Haskell을 이용한 병렬 프로그래밍

부산대학교 ▌김연어
경성대학교 ▌변석우*
부산대학교 ▌우 균*

1. 개 요

무어의 법칙이 무너지면서 고성능 컴퓨터의 개발은 단순히 집적도를 높여 속도를 올리던 시대에서 벗어나 여러 개의 코어를 연결하는 다중코어 아키텍처가 일반화되고 있다. 최근 들어서는 개인용 컴퓨터에서도 4코어 이상의 CPU가 널리 사용되고 있을 뿐만 아니라 스마트폰과 같은 휴대용 장치에서도 다중코어를 이용하는 아키텍처가 일반화되어 사용되고 있다. 또한, GPU에서는 이미 수천 개의 코어를 이용하는 CUDA 프로세서와 같은 구조가 널리 사용되고 있다. 가까운 미래에는 연산 장치를 사용하는 모든 기계에서 코어의 수가 수백에서 수천 개까지 확장되는 매니코어 환경이 다가올 것으로 예상한다.

하지만 매니코어 환경에서 모든 프로그램이 속도 상승의 혜택을 보는 것은 아니다. 오직 병렬 프로그래밍을 통해 작성된 프로그램만이 제 성능을 발휘할 수 있으며, 이외의 프로그램 성능 향상은 미미하다. 즉 매니코어 환경에서 시스템의 모든 성능을 사용하기 위해서 프로그램에서 코드가 병렬로 실행될 수 있도록 제작해주어야 한다.

이처럼 병렬 프로그래밍이 중요해졌지만, 병렬 프로그램이 쉽게 늘어나지는 못하고 있다. 왜냐하면, 기존에 널리 사용되고 있는 C나 Java와 같은 명령형 프로그래밍 언어를 병렬화하는 것이 어렵기 때문이다. 프로그램을 병렬화하기 위해서는 제어 흐름을 이해하여 병렬화할 수 있는 모듈을 선정하여야 한다. 하지만 명령형 언어에서는 제어 흐름의 순서가 정해져 있고, 부수효과(side-effect)로 인해 제어 흐름이 더욱 복잡해질 수도 있다.

최근에는 이러한 문제점을 해결하기 위한 방안 중 하나로 함수형 언어인 Haskell이 관심을 받고 있다. Haskell은 순수한 함수형 언어로 변경 가능한(mutable) 변수 사용을 제한하고 부수효과를 없앤 언어이다. 또한, Haskell은 모나드(monad)를 통해 순수/비순수 함수를 분리하기 때문에 제어 흐름이 명확해져 병렬화에 유리하다 볼 수 있다. 이외에도 여러 병렬 프로그래밍 모델을 제공하여 다양한 프로그래밍 환경에 적절히 사용할 수 있다.

이 글에서는 Haskell을 이용한 병렬 프로그래밍 방법을 소개하고자 한다. 이를 위해 먼저 2장에서 Haskell이 지닌 병렬 프로그래밍의 특징에 관해 논한다. 그리고 3장에서는 예제를 통해 `Eval` 모나드를 이용한 Haskell의 병렬 프로그래밍(parallel programming) 방법을 소개한다. 이후 4장에서 예제를 통해 Haskell의 동시성 프로그래밍(concurrency programming) 방법을 소개한다. 그리고 5장에서는 분산 메모리 환경에서 Haskell의 병렬 프로그래밍 방법을 소개한다. 마지막으로 6장에서 결론을 맺는다.

2. Haskell 병렬 프로그래밍의 특징

Haskell은 순수한 함수형 언어로 지연 계산(lazy evaluation)을 기반으로 한 프로그래밍 언어이다. Haskell에서 함수는 순수성으로 인해 기본적으로 계산 순서가 정해져 있지 않고, 부수효과가 없기 때문에 병렬 프로그래밍에 적합하다 볼 수 있다. 하지만 입출력이나 상태 변화와 같은 부수효과가 필요한 상황을 프로그래밍하기 위해서는 이러한 특징은 제약사항으로 다가올 수 있다.

Haskell에서는 입출력과 같은 부수효과가 필요한 상황을 모나드를 이용해 해결하고 있다. 모나드는 부수효과를 제공하기 위해 Haskell에서 제공하는 일종의 타입으로 모나드 내부에서는 계산 순서가 정해져 있다. Haskell에서는 입출력을 위한 `IO` 모나드나 상태

* 종신회원

† 본 연구는 미래창조과학부의 SW컴퓨팅산업원천기술개발 사업의 일환으로 수행하였음(B0101-17-0644, 매니코어 기반 초고성능 스케일러블 OS 기초연구).

저장을 위한 ST 모나드 등 상황에 적합한 모나드를 제공하고 있다. 이외에도 상황에 따라 사용자가 직접 새로운 모나드를 정의하여 사용할 수 있다.

