ISSN 2383-630X(Print) / ISSN 2383-6296(Online)
Journal of KIISE, Vol. 44, No. 8, pp. 791-802, 2017. 8
https://doi.org/10.5626/JOK.2017.44.8.791

Haskell Eval 모나드와 Cloud Haskell 간의 성능 비교

(Performance Comparison between Haskell Eval Monad and Cloud Haskell)

김 연 어 † (Yeoneo Kim)　안 형 준 † (Hyungjun An)　변 석 우 †† (Sugwoo Byun)　우 균 ††† (Gyun Woo)

요 약 최근 CPU 시장은 단일 코어의 속도 상승에서 코어의 수를 늘려가는 방향으로 변하고 있다. 이러한 상황에서 매니코어 프로세서의 자원을 최대한 사용할 수 있는 병렬 프로그래밍에 관한 관심이 높아지고 있다. 이 논문에서는 병렬 프로그래밍에 적합한 Haskell을 이용하여 매니코어 환경에 적합한 병렬 프로그래밍 모델을 확인하고자 한다. 이를 위해 이 논문에서는 Eval 모나드와 Cloud Haskell을 이용하여 표절 검사 병렬 프로그램과 K-평균 병렬 프로그램을 개발하였다. 그리고 개발된 프로그램을 대상으로 32코어 환경, 120코어 환경에서 성능을 측정하였다. 측정 결과 적은 코어 수에서는 Eval 모나드가 유리한 것으로 나타났다. 하지만 코어 수가 늘어남에 따라 Cloud Haskell이 실행 시간 기준으로 37%, 확장성 기준으로 134% 더 우수한 것으로 나타났다.

키워드: 하스켈, 병렬 프로그래밍, 병렬 프로그래밍 모델, 매니코어, 표절 검사 프로그램, 액터 모델

Abstract Competition in the modern CPU market has shifted from speeding up the clock speed of a single core to increasing the number of cores. As such, there is a growing interest in parallel programming to maximize the use of resources of many core processors. In this paper, we propose parallel programming models in Haskell to find an advisable parallel programming model for many-core environments. Specifically, we used Eval monad and Cloud Haskell to develop two versions of parallel programs: plagiarism detection and K-means. Then, we evaluated the performance of the developed programs in 32-core and 120-core environments. The results of our experiment show that the Eval monad is highly efficient in an environment with a small number of cores. On the other hand, the Cloud Haskell runtime shows 37% improvement over Eval monad and the scalability shows a 134% improvement over Eval monad as the number of cores increases.

Keywords: Hasekll, parallel programming, parallel programming model. many core, plagiarism detection program, actor model

· 이 논문은 2017년도 정부(과학기술정보통신부)의 재원으로 정보통신기술진흥센터의 지원을 받아 수행된 연구임(No.B0101-17-0644,매니코어 기반 초고성능 스케일러블 OS 기초연구)
· 이 논문은 제43회 동계학술발표회에서 '병렬 프로그래밍 모델에 따른 Haskell 병렬 프로그램의 성능 비교'의 제목으로 발표된 논문을 확장한 것임

† 학생회원 : 부산대학교 전기전자컴퓨터공학과
yeoneo@pusan.ac.kr
hyungjun@pusan.ac.kr
†† 종신회원 : 경성대학교 컴퓨터공학과 교수
swbyun@ks.ac.kr
††† 종신회원 : 부산대학교 전기전자컴퓨터공학과 교수
LG전자 스마트 제어 센터
(Pusan Nat'l Univ.)
woogyun@pusan.ac.kr
(Corresponding author임)

논문접수 : 2017년 1월 18일
(Received 18 January 2017)
논문수정 : 2017년 5월 31일
(Revised 31 May 2017)
심사완료 : 2017년 5월 31일
(Accepted 31 May 2017)

정보과학회논문지 제44권 제8호(2017. 8)

1. 서 론

CPU 시장에서 단일 코어의 성능이 발열 문제로 한계에 봉착함에 따라 최근 개발되고 있는 CPU는 코어 수를 늘려가는 방법을 택하고 있다. 즉, 예전과 같이 CPU의 성능 향상을 통해 소프트웨어 성능이 향상되는 시대가 끝났다고 보아도 과언이 아니다. 이러한 컴퓨팅 환경 변화로 자원을 어떻게 하면 효과적으로 사용할 수 있을지가 중요해졌고 이에 따라 병렬 프로그래밍에 관한 관심이 높아지고 있다.

이러한 흐름에 따라 언어 수준에서 병렬화를 지원하는 프로그래밍 언어가 주목받고 있다. 대표적인 병렬화를 지원하는 프로그래밍 언어로는 주로 전화교환기 시스템에서 사용되고 있는 Erlang이 있다. 이외에도 최근에 개발된 Go나 Rust에서도 병렬화를 지원하고 있다.

병렬화를 지원하는 프로그래밍 언어 중 특히 Haskell은 병렬 프로그래밍에 적합하다고 볼 수 있다. Haskell은 순수한 함수형 언어로 부수효과(side effect)가 없으며, 계산 순서가 정해져 있지 않기 때문에 병렬 프로그래밍에 유리하다. 또한, Haskell은 병렬 성능이 뛰어나다고 알려진 Erlang보다 더 가벼운 경량화 스레드(lightweight thread)를 제공하여 문맥 교환(context switch)의 오버헤드가 무척 작다. 따라서 병렬화에 더욱 적합하다 볼 수 있다[1]. 이외에도 다양한 병렬 프로그래밍 모델을 제공하고 있어 상황에 맞는 모델을 선택하여 사용할 수 있다.

이 논문에서는 코어가 많은 환경 즉, 매니코어(many core) 환경에 적합한 Haskell의 병렬 프로그래밍 모델의 선정 기준을 살펴보고자 한다. 이를 위해 Haskell에서 대표적으로 사용되고 있는 병렬화 모델인 Eval 모나드와 액터 모델(actor model)을 이용하는 프로그래밍 모델 간의 성능을 비교한다. Eval 모나드는 공유 메모리 환경에서 제어 흐름을 병렬화하는 모델이다. 액터 모델은 각 액터가 고유의 메모리 공간을 가지고 액터 사이의 통신은 메시지를 통해서만 이루어지는 모델로 주로 Erlang에서 사용되고 있다[2]. Haskell은 Cloud Haskell이라는 클라우드 컴퓨팅 모델을 통해 액터 모델을 제공하고 있다.

이 논문에서는 두 모델 간의 성능을 비교하기 위해 같은 병렬 알고리즘을 각 모델을 이용하여 두 개의 응용 프로그램으로 제작한다. 작성한 프로그램은 표절 검사 프로그램과 K-평균(K-means) 프로그램이다. 두 프로그램 모두 데이터 의존적인 프로그램으로서 병렬 성능을 측정하기에 적합한 프로그램이다. 그리고 같은 환경에서 실행 성능을 비교하기 위해 두 모델 모두 공유 메모리 상에서 실행하여 성능을 비교하였다.

이 논문은 다음과 같이 구성되어 있다. 2장에서는 관련 연구로 Haskell의 병렬화 방법에 대해 알아본다. 3장에서는 Eval 모나드와 Cloud Haskell을 이용한 응용 프로그램을 살펴본다. 4장에서는 두 모델을 이용하여 Haskell 병렬 프로그래밍 방법을 정성적으로 비교한다. 그리고 5장에서는 구체적인 매니코어 환경에서 두 병렬 프로그램의 성능을 정량적으로 비교한다. 6장에서는 실험 결과에 대해 토의하고, 마지막으로 7장에서 결론을 맺는다.

