FFI를 이용한 Haskell 병렬 프로그래밍 라이브러리

김연어[1], 천준석[1], 류샤오[1], 허태광[1], 변석우[2], 우균[1]
부산대학교 전기전자컴퓨터공학과[1], 경성대학교 소프트웨어학과[2]
e-mail: {yeoneo,jscheon,liuxiao,vhxpffltm,woogyun}@pusan.ac.kr[1], swbyun@ks.ac.kr[2]

The Haskell Parallel Programming Library Using the FFI

Yeoneo Kim[1], Junseok Cheon[1], Xiao Liu[1], Taekwang Hur[1], Sugwoo Byun[2], Gyun Woo[1]
Dept. of Electrical and Computer Engineering, Pusan National University[1],
Dept. of Computer Science, Kyungsung University[2]

Abstract

Recently, software development use multiple programming language. Haskell also provides a communication feature into other programming language, it called FFI. However, the parallel programming using FFI is difficult. In this paper we propose a parallel library that can be easy programming using FFI. The proposed library provide a map-style parallel function based-on forkOS. The results of our experiment show that the our library reduced the execution time by about 48 times in one core environment. Also the results of our experiment show the our library guarantees the scalability as the number of cores increases.

I. 서론

최근 소프트웨어는 하나의 프로그래밍 언어를 선택하여 개발하기보다는 여러 언어를 혼합하여 개발되고 있다. 이러한 대표적인 예로는 텐서플로(TensorFlow)가 있다. 텐서플로는 성능과 밀접한 핵심 코어는 C++로 작성되었으며, 이외의 다른 부분은 여러 언어로 개발할 수 있다. 특히 Python이 많이 사용되고 있다. 이처럼 최근 프로그래밍 언어는 성능이나 기존의 라이브러리를 효과적으로 활용하기 위해 다른 언어 사이의 통신 기능을 제공하고 있다. 이러한 언어 사이의 통신 기능은 함수형 언어인 Haskell에서도 FFI(foreign function interface)라는 이름으로 제공하고 있다[1].

Haskell은 엄격한 타입 시스템을 제공하는 순수한 함수형 언어이다. Haskell은 순수한 함수를 제공하는 언어이기 때문에 병렬화에 적합하며, 언어 차원에서도 다양한 병렬 프로그래밍 기법을 제공하기 때문에 최근 컴퓨팅 환경에 적합하다고 볼 수 있다. 하지만 Haskell은 함수형 언어이기 때문에 상태 저장이 자주 발생하는 동적(dynamic) 프로그래밍에서는 성능이 저하되는 문제가 있다.

Haskell에서는 이러한 성능 저하 문제를 해결하기 위해서 저장 가능한(mutable) 타입을 사용하거나 FFI를 사용할 수 있다. 이 중 FFI를 사용하는 방법은 C 프로그램의 성능을 그대로 가져올 수 있기 때문에 매우 효과적이라 볼 수 있다. 하지만 FFI를 사용하면서 Haskell의 장점인 병렬 프로그래밍 기법을 적용하는 함수는 프로그래머가 별도로 작성해야 한다.

이 논문에서는 FFI 기능을 사용하면서 이를 병렬로 처리할 수 있는 라이브러리를 제안하고자 한다. 제안 라이브러리는 Haskell의 고차 함수(higher-order function)인 `map`과 같은 형태의 함수를 제공한다. 그리고 제안 라이브러리의 함수를 이용하면 FFI 함수가 런타임 시스템에 정의된 코어의 수만큼 병렬로 실행되게 된다.

이 논문의 구성은 다음과 같다. 우선 2장에서 관련 연구로 기존 Haskell의 병렬 프로그래밍 모델을 소개하고자 한다. 그리고 3장에서는 제안 라이브러리의 설계를 위한 기존 라이브러리의 문제점을 분석하고 이를 해결한 라이브러리를 구현하고자 한다. 4장에서는 실험을 통해 기존 프로그램의 성능이 얼마만큼 개선되는지를 보이고자 한다. 이후 5장에서는 구현한 라이브러리에 대해 논의하고 6장에서 결론을 맺고자 한다.