Haskell은 부수효과가 없는 함수와 모나드를 타입 시스템을 통해 분리하여 순수성과 결정성(determinism)을 유지하고 있어 병렬 프로그래밍에 적합하다. 이러한 특성은 한 프로그램에서 함수와 모나드를 혼합하여 작성하여도 그대로 유지된다. 특히 타입 시스템에 의해 순수성과 결정성이 유지되기 때문에 컴파일 시간에 특정 함수가 병렬로 실행될 수 있는지 없는지를 판단할 수 있어 병렬 프로그래밍에 유리하다[1].

이외에도 Haskell에서는 언어 수준에서 경량화 스레드(lightweight thread)를 제공하고 있다. 최근 프로그래밍 언어에서는 언어 수준에서 경량화 스레드를 제공하여 동시성을 높이고 있는데 대표적인 예로 Haskell, Go, Erlang이 있다. 특히 Haskell에서는 다른 언어의 경량화 스레드보다 더 가벼운 스레드를 제공하여 문맥 교환(context switch)의 오버헤드가 매우 작기 때문에 병렬화에 더욱 적합하다[2].

Haskell의 병렬 프로그래밍을 이해하기 위해서는 병렬성(parallelism)과 동시성(concurrency)을 분리해서 이해해야 한다. 많은 분야에서 병렬성과 동시성을 같은 의미로 사용하고 있지만, Haskell 병렬 프로그래밍에서는 그렇지 않다. 병렬성은 어떻게 나누면 더 빠르게 실행할 수 있을지 그 방법에 관한 내용을 다룬다. 하지만 동시성은 동시에 실행되는 스레드를 다루는 방법에 관한 내용을 다룬다. 즉 병렬성은 효율성과 관련이 있으며, 동시성은 독립적인 스레드 간의 상호 작용과 관련이 있다.

3. Eval 모나드를 이용한 병렬화 방법

Eval 모나드를 이용하는 방법 공유 메모리를 기반으로 제어 흐름을 병렬화하는 프로그래밍 모델이다. Eval 모나드는 병렬화를 수행하고 싶은 수식에 특정한 계산 전략(evaluation strategy)을 적용하고 이렇게 변환된 수식은 실행 중에 스파크(spark)로 변경된다. Haskell에서 스파크는 병렬로 실행될 가능성이 있는 계산 단위이다. 스파크는 경량화 스레드보다 더 작은 단위로 비용이 무척 적게 든다는 특징이 있다. Haskell에서 스파크는 그림 1과 같이 처리된다.

그림 1은 스파크가 실제 CPU에 전달되기까지의 변환 과정을 나타낸 것이다. 스파크는 계산 전략에 의해 생성되어 스파크 풀(spark pool)로 전달된다. 그리고 전달된 스파크는 프로그램 실행 도중 RTS(run-time system)의 상황을 보고 경량화 스레드로 변경될지를

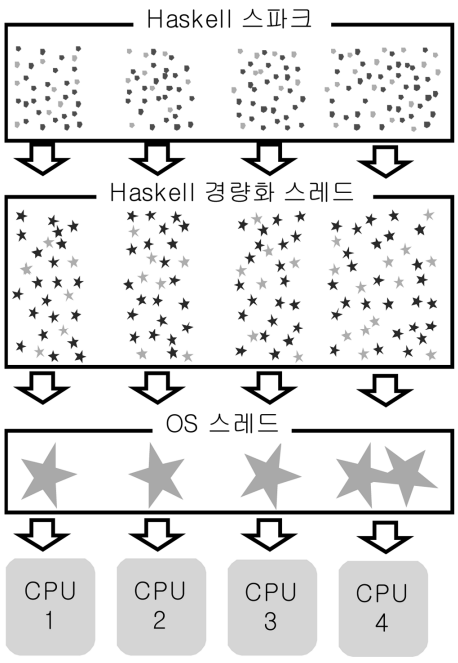

그림 1 Haskell 스파크의 변환 과정

판단하게 된다. 이때 RTS의 상황은 스파크 풀의 가용 여부나 계산 과정에서 스파크화된 수식의 사용 여부 등을 의미한다. 그리고 프로그램 종료까지 계산되지 못한 스파크나 다른 함수에 의해 이미 계산된 스파크의 경우 가비지 컬렉터(GC: garbage collector)에 전달되어 처리된다. 경량화 스레드로 변경된 스파크는 OS 스레드의 가용 여부를 보고 OS 스레드로 전달되어 CPU를 통해 실행되는 과정을 거친다. 그리고 Haskell에서는 크게 3가지 타입의 계산 전략을 제공하고 있으며 각 계산 전략의 타입과 Eval 모나드의 사용 예는 코드 1과 같다.

```
1:  rpar :: Strategy a
2:  rseq :: Strategy a
3:  rdeepseq :: NFData a => Strategy a
4:
5:  test = runEval $ do
6:    a <- rpar f1
7:    b <- rseq f2
8:    return (a+b)
```
코드 1 Haskell의 계산 전략과 Eval 모나드의 사용 예

