정하려는 경우 매우 유용하다. *parfor* 문의 문법구조는 *for* 대신에 *parfor*를 사용하는 것 이외에는 C의 *for* 문과 같다. 이 문장은 초기화, 종료, 인수(index) 수정 및 반복 수행문 (loop body)으로서 구성되어 있다. 반복 수행문은 순차적이거나 병렬적으로 구성될 수 있다. 그 예는 다음과 같다.

```
parfor (int index = 0 ; index < N ; index++) {
  a1(index);
  a2(index+1);
  a3(index);
}
```

parfor 문이 한 번 반복 수행될 때 마다 새로운 쓰레드가 생성되는 데, 이 새로 생성된 쓰레드는 이미 생성된 쓰레드와 함께 동시에 수행된다. 생성된 쓰레드들은 *par*의 쓰레드들 처럼 interleaving 방식에 의해서 처리된다. *parfor*문은 모든 반복 수행문들이 종료될 때 종료된다. 위의 예에서 처럼, index 변수는 *parfor*내의 블럭들에 나타날 수 있다. 반복 수행문에 나타나는 index 변수의 값은 각 쓰레드 생성 당시의 초기화된 값이다.

3 *spawn*

CC++에서 병렬 수행을 표현하는 세 번째 방법으로서 *spawn*이 있다. *spawn*은 그 다음에 오는 문장을 수행하기 위한 쓰레드를 하나 새로 만든다. 그 결과 두 개의 쓰레드가 존재하는 데, *spawn*에 의해 새로 만들어진 쓰레드에서는 그 *spawn* 문의 처음을 수행하며, 원래의 쓰레드에서는 *spawn*문 다음에 오는 문장을 수행한다. 이렇게 생성된 두 쓰레드들은 *par*나 *parfor*와 같이 interleaving 방식에 의해서 수행된다. 그러나, *par*나 *parfor* 와는 달리 spawn 에서는 종료될 때 join에 해당되는 barrier의 개념이 존재하지 않는다. 또한 *spawn*에 의해서 생성된 두 쓰레드 중 어느 하나가 종료되는 것은 나머지 다른 하나의 종료와는 무관하다. *spawn*에 의해서 새로 생성된 쓰레드는 그 문장의 수행이 완료됨과 동시에 종료되며 나머지 다른 쓰레드의 종료에는 아무런 영향을 주지 않는다.

*spawn*이 사용되는 분야 중에 하나로서, 네트워크 서버가 들어오는 통신을 처리하는 방법에서 사용될 수 있다. 한 네트워크의 연결은 다른 네트워크의 연결의 종료에 영향을 받지 않는다.

4 동기화 및 통신

CC++의 동기화 및 통신은 공유 변수를 기반으로 하고 있다. CC++의 *const* 변수들은 동시성의 동기화 기법에서 볼 때 매우 시사하는 바가 많다. 일단 *const* 객체로서 생성된 값은 그 객체가 살아 있는 한 변하지 않고 끝까지 유지된다. *const* 변수에서는 write race가 발생하지 않으므로 모든 쓰레드는 그 변수를 제약없이 여러 개의 프로세서가 일관성에 대한 염려없이 언제나 사용할 수 있다.

그러나 *const* 객체의 초기화가 그 것이 생성될 때 이루져야 한다는 것은 병렬 프로그래밍의 통신 기법으로서는 너무 큰 제약이다. CC++에서는 좀 더 유연성있는 통신 구조를 제시하기 위하여 *sync*라는 타입 수정문 (type modifier)을 이용한다. CC++의 *sync* 변수는 C++의 *const* 변수와 다음의 두 가지를 제외하고는 같다.

- *sync* 객체의 초기값은 그 것이 생성될 때 정해질 필요가 없고 추후 프로그램 수행시 정의될 수 있다. *sync* 객체의 초기화는 그것에 값을 할당 (assign) 함으로써 이루어 진다.
- 초기화 되지 않은 값을 읽으려고 하면 *sync* 객체가 초기화될 때 까지 대기 (wait)하여야 한다.

구현하는 입장에서 보면, 한 *sync* 객체가 여러 프로세서에 복사될 수 있다는 점에서 *sync* 객체는 *const* 객체와 비슷한 특성을 갖는다.