2. 관련 연구

2.1 Eval 모나드 병렬 프로그래밍 방법

Haskell은 순수한 함수형 언어로 지연 계산(lazy evaluation)을 기반으로 한 프로그래밍 언어다. Haskell에서는 기본적으로 함수의 계산 순서가 특별히 정해져 있지 않다. 단 모나드(monad)만 예외이다. 즉, 모나드를 제외한다면 제어 흐름을 고려하지 않고 병렬로 실행할 수 있다. 또한, Haskell은 다른 프로그래밍 언어보다 더욱 경량화된 스레드를 사용하기 때문에 문맥 교환의 오버헤드가 작고 따라서 병렬화에 유리하다.

Eval 모나드를 이용하는 방법은 공유 메모리를 기반으로 제어 흐름을 병렬화한다. Eval 모나드는 다른 모나드와 달리 모나드 내부에서 사용하는 병렬화 전략(strategy)에 따라 계산 순서가 결정된다. 그리고 Eval 모나드에서 스레드 간의 통신은 MVar라는 변경 가능한(mutable) 변수를 이용하여 이루어진다[3]. 또한, Eval 모나드를 이용하면 고차 함수(higher-order function)를 병렬 고차 함수로 변경하는 것만으로도 이미 작성된 프로그램을 병렬화할 수 있다는 장점이 있다.

하지만 전체 스레드가 하나의 VM(virtual machine)에서 동작하여 많은 메모리를 사용하는 프로그램은 메모리 재사용(GC: garbage collection) 시간이 오래 걸리는 단점이 있다. 그리고 프로그램 실행 도중 오류가 발생하면 전체 프로그램을 다시 실행해야 하는 단점도 존재한다.

2.2 Cloud Haskell 병렬 프로그래밍 방법

Cloud Haskell은 Well-Typed에서 제공하고 있는 클라우드 컴퓨팅을 위한 Haskell 플랫폼으로서 Erlang의 액터 모델을 기반으로 제작된 방법이다[4, 5]. Cloud Haskell은 분산 메모리 환경에서 동작하는 Process 모나드를 기반으로 프로그램을 작성한다. 이때 각 노드 간의 통신은 메시지를 통해 이루어지며 메시지는 `Binary`, `Typeable` 클래스의 특징을 지녀야 한다. 그리고 기존 프로그램을 Cloud Haskell 버전으로 변경하기 위해서는 Eval 모나드와 달리 분산 실행할 함수를 선택하여 별도의 코드를 작성해야 한다는 단점이 있다.

하지만 Cloud Haskell은 본래 분산 모델을 기반으로 하고 있어 이로 인한 장점이 많다. 가장 큰 장점은 오류에 대한 강건성(fault-tolerant)이다. 즉 어느 한 노드에서 오류가 발생하더라도 오류가 발생한 노드만 다시 실행하면 된다. 또한, RTS(run-time system)도 분산 실행되어 자연스럽게 병렬로 실행된다는 장점이 있는데 특히 RTS의 일부인 GC도 분산 작업 수에 따라 나뉘어서 수행되기 때문에 자연스럽게 확장성이 높아진다는 장점이 있다.

3. Haskell을 이용한 응용 프로그램

이 장에서는 Haskell의 Eval 모나드와 Cloud Haskell을 사용하여 표절 검사 프로그램과 K-평균 프로그램을 개발하고자 한다. 표절 검사 프로그램을 선택한 이유는 데이터 의존적인 프로그램으로 병렬 성능 평가에 적합하기 때문이다. 또한, K-평균 프로그램도 데이터 양에 따라 계산량이 변경되기에 병렬 성능 평가에 적합하다.

3.1 표절 검사 프로그램의 개발

표절 검사 프로그램은 기존의 다양한 표절 검사 기법 중 SoVAC(software verification and analysis center)의 기법을 이용하고자 한다. SoVAC은 프로그램을 DNA라 명하는 특별한 토큰의 형태로 변경하여 표절 여부를 판단하는 방법으로 구조는 그림 1과 같다[6].

SoVAC은 그림 1과 같이 크게 두 가지 모듈로 구성되어 있다. DNA 생성기는 SoVAC의 전단부로 프로그램의 소스코드를 입력받아 이를 DNA 형태로 변환해준다. 그리고 유사도 점수 계산기는 후단부에 해당하며 DNA 생성기에서 생성한 DNA 집합을 대상으로 지역 정렬(local alignment) 알고리즘을 통해 각 DNA 간의 표절 여부를 판단한다.

Haskell을 이용해 SoVAC을 구현하기 위해서는 전·후단부를 Haskell로 작성해야 한다. 그리고 이 논문에서는 전단부의 대상 프로그램을 즉, 표절 검사 대상을 Java 프로그램으로 한정하여 개발한다. 이를 위해 전단부에서는 language-java 파서 라이브러리를 이용한다[7].

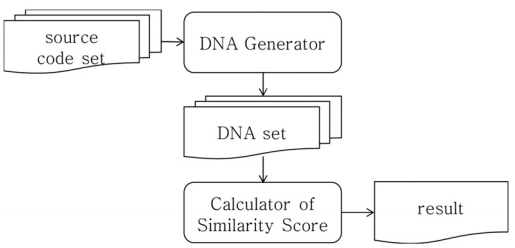

그림 1 SoVAC의 구조
Fig. 1 The overview of SoVAC

그리고 후단부에서는 지역 정렬 알고리즘을 사용하기 위해 align 라이브러리를 이용한다[8].

3.1.1 Eval 모나드를 이용한 표절 검사 프로그램

이 절에서는 Eval 모나드를 이용해 표절 검사 프로그램을 개발한다. Eval 모나드는 공유 메모리 기반이기 때문에 별도의 추가 작업 없이 표절 검사 프로그램을 구현할 수 있다. Eval 모나드를 이용하여 개발한 DNA 생성기는 코드 1과 같다.

```
1: genDNA :: String -> IO ([[String]])
2: genDNA dir = do
3:   files <- getDirectoryContents dir
4:   ···
5:   let ast = parMap rpar parseCompilationUnit cont
6:   let result = parMap rpar visit ast
7:   ···
8:   result <- compDNA result_pair
9:   return result
```

코드 1. Eval 모나드를 이용한 DNA 생성기
Code 1. The DNA generator using Eval monad

코드 1의 `genDNA` 함수는 Eval 모나드 버전의 DNA 생성기이다. `genDNA` 함수는 분석할 대상의 위치를 입력으로 받는다. 그리고 3번째 라인에서 입력된 위치의 모든 파일 정보를 가져와 `files`에 저장한다. 이후 5번째 라인의 `parseCompilationUnit` 함수를 통해 입력받은 대상을 파싱하여 AST(abstract syntax tree)를 생성하며 이 과정을 `parMap` 함수를 통해 병렬로 실행한다. `parMap`은 Eval 모나드에서 제공하고 있는 대표적인 병렬 고차 함수로서 `parMap`의 타입 정의는 코드 2와 같다.

```
1: parMap :: Strategy b -> (a -> b) -> [a] -> [b]
2: rseq :: Strategy a
3: rpar :: Strategy a
4: rdeepseq :: NFData a => Strategy a
```

코드 2. parMap 및 병렬화 전략
Code 2. parMap and parallel evaluation strategies

코드 2는 Eval 모나드에서 주로 사용되는 병렬 고차 함수인 `parMap`과 병렬화 전략의 타입 정의이다. `parMap`이 `map` 함수와 다른 점은, 리스트에 적용할 함수 대신 병렬화 전략을 첫 번째 인수로 받는다는 점이다. Eval 모나드의 병렬화 전략은 모나드 내부의 함수가 어떤 순서 혹은 방식으로 계산할지를 정의한 것으로 코드 2의 2~4번째 라인과 같은 전략이 주로 사용된다. `rseq`는 최상위 정규형(WHNF: weak head normal form)까지 계산한 뒤 다음 계산을 진행하는 전략이다. 그리고 `rpar`는 모나드가 반환되기 전까지 계산을 미루는 전략이다. `rdeepseq`는 `rseq`와 유사하지만 정규형

(normal form)까지 계산을 진행하는 전략이다.

