2014
Electronics and
Telecommunications
Trends

매니코어 시대를 대비하는 Haskell 병렬 프로그래밍 동향

Technology Trends of Haskell Parallel Programming in the Manycore Era

Ⅰ. 서론
Ⅱ. Haskell 함수 언어
Ⅲ. Haskell 병렬 프로그래밍
Ⅳ. 결론

김진미 (J.M. Kim) SW 기초연구센터 책임연구원
변석우 (S.W. Byun) 경성대학교 컴퓨터공학부 교수
김강호 (K.H. Kim) SW 기초연구센터 책임연구원
정진환 (J.H. Jeong) SW 기초연구센터 선임연구원
고광원 (K.W. Koh) 고성능컴퓨팅 SW 연구실 선임연구원
차승준 (S.J. Cha) 미래연구팀 선임연구원
정성인 (S.I. Jung) SW 기초연구센터 센터장

* 본 연구는 미래창조과학부의 SW컴퓨팅산업원천기술개발 사업의 일환으로 수행하였음(14-824-09-011, 매니코어 기반 초고성능 스케일러블 OS 기초연구).

매니코어 구조의 고성능 컴퓨팅 시대가 시작되고 있다. 매니코어의 성능을 활용하기 위해서는 병렬 프로그래밍이 필수적인데, 이 방식은 기존 프로그래밍에 비해 훨씬 더 복잡하고 어렵다. 또한 컴퓨터의 성능이 높아짐에 따라 소프트웨어의 규모와 복잡도 또한 증가하게 되며, 소프트웨어를 에러 없이 안전하게 개발하는 것은 매우 어려운 문제가 되고 있다. 이 문제해결에 도움을 줄 수 있는 한 방법으로 기존의 명령형 프로그래밍 언어 대신 Haskell과 같은 순수 함수형 언어의 이용을 고려한다. Haskell은 지난 수십 년 동안 람다 계산법, 타입 이론, 의미론 등의 강력한 이론적 배경하에 최신 기술을 수용하면서 발전하고 있는 순수 함수형 언어이다. 함수의 순수성은 결정적(deterministic) 병렬 프로그래밍을 표현하는데 매우 유리하다. 최근 이와 관련된 매우 고무적인 연구결과가 발표되고 있으며 여러 응용프로그램들이 개발되고 있다. Haskell은 여러 강력한 이론 덕택으로 병렬 프로그래밍뿐만 아니라 소프트웨어의 생산성 및 안정성과 관련된 많은 문제에 도움을 줄 수 있는 다목적 언어로써 주목 받고 있다.

© 2014 한국전자통신연구원

Ⅰ. 서론

산업의 발전으로 데이터를 처리하기 위한 계산량이 늘어나면서 고성능 컴퓨팅을 위한 멀티코어 및 매니코어 시스템 환경에 대한 요구가 증가하고 있다. 이러한 고성능의 하드웨어를 기반으로 최고의 성능을 내기 위해서는 운영체제에서부터 계산 라이브러리까지 소프트웨어 지원이 필수이다. 더욱이 이를 효과적으로 활용하기 위해서는 그 시스템에 최적화된 병렬 프로그래밍 모델뿐만 아니라 실행시스템, 컴파일러, 디버거, 라이브러리 등이 필요하며 이미 많은 기술들이 연구되고 개발되어 사용되고 있다. 이에 반해 국내에서는 아직 고성능의 병렬 하드웨어를 활용하기 위한 대비가 미흡한 실정이다[1].

고성능 컴퓨팅을 활용하기 위한 병렬 프로그래밍 기술은 더욱 중요해지고 있으며 이에 따라 효과적인 프로그래밍 언어가 주목 받고 있다. 프로그래밍 언어의 접근방식은 모두 장단점이 있으며 일반적으로 사용하는 전통적인 정적 언어인 명령형 프로그래밍 언어에서 객체지향 언어, 그리고 지난 수년간은 생산성이 높은 인터프리터 방식의 언어들을 많이 사용하였다. 하지만 멀티코어 및 매니코어 프로세서에서 전통적인 병렬 프로그래밍은 프로그래머가 가능한 모든 상태 변화를 생각하지 못하면 그 코드는 에러 발생 가능성이 높으며 안정성이 검증되지 않아 사실상 매우 어려운 작업이다. 병렬 프로그래밍은 프로그램의 흐름을 예측하기가 어려우며 그 이유로 특정 상황에 대한 재현이 어렵다.

최근 병렬 컴퓨터 구조가 각광을 받게 되면서 병렬화 구조에 적합한 함수형 언어가 관심을 받기 시작했다. 그 중 Haskell은 함수 언어가 가진 특성으로 병렬 프로그래밍을 좀 더 효율적으로 할 수 있게 해주고, 신뢰할 수 있는 동시성 프로그래밍을 가능하게 하는 기능을 제공한다. Haskell은 소프트웨어 트랜잭션 메모리 기능과 명시적인 동시성 프로그래밍으로 쓰레드를 생성하고 제어하는 기능을 제공하며 멀티코어 및 매니코어 시스템의 환경변화에 적합한 프로그래밍 언어로써의 가능성을 보여주기에 적합하다.