한 병렬 프로그램이 여러 프로그램 모듈로서 구성되어 있다고 가정하자. 한 모듈 M 이 다른 모듈 N과 동시에 병렬 수행될 때 M이 보유하고 있는 특성이 *par* {*M, N* } 에서도 유지될 수 있는 가의 문제가 제기될 수 있다. 간단히 말하여, M의 특성이 유지되지 않는 현상은 N이 M에게 어떤 영향 (혹은 간섭)을 주기때문에 발생한다. CC++에서 쓰레드 사이의 통신은 공유 변수들을 통해서 이루어진다. 모든 공유 변수들이 *sync*로서 선언되었다면, 병렬 조합 (parallel composition)되어 구성된 각 병렬 모듈은 다른 모듈에게 간섭을 하지 않으며, 병렬 프로그램은 각 부분 프로그램의 특성을 그대로 유지할 수 있게 된다. 이 특성은 매우 중요한 의미를 함축하고 있다. 예를 들어, 병렬 프로그램을 구성하는 데 있어서, 각 기본 모듈이 결정적이고 모든 공유 변수들이 *sync*로서 선언되어 있다면 그 병렬 프로그램의 결정성이 보장될 수 있음을 의미한다.

5 비결정성

결정적 계산 방식은 사용자가 프로그램 제어를 확실히 파악하고 있으므로 디버깅, 테스팅, 및 에러 수정에 유리하다. 또한 대부분의 병렬 프로그램은 기존 순차 프로그램에서와 같이 결정성적으로 수행되어야 한다. 그러나 모든 프로그램이 결정적이어야 할 필요는 없다. CC++에서는 여러 비결정적 계산이 가능하며, 특히 쓰레드들 사이에 변수가 공유되는 경우, 그 변수들을 *sync*로서 정의하지 않고 여러 번 그 변수들의 값을 바꿀 경우 프로그램은 비결정적일 가능성이 높다.

6 원자적 제어 흐름

공유 변수들의 값이 비결정적으로 바뀐다 하더라도 원자적 제어 흐름 (atomic flow control)의 기능이 필요하다. CC++에서는 이것을 *atomic* 함수를 사용하여 해결한다. *atomic* 함수의 동기는 그것이 수행되는 과정에 있어서 다른 프로세스와 interleaving되지 않는 것을 의미한다. *atomic* 함수는 대부분 C++ 클라스의 member 함수에 이용되고 있다. *atomic* 함수는 *sync* 변수들을 사용할 수 없으며 (그것이 초기화되지 않았으면 대기(wait) 상태로 들어 가서 데드락이 발생함.) *atomic* 함수가 속한 객체 밖의 변수들을 사용하지 못한다. 이러한 제한때문에 한 클라스당 한 순간에 오직 한 *atomic* 함수만이 수행될 수 있다.

7 Heterogeneity

C++ 프로그램은 여러 선언(declaration)문을 포함하고 있다. 이 선언문은 변수, 함수, 클라스 및 데이타 타입을 정의한다. 전역적으로 선언된 것은 그 범위가 프로그램 전체에 걸쳐 알려진다. CC++에서는 C++의 전역선언을 logical 프로세서 클라스를 이용하여 구현한다. 이것은 linker등의 시스템 구성 유틸리티에 의해서 생성된다. C++에서 전역적으로 알려진 함수들은 CC++에서는 logical 프로세서 클라스의 public member 함수로서 등록되어 구현된다. 전역적으로 정의된 클라스나 타입에 대해서도 같은 방법이 적용된다.

CC++에는 여러 개의 프로세서 클라스가 존재할 수 있는 데, 한 프로세서 객체가 다른 프로세서 객체를 @ 기호, 전역 포인터 및 *sync* 포인터를 이용하여 접근할 수 있는 것을 제외하고는 프로세서 객체 각각은 다른 프로세서 객체로부터 완전히 독립되어 있다. C++와 비교하면, 프로세서 클라스의 모든 member 함수는 내부적 연결을 가지고 있다고 볼 수 있다. 특히, 정적 (static)으로 선언된 변수는 같은 instance인 프로세서 객체들 사이에서 공유되지 않는다.

프로세서 객체의 member들 CC++에서 객체를 접근하는 데 사용되는 표현은 C++와 같지만, 프로세서 객체를 접근하는 데는 @ 기호가 사용된다. C++에서 객체 k의 data member x를 접근하기 위해서는 $k.x$ 라고 쓰지만, k가 프로세서 객체이면 프로세서 객체에서는 $x@k$ 라고 쓴다. 비슷한 방법으로 프로세서 객체 k의 member 함수 h는 $h@k(...)$이라 쓴다.