코드 1의 나머지 코드는 6번째 라인에서는 **visit** 함수가 AST를 순회하며 각 노드에 맞는 DNA 토큰을 병렬로 생성한다. 마지막으로 8번째 라인에서 DNA 점수 계산기로 DNA 토큰을 넘겨주기 위해 **compDNA** 함수를 호출한다. 그리고 **compDNA**는 코드 3과 같이 정의하였다.

```
 1: compDNA :: [(String, String)] -> IO ([[String]])
 2: compDNA pair = return result where
 3:   dna = parMap rpar (\(x,y) -> (tok x), y) pair
 4:   test_set = cartProduct dna dna
 5:   normal_set = M.fromList (parMap rpar
 6:               (\(x,y) -> (y, (local_align x x))) cDNA)
 7:   ···
 8:   simAB = parMap rpar (\((a,b),(c,d)) -> (LA b d)) test_set
 9:   sim = parMap rpar (\(a,b) -> (a/b)) (zip simAB min_set)
10:   ···
11:   result = reverse (sortBy (comp fst) mid)
```

코드 3. Eval 모나드를 이용한 유사도 점수 계산기
Code 3. The calculator of similarity score using Eval monad

코드 3의 **compDNA** 함수는 유사도 점수 계산기의 메인 함수로 DNA를 입력받아 표절 여부를 반환한다. 이를 위해 3번째 라인에서 **tok** 함수를 통해 DNA를 지역 정렬하기 편한 형태로 변환한다. 그리고 4번째 라인에서 **cartProduct** 함수를 이용해 표절 검사를 수행할 검사 쌍을 생성한다. 그리고 검사 쌍의 표절 여부 판단은 식 1의 유사도를 기준으로 판단한다.

$$SIM_{AB} = \frac{LA(A,B)}{\min\{LA(A,A), LA(B,B)\}} \quad (1)$$

식 (1)은 검사 대상 A, B의 유사도를 정규화를 통해 백분율로 나타내는 방법이다. 여기서 $LA(A,B)$ 는 지역 정렬 알고리즘을 통한 A와 B의 유사도 점수를 의미하며 두 대상의 길이가 다를 수 있으므로 정규화하여 계산한다. 이를 구현한 것이 코드 2의 5~9번째 라인이다. 5~6번째 라인에서는 자신의 지역 정렬 점수를 미리 계산한다. 그리고 8번째 라인에서는 검사 쌍에 지역 정렬 알고리즘 함수인 **local_align**을 병렬로 적용한다. 이후 9번째 라인에서 각 검사 쌍에 대한 유사도 백분율 점수를 계산하고 그 값을 11번째 라인에서 정렬하여 반환한다.

3.1.2 Cloud Haskell을 이용한 표절 검사 프로그램

이 절에서는 Cloud Haskell 버전의 표절 검사 프로그램을 설명한다. Cloud Haskell은 네트워크를 이용한 모델이기 때문에 마스터/슬레이브 구조를 채택하고 있다. 따라서 표절 검사 프로그램도 이 구조를 이용하여 정의해야 하는데 Cloud Haskell의 표절 검사 프로그램 구조를 그림으로 나타내면 그림 2와 같다.

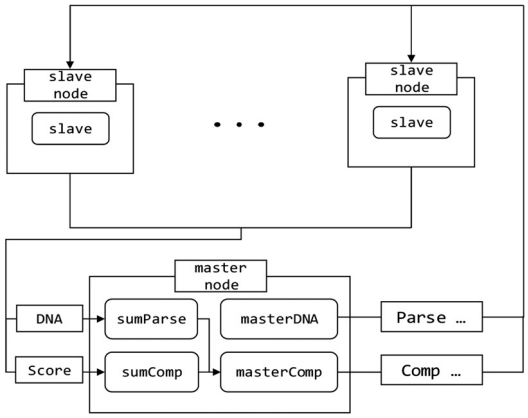

그림 2 Cloud Haskell 프로그램 모델
Fig. 2 The programming model of Cloud Haskell

그림 2에서 둥근 네모를 포함하고 있는 큰 박스는 노드(node)를 나타낸다. 마스터 노드에서는 각 모듈에서 계산량이 많은 연산을 슬레이브 노드에 메시지 형식으로 계산을 전달하고 처리된 값을 받는다. 그리고 슬레이브 노드에서는 마스터 노드에서 전달된 메시지 형식을 보고 그에 알맞은 연산을 수행한다. 이 버전은 Cloud Haskell의 특성을 고려하여 기존 SoVAC과 달리 네 모듈로 정의되었다.

첫 번째 모듈은 마스터 노드에서 DNA를 생성하는 함수로 코드 4와 같다.

```
 1: masterDNA :: String -> [NodeId] -> Process [(String,String)]
 2: masterDNA dir slaves = do
 3:   ast <- liftIO(readParse dir)
 4:   us <- getSelfPid
 5:   slaveP <- forM slaves (\nid -> spawn nid
 6:            ($(mkClosure 'slave) us))
 7:   let msg = map (\(s1,s2) -> Parse s1 s2) ast
 8:   let n = length ast
 9:   spawnLocal (forM_ (zip msg (cycle slaveP))
10:              (\(m, them) -> send them m))
11:   sumParse (fromIntegral n)
```

코드 4. 마스터 노드의 DNA 생성 모듈
Code 4. The DNA generator module of the master node

코드 4의 **masterDNA** 함수는 마스터 노드의 DNA 생성기를 정의한 것이다. **masterDNA**는 Eval 모나드의 DNA 생성기와 같은 알고리즘을 이용해 구현하였다. **masterDNA**의 3번째 라인에서는 입력받은 위치의 파일을 대상으로 AST를 생성한다. 그리고 4번째 라인에서 노드 사이의 통신을 위해 본인의 프로세스 식별자를 가져온다. 이후 5~6번째 라인에서는 슬레이브에 전달할 메시지를 **spawn** 함수를 통해 생성한다. **spawn**은 슬레

이브에서 실행할 클로저(closure)를 생성하는 함수로 코드 5와 같이 정의되어 있다.

```
1: spawn :: NodeId -> Closure (Process ()) -> Process ProcessId
```

코드 5. spawn의 타입 정의
Code 5. The type definition of spawn

`spawn`은 코드 5와 같이 다른 노드의 정보와 다른 노드에서 실행할 클로저 함수를 입력으로 받는다. Cloud Haskell에서는 일반 함수를 템플릿 Haskell의 기능을 이용하여 클로저로 변환한다. 템플릿 Haskell은 안전한 타입의 메타 프로그래밍을 제공한다[9]. 템플릿 Haskell을 이용하면 새로운 문법을 컴파일러 수정 없이 라이브러리 수준에서 제공할 수 있는데 GHC는 이를 확장 기능으로 구현하고 있다. 코드 4의 6번째 라인에서 코드 `$(mkClosure 'slave)`는 컴파일 시 클로저인 `slave_closure`로 변경된다. 이는 `slave` 함수를 다른 노드에서 실행시키기 위해 필요한 것으로 이 클로저는 다른 노드에서 실행되기 위한 정보인 자유 변수(free variable)와 해당 함수의 타입을 변·복조할 수 있는 정보를 가지고 있는다.