Haskell은 순수 함수형 프로그래밍 언어이다. C와 C++와 같은 명령형 프로그래밍 언어에서는 프로그램이 실행되는 동안 컴퓨터는 상태를 변경하여 해당 작업을 실행할 수 있지만 순수 함수형 프로그래밍은 어떤 작업을 하라고 컴퓨터에게 명령을 주는 것이 아니라 그 작업이 무엇인지를 함수로 표현하여 알려준다. 함수가 할 수 있는 것은 어떤 것을 계산하여 그 결과를 반환하는 것으로 상태 변경으로 인한 효과(side-effect)가 없으며 함수형 언어에서는 변수에 하나의 값을 설정하고 이후에 다른 값으로 설정할 수 없다. 그래서 함수는 한번 계산하면 다시 계산할 필요 없이 항상 같은 값을 가지게 되며 이러한 특성이 병렬 프로그래밍에 큰 장점을 가지게 된다. 명령형 언어가 많은 코어를 제대로 사용하기 위해서는 쓰레드 제어를 하기 위한 동시성 프로그래밍으로 프로그램을 복잡하게 만들어야 하는 데 비해 함수형 언어는 효과가 없는 언어의 특성을 살려 훨씬 쉽게 멀티코어 및 매니코어 이용이 가능할 수 있을 거라는 기대감을 가질 수 있다.

본고에서는 함수 언어인 Haskell을 중심으로 병렬 프로그래밍의 전반적인 기술현황과 발전방향을 살펴보고자 한다. 먼저 II장에서는 Haskell 함수 언어의 특성을 살펴보고, III장에서는 Haskell 병렬 프로그래밍의 기능과 기술방향을 파악하여 이를 토대로 차세대 매니코어 시대를 대비하는 병렬 프로그래밍 연구 및 응용분야를 전망하고자 한다.

Ⅱ. Haskell 함수 언어

1. Haskell 함수 언어 특성

가. 순수 함수형 프로그래밍의 결정성

Haskell은 순수 함수형(pure functional) 언어로 다음과 같은 특징을 갖는다. 첫째, Haskell에서는 할당문

(assignment statement)과 동적으로 변화하는 변수(mutable variable)를 사용하지 않는다. 따라서, 명령형 프로그래밍과는 달리, 상태(state)나 효과(effect, 혹은 side-effect)를 이용하는 프로그래밍을 기반으로 하지 않는다(필요에 따라 모나드에 의한 효과를 이용하는 비순수한 프로그래밍을 적용할 수도 있으나, 그럼에도 불구하고 순수성은 분리되어 유지될 수 있다).

둘째, 모든 수식은 유일한 값을 갖는다. 예를 들어, 한 수식 f(2,3)의 값이 7이라면 이 수식의 값은 문맥(context)과 계산순서에 독립적으로 언제나 7을 유지한다. 이 현상은 람다 계산법(The Lambda Calculus)에서 Church-Rosser의 특징이라고 불린다.

셋째, 각 수식은 유일한 값을 가지므로, Haskell에서는 치환(substitution)에 의한 계산원리를 적용할 수 있다. 예를 들어, f(2,3)의 값이 7이라고 가정할 때, g(4, f(2,3), 5)의 계산은 치환에 의한 계산 g(4,f(2,3), 5) = g(4,7,5)를 적용한다.

넷째, Haskell의 계산에서는 계산순서를 정의하지 않으며, 또한 계산순서의 변경이 결과값에 영향을 주지 않는다. Haskell에서 치환을 적용할 수 있는 부분식을 레덱스(Redex, REDucible EXpression)라고 부른다. 예를 들어, (1+2)×(3+4)는 두 개의 레덱스 (1+2)와 (3+4)를 갖는다. Haskell에서는 이 두 레덱스 중에서 어떤 것을 먼저 치환하도록 하는 계산순서를 정의하지 않는다. 계산순서에 상관없이 결과값은 앞서 언급한 Church-Rosser의 특징에 따라 언제나 일정하다. 이런 현상이 산술식뿐만 아니라 모든 경우의 계산에 적용될 수 있다. 병렬 프로그래밍의 관점에서 볼 때, 한 수식이 여러 레덱스를 내포하고 있을 때, 이들을 동시에 계산해도 결과값이 달라지지 않는다. 이런 점에서 Haskell은 병렬성을 내재하고 있다. 수행 결과 언제나 동일한 결과값을 갖는 결정적(Deterministic) 계산의 특징을 갖는 응용 프로그램을 병렬 프로그래밍할 때, Haskell은 큰 강점을 갖게 된다.

