

매니코어 병렬프로그래밍 모델

Parallel Programming Model for Manycore

김진미 (J.M. Kim)	SW 기초연구센터 책임연구원
변석우 (S.W. Byun)	경성대학교 컴퓨터공학과 교수
김강호 (K.H. Kim)	SW 기초연구센터 책임연구원
고광원 (K.W. Koh)	고성능컴퓨팅 SW 연구팀 선임연구원
차승준 (S.J. Cha)	SW 기초연구센터 선임연구원
정연정 (Y.J. Jeong)	SW 기초연구센터 책임연구원
정성인 (S.I. Jung)	SW 기초연구센터 센터장

* 본 연구는 미래창조과학부의 SW컴퓨팅산업원천기술개발 사업의 일환으로 수행하였음(B0101-15-0644, 매니코어 기반 초고성능 스케일러블 OS 기초연구).

매니코어는 단순한 기능을 가진 수백~수천 개 코어를 하나의 CPU에 집적하여 성능을 구현하는 것으로 근본적으로 이를 활용할 병렬프로그래밍이 필요하다. 단순히 속도를 높이는 방향으로 발전하던 하드웨어는 병렬성을 증대하는 방향으로 발전하고 있고 이에 따라 프로그래밍 패러다임 역시 변하고 있다. 병렬화를 위한 여러 기술이 하드웨어에 구현되고 프로그래머가 이를 보다 적극적으로 활용할 수 있게 하는 유용한 병렬프로그래밍 모델이 필요하다. 또한, 컴퓨팅 환경은 자원의 활용도를 중시하는 시스템 중심에서 응용 및 서비스 중심으로 변화하고 있으므로, 그 도메인에 적합하게 프로그래밍할 수 있는 환경이 요구된다. 매니코어에서 병렬시스템 구조를 활용하는 방법을 결정하는 병렬프로그래밍 모델은 그 목적에 유연하게 제공되고 또한 컴퓨팅 환경 변화에 따라 새로운 개념의 모델을 정립하는 데 있어 유용해야 한다.

2015
Electronics and
Telecommunications
Trends

SW·콘텐츠 기술동향 특집

- I. 서론
- II. 매니코어 병렬프로그래밍 현황
- III. Haskell 병렬프로그래밍 모델 기술동향
- IV. 결론

I. 서론

컴퓨팅 산업은 최근 10년 동안 클라우드 컴퓨팅, 모바일 컴퓨팅, 고성능 컴퓨팅 분야에서의 시스템 환경을 중심으로 진행되고 있다. 계산량이 늘어나는 이러한 시스템 환경은 멀티코어 및 매니코어를 기반으로 더욱 발전하고 있으며 멀티코어 및 매니코어는 슈퍼컴퓨터와 같은 고성능컴퓨터뿐만 아니라 다양한 모바일 기기 및 IT 기술개발에 활용할 수 있다[1][2].

하드웨어의 발전으로 매니코어를 활용하고자 하는 컴퓨팅 분야에서의 최근 요구경향은 시스템 중심에서 응용 및 서비스 중심으로 변화하고 있다. 클라우드 컴퓨팅의 경우를 살펴보면 예전에는 컴퓨팅 및 스토리지 자원의 사용비용이 매우 중요했으나 최근에는 네트워크와 보안에 필요한 비용의 중요성이 더 부각되고 있다. 즉, 매니코어를 사용함에 있어서 코어의 사용량 수준을 보는 활용성보다는 클라우드 컴퓨팅 환경에서 서비스의 스피드업 실행을 위해 메모리에서 데이터를 이동하는 비용 및 서비스의 보안이 더 중요해지고 있는 것이다.

고성능컴퓨팅의 매니코어 활용에서도 컴퓨팅의 중요성에 더해 파워효율성, 확장성, 신뢰성 등이 중요한 이슈로 부각되고 있다. 시대적 요구 반영을 볼 때 응용 및 서비스 관점에서 확장성, 성능, 에너지 효율성, 신뢰성 처리가 중요해졌음을 알 수 있으며, 이러한 시스템을 효과적으로 활용하기 위해 프로그래밍 분야 역시 적합한 대비가 필요하다. 더 빠르고 좋은 프로그램을 위해 프로그래머는 시스템의 자원을 보다 적극적으로 활용해야 한다[3].

그러나 병렬컴퓨터의 효과적인 활용을 위해 하나의 작업을 병렬로 처리할 수 있도록 프로그래밍하는 일은 복잡하고 어렵다. 병렬프로그래밍에서는 다수의 프로세서에 동시 동작 할 수 있도록 해야 할 뿐 아니라 동시에 동작하는 작업의 실행 순서가 정해지지 않으므로 컴퓨팅 자원의 올바른 상태를 유지하기 위해서 동기화 처리

또한 염두에 두어야 한다. 이러한 이유로 효과적인 병렬 프로그래밍을 위해서는 프로그램이 수행될 프로세서의 구조와 동기화 및 통신 처리방법을 지원할 수 있도록 시스템에 최적화된 프로그래밍 모델이 필요하다.