코드 4의 나머지 부분을 설명하면 다음과 같다. 7번째 라인에서 AST를 Cloud Haskell의 메시지로 변환하고 8번째 라인에서 보내는 메시지의 전체 길이를 저장한다. 그리고 9~10번째 라인에서 `spawnLocal` 함수와 `send` 함수를 통해 실제 슬레이브로 메시지인 클로저를 전달한다. 이후 11번째 라인에서 슬레이브로부터 생성된 정보를 `sumParse` 함수를 통해 전달받는다.

두 번째 모듈은 마스터 노드의 유사도 점수 계산기 모듈로 코드 6과 같다.

```
1: masterComp :: [(String, String)] -> [NodeId] -> Process [[String]]
2: masterComp pairs slaves = do
3:   ···
4:   let (ns, t_set) = compDNANormal pairs
5:   let set = parMap rseq (\((a,b),(c,d))->
6:             Comp (a,c) (min ns b d) (b,d) ) t_set
7:   ···
8:   mid <- sumComp (fromIntegral n)
9:   return (reverse (sortBy (comp fst) mid))
```

코드 6. 마스터 노드의 유사도 점수 계산기 모듈
Code 6. The calculator of similarity score module of the master node

코드 6의 `masterComp` 함수는 마스터 노드의 유사도 점수 계산을 담당하고 있다. `masterComp`는 DNA 생성 모듈에서 처리된 결과, 즉 DNA 정보와 슬레이브 노드 정보를 입력으로 받는다. 코드 6의 `masterComp` 함수는 기본적으로 `masterDNA` 함수와 구조가 유사하므로 여기서는 메시지를 생성하고 보내는 부분을 생략하였다. 4번째 라인의 `compDNANormal` 함수를 통해 검사 쌍에서 정규화를 적용할 값을 계산하게 된다. 그리고 5~6번째 라인에서 슬레이브에 전달할 메시지를 생성하게 된다. `masterComp` 함수에서는 검사 쌍과 정규화에 사용될 값을 메시지로 전달한다. 이후 8번째 라인에서 `sumComp` 함수를 통해 계산된 결과를 받아 통합하고 9번째 라인에서 이를 정렬한다.

그리고 세 번째 모듈은 마스터/슬레이브 노드 사이에 사용되는 메시지를 정의한 모듈로 코드 7과 같다.

```
1: {-# LANGUAGE DeriveGeneric #-}
2: module PlagMessage where
3: data PlagMessage = Parse String String
4:   | Comp (String, String) Double (String, String)
5:   deriving (Show, Typeable, Generic)
6: instance Binary PlagMessage
```

코드 7. 표절 검사에서 사용되는 메시지 정의
Code 7. The definition of the message used in the plagiarism detection

코드 7의 `PlagMessage`는 표절 검사 프로그램 내부에서 사용되는 메시지를 정의한 것이다. `PlagMessage`는 크게 두 종류로 나뉜다. `Parse`는 DNA 생성기에서 사용하는 메시지이며, `Comp`는 유사도 점수 계산기에서 사용하는 메시지이다. 그리고 정의한 `PlagMessage`를 Cloud Haskell용 메시지로 사용하기 위해 `Typeable`, `Generic`을 유도(deriving)받는다. `Generic`은 `Binary`를 자동으로 생성하기 위한 클래스로 컴파일 시 `PlagMessage`의 `Binary` 인스턴스가 자동으로 생성된다.

마지막 모듈은 슬레이브 노드에서 사용되는 모듈로 코드 8과 같다.

```
1:  slave :: ProcessId -> Process ()
2:  slave them = forever (do
3:    m <- expect :: Process PlagMessage
4:    case m of
5:      Parse filename file_content -> do
6:        let ast = parseCompilationUnit file_content
7:        let result = genDNA ast
8:        send them (result, filename)
9:      Comp (a,b) min name -> do
10:       let simAB = local_align a b
11:       let sim = simAB / min
12:       send them (sim, name))
```

코드 8. 표절 검사에서 사용되는 슬레이브 노드의 모듈
Code 8. The definition of the slave nodes in the plagiarism detection

슬레이브 노드는 마스터로부터 계산할 대상을 받아 처리하고 다시 전송하는 단순한 노드이다. 슬레이브 노드에서 사용되는 slave 함수는 expect 함수를 통해 마스터 노드로부터 메시지를 전달받는다(라인 3). 이때 받는 메시지는 앞서 코드 7에서 정의한 PlagMessage 타입으로 이외의 타입이 전송되면 무시하게 된다. 그리고 받은 메시지를 분석하여 Parse 타입이라면 AST를 생성하고(라인 6) genDNA 함수를 통해 AST를 DNA로 변경한다(라인 7). 그리고 send 함수를 통해 마스터 노드로 결과를 전송한다(라인 8).

그리고 전달받은 메시지의 타입이 Comp 타입이라면 검사 쌍에 지역 정렬 알고리즘 함수인 local_align을 적용하고(라인 10) 이를 정규화한다(라인 11). 그리고 send 함수를 통해 검사된 백분율 값을 전송한다(라인 12).

3.2 K-평균 프로그램의 개발

K-평균 프로그램은 주어진 데이터를 K개의 클러스터로 나누는 프로그램이다. K-평균 프로그램은 우선 초기에 주어진 K개의 중심점을 이용하여 클러스터를 구성한다. 그리고 주어진 클러스터에서 새로운 중심점을 구하고 클러스터를 재구축하는 연산을 클러스터의 변경이 없을 때까지 반복 수행하는 프로그램이다. 즉 이 프로그램 연산이 반복 수행되기 때문에 병렬 모델의 성능을 알아보기에 적합한 프로그램이다.

3.2.1 Eval 모나드를 이용한 K-평균 프로그램

이 항에서는 Eval 모나드를 이용한 K-평균 프로그램에 대해 알아보고자 한다. 이 논문에서는 Eval 모나드를 이용한 K-평균 프로그램의 경우 Simon Marlow가 구현한 버전을 이용하고자 한다[3]. Simon Marlow가 제작한 K-평균 프로그램은 다양한 병렬 방법을 소개하고 있지만, 이 논문에서는 가장 성능이 좋게 나오는 strat를 이용한 방법을 선택한다. Simon Marlow가 구현한 K-평균 프로그램의 핵심 코드는 코드 9, 10과 같다.

```
1: kmeans_strat :: Int -> Int -> [Point] -> [Cluster] -> IO [Cluster]
2: kmeans_strat numChunks nclusters points clusters =
3: let
4:   chunks = split numChunks points
5:   loop :: Int -> [Cluster] -> IO [Cluster]
6:   loop n clusters | n > tooMany = do ... return clusters
7:   loop n clusters =
8:   ...
9:     let clusters' = parSteps_strat nclusters clusters chunks
10:    if clusters' == clusters
11:      then return clusters
12:      else loop (n+1) clusters'
13: in loop 0 clusters
```