마지막으로 Haskell은 지연 계산(lazy evaluation, 혹은 call-by-name)을 적용한다. 지연 계산에서는 함수의 인수 계산을 계산하지 않고 수식 그 자체로 패싱하는 방법을 적용하며, 이 지연된 계산은 후에 '계산이 반드시 필요할 경우'에 수행되도록 한다. 예를 들어, f (x, y) = x 라는 함수를 가정해 보자. 이 함수는 두 개의 인수 x와 y 중에서 두 번째 인수는 전혀 사용하고 있지 않다. 이때, 함수 호출 f(3, g(1,2)) 이 발생하고, g(1,2)의 계산이 종료되지 않는다면 이 계산은 call-by-value로는 계산을 할 수 없으나, call-by-name 방식으로는 계산을 끝낼 수 있다. 이와 같이, 지연계산은 기존의 call-by-value 방식 보다 더 정확한 계산을 할 수 있다고 볼 수 있다.

지연 계산은 무한 개의 원소로 구성되는 리스트 및 데이터 구조를 표현할 수 있도록 한다. 예를 들어, 짝수들로 구성된 자연수들의 리스트는 [2,4..]로 표현되며, 이 리스트에서 맨 처음의 값을 찾아내는 함수 head를 적용한다면 head [2,4..] = 2가 된다. 이 또한 call-by-value 방식으로는 계산될 수 없는 경우이다. 실용적인 측면에서 볼 때, 무한 계산은 스트리밍 형태의 데이터 처리에 유용하게 적용될 수 있다. 이론적인 측면에서 볼 때, 지연 계산은 의미 없는 값(undefined value)을 해석하는 문제와 연관되어 있다. 일반적으로 무한히 많이 반복되는 계산들은 의미 없는(혹은 잘못 작성된) 프로그램으로 해석하는 경우가 있지만, 모든 경우의 무한 계산을 의미 없다고 판단할 수는 없다. 예를 들어, 운영체제의 커널은 종료되지 않는 계산을 하고 있다. 람다 계산법 등에서 이 개념은 head normal form 등으로 잘 정리되어 있으며, 이 개념이 Haskell에서 충실히 반영되어 구현되었다.

이론적으로 많은 장점을 가지고 있음에도 불구하고 구현이 어렵고 성능이 저하될 수 있는 문제 때문에 지연

계산을 채택하는 언어는 극히 일부에 지나지 않았다. 그러나 이 문제를 해결할 수 있는 방안이 제시되고, Miranda와 Haskell 컴파일러 구현을 통하여 이 문제가 어느 정도 해결되었으며, Haskell뿐만 아니라 최근 Scala 언어 또한 이 방법을 채택하고 있다.

나. 모나드에 의한 비결정성

Haskell은 순수 함수형 프로그래밍 언어로써 본질적으로 효과(effects, 혹은 side-effects)를 사용하지 않는다. 그러나 입출력처럼 경우에 따라서는 효과를 사용하는 것이 자연스럽고 편리하다. Haskell에서는 모나드(Monad)와 애로우(Arrow)를 이용하여 효과를 이용하는 프로그래밍 환경을 제공하고 있다. 효과는 응용프로그래밍의 특징에 따라 다양한 형태로 정의될 수 있다. 입출력(IO Monad)과 상태 변환(State Monad)을 명령형 프로그래밍처럼 자연스럽게 표현할 수 있다. 또한, 원소의 수가 일정하지 않는 리스트(List Monad)와 비결정적 형태의 문법을 파싱(Parser Monad)으로 쉽게 표현할 수 있다. 이 밖에 사용자는 필요에 따라 새로운 형태의 모나드를 정의할 수 있다.

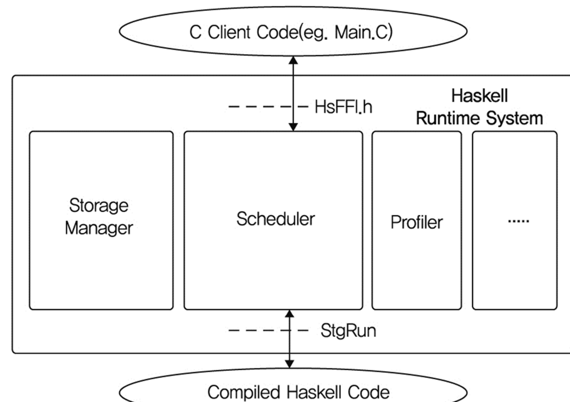
모나드를 사용하는 경우 함수의 순수성이 유지되지 못하므로 함수라는 표현 대신 액션(action)이라고 부른다. 액션은 결정성(determinism)을 유지하지 못하므로 결정적 병렬프로그램을 위해서는 액션의 사용은 제한적이어야 한다.