본고에서는 매니코어 시스템의 효과적 활용을 위한 병렬프로그래밍의 일반적인 현황과 매니코어 병렬프로그래밍 모델에 필요한 요구사항을 살펴보고자 한다. 먼저 II장에서는 병렬프로그래밍의 일반적인 특성과 시스템의 특성에 맞는 프로그래밍 모델을 살펴본다. III장에서는 매니코어 시스템을 기반으로 병렬화 구조에 적합하다고 알려진 함수언어인 Haskell을 중심으로 현재 제공하고 있는 병렬프로그래밍 모델을 살펴볼 것이다. 아울러 병렬하드웨어에서 효과적으로 활용하기 위해 적용 가능한 기능을 파악해보고자 한다. 또한, 이를 기반으로 매니코어 환경에서 활용할 수 있는 병렬프로그래밍 모델의 연구방향을 제시해보고자 한다.

II. 매니코어 병렬프로그래밍 현황

1. 병렬프로그래밍 기술

병렬프로그래밍을 잘하기 위해서는 병렬하드웨어를 활용하여 처리 속도를 높여주는 방안을 생각해야 한다. 그러기 위해 기본적으로 프로그램을 병렬로 실행할 수 있도록 더 작은 여러 개의 문제로 잘 나누어주는 분해기술(decomposition)과 잘 나누어진 단위를 병렬로 처리할 때 실행 순서를 보장하여 공유하는 데이터를 바르게 관리해주는 동기화(synchronization) 기술이 필요하다. 즉 병렬프로그래밍의 우선 단계로 소프트웨어를 다수의 프로세서에 실행하기 위해서는 가용한 프로세서에 병렬로 실행할 수 있도록 코드덩어리를 적합한 단위조각으로 나누는 것이 필요하다. 명령형 언어가 분해 및 동기화 기술로 프로그램을 복잡하게 만들어야 하는 데 비해 함수형 언어는 상태 변경으로 인한 효과(side-effect)가 없는 언어의 특성으로 언어의 특성 자체가 분해 및 동기

화 기술적용이 유리하다. 병렬프로그래밍을 위한 분해 및 동기화 기술은 그 대상 및 목적에 따라 병렬프로그래밍 모델을 결정하는 가장 기본 요인이 된다.

가. 분해(Decomposition)기술

분해기술의 목적은 다수 코어에 병렬로 실행하는 태스크를 만드는 것이다. 분해기술의 유형으로는 기능적분해(functional decomposition)와 데이터분해(data-based decomposition)가 있으며 이 두 유형은 해결하고자 하는 문제의 도메인에 따라 어떤 유형이 적합한지 판단하여 활용한다. 기능적분해는 각 프로세스가 서로 다른 함수의 계산을 수행하는 것으로 태스크 병렬성을 줄 수 있고, 데이터분해는 전체 데이터를 가능한 동일한 크기로 나누어 각 프로세스에 할당하여 계산을 수행하는 것으로 데이터 병렬성을 가질 수 있다.

즉, 서로 독립적이고 단순한 계산 작업이 아주 많은 경우에는 기능적분해가 유리하다. 기능적분해를 설명하기 위한 좋은 예로는 프로그램에서 서로 연관성이 전혀 없는 실행 단위를 함수로 각각 작성하고 각각의 함수를 독립적인 프로세서에서 실행하는 것이다. 기능을 분해하는 경우 정적인 경우와 동적인 유형으로 처리할 수 있다. 정적처리의 기능적분해는 프로세서 코어가 많아지더라도 작업이 더 많은 코어를 사용할 수 없어 확장성에 제약이 따른다. 반면에 동적기술을 적용하면 코어 수에 얽매이지 않고 가용한 프로세서 코어에 작업을 분해하여 스피드업을 가져올 수 있다. 동적처리의 기능적분해는 Open Multi-Processing(OpenMP), Cilk, 인텔의 스레드 빌딩 블록과 같은 태스크 기반 모델에서 적합하게 처리할 수 있다.

데이터분해는 아주 방대한 데이터를 처리해야 할 때 적합한데 동일한 함수에서 데이터를 나누어 병렬실행하는 것으로 배열연산을 들 수 있다. 데이터를 분해하는 경우에는 다수 프로세서 코어 환경에서 데이터를 나누어 각각의 코어에서 동일한 계산을 실행하는 것으로 일

반적으로 작업 간 종속성이 존재하지 않는다. 다만 텔레커뮤니케이션 시스템과 같이 호출 이벤트가 발생할 때 음성압축처리와 같은 데이터 처리를 하는 이벤트 기반 시스템과 같은 유형에서는 태스크 기반 모델을 같이 적용할 수 있다. 또한, 데이터의 처리가 다음 단계로 이동하는 파이프라인 모델의 경우에는 각각의 코어가 하나의 처리를 담당하게 정적 기능분해를 활용하고 데이터 분해를 통합하여 처리할 수 있다.

나. 동기화 기술

프로그램이 분해되어 동시에 병렬로 실행되는 작업에서도 동일한 데이터를 접근하고 상태를 변경하는 작업이 필요하며 이 경우 하나의 작업이 끝날 때까지 기다려 동기화 처리를 하게 된다. 동기화 처리는 맵리듀스 알고리즘의 예와 같이 각각의 기능분해에 따른 병렬계산에서 계산의 일부가 다른 계산의 결과를 필요로 하기 때문에 요구되기도 하고, 데이터를 공유할 때도 필요하다. 동기화는 성능 향상에도 많은 영향을 미치므로 동기화 처리에 따른 비용을 줄이기 위해 다양한 해결 방안들이 제안되고 있다.