코드 1과 같이 Haskell에서는 대표적으로 3종류의 계산 전략을 제공하고 있다. rpar는 입력받은 값을 당장 계산하지 않고 스파크만 발생시키는 전략이다. 이는 Haskell의 지연 계산의 특징과 연관된 것으로 값이

필요할 때까지 계산을 연기시키는 전략이다. 그리고 rseq와 rdeepseq는 rpar와 달리 당장 계산을 수행하고 다음 단계로 넘어가는 전략이다. 두 계산 전략의 차이는 rseq는 최상위 정규형(WHNF: weak head normal form) 수준까지 계산하고 rdeepseq는 정규형(normal form) 수준까지 계산한다는 점이다.

최상위 정규형과 정규형은 Haskell의 수식 계산을 위해 사용되는 개념이다. Haskell에서는 수식 계산을 위한 과정을 그래프 축약(graph reduction)을 통해 표현하고 있다. 정규형은 그래프 축약에서 계산할 식(redex)이 더 이상 없는 그래프를 의미하며, 이 경우가 모든 계산이 완료된 상태이다. 하지만 Haskell에서는 최상위에 (:), Just 등과 같은 생성자(constructor)가 오게 되면 계산을 멈추게 되는데 이 경우가 최상위 정규형에 해당한다. 최상위 정규형의 예는 그림 2와 같다.

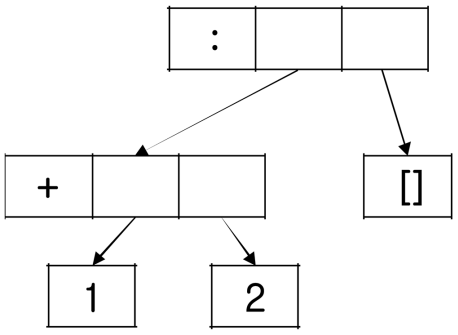

그림 2 (1+2):[]에 대한 그래프 리덕션

그림 1의 그래프는 (1+2):[] 라는 Haskell 수식을 표현한 것이다. Haskell에서는 이와 같은 수식의 경우 (:) 생성자로 인해 수식 트리 최상위의 계산이 완료된 것으로 간주하기 때문에 (1+2)의 값이 필요하기 전까지 더 이상 계산이 진행되지 않는다. Haskell에서는 이와 같은 형태를 WHNF라고 정의하고 있다. 즉, rseq를 사용한 경우 (+) 함수가 바로 실행되지 않기 때문에 병렬화가 정상적으로 이루어지지 않을 수 있다. Haskell에서는 이러한 수식의 형태를 고려하여 계산 전략을 적절히 사용하여야 한다.

Eval 모나드의 강력한 장점 중 하나는 기존에 개발된 프로그램을 병렬로 전환하기 쉽다는 점이다. 이는 Eval 모나드에서 다양한 병렬 고차함수(high-order function)를 제공하기 때문이다. 많은 Haskell 프로그램이 고차함수를 이용하여 구현되어 있어서 Eval 모나드를 사용하면 기존의 고차함수를 병렬 고차함수로 변경하는 것만으로도 병렬화 효과를 누릴 수 있다. 구체적인 예로 코드 2의 피보나치 프로그램을 통해 이를 살펴보자.

```
 1:  fibo :: Int -> Int
 2:  fibo 0 = 0
 3:  fibo 1 = 1
 4:  fibo n = fibo (n-1) + fibo (n-2)
 5:
 6:  main :: IO ()
 7:  main = do
 8:    let list = [1..40]
 9:    let results = map fibo list
10:    print results
```

코드 2 피보나치 프로그램의 예

코드 2는 Haskell로 작성된 피보나치 수를 구하는 프로그램이다. 이 프로그램은 단순하게 구성되어 있는데 fibo 함수는 피보나치 수를 구하는 것이다. 그리고 main 함수는 이를 구동시키기 위한 함수로 우선 1에서 40까지의 수를 담고 있는 list를 생성한다. 그리고 이에 map 함수를 통해 fibo 함수를 적용 시키고 출력하는 예제이다. 이 프로그램을 Eval 모나드를 이용하여 병렬 프로그램으로 다시 작성하면 코드 3과 같다.

```
 1:  import Control.Parallel.Strategies
 2:  fibo :: Int -> Int
 3:  fibo 0 = 0
 4:  fibo 1 = 1
 5:  fibo n = fibo (n-1) + fibo (n-2)
 6:
 7:  main :: IO ()
 8:  main = do
 9:    let list = [1..40]
10:    let results = parMap rseq fibo list
11:    print results
```