한 프로그램 내에서는 함수들과 액션들이 혼합되어 사용될 수 있는데, 각 모나드에 의한 효과 영향의 범위는 국지적(local)이다. 즉, 한 프로그램을 액션과 함수를 혼합하여 구성하더라도 액션의 효과가 함수의 순수성과 결정성에 영향을 미치지는 않는다. 또한, 수식의 표현이 함수인지 액션인지는 타입을 이용하여 명시적으로 표현되므로 한 함수(혹은 액션)의 결정성 여부는 정적으로 판단할 수 있다. 즉, 부분 프로그램(sub-programs)의 결정성 여부를 타입으로 제어할 수 있다.

2. Haskell 런타임 시스템

Haskell 프로그래밍 환경을 살펴보면 인터프리터와 컴파일러를 지원한다. Haskell 컴파일러인 GHC(Glasgow Haskell Compiler)의 경우 그 자체가 Haskell로 쓰여졌으며 (그림 1)의 프로그램을 실행하는 GHC 런타임 시스템(RTS: Runtime System)의 경우 5만 라인 정도의 C와 C-- 코드로 되어있다. 즉, GHC의 백엔드에서는 C--코드로 번역되어 GHC 런타임 시스템에 연결된다. C--는 함수 언어연구가인 Simon Peyton Jones와 Norman Ramsey에 의해 만들어졌으며 C가 가지고 있지 못한 tail recursion, garbage collection, 예외 처리 등을 지원하는 간편한 런타임 인터페이스를 가지고 있으면서 고수준 머신 코드를 생성하는 쉽고 편리한 어셈블리 언어(portable assembly language)의 형태를 갖는다고 할 수 있다.

런타임 시스템은 Haskell 코드를 실행하는 모든 기능과 garbage collector를 포함하는 storage manager, Haskell 쓰레드의 user-space scheduler, byte-code 인터프리터, Heap과 Time 등 여러 종류의 프로파일링 기능, STM(Software Transaction Memory)을 지원한다.



(그림 1) Haskell 런타임 시스템

Ⅲ. Haskell 병렬 프로그래밍

1. 병렬 프로그래밍 모델

프로그래밍 모델은 프로그래밍 언어와 실행시스템 그리고 라이브러리들로 구성되어 있다. 병렬 프로그래밍 모델은 병렬 프로그램이 어떻게 작성되어야 하는지를 규정하고 또 작성된 프로그램이 어떻게 병렬 시스템 구조를 활용하게 될지를 결정한다. 일반적으로 공유 메모리 모델, 메시지 전달 모델, 그리고 이 둘을 혼합한 모델 등으로 분류되며 최근에는 성능가속형 컴퓨터 구조를 위한 프로그래밍 모델들도 연구되고 있다. 공유 메모리 모델은 하나의 시스템 안에서 여러 쓰레드가 동작하고 그 시스템 안에 있는 메모리를 모든 쓰레드가 공유하는 구조이므로 너무 많은 수의 쓰레드를 사용하게 될 경우 공유 메모리에 너무 많은 작업이 몰려 성능이 저하될 수 있다. 공유 메모리 모델로는 Threads, OpenMP(Open Multi-Processing)가 일반적이다.

메시지 전달 모델은 여러 노드가 참여하여 작업을 수행하고 각 노드들 사이에 직접적인 메시지 전달을 통하여 정보를 주고 받음으로써 연산을 진행하는 방식으로 MPI(Message Passing Interface)가 대표적이다.

이 두 가지 모델이 혼합한 형태로 PGAS(Partitioned Global Address Space) 모델이 사용되며 공유 메모리 모델처럼 모든 쓰레드에서 접근 가능한 공유 메모리 영역이 존재하지만 이 메모리 영역은 논리적으로 분할되어 각 프로세스별로 더 가까운 위치의 메모리 영역이 존재하는 등 메시지 전달 모델과 비슷한 형태를 띤다. 이 외에도 성능가속형 모델로 GPU를 범용적으로 활용하기 위한 CUDA(Compute Unit Device Architecture)와 이종의 플랫폼을 활용하기 위한 OpenCL(Open Computing Language) 등이 있다[2].

기존의 이러한 병렬 프로그래밍 모델을 구현하는 명령형 언어 기반의 병렬 프레임워크는 시스템의 구조를 파악하여 특성을 이해하고 병렬화를 적용하는 사용자의 개입이 필수이며 병렬 프로그램을 작성하는데 어려움이 있다. 기존의 순차 코드를 병렬 코드로 바꾸는 일은 쉽지 않은 일이며, 상태를 공유하고 락(lock)을 사용하는 일 또한 어렵다. 이에 반해 함수형 프로그래밍은 기존 코드에 병렬화를 부여하는 일이 상대적으로 쉬우며 앞으로 매니코어 컴퓨팅 환경에서 프로그램의 생산성을 높일 수 있는 응용 프로그램 경향에 잘 적용될 수 있는 여지가 충분히 있다. 기존의 명령형 프로그래밍에 비해 Haskell 함수형 프로그래밍이 이러한 병렬 프로그래밍 모델에 적합할 수 있는지 다음 절에서 살펴보기로 한다.