2. 병렬프로그래밍 모델

병렬화의 목적은 프로그램의 실행 시간을 최대한 단축시키는 것으로 시스템 구조에 따라 최적화된 병렬프로그램을 작성하는 것이 중요하다. 병렬프로그래밍 모델은 프로그래머의 편의성을 위해 언어수준에서 해결할 수 있도록 프로그래밍 언어와 실행시스템 그리고 라이브러리로 제공된다. 프로그램의 실행에 있어 시스템의 메모리와 네트워크 구조 현황은 프로그램을 작성하기 위한 병렬프로그래밍 모델을 결정하는데 주요한 요인이 된다.

시스템 구조 측면에서 메모리 구조에 따라 프로그래밍 모델을 일반적으로 분류하자면 공유메모리 환경에서 주소 공간을 공유하여 프로그램을 작성하고 실행하는

공유메모리 병렬프로그래밍 모델과 병렬시스템을 구성하는 노드(Processing Element)들이 서로 주소 공간을 공유하고 있지 않아 노드 간 서로 연결된 네트워크를 통하여 데이터를 주고받는 메시지패싱 병렬프로그래밍 모델이 있다. 또한, 한 프로그램에서 이 두 모델을 모두 이용하여 프로그래밍하는 하이브리드 병렬프로그래밍 모델이 있다.

언어 수준에서의 병렬프로그래밍 모델은 앞서 설명한 시스템 구조와는 독립적으로 언어 자체에 내제되어 추상화된 형태로 제공되며 병렬실행이 유용하도록 응용의 디자인 및 특성에 따라 선택적으로 사용된다.

Communication Sequential Processes(CSP)와 Actor 모델의 경우에는 태스크들 각각이 독립적으로 동작하면서 메시지로 통신하는 유형의 응용에 적합하다. CSP는 먼저 정의된 채널에서 미리 준비된 수신자(receiver)에 송신자(sender)가 메시지를 보내지만, Actor 모델은 actor가 동적으로 생성되어 비동기적으로 통신한다. 기본적으로 분산되어 있고 느슨한(loosely coupled) 형태의 시스템에 적합한 모델이라 할 수 있다. 태스크 기반의 병렬모델은 공유 메모리 상에서 자주 상태 공유를 해야 하는 응용에 적합하다. 이러한 모델들은 아주 오래전부터 제안되었으며 지금까지 기본 모델로서 응용되어 활용하고 있다.

가. 프로세스 계산법(Process Calculus)

Haskell과 같은 함수형 프로그래밍의 계산 원리는 흔히 'substituting equals for equals'라는 말로 설명되고 있다. 어떤 한 식을 같은 값을 갖는 다른 식으로 치환(substitution)하는 방식의 계산을 의미한다.

병렬처리나 동시성을 표현하기 위해서는 프로세스의 개념이 필수적인데, 프로세스의 계산을 치환으로 설명하는 것은 자연스럽지 않다. 프로세스 계산의 핵심은 여러 프로세스가 동시에 동작하면서, 필요에 따라 프로세스 간의 메시지패싱(message passing)을 표현하는 것이

다. 이를 정형적으로 표현하려는 방법으로서 프로세스 계산법이 소개되었다. CSP, Calculus of Communicating Systems(CCS), Algebra of Communicating Processes(ACP), Pi-calculus 등이 있으며, 정형적이지는 않지만 많은 개념의 제안이 Actor 모델을 통해서 이루어졌다. 현재 가장 많은 관심을 받는 것은 Pi-calculus라고 할 수 있다.

람다 계산법에서 채택되고 있는 함수는 고차함수(higher-order functions)로서, 함수와 값이 구분되지 않고 동일한 수준으로 다루어지며, 함수가 수와 마찬가지로 함수의 인수나 결과값으로 이용될 수 있다. 흔히 이런 개념의 함수를 'first-class citizens'이라고 일컫는데, 이 개념은 대부분의 최신 프로그래밍언어에서 채택되고 있을 정도로 유용성을 인정받고 있다. 이 원리에 따라 람다 계산법의 구문은 변수(variables), 변수에 대한 추상화(abstraction), 함수의 적용(application)의 세 가지 형태로 간결하게 정의되고 있다.

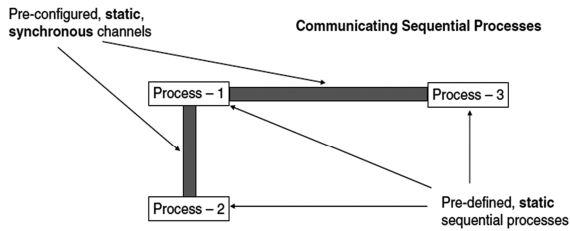
프로세스 계산법에서도 프로세스와 값을 구분하지 않고 동일하게 채택함으로써 프로세스의 표현을 자유롭게 하려는 연구가 진행됐다. 프로세스 사이에 전송되는 것은 수뿐만이 아니라 프로세스나 채널도 전송될 수 있다. 이 개념의 시작은 Actor 모델에서 찾을 수 있으며 값에 적용되는 오퍼레이터 및 프로세스가 모두 actor로서 표현된다. 이 개념은 후에 프로세스 계산법(process calculus)인 Pi-calculus에서 이론적으로 정리되어 표현되고 있다.