코드 3 병렬 피보나치 프로그램의 예

코드 3은 코드 2의 피보나치 수를 구하는 프로그램을 병렬화한 예제이다. 코드 3에서 달라진 점은 1번째 라인에서 계산 전략을 사용하기 위해 패키지를 포함하고 10번째 라인에서 map 함수를 parMap 함수로 변경한 뒤 계산 전략인 rseq를 적용한 것이 끝이다. 이 경우에 변환 과정이 매우 간단하지만 여전히 병렬화로 인한 속도 향상을 기대할 수 있다. 피보나치 프로그램처럼 매우 간단한 프로그램의 경우에

도 실제로 실행해 보면 코드 2는 6.27초의 실행 시간이 걸리는 반면 코드 4는 2.74초 안에 실행되는 것을 확인할 수 있었다(AMD Opteron 6272 멀티코어 프로세서 중 코어 4개 기준).

Eval 모나드에서 계산 전략의 중요성을 알아볼 수 있는 또 다른 예제로는 리만 제타 함수(riemann zeta function)를 들 수 있다. 리만 제타 함수는 독일 수학자 리만이 기존의 제타 함수를 확장한 급수형 함수이다[3]. 리만 제타 함수는 수식 1과 같은 함수다.

$$\zeta(s) = \sum_{n=1}^{\infty} \frac{1}{n^s} \qquad \text{수식 } 1$$

수식 1의 함수는 리만이 쓴 논문[3]에서 소개되었는데 기존 제타 함수와의 차이점은 정의역이 실수에서 복소수로 확장되었다는 것이다. 리만은 이 논문에서 리만 제타 함수와 함께 "리만 제타 함수의 음의 짝수를 제외한 모든 근의 실수부는 1/2이다."라는 수학의 10대 난제로 유명한 리만 가설을 제시했다.

이 함수를 Haskell로 구현하기 위해서는 몇 가지 제약사항이 필요하다. 리만 제타 함수는 무한급수이기 때문에 모든 항을 표현하는 것은 불가능하다. 그래서 근사적으로 주어진 자연수 n에 대해 리만 제타 함수의 1부터 n항 까지를 표현하는 방식으로 이를 구현하고자 한다. 이러한 리만 제타 함수를 Haskell의 병렬 프로그래밍으로 구현하면 코드 4, 5와 같다.

```
1:  import Control.Parallel.Strategies
2:  zetaR :: (Floating a, Integral b) =>
3:    a -> (b,b) -> [a]
4:  zetaR s (x, y) = [ fromIntegral n **
5:    (-s) | n <- [x..y]]
```

코드 4 특정 범위의 리만 제타 함수의 값을 구하는 함수

```
1:  main :: IO ()
2:  main = do
3:    (t, n, s) <- getparam
4:    let ranges = cut n t
5:        results = parMap rdeepseq
6:                  (zetaR s) ranges
7:    print (sum (concat results))
```

코드 5 병렬 리만 제타 함수 실행의 예

코드 4의 zetaR 함수는 앞서 언급한 바와 같이 특정 범위의 리만 제타 함수의 값을 구하는 것이다. 그리고 코드 5는 zetaR 함수를 실행하기 위한 main 함수이다.

표 1. 계산 전략 교체에 따른 리만 제타 함수의 실행 시간

코어 수	rpar	rseq	rdeepseq
1	26.69	26.38	54.03
3	27.45	27.88	22.66
6	29.67	29.24	17.15
10	30.39	31.62	11.62

main 함수에서는 zetaR을 병렬로 실행하기 위해 입력받은 범위를 스레드의 수만큼 분리하도록 하는 cut 함수를 적용한다. 그리고 5~6번째 라인에서 rdeepseq 계산 전략을 이용하여 zetaR 함수를 적용하도록 한 코드이다. 하지만 rdeepseq 계산 전략을 다른 전략으로 변경하게 되면 리만 제타 함수의 실행 시간이 어떻게 변경될까? 계산 전략을 변경해가며 리만 제타 함수를 실행한 결과는 표 1과 같다.