코드 9. K-평균 반복 함수의 정의

Code 9. The definition of loop function in the K-means program

```
1: parSteps_strat :: Int -> [Cluster] -> [[Point]] -> [Cluster]
2: parSteps_strat nclusters clusters pointss
3:   = makeNewClusters (
4:       foldl1' combine (
5:         (map (assign nclusters clusters) pointss
6:            `using` parList rseq)))
```

코드 10. 클러스터 계산 함수의 정의

Code 10. The definition of the cluster calculation function

코드 9의 kmeans_strat 함수는 K-평균 프로그램이 실행되는 함수이다. kmeans_strat는 데이터를 일정 크기의 청크(chunk)로 분리하여 청크를 병렬 단위로 이용하는 함수이다. 이 함수는 우선 4번째 라인에서 데이터를 일정 크기의 청크로 분리한다. 그리고 실제 반복이 수행되는 loop 함수를 5번째 라인에서 정의한다.

loop 함수는 클러스터 변경이 없거나 일정 횟수 이상 클러스터 계산이 반복되면 종료하는 함수로 6번째 라인과 같이 반복 수가 tooMany의 값보다 크면 클러스터링이 종료된다. 그리고 반복 계산 수가 tooMany 이하이면 9번째 라인과 같이 parSteps_strat 함수를 이용해 새로운 클러스터 값을 계산한다. 그리고 10~12번째 라인에서 클러스터 변경이 없다면 loop 함수를 종료하고 변경이 있다면 반복 수를 증가하여 loop 함수를 다시 호출한다.

코드 10은 병렬로 클러스터값을 계산하는 parSteps_strat 함수이다. 이 함수는 우선 5~6번째 라인에서 청크의 리스트인 pointss의 각 원소에 점과 클러스터를 할당하는 함수인 assign을 병렬로 적용한다. 이때 리스트에 병렬화 전략을 적용해야 하므로 parList 함수를 이용하여 rseq 전략을 리스트에 적용한다. 4번째 라인에서는 combine 연산을 통해 각 청크에 할당된 결과를 하나로 합친다. 그리고 그 결과를 이용해 3번째 라인에서 새로운 클러스터를 생성한다.

3.2.2 Cloud Haskell을 이용한 K-평균 프로그램

이 항에서는 Cloud Haskell을 이용한 K-평균 프로그램을 개발하고자 한다. 이를 위해 Eval 모나드 버전의 프로그램이 이용되며 전체 구조는 마스터/슬레이브 구조를 사용한다. 그리고 Cloud Haskell용 K-평균 프로그램은 네트워크 오버헤드를 줄이기 위해 각 슬레이브가 미리 청크 정보를 가지고 있도록 설계한다. 즉, 마스터 노드는 계산할 청크 위치만 전송하여 네트워크 사용량을 줄이도록 설계되었다.

마스터 노드에서 동작하는 K-평균 프로그램의 핵심 코드는 코드 11, 12와 같다.

코드 11의 kmeans_cloud 함수는 코드 9의 kmeans_strat 함수를 Cloud Haskell 버전으로 개발한 것이다.

이 함수는 마스터 노드에서 동작하며 Eval 버전과 달리 NodeID를 추가로 받고, 반환 타입이 Process 모나드이다.

```
1: kmeans_cloud :: Int -> Int -> [Point] -> [Cluster] -> [NodeId]
2:                  -> Process [Cluster]
3: kmeans_cloud numChunks nclusters points clusters slaves =
4:   let
5:     ...
6:     loop :: Int -> [Cluster] -> [ProcessId] -> Process [Cluster]
7:     ...
8:     loop n clusters sp = do
9:       clusters' <- cloudSteps nclusters clusters chunks sp
10:    ...
11: in do
12:   us <- getSelfPid
13:   slaveP <- forM slaves (\nid -> spawn nid
14:             ($(mkClosure 'slaveKmeans) (us, numChunks))
15:   loop 0 clusters slaveP
```

코드 11. Cloud Haskell K-평균 반복 함수의 정의
Code 11. The definition of loop function in the K-means program in the Cloud Haskell

```
1: cloudSteps :: Int -> [Cluster] -> [[Point]] -> [ProcessId]
2:                   -> Process [Cluster]
3: cloudSteps nclusters clusters pointss slaveP = do
4:   let n = length pointss
5:   spawnLocal (forM_ (zip [1..n] (cycle slaveP))
6:             (\(m, them) -> send them  (nclusters, clusters, m)))
7:   unsortedRet <- sumPoint n
8:   let sorted = map (\x -> snd x) (sort unsortedRet)
9:   return (makeNewClusters (foldr1 combine sorted))
```

코드 12. Cloud Haskell의 클러스터 계산 함수의 정의
Code 12. The definition of the cluster calculation function in the Cloud Haskell

좀 더 자세히 살펴보면 loop 함수의 정의도 ProcessId를 추가로 받으며 반환 타입도 Process 모나드로 변경된다(라인 6). loop 함수는 기본적으로 Eval 모나드 버전과 동일하게 동작한다. Eval 모나드 버전과 차이점은 클러스터를 계산하는 함수가 parSteps_strat이 아닌 cloudSteps를 호출하는 점이다(라인 6).

kmeans_cloud 함수의 내부도 추가 코드가 들어가게 된다. 우선 노드 사이의 통신을 위해 자기 자신의 프로세스 식별자를 가져온다(라인 12). 그리고 슬레이브 노드에 전달할 메시지를 spawn 함수를 통해 생성한다. 이때 슬레이브 노드에서 파일 처리를 위해 청크의 크기도 같이 전달되게 된다(라인 13~14). 그리고 마지막으로 loop 함수를 호출한다(라인 15).

코드 12의 cloudSteps 함수는 Eval 모나드의 parSteps_strat 함수에 대응되는 것이다. 이 함수는 앞선 코드 11의 loop 함수와 마찬가지로 ProcessID를 추가로 전달받고 반환 타입이 Process 모나드로 변경된다(라인 1~2). cloudSteps 내부에서는 우선 슬레이브에서 청크를 전달하기 위해 청크의 길이를 저장한다(라인 4). 이후 슬레이브 노드로 클러스터 정보와 계산할 청크의 위치를 전송한다(라인 5~6). 그리고 슬레이브 노드에서 계산된 정보를 sumPoint 함수를 이용해 전달 받는다(라인 7). 이후 청크 정보를 정렬하고(라인 8) combine 함수와 mkNewClusters 함수를 통해 새로운 클러스터 값을 생성한다.

슬레이브 노드에서 동작하는 K-평균 프로그램의 핵심 코드는 코드 13과 같다.

```
1: slaveKmeans :: (ProcessId, Int) -> Process ()
2: slaveKmeans (them, numChunks) = do
3:   points <- liftIO $ decodeFile "points.bin"
4:   let chunks = split numChunks points
5:   forever $ do
6:     (ncls, clusters, i) <- expect :: Process (Int, [Cluster], Int)
7:     let pointss = chunks !! (i-1)
8:     let vec = assign ncls clusters pointss
9:     let ret = Vector.toList vec
10:    send them (n, ret)
```

코드 13. K-평균에서 사용되는 슬레이브 노드의 모듈
Code 13. The definition of the slave nodes in the K-means

코드 13의 slaveKmeans는 K-평균 프로그램을 실행하기 위한 슬레이브 노드의 함수이다. 이 함수가 실행되면 우선 모든 노드에서 공통적으로 사용되는 데이터를 읽어와 지정된 크기의 청크로 분리한다(라인 3~4). 이후 expect 함수를 통해 마스터 노드로부터 클러스터의 수, 클러스터 정보, 청크 인덱스 정보가 담긴 메시지가 전달되기를 계속 대기한다(라인 5~6). 그리고 전달받은 메시지에서 청크 정보를 가져오고(라인 7) assign 함수를 통해 해당 청크의 점을 클러스터에 할당한다(라인 8). 이때 assign 함수로부터 생성되는 데이터는 변경 가능한 변수(mutable variable)인 Vector 타입의 값이 반환된다.