나. CSP

CSP는 Tony Hoare가 동시성(concurrency) 프로세스 간 통신을 기술하기 위해 고안된 형식 언어로 처음에 멀티프로세서 머신을 위해 설계되었다. 시스템을 검증하기 위한 도구로도 이용되고 있다[그림 1] 참조].

- Occam

CSP를 기반한 프로그래밍 언어로 초기 트랜스퓨



(그림 1) CSP 모델[3]

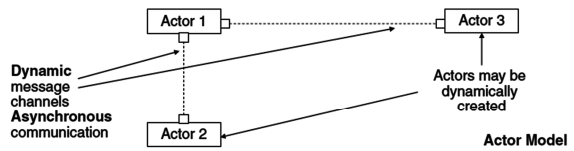
터 프로세서 구조에서 제안되었다. 메시지 통신을 하기 위해 병렬프로세스들 간의 관계를 정의하고 수행 방식, 프로세스 매핑 정의, 프로세스 간 통신 채널 및 통신 방식 등을 제공한다.

• The Go Language

구글이 개발한 Go 언어는 멀티코어 CPU에 적합한 시스템 프로그래밍 언어를 목표로 설계되었다. 대체로 C와 유사한 문법을 가지고 있고 프로그램들이 서로 통신하면서 상태를 공유하는 동시성 프로그램을 쉽게 만들 수 있다. 동시성과 관련하여 사용하는 channel/channel input들은 Tony Hoare의 CSP로부터 도용되어 만들어졌다. 기본적으로는 정적타입 언어이지만 동적타입 언어의 속성 또한 지원한다. 즉, 정적타입 컴파일 언어의 효율성과 쉽게 프로그래밍 지원하도록 동적언어의 특성을 동시에 가지고 있다. 타입 안정성과 메모리 안전성, 동시성 및 통신 지원, 효율적인 메모리 재활용(garbage collection), 빠른 컴파일을 지향하는 언어이다.

다. Actor 모델

Actor 모델은 독립적인 다수의 프로세서 환경에서 actor를 기본 단위로 메시지패싱을 이용하여 행위 (behavior)를 비동기적으로 실행하는 모델이다[(그림 2) 참조]. Actor 모델은 통신시스템(telecommunication)과 같은 이벤트 기반 연성 실시간 시스템(event-driven soft real-time system)에도 넓게 사용된다. Actor 모델은 메시지를 받으면 그에 맞는 행위를 실행한다는 매우



(그림 2) Actor 모델[3]

간단한 동작원리와 공유되는 자원이 존재하지 않으므로 다른 것에 영향을 받지 않는다는 특징 때문에 실행 순서를 이해하고 결과를 예측하기 매우 쉽다. Actor 모델은 비동기 방식을 사용하여 병목이 생기지 않으며 내부 큐를 사용하여 많은 메시지를 한 번에 처리할 수 있기 때문에 대용량 작업 처리에서 많은 장점을 가진다. 하지만 속도 면에서는 단점이 될 수 있다.

• Erlang

에릭슨에서 스위칭 소프트웨어에 사용하기 위해 개발되었으며 병렬프로그래밍이 가능한 함수형 언어이다.

• Haskell

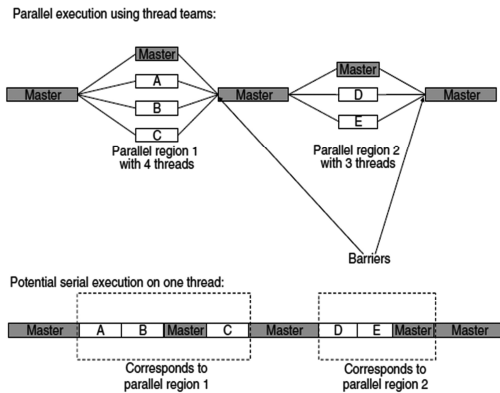
기본 동시성 기법을 확장하여 고유의 함수 언어가 가지는 특성 이외에도 CSP 혹은 Actor 모델을 지원하는 형태의 라이브러리를 제공하여 동시성을 지원하는 다양한 형태의 병렬프로그램을 할 수 있다.

• Scalar

Scalar의 Actor 모델은 Erlang과 동일한 디자인으로 성능과 안정성에서 높은 보장성을 가지고 있다.

라. Task Based Programming Model

태스크 수준의 병렬성은 임의의 코드 영역을 동시에 수행할 수 있을 때 적용할 수 있다. 하드웨어의 특성에 맞게 부하 분산이나 스케줄링을 런타임 시스템에서 지원하여 태스크 병렬성을 수행한다. 잘 알고 있는 OpenMP 모델이 있으며 최근에 X10 등의 언어가 사용된다[(그림 3) 참조].



(그림 3) OpenMP 병렬실행[3]

- X10
IBM에서 고성능 컴퓨팅을 위한 프로그래밍을 목표로 Partitioned Global Address Space(PGAS) 모델을 이용한다.
- Chapel
크레이에서 개발하였고 태스크를 기반으로 PGAS 모델에 기반한 X10과 유사하다.