II. 관련 연구

이 장에서는 관련 연구로 Haskell 병렬 프로그래밍을 소개하고자 한다. Haskell은 순수한 함수형 언어이기 때문에 모나드를 제외하고는 함수 사이의 계산 순

서가 없다. 또한, Haskell은 일반 함수와 모나드가 엄격한 타입 시스템으로 분리되어 있기에 병렬화가 쉬운 편이다. 이외에도 Haskell은 자체적인 경량화 쓰레드(lightweight thread)를 사용하기 때문에 문맥 교환(context switch)에 대한 부담이 작아 병렬화 성능이 우수하다.

또한, Haskell에서는 상황에 따라 적절하게 사용할 수 있는 다양한 병렬 프로그래밍 모델을 제공하고 있으며, 이 논문에서 관련 연구로 소개할 병렬 모델은 총 네 가지이다. 첫 번째 모델은 Eval 모나드이다. Eval 모나드는 병렬 전략(strategy)을 이용하여 함수를 병렬로 실행하는 모델로 Haskell의 가장 기본적인 병렬화 기법이다[2]. 이 모델은 사용이 편리하지만, 실행 시 상황을 보고 병렬화가 이루어지기 때문에 항상 병렬 실행이 보장되지 않는다.

두 번째 모델은 `forkIO` 및 `forkOS`를 이용한 병렬화이다[2]. 이 모델은 C의 `fork` 함수와 유사하게 새로운 Haskell 쓰레드를 생성하여 함수를 병렬로 실행하는 방법이다. 이 방법은 항상 병렬 실행이 보장되지만 실행 결과를 원래 함수로 전달받기 위해서는 별도의 코딩이 필요하다.

세 번째 모델은 Repa(regular parallel array)이다. 이 모델은 주로 데이터 병렬화에 사용되는 병렬화 모델이다[2]. 이 방법은 Haskell의 배열 구조를 대상으로 병렬화를 실행하는 방법으로 `computeP`나 `foldP` 함수를 통해 배열 내 데이터를 병렬로 계산한다. 이 방법은 사용이 간편하고 성능이 우수하지만 모든 데이터가 배열로 표현 가능할 때만 사용할 수 있다.

마지막 모델은 Cloud Haskell[3]이다. 이 모델은 분산 컴퓨팅을 위한 병렬화 모델로 액터 모델(actor model)[4]을 기반으로 하고 있다. 이 방법은 함수를 OS 수준의 프로세스에서 병렬로 실행하고 데이터는 소켓 통신을 통해 주고받는 방법이다. 이 모델은 각 프로세스가 비동기화 방식으로 실행되기 때문에 높은 병렬성을 보이지만 단일 노드에서도 소켓 통신을 사용해야 되기 때문에 오버헤드(overhead)가 존재한다.

Ⅲ. 문제점 분석 및 구현

이 장에서는 제안 라이브러리의 설계를 위한 문제점 분석 및 구현에 대해 소개하고자 한다.

3.1 기존 라이브러리 문제점 분석

관련 연구에서 살펴봤듯이 Haskell에서는 다양한 병렬 프로그래밍 모델을 제공하고 있지만, FFI를 이용하는 경우 이를 효과적으로 활용할 수 없다. 우선 FFI를 이용하는 경우 Haskell의 `forkOS`를 이용한 병렬화 기법을 이용해야 되는데 `forkOS`를 사용한 경우 계산 결과를 전달받기 위해서는 별도의 저장 가능한 변수를 사용하고 전달받는 과정이 필요하다.

또 다른 문제점은 쓰레드 동기화 문제이다. `forkOS`를 이용하여 FFI 함수를 실행한 경우 메인 쓰레드는 FFI 함수가 언제 종료될지를 알 수 없다. C 언어의 경우 `wait` 계열의 함수를 이용해 쓰레드 종료 시점을 기다릴 수 있지만 Haskell에서는 해당 함수를 제공하지 않고 `MVar`를 이용한 동기화를 제공하고 있다. 그렇기 때문에 `forkOS`를 이용한 후 동기화를 위해서는 `MVar`로 별도의 코딩이 필요하다.

이처럼 기존 라이브러리를 이용하여 FFI 함수를 병렬로 실행하기 위해서는 부수적인 코딩이 많이 필요한 상황이다. 이러한 부수적인 코딩은 FFI를 이용한 병렬 프로그래밍 진입 장벽이 될 수밖에 없다. 그렇기 때문에 이 논문에서는 FFI 함수도 `map` 함수를 이용하여 간단하게 병렬로 실행할 수 있는 라이브러리를 구현하고자 한다.

