JKSCI

한국컴퓨터정보학회논문지
Journal of The Korea Society of Computer and Information
Vol. 26 No. 1, pp. 11-17, January 2021
https://doi.org/10.9708/jksci.2021.26.01.011

Parallelization of a Purely Functional Bisimulation Algorithm

Ki Yung Ahn*

*Assistant Professor, Dept. of Computer Engineering, Hannam University, Daejeon, Korea

[Abstract]

In this paper, we demonstrate a performance boost by parallelizing a purely functional bisimulation algorithm on a multicore processor machine. The key idea of this parallelization is exploiting the referential transparency of purely functional programs to minimize refactoring of the original implementation without any parallel constructs. Both original and parallel implementations are written in Haskell, a purely functional programming language. The change from the original program to the parallel program is minuscule, maintaining almost original structure of the program. Through benchmark, we show that the proposed parallelization doubles the performance of the bisimulation test compared to the original non-parallel implementation. We also shaw that similar performance boost is also possible for a memoized version of the bisimulation implementation.

▸ **Key words**: Multicore, Deterministic Parallelism, Labelled Transition Systems, Bisimulation, Behavioral Equivalence, Functional Programming

[요 약]

본 논문에서는 순수 함수형 언어로 작성된 쌍방시뮬레이션 알고리듬의 성능을 멀티코어 프로세서 컴퓨터에서 병렬화로 향상시키는 방법을 연구한다. 이 병렬화에 있어 핵심 아이디어는 순수 함수형 프로그램의 참조 투명성을 십분 활용하면 병렬화가 전혀 고려되지 않고 작성된 초기 구현으로부터 최소한의 수정만으로 성능 개선 효과를 기대할 수 있다는 것이다. 초기 구현과 병렬화 구현 둘 다 순수 함수형 언어인 하스켈로 작성되었다. 초기 구현을 병렬화할 때 변화는 아주 적어서 병렬화된 구현에서도 초기 구현의 프로그램 구조가 거의 그대로 유지되었다. 벤치마크를 통해 제시된 간단한 병렬화만으로도 초기 구현과 비교해 두 배 이상의 성능 개선을 확인했다. 또한, 병렬화와는 별개의 최적화 기법인 메모이제이션이 적용된 버전의 쌍방시뮬레이션 구현에도 같은 방식의 병렬화를 적용함으로써 마찬가지로 성능을 개선할 수 있음을 확인하였다.

▸ **주제어**: 멀티코어, 결정적 병렬성, 라벨된 전이시스템, 쌍방시뮬레이션, 행위적 동일성, 함수형 프로그래밍

• First Author: Ki Yung Ahn, Corresponding Author: Ki Yung Ahn
*Ki Yung Ahn (kya@hnu.kr), Dept. of Computer Engineering, Hannam University
• Received: 2020. 12. 22, Revised: 2020. 12. 30, Accepted: 2021. 01. 04.

http://www.ksci.re.kr pISSN:1598-849X | eISSN:2383-9945

I. Introduction

컴퓨팅 관련 분야 시스템의 엄밀한 설계나 검증을 위한 명세로 **라벨된 전이 시스템**(labelled transition system, LTS)이 널리 쓰인다. 예컨대, 계산이론의 오토마타도 일종의 LTS로 볼 수 있다. 설계/검증하려는 시스템을 모델링하는 방법은 유일하지 않다. 이를테면 같은 시스템을 초기 설계자는 성능보다 명확성을 추구한 레퍼런스 모델 S_0를, 제품 엔지니어는 실제 구현의 성능을 고려한 모델 S_1을 작성했다고 하자. 그렇다면 성능이 개선된 S_1이라는 LTS가 초기설계를 충실히 반영하는 S_0라는 LTS와 모든 상황에서 똑같이 동작함을 어떻게 확신할 수 있을까?

하나의 대상이 되는 시스템에 대한 복수의 모델이 존재하는 상황은 다양하다. 같은 HW에 대한 고수준 논리 모델부터 소자 배열을 고려하는 저수준 모델에 이르기까지 여러 층위의 모델이 있다. 통신/보안 프로토콜에도 고수준 명세가 있는가 하면 어떤 기반 기술/인프라를 사용하는지 고려한 실제 구현과 더 가까운 명세도 있다. 그리고 같은 수준의 모델이라도 S_1에서 더 효율적인 구현 가능성을 반영한 후속 모델 S_1'이 이후에 작성될 수 있다. 이럴 때, S_0, S_1, S_1'이 과연 원하는 대로 똑같이 동작하는지 확신하기란 간단한 문제가 아니다.