III. Haskell 병렬프로그래밍 모델 기술동향

1. Haskell 병렬프로그래밍 모델

수천 개 코어를 가지는 매니코어 시스템에서 병렬프로그래밍 모델을 제공받지 않고 병렬프로그램을 하는 것은 해당 시스템의 메모리구조와 네트워크 현황 등을 정확하고 상세하게 알아야 가능한 매우 까다롭고 복잡한 작업이다. 함수언어인 Haskell은 함수의 순수성이 결정적 병렬프로그래밍을 표현하는 데 매우 유리하여 기존의 코드에 병렬화를 부여하는 일이 기존의 명령형 프로그래밍에서보다 상대적으로 어렵지 않다.

Haskell은 순수 함수형 언어로서 레택스의 선택은 결과값에 영향을 주지 않는다. 즉, 레택스에 대한 병렬계산 때문에 결과값이 달라지지 않는 장점이 있다. 이러한 Haskell의 순수성은 병렬계산에 매우 유리하다.

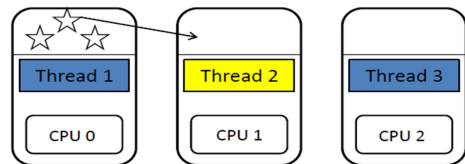
이러한 좋은 특징 때문에 Haskell을 이용한 완전 자동 병렬화를 생각할 수도 있지만, 여기에는 granularity

문제에 대한 어려움이 있다. 병렬계산을 프로세스나 스레드로서 구현할 경우 이에 오버헤드가 발생할 수밖에 없으며, 특히 IPC에는 많은 비용이 발생하게 된다. Granularity가 너무 작으면 병렬계산에 의한 오버헤드가 너무 많아서 충분한 스피드 업을 얻을 수 없게 된다. 이 오버헤드 비용은 하드웨어의 구조 및 코어의 수 등에 영향을 받게 되므로 프로그래밍 과정에서 granularity를 결정하기는 어렵다.

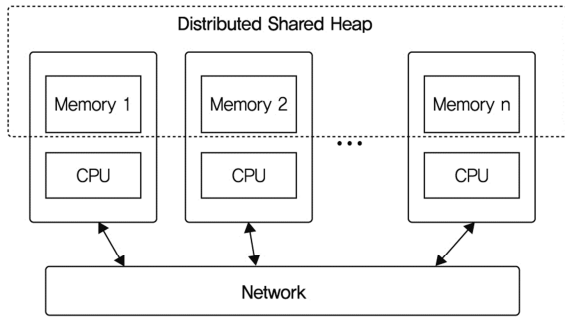
Haskell의 컴파일러인 Glasgow Haskell Compiler(GHC)에서 병렬프로그램을 할 수 있도록 확장한 Glasgow Parallel Haskell(GpH)에서는 반명시적인 병렬 프로그래밍(semi-explicit parallel programming)을 할 수 있게 한다. 즉, 프로그램에서 어느 부분이 동시적으로 수행될 수 있는지 알려주는 ‘par’와 ‘pseq’를 프로그램에 삽입하여 병렬화하는 결정적인 프로그램 방식을 제공한다. GpH는 기본 병렬프로그래밍 방식에서 시스템의 하부 구조를 알지 못해도 프로그래밍을 할 수 있도록 공유메모리, 분산메모리 구조의 시스템에서 병렬성을 제공할 수 있는 Symmetric MultiProcessor(SMP), Graph reduction for a Unified Machine Model(GUM) 병렬프로그래밍 모델을 확장하여 제공한다[4][5].

가. GHC-SMP

Glasgow Haskell Compiler-Symmetric MultiProcessor(GHC-SMP)는 주소공간이 공유되는 시스템 환경을 대상으로 한다. GpH의 공유메모리 구현을 최적화하여 런타임 환경에서 Haskell의 경량 스레드를 코어를 담당하는 OS 스레드와 협력하여 효율적으로 처리한다. Haskell 스레드는 (그림 4)와 같이 스케줄링 정책에 따



(그림 4) GHC-SMP 작업 분배[5]



(그림 5) Virtual Shared Heap 구조[5]

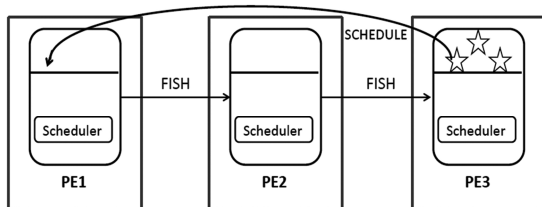
라 런타임 시 다른 코어에 작업을 이전할 수 있고 물리적인 공유메모리에서 지역 스레드 간의 동기화를 위해 뮤텍스를 사용한다. 워크푸싱(work pushing) 스케줄링 방식이다.

나. GHC-GUM

GpH의 RunTime Environment(RTE)를 확장하여 기본적으로 분산메모리 병렬화를 지원한다. 분산된 힙을 추상화하여 프로그래머에게 가상공유힙(virtual shared heap)에서 프로그래밍하도록 한다(그림 5) 참조.

메시지패싱으로 추상적인 그래프 리덕션 기계를 구현하고, 그래프를 변환하여 결과를 생성하도록 그래프의 각 노드에는 실행단위를 구성하고 병렬실행하여 그래프를 리덕션한다. 가상공유그래프를 활용하여 공유메모리, 분산메모리 및 분산공유메모리 시스템 환경에서도 실행할 수 있다.