3.2 라이브러리 구현

이 논문에서 제안하는 라이브러리의 목표는 FFI 함수를 손쉽게 병렬로 사용할 수 있게 하는 것이다. 그렇기 때문에 프로그래머는 일반 `map` 함수를 사용하듯이 FFI 함수를 사용할 수 있어야 한다. 이를 고려하여 구현한 함수는 코드 1과 같다.

```
 1: parMapFFI :: (a -> b) -> [a] -> IO ([b])
 2: parMapFFI f xs = do
 3: cap <- getNumCapabilities
 4: ...
 5: writeList2Chan iCh xs
 6: p_id <- mapM (\i -> forkOS $
                    go f iCh oCh sync n) ns
 7: waitN n sync
 8: ret <- mapM (\i -> readChan oCh) [1..n]
 9: return ret
10: where
11:   go :: (a->b) -> Chan a -> Chan b ->
          MVar Int -> Int -> IO ()
12:   go f iCh oCh v n = do
13:     val <- readChan iCh
14:     writeChan oCh $ f val
15:     ...
16:     when ((i+1) /= n) (go f iCh oCh v n)
17:     return ()
```

코드 1. FFI를 이용한 Haskell 병렬 map 함수

코드 1의 `parMapFFI` 함수를 살펴보면 기존 `map` 함수의 타입인 `(a->b) -> [a] -> [b]`와 유사한 것을 확인할 수 있다. 차이점은 반환 타입이 IO 모나드인 것으로 이는 `parMapFFI`가 `forkOS`를 사용하기 때문에 발생한 차이점이다. 이외에 입력 타입은 `map` 함수와 같기 때문에 `map` 함수를 사용하듯이 사용하면 입력 함수가 병렬로 실행될 수 있다.

세부 코드를 좀 더 살펴보면 3번째 라인에서는 현재 프로그램에서 사용한 가능한 코어의 수를 가져온다. 그리고 6번째 라인에서 코어 수만큼 `forkOS`를 실행하면서 입출력 값 전달을 위해 `go` 함수가 실행된다. `go` 함수는 `Chan` 타입의 입력 값을 하나씩 값을 가져와 입력 값이 없어질 때까지 입력 함수인 `f`를 적용하고 이를 출력값으로 전달하는 함수이다. `forkOS`를 실행한 뒤 메인 쓰레드에서는 `waitN` 함수를 통해 값이 모두 계산되기를 기다리게 된다. 그리고 값이 모두 전달되면 8번째 라인에서 출력 값을 가져오고 이를 9번째 라인에서 반환하게 된다.

Ⅳ. 실험

이 장에서는 논문에서 제안한 라이브러리의 성능을 확인하고자 한다. 이 논문에서는 실험을 위해 소스코드 표절 검사 프로그램[5]을 대상으로 사용하였다. 소스코드 표절 검사 프로그램은 검사 대상을 늘리면 계산량이 늘어나기 때문에 병렬화 성능 검사에 효과적이며, 동적 알고리즘 중 하나인 부분 정렬(local alignment) 알고리즘을 사용하였기 때문에 FFI 사용 예제로 적용하기에도 적합하다.

그리고 실험 환경으로는 32코어 CPU(AMD Opteron 6276 2개), 82GB RAM, OS는 Ubuntu 18.04.1 LTS를 사용하였다. 빌드 환경은 GHC 8.4.3 버전을 사용하였다. 그리고 입력으로는 150개의 Java 프로그램을 입력으로 하여 표절 검사 프로그램을 실행하였다. 그 결과 두 프로그램의 실행 시간 측정 결과는 표 1과 같다.