이러한 문제, 즉 한 시스템에 대한 다수의 LTS 명세가 항상 같은 동작을 보장하는지 확인하기 위한 강력한 도구의 하나로 **쌍방시뮬레이션**(bisimulation)이 있다. 하지만 쌍방시뮬레이션 검사 알고리듬의 복잡도가 높은 편이라 아직은 활용 범위가 다소 제한적인 편이다. 이 논문은 간단한 LTS에 대한 전혀 최적화되지 않은 쌍방시뮬레이션 알고리듬을 순수 함수형 언어인 하스켈(Haskell)로 작성한 상태에서 아주 적은 코드 변경만으로 두 배의 가량의 성능 개선을 보여줌으로써 병렬화된 쌍방시뮬레이션 알고리듬 고성능 구현에 순수 함수형 언어 활용 가능성을 시사한다.

II. Preliminaries

1. Parallelism in Functional Programming

함수형 프로그래밍의 대표적 장점은 상태 변화에 대한 의존성을 최소화함으로써 프로그램 작성시 오류를 줄이고 모듈성이 높아진다는 점이다. 변수에 대한 잦은 대입으로 상태 변화 의존적인 명령형 프로그래밍에서도 전역변수를 피함으로써 서로 다른 모듈 간의 의도치 않은 상호작용을 방지하는 것을 좋은 습관으로 권장한다, 함수형 프로그래밍은 이를 프로그램 전반에 적용하자는 패러다임으로 이해해도 좋을 것이다. 순수 함수형 언어인 Haskell의 경우는 모든 변수가 기본적으로 대입이 불가하여 이러한 함수형 프로그래밍의 장점을 언어적으로 강제한다.

대부분의 기존 SW가 직렬로 작성되어 있어 멀티코어 CPU의 대중화 이후 풍부해진 병렬 컴퓨팅 HW를 충분히 활용하지 못하고 있다. 순차적 상태 변화를 따라 진행하는 명령형 프로그램을 병렬 컴퓨팅에 맞게 재구성하기는 일반적으로 매우 번거롭다. 반면, 애초에 상태 변화를 터부시하는 함수형 프로그램은 상대적으로 병렬화하기 좋다. 프로그램에 나타나는 부분식 계산 순서에 무방하게 같은 결과를 보장하기 때문이다. 예컨대, e1 + e2에서 e1를 먼저 계산하든 e2를 먼저 계산하든 아니면 둘을 동시에 병렬적으로 계산하든 최종 결과가 같다.

2. Bisimulation

이론적 정의와 증명을 다루려는 논문이 아니므로 최대한 직관적이고 간결하게 설명하려 한다. P와 Q라는 두 LTS가 쌍방시뮬레이션 관계(P ~ Q)임을 설명하기에 앞서 시뮬레이션 및 상호시뮬레이션 관계에 대해 알아보자. 그 다음 쌍방시뮬레이션과 이들 두 관계의 차이를 파악하자.

2.1 Simulation

P가 먼저 한걸음씩 가면 Q도 그에 대응하는 같은 동작으로 따라갈 수 있을 때 P를 Q로 시뮬레이션할 수 있다고 말한다 (P≤Q), 비유하자면 최대속력 60km/h 자동차 P의 모든 동작을 최대속력 80km/h 자동차 Q로는 따라갈 수 있지만 반대로 Q가 70km/h로 달려버리면 P는 따라 할 수가 없다.

2.2 Mutual simulation

P가 먼저 한 동작씩 하면 Q도 같은 동작으로 따라갈 수 있고, 반대로 Q가 먼저 한 동작씩 가면 P도 같은 동작으로 따라갈 수 있을 때 **상호시뮬레이션**(mutual simulation) 관계라 한다. 즉, P≤Q이면서 P≥Q인 경우이다. 직관적으로는 어떤 작업을 P로 처음부터 끝까지 처리했다면, 다음에 같은 작업을 위해 Q로 대신해서 처음부터 끝까지 같은 동작으로 진행해 처리할 수 있으며, 그 반대도 가능하다는 뜻으로 이해할 수 있다.