하지만 Cloud Haskell에서는 Vector 타입과 같은 변경 가능한 데이터 타입은 메시지로 전송할 수 없다. 이는 변경 가능한 변수는 그 자체가 공유 메모리의 특성을 지니기 때문에 각 노드가 고유한 메모리를 가지는 Cloud Haskell 특성과 상충하기 때문이다. 이를 해결하기 위해 Vector를 리스트로 형태로 변환(라인 9)하여 마스터로 전송한다(라인 10).

4. HasekII 병렬 프로그래밍 방법 비교

이 장에서는 Haskell의 병렬 프로그래밍 방법인 Eval 모나드와 Cloud Haskell을 비교하고자 한다. 두 모델에 대해서는 앞서 2~3장을 통해 자세히 살펴보았다. 이를

표 1 Eval 모나드와 Cloud Haskell의 비교
Table 1 A comparison of Eval monad and Cloud Haskell

	Eval monad	Cloud Haskell
Memory model	shared memory	distributed memory
Data type	Eval monad, MVar	Process monad
How to convert sequential programs	using the parallel high-order functions	rewriting the code
Number of RTSes	1	dependent on the number of nodes
Fault-tolerancy	unsupported	supported

바탕으로 두 모델의 특징을 비교한 결과는 표 1과 같다.

표 1에서 볼 수 있는 것과 같이 Eval 모나드는 공유 메모리를 기반으로 한 병렬화 방법으로서 기존 Haskell 프로그래밍 방법과 거의 유사하게 병렬 프로그래밍을 할 수 있는 방법을 제공한다. 또한 Eval 모나드는 다양한 병렬 전략을 제공하여 상황에 적합한 프로그래밍을 도와준다. 하지만 병렬 실행이 항상 보장되는 것은 아니라는 단점이 있다.

Cloud Haskell은 분산 메모리 기반의 병렬화 방법이지만 공유 메모리 환경에서도 실행할 수 있다. Cloud Haskell은 RTS를 분산 실행하기 때문에 오류에 대한 강건성이 높고 GC의 부하에서 비교적 자유로우며 병렬 함수는 항상 병렬로 실행된다는 장점이 있다. 하지만 기존 프로그램을 변경하기 위해서는 별도의 코드 재작성이 이루어져야 하는 단점이 있다. 또한, 변경 가능한 변수의 경우 Cloud Haskell을 통한 전달이 불가능하기 때문에 공유 메모리 사용에 제한이 있다.

5. 실 험

이 장에서는 Eval 모나드와 Cloud Haskell을 이용하여 개발된 프로그램의 성능을 비교하여 매니코어 환경에 어떤 모델이 적합한지 생각해보고자 한다. 이를 위해 앞선 장에서 개발한 표절 검사 프로그램과 K-평균 프로그램을 대상으로 실험을 진행하고자 한다. 그리고 동일한 조건에서 성능을 평가하기 위해 두 모델 모두 로컬 네트워크 환경에서 실험을 진행한다. 그리고 코어 수에 따른 성능을 분석하기 위해 32코어 환경과 120코어 환경을 대상으로 실험을 진행한다.

Eval 모나드를 이용한 병렬 프로그램과 Cloud Haskell의 메시지 전달을 이용한 병렬 프로그램 모두 Haskell Stack 1.3.2를 통해 제작되었으며 빌드 시 사용된 컴파일러의 버전은 GHC 8.0.1이다. 그리고 유사도 검출에 사용된 대상 프로그램은 2012년 부산대학교 객체지향프로그래밍 수업에서 과제로 제출된 Java 코드를 이용한다. K-평균 알고리즘에 사용된 입력은 좌표 평면상에 임의로 생성된 점을 이용한다.

5.1 32코어 환경

32코어 환경의 실험은 CPU Opteron 6272 2개, RAM 96GB, SSD 256GB, OS는 Ubuntu 14.04.1 LTS에서 진행하였다. 표절 검사 프로그램은 108개의 Java 코드를 입력으로 이용하여 실험을 진행하였다. 그리고 표절 검사 프로그램의 실행 시간과 확장성은 그림 3, 4와 같다.

두 프로그램의 실행 시간은 그림 3과 같이 측정되었다. 그리고 이를 기반으로 확장성 비교를 위해 속도 상승을 계산한 결과는 그림 4와 같다. 실행 시간은 단일 코어의 경우 Eval 모나드가 134.20초로, Cloud Haskell이 172.93초로 측정되었다. 이는 같은 알고리즘을 사용하였지만 Eval 모나드를 이용한 방법이 Cloud Haskell을 이용한 방법보다 빠르다는 것을 뜻한다. 하지만 32코어를 사용한 실행 환경에서는 Eval 모나드가 11.77초, Cloud Haskell이 15.53초로 측정되어 실행 시간의 차이가 많이 줄어든 것을 확인할 수 있다. 이를 기반으로 확

그림 3 32코어 환경에서 표절 검사 프로그램 실행 시간 비교
Fig. 3 The execution time of the plagiarism detection program in 32 cores

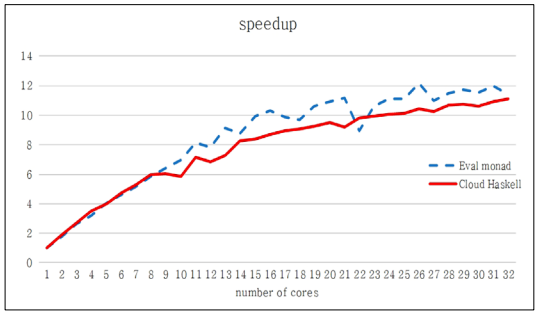

그림 4 32코어 환경에서 표절 검사 프로그램 확장성 비교
Fig. 4 The scalability of the plagiarism detection program in 32 cores

그림 5 32코어 환경에서 K-평균 프로그램 실행 시간 비교
Fig. 5 The execution time of the K-means program in 32 cores

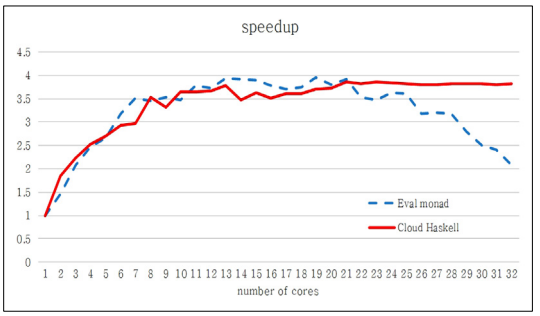

그림 6 32코어 환경에서 K-평균 프로그램 확장성 비교
Fig. 6 The scalability of the K-means program in 32 cores

장성을 계산하면 Eval 모나드의 경우 최대 확장성은 11.96배이며 Cloud Haskell은 11.14배로 나타났다.