모든 연결은 한 프로세서 객체내에서 일어나며, @ 기호를 사용하지 않는 한 프로세서 객체 범위밖의 이름은 접근할 수 없다. C++에서 x라는 이름의 전역(global) 객체는

::x

로서 접근되지만, CC++에서 그것은

::x@k

로서 번역되는 데, 여기서 k는 그 표현을 수행하는 프로세서 객체의 id이다. 한 프로세서 객체 내에서의 모든 객체의 생성은 *new* 문을 사용하여 이루어 진다. 이러한 방법에 따라서, 모든 (동적 혹은 정적으로) 만들어 진 객체들은 어떤 한 프로세서 객체에 지역적(local)이다.

Logical 프로세서와 Physical 프로세서 IDs. logical 프로세서의 타입은 구현방법에 달려있지만 일반적으로 standard include 화일에 명시된다. 각 logical 프로세서는 고유의 id를 갖는다. 프로그램에서 logical 프로세서를 접근하려 할 때는 logical 프로세서를 직접 접근하는 대신에 그것의 id를 이용한다. 따라서 logical 프로세서를 접근하는 동작은 overloading 될 수도 있다. 이러한 방법에 의한 logical 프로세서의 어드레싱은 유연성을 갖게 된다.

프로그램의 수행이 처음 시작할 때, (단 하나의) 프로세서 객체는 한 (physical) 프로세서에 사상된다. 다른 프로세서 객체들은 후에 *new*에 의해서 동적으로 생성될 수 있다. 프로세서 객체는 *new* 함수에 의해서 어떤 특정 하드웨어 (즉, physical 프로세서)에 위치하게 된다. 예를 들면,

*proc_t * sPrt = new (23) Solver;*

라는 statement는 프로그램 *Solver* 라는 새 instance를 만들고 그것을 노드 23에 위치시킨다.

Global pointers CC++에서 포이터 변수는 logical 프로세서를 사용함에 따라서 포인터의 범위가 그 logical 프로세서의 안인지 혹은 밖인지를 구별해야한다. 전자를 지역 포인터, 후자를 전역 포인터라고 부른다. 따라서, CC++에서 포인터는 지역 포인터, 전역 포인터, 혹은 *sync* 포인터 중에 하나이다.

전역 포인터는 *global*이라는 구문을 이용하여 다음의 예와 같이 사용된다.

*int * global g_ptr;*

포인터가 다른 logical 프로세서 객체를 포인트하는 경우 그것은 전역 포인터이거나 *sync* 포인터이다. 그것이 같은 logical 프로세서 객체이내 이라면 그것은 셋 중의 어느 하나가 될 수

있다. 만약 전역 포인터가 *sync* 객체를 포인트하면 그 포인터를 포함하는 프로세서 객체에 있는 포인터의 값을 캐쉬함으로써 구현한다.

전역 포인터는 RPC(remote procedure call)의 형태를 제공한다. 한 객체에 대한 포인터가 주어지면, 그 객체의 member 함수를 invoke할 수 있다. 이때 그 포인터가 전역 객체 포인터이면, 그 객체의 member 함수는 전역 포인터를 통해서 invoke될 수 있다. member 함수의 수행은 그 포인터를 포함하고 있는 프로세서 객체에서 이루어 지는 것이 아니라 목표 객체를 포함하는 프로세서 객체에서 이루어 진다.

제 3 절 구현 시 고려 사항

CC++는 기존의 C++ 컴파일러를 사용하여 구현될 수 있다. CC++를 프리프로세싱하는 번역기를 만들어 CC++를 C++ 코드로 변환한 다음, C++ 컴파일러 및 라이브러리를 이용한다. 쓰레드들은 lightweight 프로세스 라이브러리를 이용하여 구현될 수 있다. *atomic* 동작을 구현하기 위해서는 공유 메모리를 갖는 다중처리 시스템과 같은 실행 환경이 요구된다.

CC++의 전역적 접근의 구현은 하드웨어의 구조에 따라 다르다. 공유 메모리를 지원하는 시스템에서는 전역 접근을 위해서 특별한 것을 지원할 필요는 없다. 그렇지 않은 경우에는, 시스템의 구조에 따라서 해결하는 방법이 다른데, 운영체제에서 가상 공유 주소 공간을 지원하는 경우와 그렇지 않은 두 경우를 따로 고려할 수 있다.

가상 공유 주소 공간을 제공하지않는 시스템에서는, 전역 포인터는 소프트웨어적으로 해결된다. 한 전역 포인터는 (id, adr)로서 표현된다. (id는 노드 no 이고, adr은 그 노드내에서의 어드레스를 의미함.) 한 전역 포인터를 접근하는 경우, 그 전역 포인터는 (id, adr)을 접근하기위한 메시지 패싱을 하는 동작으로서 overloading 된다.

