표절 검사 기법을 이용한 Haskell의 병렬화 성능 비교

김연어[O1], 천준석[2], 변석우[3], 우균[4]
[1,2,4]부산대학교 전기전자컴퓨터공학과, [3]경성대학교 컴퓨터공학과, [4]LG전자 스마트 제어 센터
yeoneo@pusan.ac.kr, jscheon@pusan.ac.kr, swbyun@ks.ac.kr, woogyun@pusan.ac.kr

A Parallel Performance Comparison of Haskell Using a Plagiarism Detection Method

Yeoneo Kim[O1], Junseok Cheon[2], Sugwoo Byun[3], Gyun Woo[4]
[1,2,4]Dep. of Electrical and Computer Engineering, Pusan National University,
[3]School of Computer Science & Engineering, Kyungsung University,
[4]Smart Control Center of LG Electronics

요 약

최근 컴퓨터의 성능이 발달함에 따라 많은 프로그래밍 언어가 가상 머신 환경을 선택하고 있다. 특히 매니 코어 환경이 확대되면서 전체 컴퓨터 성능을 사용하도록 병렬화를 지원하는 프로그래밍 언어가 새로운 이슈로 떠오르고 있다. 수많은 프로그래밍 언어 중 Haskell은 부수효과가 없으므로 병렬화에 적합한 언어라 알려져 있다. 이 논문에서는 Haskell을 이용하여 병렬로 실행되는 표절 검사 프로그램을 제작한다. 그리고 기존 명령형 언어로 구현된 표절 검사 프로그램과 비교를 통해 Haskell이 병렬화에서 우수한 점을 보이고자 한다. 실험 결과 Haskell은 명령형 언어보다 확장성이 높으며 매니코어 환경에서의 실행 시간도 큰 차이가 나지 않는 것으로 나타났다. 그리고 Haskell의 코드 크기가 명령형 언어보다 약 1/74 수준으로 간결한 것으로 나타났다.

1. 서 론

최근 많은 프로그래밍 언어가 개발되어 사용되고 있다. 특히 최근 개발된 프로그래밍 언어는 이식성을 높이고 안전성을 높이기 위해, 자동으로 메모리를 관리하는 가비지 컬렉터(garbage collector)를 채택하고 있으며 따라서 가상 머신(virtual machine) 환경에서 동작하는 경우가 많다[1-3]. 가상 머신을 사용하는 프로그래밍 언어는 기계 위에서 직접 동작하는 언어에 비해 느린 편이다. 하지만 컴퓨터 환경의 발달로 가상 머신을 사용하더라도 충분한 속도가 보장되기 때문에 최근에는 큰 문제가 되지 않는다.

특히 매니코어(manicore) 환경[4]이 점차 확대되고 있는 상황에서는 오히려 컴퓨터 자원을 모두 사용하지 못하는 경우도 종종 발생하고 있다. 이 때문에 손쉽게 병렬화를 제공하는 기법에 관한 연구가 활발히 진행되고 있다. 하지만 기계 위에서 직접 동작하는 C나 C++의 경우 포인터를 이용하여 메모리를 직접 제어하기 때문에 제어 흐름이 매우 복잡하여 병렬화에 어려움이 있다. 또한, 가상 머신을 사용하는 Java도 명령형 언어의 특징상 제어 흐름이 복잡하여 병렬화를 하기가 어렵다. 더욱이 기존에 개발된 프로그램(legacy program)을 병렬화하는 것은 전체 프로그램의 흐름을 이해해야 해서 어려운 문제이다.

즉 병렬 프로그램을 만드는데 가장 큰 장애물은 복잡한 제어 흐름으로 볼 수 있다. 제어 흐름을 복잡하게 하는 요인은 여러 가지가 존재하지만, 대표적으로는 부수효과(side-effect)와 공유 메모리를 들 수 있다. 공유 메모리는 알고리즘에 의존적인 문제이기 때문에 쉽게 해결하기 어렵지만, 부수효과는 함수형 언어를 사용하면 간단하게 해결할 수 있다. 함수형 언어에서는 부수 효과가 없으므로 명령형 언어와 달리 제어 흐름이 간단하여 병렬화가 쉬운 편이다.