2. Haskell 병렬 프로그래밍 기술

병렬성 및 동시성 프로그래밍에 있어 프로그램 방법은 크게 결정적인 프로그래밍 방법과 비결정적인 프로그래밍 방법으로 나누어 볼 수 있다. 결정적인 프로그래밍 방법은 다시 반명시적인 병렬 프로그래밍(semi-explicit parallelism)과 데이터 병렬성 프로그래밍(data parallelism) 방법으로 분류할 수 있으며, 비결정적인 프로그래밍 방법으로는 명시적인 동시성 프로그래밍(explicit concurrency) 방법으로 분류할 수 있다[3].

병렬성 및 동시성에 대해 우선 간단하게 정의하자면, 병렬 프로그래밍은 일반적으로 성능 향상을 목적으로 여러 프로세서를 사용하도록 명시적인 프로그래밍을 하게 된다. 그에 반해 동시성 프로그래밍은 변화가 있는 여러 상호작용을 프로그램으로 표현하는 것으로 여러 개의 동시적인 작업이 쓰레드로 실행이 되어 그 결과가 상호작용에 의해 비결정적일 수 있다. 즉, 병렬 프로그래밍은 프로그램의 시멘틱에 전혀 영향을 미치지 않으므로 결과는 반드시 같지만 동시성 프로그래밍은 멀티쓰레드 사용 자체가 시멘틱에 포함되어 있기 때문에 쓰레드 수행에 따라 시멘틱이 달라져 결과가 다르게 나올 수 있다[4].

동시성 프로그래밍은 주로 입출력 동시성, 예외 처리

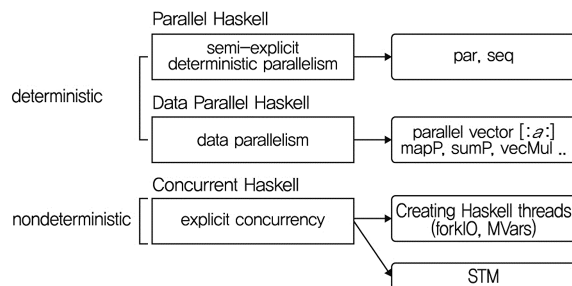
기능을 생각해 볼 수 있다. 이러한 병렬성 및 동시성 프로그래밍의 정의를 기반으로 반명시적 병렬 프로그램은 병렬성을 표시하는 단서(annotation)를 두어 OS 및 프로세서에게 병렬로 수행할 수 있도록 알려주는 방법이며, 데이터 병렬성 프로그램은 벡터 처리로 연속적인 데이터를 동일한 명령어가 동시에 실행할 수 있다. 명시적 동시성 프로그램은 작업을 동시에 수행함에 있어 공유되는 정보에 대한 처리가 이루어질 수 있도록 명시적으로 프로그램 할 수 있게 한다.

(그림 2)에서 보는 바와 같이 함수 언어인 Haskell의 병렬 프로그래밍에 대해 이 세가지 병렬 프로그래밍을 어떻게 표현하는지 살펴보았다. 이는 가장 기본적인 개념 설명을 위해 필요한 부분이며 효율적인 병렬 프로그래밍 기술은 이를 위해 리스트(list), 맵(map), 어레이(array) 등의 여러 타입과 함수를 활용할 수 있다[5][6].

가. Haskell의 반명시적인 병렬 프로그래밍

Haskell에서는 어느 부분이 동시에 수행될 수 있는지 알려주는 'par'와 'seq'를 프로그램에 삽입하여 순차적인 프로그램을 병렬화할 수 있다. par를 통해 프로그램에서 어떤 부분을 동시에 수행시킬 것인가를 알려주면 쓰레드를 생성하고 seq로 결과를 기다려 동기화를 해주게 된다. 아래는 그 타입을 보여준다.

par : : $a \rightarrow b \rightarrow b$ − *parallel composition*
seq : : $a \rightarrow b \rightarrow b$ − *sequential composition*



(그림 2) Haskell 병렬화 분류

순차적으로 작성된 프로그램을 병렬화하는 방식으로 단서로 사용하는 함수를 활용하여 병렬성을 보여주는 이러한 방식은 상태를 공유하지 않고 프로그램 되어있는 함수 프로그램에서 상대적으로 쉬울 수 있다. 그러나 상태를 가져야 하는 계산에는 적합하지가 않다.

나. Haskell의 데이터 병렬성 프로그래밍

데이터 병렬성 프로그래밍은 어레이 계산을 생각해보면 된다. Haskell에서는 대표적으로 Repa(REgular Parallel Arrays) 패키지의 라이브러리를 활용하는데 이는 어레이를 생성하여 병렬로 동작하게 한다. computeP, foldP등은 어레이 병렬 계산을 멀티/매니코어를 활용하여 효율적으로 할 수 있다.