GUM의 부하분산 스케줄링은 작업이 없는 노드(Processing Element)에서 수행할 작업을 찾기 위해 랜덤하게 메시지를 보내어 작업을 찾게 되면 처음 요청한



(그림 6) GHC-GUM 작업 분배[5]

노드에 작업을 전달하게 된다. 워크스틸링(work stealing) 스케줄링 방식이다(그림 6) 참조.

다. GUMSMP

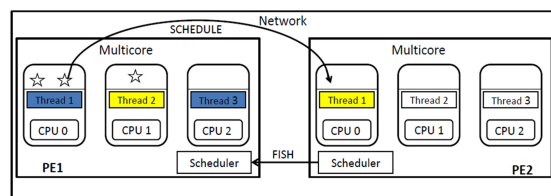
GHC-SMP가 공유메모리 대상으로 병렬화를 지원하고 Glasgow Haskell Compiler-Graph reduction for a Unified Machine Model(GHC-GUM)의 경우 분산메모리 대상으로 병렬화를 지원하는 반면 Compiler-Graph reduction for a Unified Machine Model Compiler-Symmetric MultiProcessor(GUMSMP)는 멀티코어의 클러스터 환경을 지원하는 멀티레벨의 병렬성을 지원한다.

GHC-SMP와 GHC-GUM의 장점을 통합하였고 GHC-GUM의 가상공유힙 기능으로 메모리 관리를 하고 커뮤니케이션 역시 GUM과 동일한 메커니즘을 제공한다. 단일 프로그래밍 모델로서 스케일러블을 지원하며 GHC-SMP와 GHC-GUM의 부하분산 기능이 결합된 형태로 자동 부하분산 기능을 제공한다(그림 7) 참조.

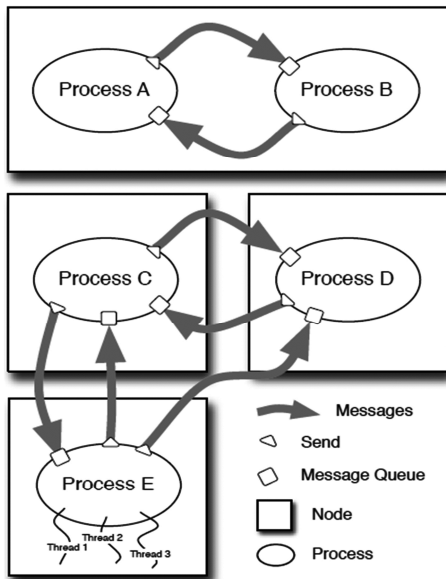
2. Haskell 병렬화 적용기술

가. Cloud Haskell

Haskell은 Embedded Domain Specific Language(EDSL)로 특정영역별 라이브러리를 제공하여 마치 새로운 언어를 정의하듯 프로그래밍 환경 구축이 가능한 언어이다. Cloud Haskell은 EDSL형태로 분산메모리 컴퓨팅 환경에서 프로그래밍할 수 있게 라이브러리를 제공한다. 기본적으로 Actor 모델을 따르며 메시지패싱에 의해 독립적인 프로세서 동작을 실행한다. (그림 8)의 구조에서 볼 수 있듯이 클라우드 도메인에서 노드, 프로



(그림 7) GUMSMP 작업 분배[5]



(그림 8) Cloud Haskell 구조[6]

세스, 커뮤니케이션 관계의 분산컴퓨팅 인터페이스를 정의하고 메시지패싱에 의해 작업을 수행한다. 메시지에는 프로세스 식별자와 포트 채널을 포함하여 위치를 식별할 수 있다. Cloud Haskell은 현재 진행 중인 프로젝트로 대용량 데이터 처리를 위한 Map-Reduce 분산 프레임워크를 정의하여 사용할 수 있다[6].

나. FRP

게임처럼 시간의 변화와 외부의 이벤트 입력에 따라 반응하면서 동작하는 프로그램을 위해서 Functional Reactive Programming(FRP)에서는 Behavior 타입을 정의하여 사용한다. 이 타입과 Arrow 프로그래밍을 이용함으로써 FRP에서는 이러한 동적인 변화를 표현하는 프로그래밍을 비교적 쉽고 추상적으로 표현할 수 있도록 한다.

FRP에서 적용하고 있는 Arrow는 데이터플로우 프로그래밍의 구조를 표현하고 있으며, 함수들 사이의 종속성과 병렬성이 쉽게 드러나게 된다. 따라서 병렬성을 적용하기가 적합하다.

FRP를 게임 프로그래밍에 적용할 경우, 사용자의 입

력 이벤트에 의해 생성되는 각 이벤트 객체들은 시간에 따라 변화하면서 동시에 외부의 이벤트에 반응하면서 작동하는 동시성을 갖게 된다. 이 상황은 FRP에 의한 게임 프로그래밍에 병렬처리가 적용될 수 있음을 제시하고 있다.

FRP는 게임뿐만 아니라 로봇 제어, 시뮬레이션 등의 여러 분야에 적용될 수 있는 중요 기술이다[7]. 이들은 시간에 따라 변환하고 외부의 반응에 대응하는 특징을 갖고 있는데, 이들을 프로그래밍하는 데는 여러 번거롭고 까다로운 점이 있으나 FRP를 적용함으로써 이 프로그래밍을 추상적인 관점에서 쉽고 간편하게 표현할 수 있다. 특히, Haskell의 Domain Specific Language 특징을 적용하면 구문을 훨씬 더 자연스럽게 편리하게 표현할 수 있다.