다양한 함수형 언어 중 Haskell은 병렬화에 적합한 프로그래밍 언어이다. Haskell은 부수효과가 없어 수식의 계산 과정을 병렬로 수행할 수 있다. 또한, 다른 함수형 언어와 달리 느긋한 계산(lazy evaluation)으로 불필요한 계산을 줄일 수 있으며, 다양한 병렬화 환경을 제공하고 있다.

이 논문에서는 병렬 표절 검사 프로그램을 Haskell로 개발함으로써 Haskell이 병렬화에 효과적임을 보이고자 한다. 프로그램 표절 검사는 입력된 프로그램 간의 표절 여부를 판단하는 도구로 입력되는 방대한 양의 프로그램, 즉 데이터에 의존적인 프로그램이다. 데이터 의존적인 프로그램은 비교적 제어 흐름이 간단하여 병렬화에 적합하다.

이 논문은 다음과 같이 구성되어 있다. 2장에서는 관련 연구로서 기존 표절 검사 프로그램 및 Haskell 병렬화 기법에 대해 알아본다. 3장에서는 Haskell을 이용하여 개발한 표절 검사 프로그램에 대해 알아본다. 그리고 4장에서는 명령형 언어로 작성된 표절 검사 프로그램과 비교하여 병렬화 환경에서 Haskell의 장점을 논의한다. 마지막으로 5장에서 결론을 맺는다.

2. 관련 연구

2.1 표절 검사 기법

기존 표절 검사 기법은 주로 소스 코드 실행 코드에서 특징을 추출하여 표절 여부를 판단한다. 기존에 연구된 표절 검사 기법은 프로그램의 특징을 추출하는 방법에 따라 문자열 기반, 토큰 기반, AST(Abstract Syntax Tree) 기반, PDG(Program Dependency Graph) 기반, 버스마크(birthmark) 기반으로 나눌 수 있다[5,6]. 이 논문에서는 이 중 기존에 확보가 가능한 토큰 기반의 방법의 하나인 SoVAC(Software Verification and Analysis Center)을 이용하기로 한다[7]. SoVAC은 소스 코드에서 사용된 토큰의 나열을 DNA로 변경한 뒤 생명 공학 분야에서 널리 사용되고 있는 지역 정렬(local alignment) 알고리즘을 이용하여 DNA를 비교하는 것으로 표절을 판단한다.

2.2 Haskell 병렬화

Haskell은 기존 명령형 언어보다 간단하게 병렬화를 할 수 있는 프로그래밍 언어이다. 이는 Haskell이 부수효과가 없어 함수별 병렬화가 가능하며 모나드를 제외한 경우 수학과 같이 계산 순서의 제약이 없기에 제어 흐름을 고려하지 않고도 병렬화가 가능하다. 그리고 Haskell은 단순히 고차 함수(higher-order function)를 변경하거나 병렬 자료 구조로 변경하는 것만으로 병렬화를 할 수 있다.

또한, Haskell은 언어 수준에서 가벼운 스레드(thread)를 이용하여 병렬화에 적합하다[8]. 기존의 많은 프로그래밍 언어는 언어 수준의 스레드가 아닌 OS 수준의 스레드[9]를 이용한다. OS 스레드는 사이즈가 크기 때문에 문맥 교환(context switch) 시 오버헤드가 큰 것으로 알려져 있다. 하지만 Haskell은 언어 수준에서 스레드를 제공하며, 이 또한 다른 언어에 비해 가볍기에 문맥 교환에 유리하다. 또한, Haskell에서는 자체적인 스케줄러를 이용하여 가벼운 스레드에 적합한 스케줄링 알고리즘을 사용하고 있다.

3. 표절 검사 프로그램 개발

이 장에서는 Haskell을 이용하여 토큰 기반의 표절 검사 프로그램을 개발하고자 한다. 이 논문에서는 여러 표절 검사 기법 중 SoVAC의 기법을 이용하여 병렬 표절 검사 프로그램을 개발하고자 한다. SoVAC의 구조는 그림 1과 같이 구성된다.