2.3 Bisimulation

쌍방시뮬레이션(bisimulation)은 앞서 소개한 상호시뮬레이션보다 더 강력하게 동등성을 보장한다. P가 먼저 한

동작씩 하면 Q가 같은 동작으로 따르다 어느 동작부터는 앞서는 쪽을 바꿔서 Q가 먼저 동작해도 P가 같은 동작을 따라 할 수 있고, 언제든 몇 번이고 앞선 쪽과 따르는 쪽을 바꿔도 항상 앞서는 쪽의 동작을 같이 뒤따를 수 있어야 쌍방시뮬레이션 관계이다. 직관적으로는 P와 Q를 도중에 언제든 바꿔쳐도 괜찮다고 이해할 수 있다.

휘발유차 P와 전기차 Q로 지점 주차장으로 이동해 주차하는 동작(a), 지점에서 일보는 동작(b), 본점 복귀 동작(c)으로 출장업무를 모델링하자. P는 왕복거리 주행 가능하며 Q는 편도 주행만 되지만 일보는 동안 충전으로 복귀할 수 있다고 하자. 이 경우 상호시뮬레이션 관계가 성립하더라도, 쌍방시뮬레이션 관계는 성립하지 않을 수 있다. Q로 동작 a를 하면 전기차 충전 플러그가 닿는 곳에 주차했을 것이므로 P로 바꿔쳐도 b와 c를 계속할 수 있다. 그런데 P로 동작 a를 하면 충전 플러그가 닿지 않는 곳에 주차했을 가능성이 있어, 이런 경우에 Q로 바꿔치면 일보고(b) 나서 복귀(c)하지 못하는 경우가 발생할 수 있다.

III. Benchmark

1. Vanilla (non-memoizing) implementation

방향 비순환 그래프(directed acyclic graph, DAG)에 대한 쌍방시뮬레이션 알고리듬의 Haskell 코드가 Fig. 1에 나타나 있다. Haskell 문법은 자세히 설명하지 않고 쌍방시뮬레이션 코드의 전체적인 구조와 병렬화 구현에서 달라지는 점을 중심으로 살펴보기로 하자.

우선 boldrm bisim p q = ... 이후로 나타나는 코드에서 쌍방시뮬레이션의 대칭적 구조가 드러난다. 논리 연산자 && 전까지는 P가 먼저 진행하는 모든 경우에 대해 Q가 똑같은 동작으로 따르는 경우가 존재함을 의미하며, && 이후로는 Q가 먼저 진행하는 모든 경우에 대해 P가 똑같은 동작으로 따르는 경우가 존재함을 의미한다. 그리고 양쪽 모두 그렇게 p와 q로부터 한 동작씩 진행한 직후 상태 p1과 q1을 기준으로도 쌍방시뮬레이션 관계가 재귀적으로 성립해야 함(boldrm bisim p1 q1)을 나타낸다.

Fig. 1을 보면 알다시피 boldrm bisimilar와 그 병렬화 구현인 boldrm bisimilar′의 내용이 대동소이하다. 차이점이라고는 &&를 대신해 `parAnd`를 사용했을 뿐이다. `parAnd`는 &&와 같은 결과를 계산하되, 양쪽 인자를 병렬로 처리한다. 병렬 연산을 촉발하는 `par`는 왼쪽과 오른쪽의 계산이 병렬로 진행 가능하며 오른쪽의 결과를 전체의 값으로 한다. 즉, e2 `par` (e1 && e2)의 의미는 e2과 (e1 && e2)를 병렬처리해 (e1 && e2)의 값을 구한다. 대부분 프로그래밍 언어에서와 마찬가지로 논리 연산자 &&는 e1의 결과값이 True로 계산됨을 확인한 다음 e2를 계산한다. 즉, `par` 연산자는 왼쪽에 e2를 배치함으로써 e1의 계산 완료 전에 e2와 e1을 병렬처리할 수 있도록 하스켈 컴파일러(GHC)에게 알려주는 역할을 한다. 컴파일 옵션으로 -rtsopts를 포함해야 `par`가 요청하는 병렬처리 지원이 활성화되며 실행옵션인 +RTS와 함께 -s와 -N옵션을 활용해 실행시 벤치마킹 데이터를 얻고 활용할 최대 코어/스레드 개수를 지정할 수 있다. GHC 버전은 8.8.4를 사용하였다.