최근 범용 CPU보다 계산에 효율적인 GPU 프로그래밍으로 데이터 병렬성 프로그래밍을 많이 하고 있는데 CUDA와 OpenCL로 CPU와 GPU 메모리 간 데이터 이동을 일일이 프로그램해야 하는 등 저수준의 프로그래밍을 해야 하는 어려움이 있다. Haskell에서는 CUDA 혹은 OpenCL의 코드로 프로그래밍 하지 않고 GPU를 프로그래밍하기 위해 EDSL(Embedded Domain-Specific Language)의 라이브러리를 사용한다. EDSL은 어레이 계산을 Haskell syntax로 GPU 환경에서 가능하게 해주는 엑셀러레이트 라이브러리이다. GPU를 가지지 않은 시스템 환경에서도 계산은 가능하며 그럴 경우 당연히 성능은 GPU 환경에서보다 떨어지게 된다.

다. Haskell의 명시적인 동시성 프로그래밍

Haskell 의 동시성 프로그래밍은 다중 쓰레드를 제어하는 프로그래밍 기술이다. 보다 많은 CPU를 사용하여 프로그램을 더 빨리 실행시키려는 병렬 프로그래밍과는 달리 동시성 프로그래밍은 실생활에서 많이 일어나는 동시적인 행위에 대해 여러 상호작용을 표현하고 실행하는 응용에서 사용된다. 즉, 락(lock)과 컨디션 변수

(condition variable)를 사용하여 여러 쓰레드가 공유한 메모리 자원에 데이터를 읽고 쓰며 접근을 통제하는 멀티쓰레드 프로그래밍이 가장 일반적이다. Haskell에서는 오버헤드를 최소화하는 경량의 쓰레드를 제공하는데 명시적인 동시성 프로그래밍으로 쓰레드를 생성하기 위해 forkIO와 같은 라이브러리 함수를 활용하며 상호작용하는 쓰레드 간 통신을 위해서는 자체적으로 동기화 기능이 제공되는 MVars와 같은 공유 변수를 사용한다.

또한, Haskell은 락 기반의 멀티쓰레드 프로그래밍을 대신한 새로운 병렬 프로그래밍 모델로 STM을 지원한다. Haskell의 STM은 DB의 트랜잭션 개념을 가지고 있어 프로그래밍 형태로 여러 사용자가 작업의 상태를 동시에 수정했을 때 정상적으로 동작하도록 한다. 이 경우 어떤 상황에도 작업의 일관성이 없는 상태를 가지지 않도록 처리하게 되어 데드락의 문제가 발생하지 않고 에러가 발생할 때 복구가 자동으로 이루어진다. 트랜잭션의 일반적인 예로 예금의 입출력에서 이 두 작업을 하나의 트랜잭션으로 처리하기 때문에 이 작업은 성공과 실패만 존재하며 실패 시에는 항상 작업 이전의 상태로 안전하게 복구하게 된다. 이와 같이 STM은 실행 도중 다른 쓰레드에 의해 방해 받지 않는 단위로 데이터를 처리하기 위해 DB의 트랜잭션 개념을 병렬 프로그래밍에 적용하였다. Haskell의 STM을 활용한 프로젝트는 현재 200개 이상 진행되고 있는데 대표적인 프로젝트로는 웹 프레임워크인 Happstack[7], chat 프로그램인 Baracuda[8], P2P VPN을 구현한 Scurry 패키지[9] 등이 있다.