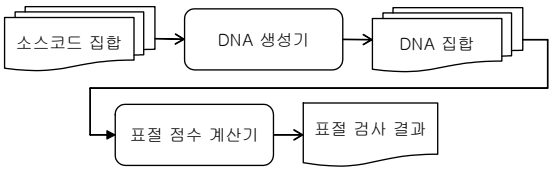

그림 1. 표절 검사 프로그램의 구조

SoVAC은 그림 1과 같이 소스 코드 집합을 입력으로 받아 표절 검사 결과를 반환하는 구조로 이루어져 있다. Haskell로 SoVAC의 기법을 구현하기 위해서는 크게 두 모듈을 제작하여야 한다. 첫 번째 모듈은 소스코드를 DNA 형태로 변환해주는 DNA 생성기이다. 그리고 두 번째 모듈은 DNA에서 표절 점수를 계산해주는 표절 점수 계산기이다.

DNA 생성기를 개발하기 위해서는 분석 언어에 맞는 파서 제작이 필요하다. 이 논문에서 표절 검사 대상 언어를 Java로 한정하기로 한다. 즉 DNA 생성기는 구문별 DNA를 생성하는 Java 파서를 만들어야 한다. Haskell에서는 Java 파서를 language-java-0.28 라이브러리를 통해 제공하기에 이를 이용하여 DNA 생성기를 제작한다[10]. 그리고 제작된 DNA 생성기 코드는 코드 1과 같다.

```
1: genDNA :: String -> String -> IO ([[String]])
2: genDNA dir ext = do
3:   files <- getDirectoryContents dir
4:   ...
5:   let tokens = parMap rseq parseCompilationUnit
                  file_contens
6:   let result = parMap rseq genDNA tokens
7:   ...
8:   result <- compDNA result_pair
9:   return result
```

코드 1. Haskell 버전 DNA 생성기

코드 1의 genDNA 함수는 소스 코드가 위치한 폴더와 분석할 언어의 확장자를 입력으로 받아 DNA를 생성하고 비교 결과를 반환하는 함수이다. 5번째 라인에서 입력된 폴더 내부의 모든 Java 파일을 파싱하여 AST를 생성한다. 그리고 AST 생성 단계는 병렬화가 가능하기에 병렬 맵(map) 함수인 parMap을 이용하여 파싱 작업을 병렬로 수행한다. 6번째 라인에서는 AST를 순회하며 각 노드에 맞는 토큰을 생성하는데 이 작업 역시 병렬화가 가능하기에 parMap을 이용한다. 그리고 8번째 라인에서는 표절 점수 계산기 함수인 compDNA를 호출한다. compDNA는 코드 2와 같이 구현된다.

```
 1:  compDNA :: [(String, String)] -> IO ([[String]])
 2:  compDNA pairs = do
 3:    let cDNA = parMap rpar (\(x,y) -> (procToken x),
                  y) pairs
 4:    let test_set = cartProduct cDNA cDNA
 5:    let nomal_set = M.fromList $ parMap rpar (\(x,y)
                    -> (y, (LA x x))) cDNA
 6:    let  min_set  =  parMap  rpar  (\((a,b),(c,d))  ->
                     select nomal_set b d) test_set
 7:    let simAB = parMap rpar (\((a,b),(c,d)) ->
                  (LA b d)) test_set
 8:    let sim = parMap rpar (\(a,b) -> (a/b)) $ zip simAB
              min_set
 9:    ...
10:    let sort_result = reverse $ sortBy (comparing fst)
                     tmp_result
11:    ...
12:    return result
```

코드 2. Haskell 버전 표절 점수 계산기

compDNA는 파일명과 토큰 쌍을 입력으로 받아 표절 결과를 반환하는 함수이다. compDNA는 3번째 라인에서 procToken 함수를 이용하여 파싱된 토큰을 DNA 형태로 변환된다. 그리고 4번째 라인에서 cartProduct 함수를 이용하여 중복되는 조합이 없도록 비교 쌍을 생성한다. 그리고 이 논문의 비교 쌍 유사도를 백분율로 표기하는 방법은 식 1과 같다.