32코어 환경에서 K-평균 알고리즘의 성능 평가는 임의로 생성된 32만 개의 점을 입력으로 이용하여 실험을 진행하였다. 그리고 K-평균 알고리즘의 실행 시간과 확장성 그래프는 그림 5, 6과 같다.

두 프로그램의 실행 시간은 그림 5와 같이 측정되었으며 이를 기반으로 계산한 확장성은 그림 6과 같이 측정되었다. 실행 시간은 단일 코어에서 Eval 모나드가 39.61초, Cloud Haskell이 126.19초로 나타났다. 그리고 32코어에서는 Eval 모나드가 19.13초, Cloud Haskell이 32.99초로 나타났으며 최대 확장성은 Eval 모나드가 3.95배, Cloud Haskell이 3.83배로 나타나 표절 검사 프로그램과 유사하게 나타났다. 즉 32코어 환경까지는 Eval 모나드가 유리하다고 볼 수 있다.

5.2 120코어 환경

120코어 환경의 실험은 CPU Intel Xeon E7-8870 v2 8개, RAM 792GB, SSD 1TB, OS는 Ubuntu 15.04에서 진행하였다. 그리고 120코어 환경에서는 32코어 환경과 사양 차이를 고려하여 실험 데이터를 늘릴 필요가 있다. 그렇기 때문에 표절 검사 프로그램은 150개의

그림 7 120코어 환경에서 표절 검사 프로그램 실행 시간 비교
Fig. 7 The execution time of the plagiarism detection program in 120 cores

그림 8 120코어 환경에서 표절 검사 프로그램 확장성 비교
Fig. 8 The scalability of the plagiarism detection program in 120 cores

Java 코드를 대상으로 실험을 진행하였다. 150개 코드를 대상으로 프로그램의 실행 시간과 확장성을 측정한 결과는 그림 7, 8과 같다.

두 프로그램의 실행 시간은 그림 7과 같이, 확장성은 그림 8과 같이 측정되었다. 실행 시간은 단일 코어의 경우 Eval 모나드는 246.86초, Cloud Haskell은 326.05초로 32코어 환경과 유사하게 Eval 모나드가 빠르게 나타났다. 하지만 코어 수가 늘어남에 따라 이 결과는 역전되어 120코어에서 실행 시간은 Eval 모나드가 25.84초로, Cloud Haskell이 10.51초로 나타났다.

이를 기반으로 확장성을 계산한 결과 Eval 모나드의 최대 확장성은 18.65배 정도로 나타났으며, Cloud Haskell의 최대 확장성은 31.02배 정도로 나타났다. 즉 120코어 환경에서는 표절 검사 프로그램을 기준으로 Cloud Haskell의 실행 시간은 59.33% 감소하였으며 확장성은 66.33% 증가한 것을 확인할 수 있다.

K-평균 알고리즘은 임의로 생성된 1,500만 개의 점을 입력으로 이용하였으며 그 결과는 그림 9, 10과 같이 나타났다.

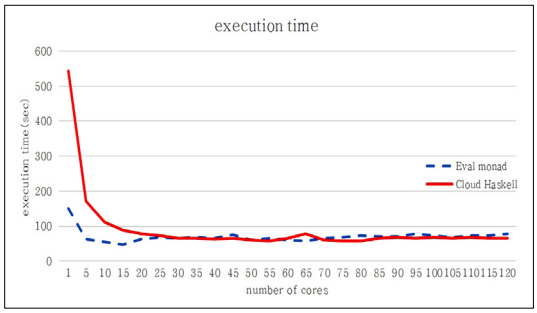

그림 9 120코어 환경에서 K-평균 프로그램 실행 시간 비교
Fig. 9 The execution time of the K-means program in 120 cores

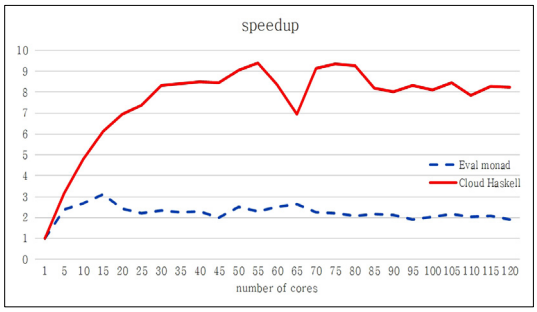

그림 10 120코어 환경에서 K-평균 프로그램 확장성 비교
Fig. 10 The scalability of the K-means program in 120 cores

두 프로그램의 실행 시간은 그림 9와 같이 나타났으며, 확장성은 그림 10과 같이 나타났다. 실행 시간은 단일 코어에서 Eval 모나드가 149.51초, Cloud Haskell이 542.80초로 나타났으며, 120코어에서는 Eval 모나드가 77.61초, Cloud Haskell이 65.79초로 나타났다. 그리고 최대 확장성은 Eval 모나드는 3.12배로, Cloud Haskell은 9.41배로 나타났다. 즉 120코어 환경에서 K-평균 프로그램은 Cloud Haskell을 기준으로 실행 시간은 15% 감소하였으며, 확장성은 201.60% 증가하였다.

5.3 실험 결과 분석

전체 실험 결과 코어 수가 32코어 정도로 적은 환경에서는 Eval 모나드가 우수한 것으로 나타났다. 그러나 코어 수가 많은 환경에서는 Cloud Haskell이 우수한 것으로 나타났다. 구체적으로 살펴보면 표절 검사 프로그램을 기준으로 Cloud Haskell 프로그램이 32코어에서는 Eval 모나드 버전보다 약 4초 정도 느리게 나타났지만 120코어 환경에서는 반대로 Eval 모나드 버전보다 15초 정도 빠르게 나타났기 때문이다. K-평균 알고리즘도 32코어에서는 약 12초 정도 느리게 나타났지만 120코어에서는 약 11초 정도 빠르게 나타났기 때문이다. 즉 120코어를 기준으로 실행 시간은 37%가 감소되었으며, 확장성은 134% 증가하였다.

또한, Eval 모나드는 코어 수가 적을 때 확장성은 보장되지만, 코어 수가 늘어나면 확장성이 보장되지 못한다. 반면에 Cloud Haskell은 코어가 증가함에 따라 꾸준히 확장성이 보장되는 것으로 나타났다. 즉 코어가 많은 환경에서는 공유 메모리 환경일지라도 Cloud Haskell과 같은 메시지 전달 모델을 사용하는 것이 유리하다.

6. 고 찰

응용 프로그램을 이용하여 실험한 결과 Eval 모나드의 경우 32코어 환경과 달리 120코어 환경에서 성능 저하 문제가 관측되었다. 이 문제를 해결하기 위해 Eval 모나드를 사용한 두 응용 프로그램을 프로파일링한 결과 그림 11과 같이 GC 시간이 증가하는 것을 관측할 수 있었다.

그림 11은 120코어 환경에서 Eval 모나드로 작성한 응용 프로그램의 GC 소요 시간이다. 표절 검사의 경우 GC의 소요 시간은 처음에는 2초 이내로 전체 실행 시간의 0.7%밖에 차지하지 않지만 40코어 환경에서부터 그 시간이 점차 늘어나 120코어에서는 전체 실행 시간의 40%를 차지하게 된다. K-평균 프로그램의 경우에도 처음에는 GC의 소요 시간이 4.84%을 차지하지만 120코어에서는 50.65%를 차지하게 된다. 물론 실행 시간이 점차 줄어들어 GC의 비중이 높아지는 것도 있지만, 실제 GC 시간의 증가도 무시할 수 없다. 32코어 환경의 실험에서도 특정 코어 수에서 일부 코어에서 성능 저하 문제가 관측되었는데 이 또한 GC의 문제로 판단된다[10].