표 1은 리만 제타 프로그램 계산 전략 교체에 따른 실행 시간을 측정한 결과이다. 그리고 리만 제타 함수의 실행값은 5천만으로 하였다. 실행 결과 rpar와 rseq의 경우에는 코어 변화에 따른 실행 시간 감소가 없이 오히려 실행 시간이 늘어난 것을 확인할 수 있다. 그리고 rdeepseq는 코어 1개에서 실행 시간은 다른 두 계산 전략보다 오래 걸리지만, 코어 수가 늘어남에 따라 실행 시간이 줄어드는 것을 확인할 수 있다. 그 이유는 rpar와 rseq의 경우 리스트 생성자인 (:)로 인해 WHNF 형태까지만 계산되어 실제 연산은 나중에 일어났기 때문이다. 그리고 rdeepseq의 경우에는 정규형까지 계산되어 정상적으로 병렬 실행이 되어 실행 시간이 감소하였다. 이외에도 코어 1개에서 실행 시간이 2배가량 차이가 나는 것은 rdeepseq는 즉시 정규형까지 계산되어 가비지 컬렉션에 소모되는 시간이 늘어나 발생한 문제이다.

4. Haskell을 이용한 동시성 프로그래밍 방법

Haskell의 동시성 프로그래밍은 우리가 흔히 아는 명령형 언어의 스레드 프로그래밍과 유사하다고 볼 수 있다. 명령형 언어에서 스레드를 사용하기 위해서는 C의 경우 스레드에서 실행할 함수를 타입에 맞게 선언하고 이를 pthread_create와 같은 함수를 이용하여 스레드를 실행시킨다. Haskell에서도 이와 유사하게 동시에 실행할 함수를 정의하고 이를 스레드를 실행하는 함수를 통해 실행하게 된다. 특히 Haskell에서는 언어 수준의 경량화 스레드와 OS 수준의 스레드와 같이 두 종류의 스레드를 이용하기 때문에 이에 대응하는 Haskell의 함수가 별도로 정의되어 있다. 그리고 스레드 실행 함수와 동기화를 위한 자료구조, 관련 함수도 코드 6과 같이 정의되어 있다.

```
1:  import Control.Concurrent
2:  import Control.Concurrent.MVar
3:  forkIO :: IO () -> IO ThreadId
4:  forkOS :: IO () -> IO ThreadId
5:  data MVar a
6:  newMVar :: a -> IO (MVar a)
7:  takeMVar :: MVar a -> IO a
8:  putMVar :: MVar a -> a -> IO ()
```

코드 6 Haskell 동시성 프로그램을 위한 함수 및 자료 구조

코드 6은 Haskell에서 동시성 함수를 실행하기 위해 사용되는 함수와 자료 구조를 정의한 것이다. Haskell에서 언어 수준의 경량화 스레드를 사용하기 위해서는 forkIO라는 함수를 사용하고 OS 수준의 스레드를 사용하기 위해서는 forkOS를 사용한다. forkIO와 forkOS 둘 다 같은 형태로 사용되지만 forkOS의 경우 Haskell의 RTS의 스케줄러에 영향을 받지 않고 실행된다.

Haskell에서 스레드 사이의 데이터를 주고받기 위해서는 코드 6의 5번째 라인에 정의된 것처럼 MVar 자료 구조를 이용해야 한다. MVar는 Haskell의 컴파일러인 GHC에서 제공하고 있는 변경 가능한 자료 구조이다. 즉, MVar는 이러한 특징을 살려 스레드 사이의 데이터 공유나 동기화를 위해 사용된다.

Haskell에서 MVar를 이용한 동기화는 takeMVar와 putMVar를 이용해 이루어진다. takeMVar는 MVar의 자료를 가져오는 함수로 스택의 pop 명령어와 유사하다. takeMVar 함수는 가져올 자료가 없다면 블록 상태가 되어 가져올 자료가 있을 때까지 대기하게 된다. 그리고 putMVar는 MVar에 자료를 집어넣는 함수이다. putMVar는 이미 자료가 있으면 데이터가 없어질 때까지 블록 상태가 되어 대기한다. Haskell에서는 이와 같은 MVar의 함수를 이용하여 스레드 간의 상태를 동기화할 수 있다.

Haskell의 이러한 동기화 기능을 이용한다면 식사하는 철학자들(dining philosophers) 문제를 풀 수 있다. 식사하는 철학자들 문제는 OS 분야에서 동시성과 교착상태(deadlock)를 설명하기 위해 널리 알려진 문제이다 [4]. 이 문제에서 철학자들은 원형 식탁에 앉아 식사하려고 하며, 철학자의 양옆에는 포크가 한 개씩 존재한다. 그리고 철학자들이 식사를 하기 위해서는 양옆의 포크를 이용해야 한다. 여기서 문제는 양옆의 포크가 공유되는 자원이기 때문에 교착상태에 빠질 수 있다는 점이다. 여기서 문제를 Haskell의 MVar를 이용하여 구현한 자료 구조는 코드 7과 같다.

```
1:  import Control.Concurrent.MVar
2:  type Fork = MVar Int
3:  data Philosopher = Ph String Fork Fork
4:  ph_name = ["Kant", "Marx", "Russel"]
```