다. 종속형 타입(Dependent Type)을 이용한 분산배열 프로그래밍(Distributed Array Programming)

메모리 공간이 분리된 분산 메모리 환경에서 분리된 각 노드에 코드와 배열(array) 데이터를 위치시킬 때 코드의 배열 인덱스(array index)값이 지정된 경계(boundary)를 넘지 않도록 프로그래밍하는 것은 프로그램 안전성 측면에서 매우 유용하다.

타입은 자료구조의 표현과 프로그램의 안전성에 중요한 역할을 한다. 현재 함수형 언어 Haskell의 타입 시스템에는 Type Checking, Type Inference, Polymorphic Type, Parameterized Type, Recursive Type, Type Class에 의한 overloading 등의 기능이 구현되어 있다. 여기에 종속형 타입이 추가된다면 타입 시스템은 매우 풍부한 표현력을 갖게 된다. 아직 프로그래밍 언어에서는 이 기술이 충분히 구현되지 않았지만 Coq와 Agda 등의 정리 증명기 등을 통해서 구현됨으로써 많은 주목을 받고 있다. Haskell에서도 Generalized Abstract Data Type(GADT) 등을 통하여 이 기능의 일부가 구현되었으며 이 기술을

구현하기 위한 노력이 꾸준히 진행되고 있다.

타입을 갖는 람다 계산법에서 볼 때, 타입은 마치 함수처럼 정의되고 계산될 수 있다. 예를 들어, (List a) 타입은 임의의 리스트를 표현하는 타입으로서, 타입 변수 a가 정수인 (List Int)는 정수들로 구성된 타입을 의미하며, true, false 등의 원소로 구성되는 리스트는 (List Bool)의 타입을 가진다. 기존의 Parameterized 타입의 경우, a가 취할 수 있는 타입은 오직 타입에 국한되어 있다. 이 기능을 확장하여 타입 인수가 타입뿐만 아니라 수 등을 표현하는 함수식 (term)을 취할 수 있다면, (List a n) 등의 형태가 될 수 있다. a가 정수, n이 수인 식, 예를 들어, (List Int 10)은 10개의 정수로 구성된 배열에 해당된다. 이 기술이 구현된 언어의 컴파일러는 컴파일 시에 배열의 인덱스가 그 범위를 넘어가지 못하도록 점검할 수 있다(C 언어는 이를 점검할 수 없으며, Java의 경우 컴파일러가 아닌 런타임에 점검되어 예외상황이 발생됨).

배열뿐만 아니라 종속형 타입이 추가된 타입 시스템의 표현력은 여러 분야에 응용 될 수 있다. 타입 이론과 증명 이론 사이의 일치성을 설명하는 Curry-Howard Isomorphism의 관점에서 볼 때, 기존 타입 시스템은 Propositional Logic에 해당되며, 종속형 타입 기능은 이를 Predicate Logic으로 확장한 것에 해당된다. 즉, 종속형 타입이 구현된 시스템은 Predicate Logic 수준의 specification과 증명(proof)을 할 수 있음을 의미하는데, 이런 기반을 바탕으로 구현된 정리 증명기로서 Coq와 Agda 등이 있다. 종속형 타입의 구현은 프로그래밍 언어와 정리 증명의 기능이 통합될 수 있는 언어 시스템을 예측할 수 있으며, 이 기술을 이용함으로써 프로그램의 안전성은 획기적으로 발전할 수 있다[8].

3. Haskell 병렬프로그래밍 모델 연구방향

병렬프로그래밍 모델은 컴퓨터 시스템을 하나의 추상화된 형태로 보여주어 프로그래밍언어와 라이브러리로 프로그래머가 병렬프로그램을 할 수 있게 한다.

매니코어로 코어가 늘어날 때 스피드업을 코어가 늘어나는 것만큼 올릴 수 있느냐는 쉬운 문제가 아니다. 병렬성을 무작정 높여 확장성을 가질 수 있다고 생각하는 것은 옳지 않은 생각이다. 시스템 측면에서 메모리 대역폭도 고려해야 하고, 통신 비용, 공유 자원 비용 역시 고려해야 하며 시스템 자원을 관리하는 운영체제에서도 이러한 문제를 해결해주고 지원해주어야 한다.

또한, 수천 개 이상의 매니코어 시스템에서는 모든 코어에서 주소 공간을 공유하는 것이 어려울 수 있다. 이럴 경우에 일반적으로 주소 공간에 따라 Message Passing Interface(MPI)와 OpenMP의 병렬프로그래밍 모델을 혼합하여 하이브리드 병렬프로그래밍 모델을 사용하게 된다.

병렬화는 시스템 구조의 영향을 많이 받기 때문에 시스템 구조를 노출할 경우 컴퓨터 시스템에 대한 이해가 높지 않은 응용 프로그래머들에게 복잡성으로 인해 큰 부담으로 적용한다. 명령형 언어의 경우에는 어떤 코드가 병렬화가 가능한지 판단하고 대부분 병렬성을 직접 노출해서 프로그램을 완전히 새롭게 구성해야 한다.