벤치마킹을 위한 DAG는 상태와 라벨 개수를 정해 놓고 에지 생성 횟수를 적절히 지정해 무작위 생성 후 중복을

```haskell
-- test whether  sysP starting from p0   is bisimilar to  sysQ starting from q0
bisimilar sysP@(p0,esP) sysQ@(q0,esQ) = bisim p0 q0
  where
  bisim p q = and [or [bisim p1 q1 | (q',b,q1) <- esQ, q==q', a==b]
                                   | (p',a,p1) <- esP, p==p']
           && and [or [bisim p1 q1 | (p',a,p1) <- esP, p==p', b==a]
                                   | (q',b,q1) <- esQ, q==q']

-- parallelized bisimilar
bisimilar' sysP@(p0,esP) sysQ@(q0,esQ) = bisim p0 q0
  where
  bisim p q = and [or [bisim p1 q1 | (q',b,q1) <- esQ, q==q', a==b]
                                   | (p',a,p1) <- esP, p==p']
     `parAnd` and [or [bisim p1 q1 | (p',a,p1) <- esP, p==p', b==a]
                                   | (q',b,q1) <- esQ, q==q']

e1 `parAnd` e2 = e2 `par` (e1 && e2)
```

Fig. 1. Vanilla Bisimulation Code (original vs. parallelized)

제거한 뒤 초기 상태인 p0에서부터 접근 가능한 부분만을 선별하였다. 에지 생성 방법은 DAG임을 감안하여 상태 개수를 n이라 할 때, $0 \leq x < y \leq n$을 만족하는 순서쌍 (x,y)를 대략 $n^2/2$회 생성해 각각에 라벨 a를 무작위로 붙여 (x,a,y) 형태로 만든 뒤 중복을 제거했다. 활용 코어 개수에 따른 쌍방시뮬레이션 코드의 실행 시간 변화 추이 확인에 적절한 값들로 상태 개수 20, 라벨 개수 2, 그리고 무작위 에지 생성 횟수는 앞서 언급한 바와 같이 상태 개수의 제곱이므로 $20 \times 19/2$개를 무작위 생성 후 중복을 제거하여 Fig. 2와 같은 DAG를 벤치마킹 데이터로 생성하였다. 그리고 이 DAG 자신에 대한 쌍방시뮬레이션 관계를 실행했다. 즉, DAG를 D20라 하면 D20 D20을 실행한 것이다. 자기 자신에 대한 쌍방시뮬레이션 결과는 뻔하게 참인데 이렇게 하는 이유는 결과가 참인 경우가 모든 경우의 수를 빠짐없이 검사하게 되기 때문이다. 쌍방시뮬레이션 알고리듬 구현에서 논리곱(and)의 인자 중 하나라도 거짓이면 다른 부분을 계산할 필요 없이 결과값이 결정되므로 오히려 실행 시간은 관계 성립이 참인 경우보다 줄어든다. 비교적 적은 개수의 상태 및 에지로 충분한 실행 시간이 나와야 병렬화의 효과를 확인하기 좋다.

두 개의 컴퓨터로 실행한 벤치마크를 Fig. 3에 도표로 나타냈다. 같은 인텔 코어i7 계열의 CPU지만 8565U보다 8700이 지원하는 멀티코어/하이퍼스레드 개수가 더 많을 뿐더러 그 컴퓨터의 메인 메모리도 많다. 하지만 메모리 재활용(GC)이나 샘플링된 최대 메모리 사용량이 최대 활용 코어/스레드 개수를 늘려도 그다지 차이가 나지 않아 이번 D20의 정도의 쌍방시뮬레이션으로는 두 컴퓨터의 메인 메모리 용량 차이로 인한 영향은 미미하다. 오히려 CPU의 계산 성능 차이로 `par`가 생성하는 스팍(GHC 런타임에서 task로 처리될 후보인 계산의 불쏘시개)으로 인한 작업(task)이 더 빠르게 소진되어 과다 스팍 발생으로