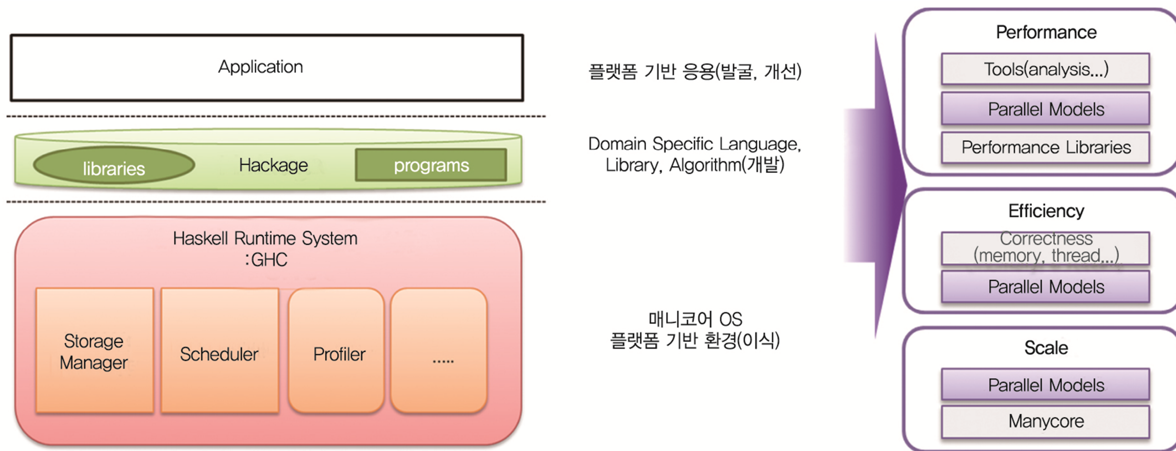
3. Haskell 병렬화 연구방향

병렬 프로그래밍의 핵심은 한 프로그램의 수행에 다중 코어들을 동시에 사용하는 것으로, 운영체제에서 제공하는 쓰레드 관련 라이브러리를 이용하는 것이 필수적이다. 이 라이브러리들로써 병렬 프로그래밍을 할 수는 있으나, 쓰레드에 의한 병렬 제어는 매우 복잡하고 프로그램의 정확성을 유지하기가 힘들다. 병렬 프로그래밍 언어는 프로그래밍 언어의 특징을 이용하여 병렬 프로그래밍을 좀 더 쉽고 안전하게 하는 기능을 제공한다.

가장 이상적인 병렬 프로그래밍은 완전 자동(automatic) 병렬 프로그래밍이다. 이 방법에서는 프로그래머가 병렬성을 표현하지 않더라도 컴파일러가 이를 자동적으로 처리해 주는 것이다. 병렬 프로그래밍의 목적은 스피드 업(speedup)인데, 자동 병렬화에 의한 성공적인 스피드 업은 효과를 얻을 수 없으므로 이 방법은 현실성이 없다고 인식되고 있다.

Haskell에서 함수로써 정의된 프로그램 코드는 효과(effects)를 사용하지 않으므로 병렬성을 갖는다. 그러나 프로그램을 병렬 수행할 때는 쓰레드 fork 등으로 인한 시간이 소모되므로 병렬 처리의 단위(granularity)를 너무 적게 설정하면 이 또한 성공적인 스피드 업을 얻을 수 없다. 한편 모든 컴퓨터의 하드웨어 구성은 동일하지 않을 수 있으며, 컴퓨터에 따라 코어의 수 또한 달라질 수 있다. 이 현상은 병렬 처리 단위 설정에 어려움이 되고 있다. 이 문제와 관련된 현실적인 방법으로서, 프로그램이 fork되는 개수를 프로그램 수행 시 이 수를 옵션으로 정의하도록 프로그래밍 하는 방법이 적용될 수 있다. 사용자는 사용 가능한 코어의 수 등을 고려하면서 프로그램을 병렬로 수행할 수 있다. 이 방법을 반자동(semi-automatic) 병렬화라고 부르며, Haskell에서는 프로그램의 특성에 따라 이 방법을 적용할 수 있다.

Haskell의 병렬 응용을 살펴보면 대규모 계산 응용에서부터 상업용 응용까지 그 시도가 다양하게 이루어지고 있다. 대규모 병렬 처리의 응용분야를 살펴보면 대용량 작업을 위한 병렬 쿼리 실행 및 인덱스 작업의 병렬 처리를 적용한 SQL(Structured Query Language) 서버와 과학용 대규모 계산 응용, 아마존 EC2 클러스터 환경에서 대용량 데이터 처리를 지원하는 MapReduce를



(그림 3) 매니코어 환경의 Haskell 병렬화 방향

Haskell 프로그래밍 모델을 적용하여 시도하고 있다. 분야별로도 교육, 웹 개발, 데이터 관리, 오디오 및 그래픽, 게임 응용까지 응용의 폭 또한 넓어지고 있다.

매니코어 시스템 환경에서는 매니코어를 지원하는 운영체제뿐만 아니라 시스템에서 응용을 실행하고 지원하는 병렬화 환경 또한 무척 중요하다. 매니코어 구조의 고성능 컴퓨팅 환경에서 성능, 효율성, 확장성을 고려할 때 Haskell 병렬 환경은 앞서 정리하였듯이 타 병렬화 프레임워크에 비해 병렬화에 유리한 요소들을 포함하고 있다.

응용의 성능은 컴파일러, 런타임 시스템, OS, 하드웨어 상호작용에 의해 성능이 결정되며 매니코어 환경에서 Haskell 병렬화를 지원하기 위해서는 (그림 3)에서 보듯이 크게 플랫폼 기반 환경을 이식하고 매니코어 기반의 함수 언어 병렬화 모델을 정립하는 것이 필요하다. 그러기 위해서 매니코어 플랫폼 기반의 환경을 이해하는 라이브러리 및 알고리즘을 개발하고 매니코어 기반의 응용을 발굴하여 개선하여야 한다.