Eval 모나드의 이러한 GC 문제는 하나의 Haskell RTS 시스템에서 실행하는 메모리의 크기가 커서 발생하며 특히 큰 객체를 다룰 때 발생하는 문제로 추정된다. 실제 표절 검사 프로그램의 경우 120코어 환경에서

그림 11 120코어 환경에서 응용 프로그램의 GC 실행 시간
Fig. 11 The GC execution time of the application program in 120 cores

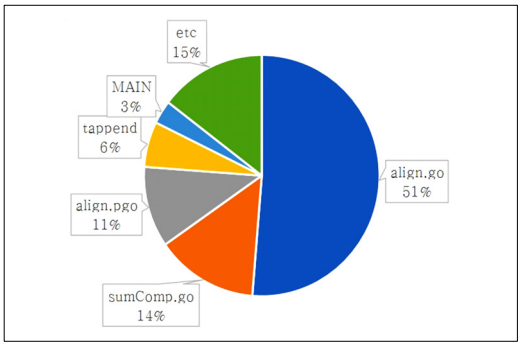

그림 12 Cloud Haskell의 표절 검사 프로그램 실행 시간 프로파일링 결과

Fig. 12 The profiling results of the plagiarism detection program execution time in Cloud Haskell

프로파일링을 통해 드러난 전체 할당된 메모리는 약 99Gbyte로 나타났다. 이 문제를 근본적으로 해결하기 위해서는 Haskell의 RTS를 수정해야 하는데 이러한 연구도 현재 진행 중이다[11]. 반면, Cloud Haskell의 경우 프로세스가 독립적인 메모리에서 실행되므로 당연히 GC도 분산 실행되며 이러한 GC 오버헤드 문제가 발생하지 않는다.

Cloud Haskell의 문제점은 코어 수가 적은 환경에서 Eval 모나드에 비해 느린 실행 시간이다. Cloud Haskell은 같은 알고리즘을 사용하여도 코어 수가 적은 환경에서는 Eval 모나드보다 현저하게 느리다. 두 응용 프로그램 중 표절 검사 프로그램을 대상으로 프로파일링한 결과 함수별 실행 시간 비중은 그림 12와 같이 나타났다.

그림 12와 같이 Cloud Haskell로 작성된 표절 검사 프로그램의 실행 시간 대부분은 세 함수가 차지하고 있음을 알 수 있다. 실행 시간의 절반 이상을 차지하는 `align` 계통 함수는 지역 정렬 알고리즘을 계산하는 함수이다. 그리고 다음으로 실행 비중이 큰 `sumComp`는 슬레이브 노드로부터 계산된 값을 전달받는 함수이다. 이 함수에서 네트워크를 이용한 통신 오버헤드가 발생한다. 이러한 오버헤드를 줄이기 위해서는 Cloud Haskell을 공유 메모리 환경에 특화시켜 네트워크를 통하지 않고 메모리를 통해 메시지를 전달하는 방법이 필요하다.

7. 결 론

이 논문에서는 병렬 프로그래밍 모델에 따른 Haskell 프로그램의 성능 비교를 통해 매니코어 환경에 적합한 프로그래밍 모델을 확인해보았다. 이를 위해 Eval 모나드와 액터 모델을 사용하는 Cloud Haskell을 이용하여 각 모델에 맞는 응용 프로그램을 제작하였다. 그리고 제작된 응용 프로그램을 이용하여 성능을 평가하였으며, 실험 결과 코어가 적은 환경에서는 Eval 모나드를 이용하는 것이 유리하며 코어 수가 많은 환경에서는 Cloud Haskell을 이용하는 것이 유리한 것으로 나타났다. 특히 코어가 많은 120코어 환경에서는 Cloud Haskell이 Eval 모나드보다 실행 시간 기준으로는 37%, 확장성 기준으로는 134% 우수한 것으로 나타났다.

향후 연구로는 고찰에서 언급한 문제의 해결 방안을 연구하고자 한다. 구체적으로 Eval 모나드의 코어 후반부에 GC로 인한 실행 시간 증가 문제를 해결하기 위해 Haskell의 RTS의 GC 모듈을 분석하고자 한다. 그리고 Cloud Haskell의 초반 실행 시간 문제를 해결하기 위해 메모리를 통해 메시지를 전달할 수 있는 함수를 제작하고자 한다.

References

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

[2] C. Hewitt, P. Bishop and R. Steiger, "A universal modular actor formalism for artificial intelligence," *Proc. of the 3rd international joint conference on Artificial intelligence*, pp. 235-245, 1973.

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

[4] 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.

[5] J. Armstrong, R. Virding, C. Wikstrom, and M. Williams, *Concurrent programming in ERLANG*, Prentice Hall, second edition, 1996.

[6] J. Ji, G. Woo and H. Cho, "A Source Code Linearization Technique for Detecting Plagiarized Programs," *ACM SIGCSE Bulletin*, Vol. 39, No. 3, pp. 73-77, 2007.

[7] V. Hanquez, Java parser and printer for Haskell, [Online]. Available: https://github.com/vincenthz/language-java. (downloaded 2017, Jan. 13)

[8] P, Robin, The align package, [Online]. Available: https://hackage.haskell.org/package/align. (downloaded 2017, Jan. 13)

[9] T. Sheard, S. P. Jones, "Template meta-programming for Haskell," *Proc. of the 2002 ACM SIGPLAN workshop on Haskell*, pp. 1-16, 2002.

[10] H. Kim, H, An, S. Byun and G. Woo, "An Approach to Improve the Scalability of Parallel Haskell Programs," *Proc. of the 2016 International Conference on Computing Convergence and Applications*, pp. 175-178, 2016.

[11] S. Marlow and S. P. Jones, "Multicore garbage collection with local heaps," *ACM SIGPLAN Notices*, Vol. 46, No. 11, pp. 21-32, 2011.

김 연 어

2010년 동아대학교 컴퓨터공학과(학사) 2012년 동아대학교 컴퓨터공학과(석사) 2012년~현재 부산대학교 전기전자컴퓨터공학과 박사과정. 관심분야는 프로그래밍 분석, 정적 분석, 표절 검사, 함수형 언어

안 형 준

2015년 연세대학교 수학과(학사). 2015년~현재 부산대학교 전기전자컴퓨터 공학과 석사과정. 관심분야는 함수형 프로그래밍, 역공학, 병렬 프로그래밍, 사용자 인터페이스

변 석 우

1980년 숭실대학교 전자계산학과(공학사) 1982년 숭실대학교 전자계산학과(공학석사). 1982년~1999년 ETRI(책임연구원) 1995년 Univ. of East Anglia(영국), Computer Science(Ph.D.). 1999년~현재 경성대학교 컴퓨터공학부 교수. 관심분야는 함수형 프로그래밍, 정형 증명, 프로그래밍 언어

우 균

1991년 한국과학기술원 전산학(학사). 1993년 한국과학기술원 전산학(석사). 2000년 한국과학기술원 전산학(박사). 2000년~2004년 동아대학교 컴퓨터공학과 조교수 2004년~현재 부산대학교 전기전자컴퓨터공학과 교수. 관심분야는 프로그래밍언어 및 컴파일러, 함수형 언어, 프로그램 분석, 프로그램 시각화, 프로그래밍 교육, 한글 프로그래밍 언어