표 1. 소스코드 표절 검사 프로그램 대상 실행 시간 비교

#	Pure Haskell			FFI Haskell		
Core	MUT	GC	Total	MUT	GC	Total
1	236.97	147.24	384.22	5.47	2.47	7.94
5	51.20	52.18	103.38	1.62	1.33	2.95
10	29.66	38.04	67.70	1.14	0.98	2.14
15	21.39	51.44	72.84	1.08	0.98	2.09
20	18.07	60.67	78.74	1.12	1.03	2.17
25	15.99	67.36	83.37	1.11	1.18	2.31
30	14.64	74.02	88.68	1.19	1.29	2.51

표 1의 결과에서 Pure Haskell은 FFI를 사용하지 않고 기존 방식으로 병렬화한 프로그램이며, FFI Haskell은 FFI를 사용하여 병렬화한 프로그램이다. 그리고 MUT는 실제 계산 시간을 의미하며, GC는 가비지 컬렉션 시간, Total은 전체 실행 시간을 의미한다. 우선 코어가 하나일 때 실행 시간을 살펴보면 FFI 사용 유무에 따라 실행 시간이 약 48배가량 차이가 나는 것을 확인할 수 있다. 그리고 코어 수가 늘어남에 따라 두 모델 다 실행 시간이 효과적으로 줄어드는 것을 확인할 수 있다.

Ⅴ. 고찰

이 장에서는 4장의 실험 결과에 대해 논의하고자 한다. 동적 프로그래밍을 사용한 프로그램에서 FFI를 사용한 경우 실행 시간이 극적으로 줄어드는 것을 실험을 통해 확인할 수 있었다. 이는 실행 시간의 많은 부분을 차지하는 지역 정렬 알고리즘이 C로 실행되었기 때문이다. 이 때문에 일부 사람들은 Haskell 사용 시 얻을 수 있는 장점에 대해 의문을 가질 수 있다. 하지만 Haskell을 사용하는 경우 얻을 수 있는 장점은 크게 두 가지를 들 수 있다.

첫 번째 장점은 병렬화의 편의성이다. 기존의 C와 같은 명령형 언어를 사용하여 병렬 프로그램을 작성하기 위해서는 변수 간의 의존성에 대한 검증이 필요하고, 이외에도 쓰레드 간의 자원 할당을 위한 작업이 필요하다. Haskell을 사용하는 경우 이러한 부수적인 작업이 없어도 손쉽게 병렬화가 가능하다.

두 번째 장점은 코드의 간결성이다. 같은 프로그램을 C와 Haskell로 작성해도 코드의 길이는 Haskell이 간결하게 작성된다. 더욱이 작성 프로그램이 병렬 프로그램이라면 Haskell은 C보다 더 간결하게 작성할 수 있다.

Ⅵ. 결론

이 논문에서는 FFI를 이용하여 프로그램을 작성 시에도 사용할 수 있는 병렬 라이브러리를 개발하였다. 제안 라이브러리는 `forkOS`를 이용하여 FFI 함수를 병렬로 실행하는 함수로 `forkOS`의 입출력 전달 문제와 동기화 문제를 해결하였다. 그리고 실험을 통해 동적 프로그래밍 환경에서 제안 라이브러리를 사용한 경우 기존 프로그램보다 1코어 환경에서는 약 48배가량 실행 시간이 줄어든 것을 확인할 수 있었으며, 코어가 늘어남에도 실행 시간이 잘 줄어드는 것을 확인할 수 있었다.

향후 연구로는 기존 명령형 언어 프로그램과도 실행 시간을 비교하고자 한다. 동적 프로그래밍의 경우 함수형 언어의 실행 시간이 매우 느려 성능을 비교하기에는 적합하지 않았다. 하지만 FFI를 이용한 라이브러리를 사용한다면 명령형 언어와도 성능을 비교할 수 있을 것으로 판단된다. 이외에도 다른 Haskell의 병렬 프로그래밍 모델에서도 FFI 함수를 적용할 수 있을지에 대한 연구를 진행하고자 한다.

ACKNOWLEDGMENT

이 논문은 2020년도 정부 (과학기술정보통신부)의 재원으로 정보통신기획평가원의지원을 받아 수행된 연구임 (No.2014-3-00035, 매니코어 기반 초고성능 스케일러블 OS 기초연구 (차세대 OS 기초연구센터)).
*교신 저자: 우균(부산대학교, woogyun@pusan.ac.kr).

참고문헌

[1] M. M. Chakravarty, *The Haskell Foreign Function Interface 1.0: An Addendum to the Haskell 98 Report*, 2003.

[2] S. Marlow, *Parallel and concurrent programming in Haskell: Techniques for multicore and multithreaded programming*, 1st Ed., O'Reilly Media, 2013.

[3] 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.

[4] 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.

[5] 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.