IV. 결론

하드웨어의 구조가 멀티코어 및 매니코어 형태로 일반화되면서 병렬 프로그래밍의 문제가 대두되고 있다. 프로그래밍 언어 또한 병렬성을 표현하는 것이 필수적이라고 보여진다. 일반적으로 병렬 프로그래밍은 까다롭고 복잡하여 프로그램의 정확성을 유지하기가 어렵다. 좋은 병렬 프로그래밍 언어는 이러한 문제해결에 어느 정도 도움을 줄 수 있을 것이다.

순수 함수형 프로그래밍 언어는 동적(mutable) 변수를 사용하지 않으므로 병렬 프로그래밍에 적합하다. Haskell은 람다 계산법(Lambda Calculus), 의미론(semantics), 타입 및 증명 이론(type and proof theory) 등의 최신 기술들을 수용하며 발전하고 있는 함수형 언어이다. 지난 약 20~30년의 연구에 의해 여러 우수한 기능을 포함하고 있으며, 최근에 연구되고 있는 병렬 프로그래밍 환경 또한 매우 성공적이라고 보여진다.

기본 함수 언어의 바탕을 충실하게 유지하면서 표현력을 확장할 수 있는 함수 언어인 Haskell은 이론과 경험을 결합하여 매니코어 시대에 대비한 병렬화 모델을 정립하는데 가장 유용한 언어가 될 수 있을 것으로 기대한다. 아직 산업에서 활발하게 활용되고 있지 않으나 Haskell로 생각하는 방법과 표현능력을 확장하여 앞으로 더욱 보편화될 고성능 컴퓨팅 시대에 병렬 프로그래밍 사고의 틀을 넓혀갈 수 있을 것이다.

용어해설

함수형 프로그래밍 명령형 프로그래밍의 변수값을 동적으로 변화시키는 계산방법과는 대조적으로 람다 계산법의 원리를 기반으로 하여, 어떤 한 식의 값은 일정하므로 한 식을 같은 값을 갖는 다른 식으로 치환(substitution)하는 방식의 계산을 적용함. 고차 함수를 적용하여 함수의 표현이 자유로우며, 잘 정립된 타입 이론을 적용함으로써 자료구조에 대한 풍부한 표현력을 가지며 프로그램의 안전성이 높음.

멀티코어/매니코어 멀티코어는 하나의 CPU 다이에 여러 개의 CPU를 집적하고 각각의 코어 간 메모리를 공유하여 모든 코어가 동일한 메모리 영역을 공유함. 매니코어는 수십~수천개의 코어를 하나의 CPU에 집적하고 분산/공유 메모리 하이브리드 방식으로 코어의 작은 그룹은 메모리를 공유하나 그룹과 그룹 간에는 메모리를 공유하지 않으므로 메모리를 공유하지 않은 코어 간에는 메시지로 데이터를 송/수신해야 함.

약어 정리

CUDA	Compute Unit Device Architecture
EDSL	Embedded Domain-Specific Language
GHC	Glasgow Haskell Compiler
MPI	Message Passing Interface
OpenCL	Open Computing Language
OpenMP	Open Multi-Processing
PGAS	Partitioned Global Address Space
Redex	REDucible EXpression
Repa	REgular Parallel Arrays
RTS	Runtime System
SQL	Structured Query Language
STM	Software Transaction Memory

참고문헌

[1] 김진미, 이재진, 최완, “고성능 컴퓨팅을 실현하는 런타임 시스템 기술 동향,” 전자통신동향분석, vol. 27. no. 6, 2012.

[2] J. Diaz, C. Muñoz-Caro, and A. Niño, “A Survey of Parallel Programming Models and Toold in the Multi and Many-Core Era,” *IEEE Trans. Parallel and Distributed Syst.*, vol. 23, no. 8, Aug. 2012, pp. 1369-1386.

[3] S.P. Jones and S. Singh, “A Tutorial on Parallel and Concurrent Programming in Haskell,” *Lectures Notes in Computer Science, Springer Verlag*, vol. 5832, 2009, pp. 267-305.

[4] S.P. Jones, “Tackling the Awkward Squad: monadic input/output, concurrency, exceptions, and foreign-language calls in Haskell,” Engineering theories of software construction, IOS Press, ISBN 1 58603 1724, 2001, pp. 47-96.

[5] S. Marlow, “Parallel and Concurrent Programming in Haskell,” O’REILLY, 2013.

[6] The Haskell Programming Language, “Haskell for multicores,” 2012. http://www.haskell.org/haskellwiki/Haskell_for_multicores

[7] SeeReason Partners, “Happstack,” 2013. http://happstack.com/page/view-page-slug/1/happstack

[8] Hackage, “Barracuda: An ad-hoc P2P chat program,” 2008. http://hackage.haskell.org/package/Barracuda

[9] Hackage, “Scurry: A cross platform P2P VPN application built using Haskell,” 2009. http://hackage.haskell.org/package/Scurry