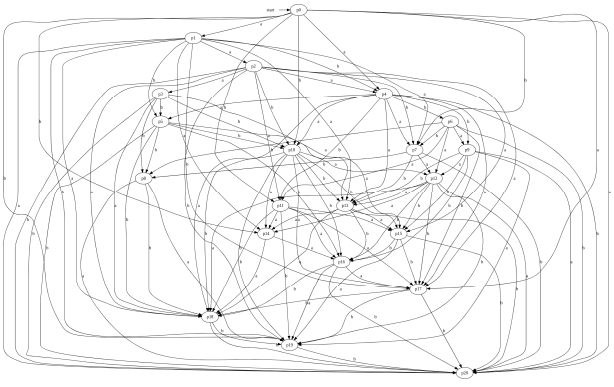

Fig. 2. Randomly generated DAG with 20 nodes for benchmarking

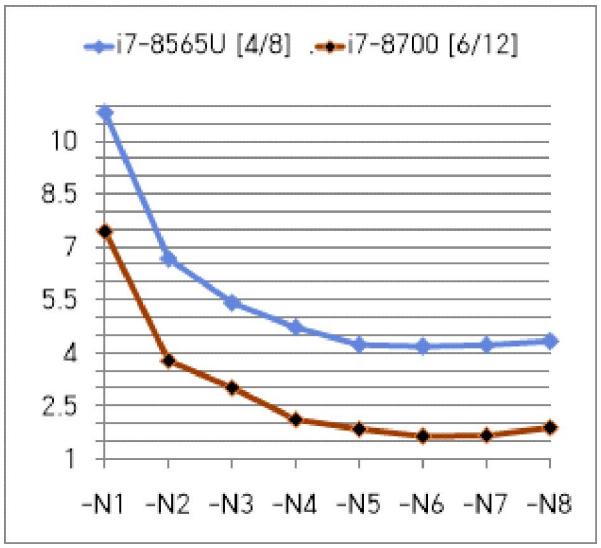

Fig. 3. Benchmarking (boldrm bisimilar' D20 D20) on multicore CPU machines

적체되는 병목도 가용 코어/스레드 개수와 함께 늘어나는 것으로 생각된다. 재귀적으로 bisim이 호출되며 `par`로 병렬화된 계산에서 한 단계 더 깊이 재귀호출이 일어나며 연쇄적으로 스팍이 생성되기 때문일 것이다. 참고로, 일정 깊이 이하 재귀함수 호출에서만 병렬화를 허용하는 등의 전략으로 과도한 스팍 생성을 줄이면 추가 성능 개선이 가능하다고 알려져 있다.

그러나 아직 병렬화 성능의 추가적인 개선까지 고민할 단계는 아니다. 그보다 훨씬 성능이 많이 개선되는 방법을 아직 적용하지 않았기 때문이다. 이는 의도된 것으로 병렬화와 무관한 성능 개선을 전혀 적용하지 않은 곧이곧대로(vanilla)의 구현에서 병렬화가 어떤 효과를 가져오는지 관찰한 후에 다음 절에서 다룰 메모이제이션 최적화를 적용한 상태에서도 병렬화의 효과가 중첩될 수 있는지를 알아보고 싶었기 때문이다.

2. Memoizing implementation

인자값에 대응되는 함수의 결과값을 기억해 놓고, 이미 처리했던 인자를 받으면 계산 과정을 생략하고 기억했던 결과를 즉각 찾아주는 방식을 특히 함수형 프로그래밍의 맥락에서 메모이제이션(memoization)[1,2]이라고 한다. 이는 동적계획법(dynamic programming)과도 일맥상통하지만, 동적계획법에서는 하위 결과를 저장하는 데이터 구조를 명시적으로 드러내지만, 메모이제이션은 저장을 위한 데이터 구조를 드러내지 않고 원래 함수 f의 실행 결과를 기억하는 f_m으로 손쉽게 대체해서 활용할 수 있는 측면을 강조한다. 특히 하스켈 같은 함수형 언어에는 f로부

```haskell
import qualified Data.MemoCombinators as MC
import Data.RunMemo

bisimilarM sysP@(p0,esP) sysQ@(q0,esQ) = runMemo mcIntPair bisim' (p0,q0)
  where
  bisim' bisim(p,q) = and [or [bisim(p1,q1)| (q',b,q1) <- esQ, q==q', a==b]
                                           | (p',a,p1) <- esP, p==p']
                  && and [or [bisim(p1,q1)| (p',a,p1) <- esP, p==p', b==a]
                                           | (q',b,q1) <- esQ, q==q']

bisimilarM' sysP@(p0,esP) sysQ@(q0,esQ) = runMemo mcIntPair bisim' (p0,q0)
  where
  bisim' bisim(p,q) = and [or [bisim(p1,q1)| (q',b,q1) <- esQ, q==q', a==b]
                                           | (p',a,p1) <- esP, p==p']
             `parAnd` and [or [bisim(p1,q1)| (p',a,p1) <- esP, p==p', b==a]
                                           | (q',b,q1) <- esQ, q==q']

mcIntPair :: ((Int, Int) -> r) -> (Int, Int) -> r
mcIntPair = MC.pair MC.integral MC.integral
```

Fig. 4. Memoized Bisimulation Code (original vs. parallelized)

터 f_{m}을 유도하는 것을 돕는 라이브러리들이 개발되어 있다. 그런 라이브러리를 활용해 메모이제이션을 적용한 구현이 Fig. 3에 나타나 있다. 여기서도 병렬화되기 전 코드인 bisimilarM과 병렬화한 코드인 bisimilarM′의 차이는 **&&**과 **`parAnd`** 뿐이다.