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

식 1에서 LA는 지역 정렬을 계산하는 함수이다. 두 프로그램 간의 유사도는 백분율로 표시하기 위해 정규화된 값을 이용한다. 이를 Haskell에 적용하기 위해 bio-0.5.3 라이브러리를 이용한다[11]. 식 1을 Haskell로 구현한 방법은 코드 2의 6~8번째 라인과 같다. 그리고 표절 점수 계산기에서는 정규화된 값을 10번째 라인에서 퀵 정렬 함수를 이용하여 내림차순으로 정렬한다. 그리고 정렬된 값을 12번째 라인에서 반환한다.

4. 실 험

이 장에서는 Haskell로 구현한 표절 검사 프로그램과 기존 명령형 언어로 구현된 프로그램을 비교를 통해 Haskell이 병렬화에서 우수한 점을 보이고자 한다. 기존 명령형 프로그램인 SoVAC은 제어 흐름이 복잡하여 병렬 처리 버전으로 만들기 어렵기에 같은 알고리즘을 이용하는 버전을 별도로 제작하였다. 이 버전에서 DNA 생성기는 기존 C++로 제작된 프로그램

을 이용하였으며, 표절 점수 계산기는 Java로 새로이 구현하였다. 그리고 두 프로그램의 병렬화 방법으로 Java의 스레드 풀을 사용하여 구현하였으며 입력 프로그램을 기준으로 스레드를 나누어 실행한다. 그리고 이 버전과 Haskell 버전과 비교하였다. 실험은 CPU Opteron 6272 2개, RAM 32GB, SSD 256GB, OS는 Ubuntu 14.04.1 LTS 버전의 32코어 실험 환경에서 실험을 진행하였다. 그리고 표절 검사 대상으로 사용한 프로그램은 2012년 부산대학교 객체지향 프로그래밍 수업에서 과제로 제출된 Java 코드 82개를 이용하였다. 82개 코드를 대상으로 표절 검사를 실행해본 결과 코어에 따른 속도 상승 폭은 그림 2와 같이 나타났다.

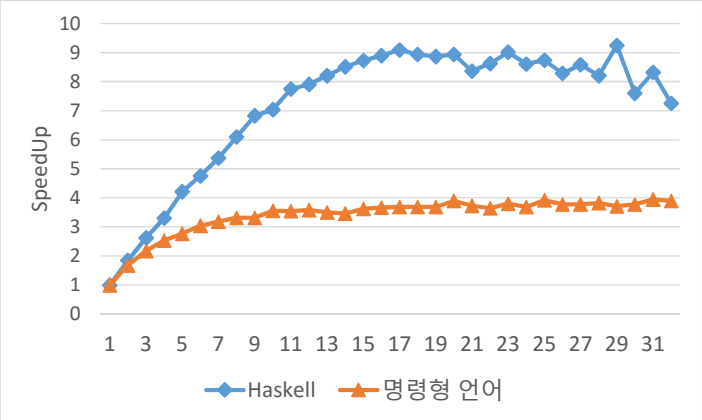

그림 2. 표절 검사 프로그램 속도 상승 비교

두 프로그램을 확장성 측면에서 비교해본다면 Haskell이 그림 2와 같이 더 높은 것으로 나타났다. Haskell의 경우 최대 속도 상승이 9.2로 측정되었으며 명령형 언어로 작성한 프로그램의 최대 속도 상승은 3.9로 측정되었다. 그리고 실제 실행 시간을 비교해보면 Haskell이 코어 1개에서는 명령형 언어로 작성된 프로그램보다 52.92초 느린 것으로 측정되었다. 하지만 코어 수가 증가해감에 따라 최대 4.82초 느린 것으로 측정되었다. 이는 전체 프로그램 중 병렬화가 되지 못한 부분과 Haskell과 명령형 언어의 속도 차이로 보인다. 즉 병렬화 환경에서는 Haskell도 명령형 언어와 큰 차이가 없는 것으로 나타난다. 그리고 두 프로그램의 크기를 비교한 결과는 표 1과 같다.