코드 7 식사하는 철학자 문제를 위한 자료 구조

코드 7은 식사하는 철학자 문제를 해결을 위해 정의한 Haskell 자료 구조이다. 식사하는 철학자 문제에서 양옆의 포크는 공유되는 자원으로 동기화가 필요하다. 그러므로 포크를 MVar Int를 통해 구현한다. 그리고 Philosopher라는 철학자 자료 구조를 정의하는데 이는 철학자의 이름과 양옆의 공유되는 포크 두 개로 정의되었다. 그리고 ph_name은 문제에서 사용될 철학자 이름이다. 그리고 각 철학자가 행동하는 동작을 정의한 함수는 코드 8과 같다.

```
1:  runPh :: Philosopher -> IO ()
2:  runPh (Ph name l r) = forever $ do
3:    putStrLn $ name ++ "hungry"
4:    left <- takeMVar l
5:    right <- takeMVar r
6:    putStrLn $ "Eat" ++ name
7:    ...
8:    putMVar l left
9:    putMVar r right
```

코드 8 철학자의 행동을 정의한 함수

코드 8은 각 철학자의 동작을 정의한 runPh 함수이다. runPh는 우선 각 철학자가 배고프다는 말을 하고 왼쪽과 오른쪽의 포크를 취하려는 행동을 개시한다. 그리고 양쪽의 포크가 취해지면 식사를 시작하고 식사가 종료되면 왼쪽과 오른쪽 포크를 내려놓는 함수이다. 이 함수에서 양쪽의 포크를 취하지 못한다면 takeMVar에 의해 블록 상태가 되므로 철학자 간의 동기화가 이루어져서 코드는 정상적으로 동작한다. 그리고 이를 구동하는 Haskell 코드는 코드 9와 같이 구현된다.

```
1:  main :: IO ()
2:  main = do
3:    forks <- mapM (\i -> newMVar i)
4:             ([1..3] :: [Int])
5:    let forkPairs = solve_deadlock $
6:          zip forks (tail . cycle $ forks)
7:        phs = map (\(n, fs) -> mkPh n fs) $
8:          zip ph_name forkPairs
9:    mapM_ (\ph -> forkIO $ runPh ph) phs
```

코드 9 식사하는 철학자 문제를 실행하는 함수

코드 9는 식사하는 철학자 문제를 실행하는 main 함수이다. 이 함수에서는 공유되는 자원인 포크를 3~4번째 라인에서 MVar를 이용해 생성한다. 그리고 각 철학자가 선택할 포크의 쌍인 forkPairs를 생성한다. 이때 교착상태를 피하기 위해, Dijkstra가 제안한 방법을 구현한 함수인 solve_deadlock을 forkPairs에 적용한다[5]. 이후 7~8번째 라인에서 각 철학자를 정의하고 9번째 라인에서 forkIO 함수를 이용하여 철학자들이 식사를 시작한다. 최종적으로 이를 실행시켜 보면 교착상태 없이 정상적으로 동작하는 것을 확인할 수 있다.

5. 분산 환경에서 Haskell을 이용한 병렬 프로그래밍

Haskell은 다중코어와 같은 공유 메모리상의 병렬 프로그래밍뿐만 아니라 네트워크로 연결된 분산 환경을 위한 병렬 프로그래밍 방법도 제공하고 있다. 기존 명령형 언어에서 널리 사용되고 있는 MPI의 Haskell 버전에서부터 Haskell만의 독자적인 라이브러리와 같이 분산 환경을 위해 다양한 병렬 프로그래밍 방법을 제공하고 있다. 다양한 분산 환경을 위한 라이브러리 중 가장 유명한 라이브러리로는 Cloud Haskell이 있다.

Cloud Haskell은 Well-typed에서 제공하고 있는 클라우드 컴퓨팅을 위한 Haskell 플랫폼으로서 Erlang의 액터 모델을 기반으로 제작된 방법이다[6,7]. 액터 모델은 각 액터가 고유의 메모리 공간을 가지고 메시지를 통해 액터 사이의 통신이 이루어지는 모델이다. 즉, 액터 간에는 공유하는 메모리가 존재하지 않아서 각 액터는 동시에 실행되어도 문제가 없으므로 동시성이 높은데, 액터 모델은 주로 Erlang에서 사용되고 있다[8].

Erlang은 ATM 시스템에서 주로 사용되던 프로그래밍 언어로 기존 프로그래밍 언어 중 동시성이 뛰어난 것으로 알려져 있다. Cloud Haskell은 Erlang과 같은 높은 동시성을 확보하기 위해 액터 모델을 기반으로 하고 있다. 더욱이 Haskell은 Erlang보다 가벼운 스레드를 제공하고 있어서 성능 측면에서 더 유리하다고 볼 수 있다.