메모이제이션은 쌍방시뮬레이션의 성능을 비약적으로 향상시킨다. 병렬화되지 않은 bisimilarM으로 상태 개수 200개로 설정해 생성한 DAG의 자기 자신에 대해 쌍방시뮬레이션 검사를 i7-8700 머신에서 실행하면 13.5초 정도 걸린다. 메모이제이션이나 병렬화를 전혀 적용하지 않고 상태 개수 20개로 쌍방시뮬레이션 검사를 하던 시간이면 메모이제이션이 적용된 코드로는 상태 개수 200개 DAG를 처리할 수 있는 셈이다. Fig. 5에 나타난 벤치마킹은 상태 개수 225개로 놓고 i7-8700으로만 진행하였다. Fig. 3과 그래프의 양상을 대조하자면 -N1에서 -N2로 급격히 시간이 감소하고 그 이후로는 다소 완만하게 감소한다. 이유를 추측해 보면, 병렬화로 인해 함수 호출이 많아지면 메모이제이션도 빠르게 진행되는 시너지 효과가 나타나 -N2만으로도 급격한 성능 개선을 보이는 것 같다. 그러나 메모이제이션의 효과를 어느 정도 본 다음에는 이미 기억하고 있는 비율이 높아 평균적으로 함수 호출에 걸리는 시간이 짧아질 것이다. 그러므로 각각의 병렬화된 작업에 걸리는 시간 자체가 짧아져 병렬화 효과 대비 오버헤드가 늘어나 그래프가 완만해지는 것으로 추측한다.

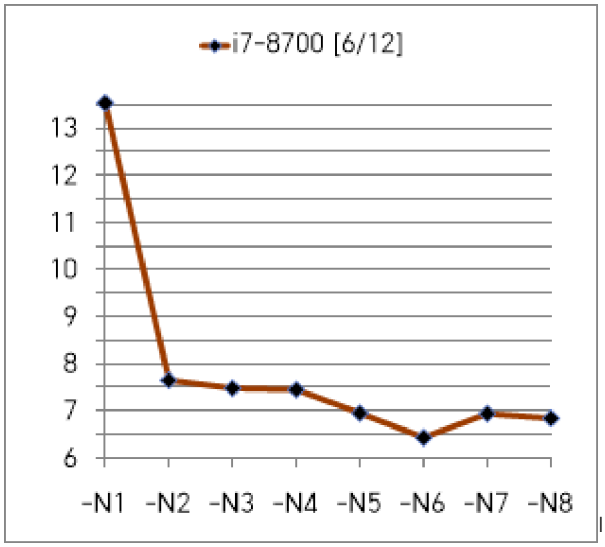

Fig. 5. Benchmarking (bisimilarM′ D_{225} D_{225}) on a 6-core/12-thread multicore CPU

IV. Related work

이 논문에서 활용한 병렬화 연산 par는 1992년 Lazy ML이라는 함수형 언어에 도입[3]한 것을 시초로, 이후에 실험적인 하스켈 컴파일러[4]에 구현되는 등 관련 연구가 이루어졌다. 이후 2000년대 후반에 들어 GHC 런타임에서 본격적으로 멀티코어를 위한 병렬 컴퓨팅이 안정적으로 지원[5]되기 시작했다. 전통적으로 스레드를 명시적으로 생성하여 관리하면서 스케줄러에 따라 다른 결과가 나올 수 있는 병렬 프로그램과 달리 정해진 계산 결과를 그대로

유지하는 것을 보장하며 성능 개선을 도모하는 방식을 결정적 병렬화(deterministic parallelism)라 부르며 par 같은 기본 병렬화 연산을 제공하는 것이 결정적 병렬화를 구현하는 대표적인 방법이다. 결정적 병렬화를 프로그래밍 언어 시스템에서 지원하려는 시도는 논리 프로그래밍을 병렬화하려는 연구[6]에서 그 기원을 찾을 수 있다.