전역 접근과 같이, *sync* 객체의 구현 역시 하드웨어의 구조에 따라 구현방법이 달라진다. J-Machine이나 Tera Machine과 같은 시스템에서는 메모리의 어떤 부분이 쓰여지는 순간 그것을 하드웨어적으로 감지하는 기능이 제공되고 있다. 이러한 기능은 *sync*를 처리하는데 도움을 준다. 그러나 대부분의 시스템에서는 이러한 기능이 제공되지 않고 있으므로, *sync*의 문제는 소프트웨어적으로 처리되고 있다. 각 *sync* 객체마다 tag을 붙이고, 소프트웨어적으로 이것을 감시함으로서 *sync*를 구현할 수 있다.

CC++에서 C++로 변환하기 위해서는 파스 트리(parse tree)를 구성해야 한다. 이 파스 트리는 C++의 파스 트리로서 바뀐 다음, textual 폼으로서 출력된다. 이 과정에서 병렬 구문의 의미를 정확하게 변환시켜야 한다.

제 4 절 향후 전망

앞으로 SMP 기능을 제공하는 하드웨어 시스템이 점차 널리 확산될 것으로 전망한다. 따라서 프로그래밍 환경 측면에서는 이러한 시스템을 위한 '표준화된' 병렬 프로그래밍 언어를 제공하는 일이 시급하다고 생각된다. 최근 HPC++ [HPC++]과 OpenMP [OpenMP] 그룹을 중심으로 병렬 언어의 표준화에 대한 논의가 활발하게 진행되고 있다.

분산 메모리 구조에 기반한 기존의 병렬 컴퓨터에서의 병렬 프로그래밍은 '데이터 병렬성'을 이용한 수치 계산 분야에서 주로 활용되어 왔으며, '제어 병렬성'은 거의 이용되지 않았었다. 그러나 프로그램의 흐름 제어가 특별히 복잡하지 않다면, SMP 머신에서 제어 병렬성 기반 프로그래밍 기법이 실용화 될 수 있을 것으로 예측한다. 따라서 병렬 프로그래밍의 응용 분야가 수치계산 분야에 제한되지 않고 일반 분야로 확장될 수 있을 것이라고 생각한다. 본 고에서 소개된 병렬 구문들은 제어 병렬성 기반 프로그래밍에 많은 도움을 줄 수 있을 것이다

지금까지 국내에서 병렬 프로그래밍을 이용한 응용 프로그램 개발 사례는 매우 적다. 병렬 컴퓨터 시스템이 일반화됨에 따라 병렬 프로그래밍에 대한 요구 또한 증가할 것이며, 이를 위한 교육 및 상용화 방안이 적극 모색되어야 할 것이다. 본 고에서 논의된 CC++의 병렬 구

문들은 매우 잘 정리된 병렬 프로그래밍 기법을 바탕으로 설계되었으므로, 향후 병렬 프로그래밍 언어의 표준화가 어떠한 방향으로 발전하더라도 직접 혹은 간접적으로 연관될 수 있을 것으로 전망한다. 최소한 이러한 언어들은 교육용 목적으로서 충분히 고려할 만한 가치가 있다고 생각한다.

참고 문헌

[CC++Web] `http://www.compbio.caltech.edu/ccpp` (or `http://www.infosphers.caltech.edu/people.mani.html`.).

[CC++Tut] Tutorioals: CC++. California Institute of Technology.

[CK93] K. Mani Chandy and C. Kesselman. Compositional C++: Compositional parallel programming. In *Fourth Workshop on Parallel Computing and Compilers*, 1993.

[CM88] K. Mani Chandy and Jayadev Misra. *Parallel Program Design: A Foundation*. Addison-Wesely, 1988.

[HPC++] HPC++ Committee. HPC++: A Draft White Paper.

[OpenMP] OpenMP Architecture Review Board. `http://www.openmp.org`.

[김진미97] 김진미. ParaC 언어의 쓰레드 기반형 병렬 구문 구현. 한국정보과학회 논문지 C. pp 78-89, vol. 3. no. 1, 1997년 2월.

[우영춘97] 우영춘, 변석우, 지동해, 윤석한. 고속병렬처리컴퓨터에서 ParaC 언어의 원격 태스크 구문을 사용한 병렬 코드에 대한 성능, 한국정보처리학회학술지, 제 4권 2호. 1997년 10월.

[정박윤98] 정용화, 박진원, 윤석한. CC-NUMA 방식의 컴퓨터 구조 고찰, 주간기술동향, 제 848호, 한국전자통신연구원, 1998년 5월.