앞의 문단에서 언급한 바와 같이 Cloud Haskell의 액터 사이의 통신은 모두 메시지를 통해서만 이루어진다. 이때 메시지는 Serializable을 상속받아 정의되어야 하는데 Serializable은 Binary와 Typeable 타입이어야 한다. Binary의 경우 메시지를 bytestring 형태로 변경하는 데 필요한 타입이며 Typeable의 경우 다른 액터에서 전송받은 데이터의 타입을 확인하는 데 필요하다. 이외에도 Cloud Haskell에서는 변경 가능한 변수의 경우 메시지의 타입으로 사용할 수 없는 특징이 있다.

Cloud Haskell은 다양한 분산 메모리 환경에서 동작하기 위해 Process 모나드를 이용한 전단부와 실제 네트워크 연결을 위한 후단부로 나뉘어 동작한다. 특히 후단부 라이브러리로는 LAN(local area network)을 위한 simplelocalnet, 공용망을 위한 p2p, Microsoft Azure를 위한 azure를 제공하고 있다. 즉, Cloud Haskell은 상황에 적합한 후단부를 선택하여 프로그래밍할 수 있다.

Cloud Haskell에서는 마스터/슬레이브 구조의 프로그래밍 모델을 제공하고 있다. 마스터/슬레이브 구조는 마스터에서 슬레이브로 작업을 보내어 처리하는 구조로 분산 메모리 환경에서 작업을 처리하는데 적합하다. 이 글에서는 피보나치 수열의 합을 구하는 프로그램을 통해 Cloud Haskell의 동작 과정을 설명하고자 한다. 이를 위한 마스터 노드의 프로그램은 코드 10과 같다.

```
1:  master :: Integer -> [NodeId] -> Process [Integer]
2:  master n slaves = do
3:    us <- getSelfPid
4:    sp <- forM slaves $ \id -> spawn id
5:            ($(mkClosure 'slave) us)
6:    let target = forM_ (zip [1..n] (cycle sp))
7:    spawnLocal $ target $ \(m, them) -> send them m
8:    mergeFiboList (fromIntegral n)
```

코드 10 피보나치 수열의 합을 구하기 위한 마스터 노드 코드

코드 10은 Cloud Haskell에서 마스터 노드에서 실행하는 master 함수이다. master 함수의 4~5번째 라인에서는 슬레이브 노드에서 실행될 slave 함수를 mkClosure 함수를 통해 직렬화(serialization)한다. 이는 Cloud Haskell에서 함수를 전달하는 방법으로 함수를 다른 노드에서 실행하기 위해서는 환경 정보도 같이 전달되어야 하기 때문에 직렬화를 수행한다. 그리고 7번째 라인에서 send 함수를 이용해 슬레이브 노드로 실제 메시지를 전달한다. 이후 mergeFiboList 함수에서 슬레이브 노드로부터 메시지를 전달받아 피보나치 수열의 값을 전달받아 합치는 작업을 수행한다. 그리고 이를 위한 슬레이브 노드는 코드 11과 같다.

```
1:  slave :: ProcessId -> Process ()
2:  slave them = forever $ do
3:    n <- expect :: Int
4:    send them (fibo n)
```

코드 11 피보나치 수열의 합을 구하기 위한 슬레이브 노드 코드

코드 11은 슬레이브 노드에서 실행되는 slave 함수를 정의한 것이다. slave 함수는 expect 함수를 이용하여 메시지를 전달받는다. 이때 expect 함수는 전달받은 메시지의 타입을 확인하는데 코드 11과 같은 경우 Int 이외의 타입이 메시지로 전달되면 무시하게 된다. 그리고 전달받은 메시지에 fibo 함수를 적용해 마스터 노드로 전달하는 단순한 함수이다. 그리고 피보나치 수열의 합을 구하는 프로그램을 실행시키는 main 함수는 코드 12와 같다.

```
1:  main :: IO ()
2:  main = do
3:    args <- getArgs
4:    case args of
5:      ["master", host, port, n] -> do
6:        backend <- initializeBackend host port rtable
7:        startMaster backend $ \slaves -> do
8:          result <- master (read n) slaves
9:          liftIO $ print result
10:     ["slave", host, port] -> do
11:       backend <- initializeBackend host port rtable
12:       startSlave backend
```

코드 12 피보나치 수열의 합을 실행하기 위한 코드

코드 12는 Cloud Haskell 프로그램을 구동시키기 위한 main 함수이다. 이 함수는 실행 시 인자를 통해 현재 실행하는 프로그램이 마스터 역할을 수행할지 아니면 슬레이브 역할을 수행할지를 구분한다. 마스터 노드의 경우 IP 정보와 포트뿐만 아니라 구하고자 하는 피보나치 수열의 합의 범위를 입력으로 받는다. 이 프로그램은 같은 네트워크상에서 슬레이브를 먼저 실행한 뒤 마스터 노드를 실행하면 정상적으로 실행되는 것을 확인할 수 있다.