이 논문에서는 멀티코어를 지원하는 컴퓨터 내에서의 병렬성만을 고려하며 물리적으로 떨어져 있어 네트워크로 통신하는 분산 컴퓨팅의 병렬성은 고려하지 않았다. 다만, 분산 환경에서도 함수형 프로그래밍 패러다임에서 비롯된 컴퓨팅 API가 (하둡 맵리듀스, 아파치 스팍 등등) 상당히 대중화되어 있으므로 분산 환경을 통해 쌍방시뮬레이션 알고리듬을 구현할 경우 비슷한 접근을 시도해 볼 여지는 충분하다. Haskell 관련 국내 연구로는 하스켈 컴파일러(GHC) 런타임에서 제공하는 기능을 활용하는 Eval 모나드와 얼랭 스타일 메시지 패싱 방식으로 분산 시스템과 멀티코어를 아우르며 병렬성을 활용할 수 있는 Cloud Haskell 프레임워크의 성능을 비교한 연구[7]가 있다. Eval 모나드는 GHC 런타임이 직접 지원하는 par 같은 병렬화 기능을 좀 더 체계적으로 구조화해 활용하기 위한 모나드로, 적은 수의 코어에서 좋은 성능을 보였지만 32~120 코어와 같이 다수의 코어에서는 Cloud Haskell의 성능이 더 나았다고 한다. 단, 병렬화를 고려하지 않고 작성된 코드에 Cloud Haskell을 적용하려면 GHC에 내장된 Eval 모나드 등을 활용할 때보다 더 많은 코드의 변경이 필요할 것이다.

쌍방시뮬레이션은 Milner[8]가 다중 프로세스로 구성된 시스템의 이론적 분석을 위한 프로세스 계산법의 의미를 컴퓨터 과학 이론 분야에 도입되었다. 이후 통신 채널의 동적 생성 등 연결이 변화하는 다중 프로세스 시스템을 모델링할 수 있는 파이계산법(pi-calculus)에 적합한 쌍방시뮬레이션 방식에 대한 연구가 이어졌는데, Sangiorgi[9]가 열린쌍방시뮬레이션(open-bisimulation)을 제시함으로써 부분 시스템의 동일성 결과를 재조합해 전체 시스템의 쌍방시뮬레이션 결과 도출에 활용 가능성이 열린 것이 관련 이론 분야의 대표적 도약이었다. 이후 이에 기반한 보안 프로토콜 분석[10] 등 구체적 분야로 응용을 시도하는 한편 최근까지도 핵심 알고리듬의 성질 개선[11]이 이루어지는 등 이론적 발전 또한 지속되고 있다.

쌍방시뮬레이션과 같은 모델 분석 외에도 다량의 데이터를 효과적으로 처리하기 위해 GHC의 병렬 기능을 활용하고 개선하는 연구도 꾸준히 지속되고 있다. 일차원 벡터 연산 병렬화를 넘어 다중 배열을 비롯해 더 복잡한 구조를 포함하는 병렬 배열 처리에 하스켈을 활용하는 연구이다. 이 주제와 관련한 최근 연구로는 Accelerate라는 eDSL로 하스켈 컴파일러(GHC)를 통해 CPU 및 GPU를 대상으로 최적화된 병렬처리 코드를 생성하여 벤치마크한 결과[12]를 들 수 있다.

V. Conclusions

이 논문에서는 병렬화를 통한 쌍방시뮬레이션 알고리듬의 성능 개선을 위해 두 단계로 벤치마크를 진행하였다. 첫째, 메모이제이션이 적용되지 않은 상태에서 대칭적인 알고리듬의 한가운데를 연결하는 논리 연산자에 하스켈 컴파일러(GHC) 런타임에서 제공하는 병렬 연산자를 적용하였다 (Fig. 1 참고). 둘째, 메모이제이션이 적용된 상태에서 마찬가지 방법의 병렬화를 지원하였다 (Fig. 4 참고). 두 단계 모두에서 병렬화로 2배 이상의 성능 개선을 확인할 수 있었다 (Fig. 3과 Fig 5).

벤치마크 결과를 종합해 보면 쌍방시뮬레이션 알고리듬 구현에서 병렬화와 무관한 최적화 기법이 적용되었더라도 병렬화를 통한 추가 성능 개선이 가능할 수 있다는 점을 확인하였다. 하지만 두 가지 다른 방식의 최적화 기법을 겹쳐서 적용하다 보면 일반적으로 소스코드의 복잡도가 높아질 수 있다는 점이 현실적인 문제이다. 상태 변화에 민감한 직렬적 구현에 병렬화와 무관한 최적화가 적용된 상태에서 스레드를 추가하는 등의 전통적 병렬화 기법을 추가로 적용하려면 코드의 수정이 상당히 많이 필요하며 원래 소스코드의 구조에서 멀어지는 경우도 발생한다.