매니코어 시스템의 경우 시스템 측면에서는 단순히 속도보다는 병렬성을 증대하는 방향으로 발전하고 있으므로 함수의 순수성으로 결정적 병렬프로그래밍을 표현하는데 매우 유리한 Haskell은 진화 방향에 적합하다고 할 수 있다. 또한, 최근 GpH 확장으로 병렬프로그래밍 모델의 기본이 되는 공유, 분산, 분산공유 모델을 제공하고 있다. 매니코어 시스템 역시 시스템 메모리 구조 및 네트워크 현황이 이러한 프로그래밍 모델의 범주를 크게 벗어나지 않을 것이다.

또한, Haskell은 Domain Specific Language를 지원할 수 있다. 즉 언어기반에서 새로운 병렬화 모델을 제시하기에 유용하다. 즉, 시스템 및 응용의 도메인에 특화된 언어의 형태를 제공할 수 있어 기본 병렬프로그래밍 모델에 확장하여, 또는 새로운 개념의 모델을 정립하는데 있어 가능성을 열어준다.

IV. 결론

매니코어 시대를 대비하여서는 매니코어를 지원하는 운영체제의 지원뿐만 아니라 응용을 병렬화하면서 겪는 프로그래밍 장벽을 해결하는 것이 중요하다. 이를 위해 프로그래밍 모델을 결정함에 있어 시스템의 구조는 아주 중요하다. 매니코어 시스템의 메모리 구조를 생각할 때 각 코어 간에 주소 공간을 공유하느냐 그렇지 않느냐에 따라 프로그래밍 방법이 달라질 것이고, 프로그래밍 모델에서는 사용자에게 하드웨어를 감추고 어느 수준의 추상화를 제공하는 게 효율적이고 편리한지 결정해야 한다. 시스템 구조에 따라 코어들 수준에서의 병렬성은 OpenMP 방식을 활용하고 노드 간 병렬성은 MPI 방식이 유리하며 매니코어 시스템에서는 하이브리드 병렬성을 고려해야 할 것이다.

Haskell은 함수 언어가 가진 특성으로 병렬프로그래밍을 좀 더 효율적으로 할 수 있게 해주고, 신뢰할 수 있는 동시성 프로그래밍을 가능하게 하는 기능을 제공하며 매니코어 시스템의 환경변화에 적합한 프로그래밍 언어로서의 가능성을 보여준다. 또한, 특화된 언어의 형태를 제공할 수 있어 시스템 및 응용의 도메인에 적합한 병렬화 모델을 제시하기에 유용하다.

용어해설

병렬프로그래밍 모델 병렬프로그램의 작성 방법을 규정하고 프로그램에 의해 병렬컴퓨터구조의 활용 방법을 결정하는 것으로 프로그래머에게 편의성을 제공하기 위해 언어수준에서 해결할 수 있도록 프로그래밍 언어, 실행시스템, 라이브러리들로 제공됨.

약어 정리

ACP	Algebra of Communicating Processes
CCS	Calculus of Communicating Systems
CSP	Communicating Sequential Processes
EDSL	Embedded Domain Specific Language
FRP	Functional Reactive Programming
GADT	Generalized Abstract Data Type
GHC	Glasgow Haskell Compiler

GHC-GUM	Glasgow Haskell Compiler-Graph reduction for a Unified Machine Model
GUM	Graph reduction for a Unified Machine Model
GHC-SMP	Glasgow Haskell Compiler-Symmetric MultiProcessor
GUMSMP	Compiler-Graph reduction for a Unified Machine Model Compiler-Symmetric MultiProcessor
GpH	Glasgow parallel Haskell
MPI	Message Passing Interface
OpenMP	Open Multi-Processing
PGAS	Partitioned Global Address Space
RTE	RunTime Environment
SMP	Symmetric MultiProcessor

참고문헌

- [1] 김진미 외, “매니코어 시대를 대비하는 Haskell 병렬프로그래밍 동향,” 전자통신동향분석, 제29권 제5호, 2014. 10, pp. 167-175.
- [2] 정진환 외, “Manycore 운영체제 동향,” 전자통신동향분석, 제29권 제5호, 2014. 10, pp. 176-185.
- [3] A. Vajda, “Programming Many-Core Chips,” Springer, 2011.
- [4] M.KH. Aswad, P.W. Trinder, and H.W. Loidl, “Architecture Aware Parallel Programming in Glasgow Parallel Haskell (GPH),” *ICCS, Procedia Computer Science*, vol. 9, 2012, pp. 1807-1816.
- [5] M. Aljabri, H.W. Loidl HW, and P.W. Trinder, “The Design and Implementation of GUMSMP: a Multilevel Parallel Haskell Implementation,” *ACM International Conference Proceeding Series*, 2013, pp. 37-48.
- [6] J. Epstein, A.P. Black, and S. Peyton-Jones, “Towards Haskell in the Cloud,” *Haskell Symposium*, Tokyo, Sept. 22nd, 2011.
- [7] E. Amsden, “A Survey of Functional Reactive Programming Concepts, Implementations, Optimizations, and Applications,” *ACM, Computing Surveys (CSUR)*, vol. 45, no. 4, Aug. 2013.
- [8] W. Swierstra and T. Altenkirch, “Dependent Types for Distributed Arrays,” *Proc. 9th Symposium on Trends in Functional Programming*, 2008.