표 1. 표절 검사 프로그램 LOC 비교

대상	파일 수	LOC
명령형 언어	273	45593
표절 점수 계산기(명령형 언어)	6	925
Haskell	4	608

표 1과 같이 두 프로그램을 LOC 측면에서 비교해보면 Haskell 버전이 1/74로 월등히 짧은 것을 확인할 수 있다. 특히 명령형 언어에서 표절 점수 계산기 모듈만 비교해보아도 Haskell의 LOC가 더 짧은 것을 확인할 수 있다. 이는 Haskell이 명령형 언어에 비해 간결하고 풍부한 라이브러리를 제공하기 때문이라 볼 수 있다.

5. 결 론

이 논문에서는 Haskell을 이용하여 병렬 표절 검사 프로그램을 제작하였으며, 기존 명령형 프로그램과 비교하여 병렬화 환경에서 Haskell의 우수한 점을 보였다. 비교 실험 결과 Haskell은 병렬화가 간단하며 확장성이 우수하여 매니 코어 환경에서는 실행 속도도 그리 뒤처지지 않는 것으로 나타났다. 또한, LOC 비교 결과 명령형 언어보다 Haskell이 약 1/74 수준인 것을 확인하였다.

실험 결과에 따르면 코어 수가 늘어남에 따라 기존 실행 시간 중 가비지 컬렉터의 비중이 높아지는 것을 확인할 수 있었다. 따라서 Haskell 컴파일러인 GHC 분석을 통하여 가비지 컬렉터 시간의 비중을 줄이는 방법을 연구하는 것이 필요한데 이는 향후 연구로 진행할 예정이다. 그리고 표절 검사 프로그램 이외에도 다른 알고리즘을 대상으로도 명령형 언어와 실행 시간 및 코드 크기를 비교해볼 예정이다.

ACKNOWLEDGMENT

본 연구는 미래창조과학부의 SW컴퓨팅산업원천기술개발 사업의 일환으로 수행하였음(B0101-16-0644, 매니코어 기반 초고성능 스케일러블 OS 기초연구).
*교신 저자 : 우균(부산대학교, woogyun@pusan.ac.kr).

참 고 문 헌

[1] B. Venners, *Inside the Java virtual machine*, McGraw-Hill, 1996.
[2] A. Kennedy and D. Syme. “Design and implementation of generics for the. net common language runtime,” ACM SigPlan Notices, Vol. 36, No. 5, pp. 1-12, 2001.
[3] M. Odersky, L. Spoon and B. Venners, *Programming in Scala: A Comprehensive Step-by-Step Guide, 2nd Edition*, Artima Inc, 2011.
[4] 김진미, 변석우, 김강호, 정진환, 고광원, 차승준, 정성인, “매니코어 시대를 대비하는 Haskell 병렬 프로그래밍 동향,” 전자통신동향분석, 제29권 제5호, pp. 167-175, 2014.
[5] C. Roy and J. Cordy, A Survey on Software Clone Detection Research, Technical Report 541, Queen’s University at Kingston. 2007.
[6] H. Tamada, M. Nakamura, A. Monden, and K. Matsumoto, “Java Birthmark - Detecting the Software Theft,” IEICE Transactions on Information and Systems, Vol.88, No.9, pp.2148-2158, 2005.
[7] 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.
[8] S. Marlow, S.P. Jones, and S. Singh, “Runtime support for multicore Haskell,” ACM Sigplan Notices. Vol. 44. No. 9. pp. 65-78, 2009.
[9] S. Marlow, Commentary/Rts/Scheduler - GHC, [Online]. Available: https://ghc.haskell.org/trac/ghc/wiki/Commentary/Rts/Scheduler. (visited 2016, Apr. 26)
[10] V. Hanquez, Java parser and printer for haskell, [Online]. Available: https://github.com/vincenthz/language-java. (downloaded 2016, Apr. 26)
[11] K. Malde, The Bio Library, [Online]. Available: http://biohaskell.org/Libraries/Bio. (downloaded 2016, Apr. 26)