하지만 이 논문에서는 쌍방시뮬레이션 알고리듬을 순수 함수형 패러다임에 따라 구현했기에 모듈화된 방식으로 메모이제이션과 병렬화를 적용하는 것이 가능하여 최적화가 전혀 적용되지 않은 구현과 두가지 모두 적용된 구현의 코드가 거의 같은 구조를 유지하는 (Fig 1.과 Fig. 4를 비교) 것이 가능했다.

후속 연구로는 상태 노드의 개수가 정해진 DAG가 아니라 더 유연하고 복잡한 시스템을 설계하는 데 활용되는 프로세스 계산법(process calculus)에 대한 최근 이론적 발전을 반영한 쌍방시뮬레이션 알고리듬[11]의 구현에서도 코드의 명확성을 크게 저해하지 않으면서 메모이제이션과 병렬화를 중첩해 성능 개선을 이룰 수 있는지 탐구하고자 한다.

ACKNOWLEDGEMENT

This work was supported by the NRF grant 2018R1C1B5046826, funded by Korea government (MSIT).

REFERENCES

[1] D. Michie. *Memo Functions: a Language Feature with "Rote-Learning" Properties*. Edinburgh University, Dept. of Machine Intelligence and Perception, 1967.

[2] J. Huges. "Lazy Memo Functions," Functional Programming languages and Computer Architecture, pp. 129-148, Springer-Verlag, 1985, DOI: 10.1007/3-540-15975-4_34

[3] K. Hammond and S. L. Peyton-Jones. *"Profiling Scheduling Strategies on the GRIP Multiprocessor"*. In Intl. Workshop on the Parallel Implementation of Functional Languages, pages 73-98, Aachen, Germany, Sept. 1992.

[4] P. W. Trinder, E. Barry Jr., M. K. Davis, K. Hammond, S. B. Junaidu, U. Klusik, H.-W. Loidl, and S. L. Peyton-Jones. "Low level Architecture-independence of Glasgow Parallel Haskell (GpH)". In Glasgow Workshop on Functional Programming, Pitlochry, Scotland, Sept. 1998.

[5] S. Marlow, S. L. Peyton-Jones, and S. Singh. "*Runtime Support for Multicore Haskell*." In ICFP 2009, pp. 65-78, Aug. 2009. ACM Press. DOI: 10.1145/1596550.1596563

[6] J. S. Conery and D. F. Kibler. "*Parallel Interpretation of Logic Programs*." In Proc. of the 1981 Conference on Functional Programming languages and Computer Architecture, pp. 163-170. 1981.

[7] Y Kim, H. An, S. Byun and G. Woo. *"Performance Comparison between Haskell Eval Monad and Cloud Haskell,"* Journal of KIISE, Vol 44, No. 8, pp. 791-802, August 2017, DOI: 10.5626/JOK.2017.44.8.791

[8] R. Milner. *"Communication and Concurrency"*. Prentice Hall. 1989. ISBN 0-13-114984-9

[9] D. Sangiorgi, "*A theory of bisimulation for pi-calculus*," Acta Informatica. Vol. 3, pp. 69–97, Springer, 1996

[10] A. Tiu, N. Nguyen and R. Horne. *"SPEC: an equivalence checker for security protocols,"* APLAS '16: Asian Symposium on Programming Languages and System, pp. 87-95, 2016

[11] R. Horne, K.Y. Ahn, S-W Lin and A. Tiu. "Quasi-Open Bisimilarity with Mismatch is Intuitionistic," LICS '18: Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science, July 2018, pp. 26–35 DOI: 10.1145/3209108.3209125

[12] R. Clifton-Everest, T. L. McDonell, M. Chakravarty, and G. Keller. *"Streaming Irregular Arrays."* Haskell 2017: Proceedings of the 10th ACM SIGPLAN International Symposium on Haskell. pp. 174–185, September 2017, DOI: 10.1145/3122955.3122971

Authors

Ki Yung Ahn joined the faculty of the Department of Computer Engineering at Hannam University, Daejeon, Korea, in 2018. His research interests are parallelism, functional programming, programming languages, type systems, and cyroptographic protocols.