5. 결 론

이상에서 Haskell이 제공하는 세 가지 병렬 프로그래밍 모델을 소개하였다. 첫 번째는 공유 메모리 환경에서 병렬성에 주안점을 둔 Eval 모나드를 이용한 방법이다. 두 번째는 공유 메모리 환경에서 동시성에 주안점을 둔 MVar를 이용한 방법이다. 마지막으로 세 번째는 분산 환경을 위한 병렬 프로그래밍 모델인 Cloud Haskell이다.

그렇다면 구체적으로 어떤 상황에서 어떤 프로그래밍 모델을 사용해야 할까? Eval 모나드는 다양한 계산 전략과 병렬 고차 함수를 제공하기 때문에 기존 프로그램을 병렬 프로그램으로 변경하기에 적합하다. MVar를 이용한 동시성 프로그램은 Haskell의 경량화 스레드의 장점을 살려 많은 스레드가 필요한 응용 분야에 적합하다. 그리고 Cloud Haskell은 빅데이터 분산 처리와 같은 응용 분야에 활용할 수 있다.

Haskell이 제공하는 함수형 언어 개념은 다소 생소할 수 있지만, 병렬 프로그래밍 관점에서만 보면 활용 가능성이 매우 높다. 성능 측면에서는 명령형 언어나 다른 함수형 언어보다 가벼운 스레드를 제공하고 다양한 병렬 모델을 제공하고 있다. 그리고 개발 편의성 측면에서는 강력한 타입 시스템으로 버그를 줄여주며, 코드 간결성이 뛰어나 유지보수에 드는 비용을 줄여준다.

마지막으로 이 논문에서 사용한 예제 코드는 https://github.com/woogyun/ParallelHaskell에 공개되어 있다.

참고문헌

[1] S. Marlow, Parallel and Concurrent Programming in Haskell: Techniques for Multicore and Multithreaded Programming, 1st Ed., O'Reilly Media, 2013.

[2] S. Marlow, S.P. Jones, and S. Singh, "Runtime support for multicore Haskell," ACM Sigplan Notices, Vol. 44. No. 9. pp. 65-78, 2009.

[3] E. C. Titchmarsh and D. R. Heath-Brown, The theory of the Riemann zeta-function, Oxford University Press, 1986.

[4] A. Silberschatz, P. B. Galvin and G. Gagne, Operating system concepts, 8th Ed., Addison-wesley, 2009.

[5] E. W. Dijkstra, "Hierarchical ordering of sequential processes," Acta informatica, Vol. 1, No. 2, pp. 115-138, 1971.

[6] J. Epstein, A. P. Black and S. Peyton-Jones, "Towards Haskell in the cloud," ACM SIGPLAN Notices, Vol. 46. No. 12. pp. 118-129, 2011.

[7] J. Armstrong, R. Virding, C. Wikstrom, and M. Williams, Concurrent programming in ERLANG, Prentice Hall, 2nd Ed., 1996.

[8] C. Hewitt, P. Bishop and R. Steiger, "A universal modular

actor formalism for artificial intelligence," In Proceedings of the 3rd international joint conference on Artificial intelligence, pp. 235-245, 1973.

약 력

김 연 어

2010 동아대학교 컴퓨터공학과 졸업(학사)
2012 동아대학교 컴퓨터공학과 졸업(석사)
2012~현재 부산대학교 전자전기컴퓨터공학과 박사과정
관심분야: 프로그래밍 분석, 정적 분석, 표절 검사, 함수형 언어
Email : yeoneo@pusan.ac.kr

변 석 우

1980 숭실대학교 전자계산학과(학사)
1982 숭실대학교 전자계산학과(석사)
1982~1999 ETRI 책임연구원
1995 Univ. of East Anglia (영국), Computer Science (Ph.D.)
1999~현재 경성대학교 컴퓨터공학부 교수
관심분야: 함수형 프로그래밍, 정형 증명, 프로그래밍 언어
Email : swbyun@ks.ac.kr

우 균

1991 한국과학기술원 전산학(학사)
1993 한국과학기술원 전산학(석사)
2000 한국과학기술원 전산학(박사)
2000~2004 동아대학교 컴퓨터공학과 조교수
2004년~현재 부산대학교 전자전기컴퓨터공학과 교수
관심분야: 프로그래밍언어 및 컴파일러, 함수형 언어, 그리드컴퓨팅, 소프트웨어 메트릭, 프로그램 분석, 프로그램 시각화
Email : woogyun@pusan.ac.kr