함수형 반응적 프로그램 병렬화 기법

송지원O1, 이동주[1], 변석우[2], 우균[34]*
[1]부산대학교 전기전자컴퓨터공학과, [2]경성대학교 컴퓨터공학과,
[3]부산대학교 전기컴퓨터공학부, [4]LG전자 스마트제어센터
{jiwon, mrlee}@pusan.ac.kr, swbyun@ks.ac.kr, woogyun@pusan.ac.kr

Parallelism Techniques for Functional Reactive Programs

Jiwon SongO1, Dongju Lee[1], Sugwoo Byun[2], Gyun Woo[34]*
[1]Department of Electrical and Computer Engineering, Pusan National University,
[2]School of Computer Science & Engineering, Kyungsung University,
[3]School of Electrical and Computer Engineering, Pusan National University,
[4]Smart Control Center of LG Electronics

요 약

함수형 반응적 프로그램은 시간에 대해 연속적이고 이벤트에 반응적인 시그널 함수로 정의되며, 애로우 프레임워크 기반의 프로그램 합성을 통해 전체 프로그램이 구성된다. 시그널 함수의 병렬 합성은 데이터 흐름으로 의미상 병렬적으로 계산되지만 하스켈 실행 시스템에서는 병행적으로 수행된다. 본 논문은 멀티코어 시스템에서 병렬 처리될 수 있도록 명시적으로 사용하는 연산자를 시그널 함수로 추상하고 애로우 컴비네이터의 합성을 통해 간결하게 병렬 처리될 수 있도록 확장하는 방법을 제안하고 성능 개선의 효율성을 점검한다.

1. 서 론

함수형 반응적 프로그래밍 언어(FRP: Functional Reactive Programming)는 함수형 패러다임의 반응적 언어(reactive language)로 순수 함수형 언어(pure functional language)인 하스켈(Haskell)에 내장된 언어(embedded language)이다. FRP는 Elliott과 Hudak이 제안한 컴퓨터 애니메이션 언어인 Fran[1]에서 기본적인 아이디어를 얻어 발전하였다. FRP 프로그램의 구현은 시간에 대해 연속적이며 이벤트에 반응하는 대상을 시그널 함수(SF: Signal Function)로 정의된다. 시그널 함수의 합성을 담당하는 연산자는 애로우(arrow) 프레임워크를 기반으로 한다[2].

시그널 함수의 합성은 계산의 순서에 따라 순차적으로 합성되거나 순서에 관계없이 병렬적으로 합성될 수 있다. FRP 구현 라이브러리인 Yampa[3]에서 제공하는 시그널 함수의 병렬 합성은 FRP 프로그램의 의미상으로 병렬화를 지원하지만 하스켈 실행 시스템(runtime system)에서는 병행적으로 실행된다. 본 논문은 멀티코어 시스템 기반의 하스켈 실행 시스템에서 실제 코어(core)수에 따라 병렬 처리가 가능하도록 시그널 함수를 확장하는 방법을 제안하고 프로그램 수행을 통해 성능 개선 결과를 살펴본다.

2. 하스켈의 병렬처리

하스켈에서 병렬 프로그램을 작성하려면 함수를 Eval 모나드(Monad) 타입으로 추상화하고 runEval 시스템 함수를 통해 병렬 계산되도록 프로그래머가 명시적으로 구현해야 한다[4]. 그림 1은 Eval 모나드와 병렬, 순차 처리를 위한 함수, Eval 모나드의 병렬 계산 처리를 위한 runEval 함수의 타입이다. rpar 함수는 입력 매개변수가 병렬 계산처리 하도록 Eval 모나드로 추상화하며, rseq 함수는 순차 계산처리 되도록 추상화 한다.

```
data Eval a
instance Monad Eval
runEval :: Eval -> a
rpar :: a -> Eval a
rseq :: a -> Eval a
```

그림 1 Eval 모나드

그림 2에서는 rpar를 이용하여 (f x) 계산과 (f y) 계산을 병렬 처리되도록 명시하고, runEval 통해 실제 적용하는 예제를 나타낸다. 그림 2의 예제에서 보듯이

병렬 프로그래밍의 핵심은 병렬 계산 영역에 대해 rpar 연산자를 통해 명시적으로 병렬 계산 영역을 선언하고 runEval에 적용하는 것이다.

```
runEval $ do
  a <-  rpar (f x)
  b <-  rpar (f y)
  return (a,b)
```

그림 2 Eval 모나드를 사용한 병렬처리 예제

3. 시그널 함수의 병렬 처리

시그널 함수는 FRP 언어의 일등급 값(first-class value)이며, 애로우[5] 인터페이스를 기반으로 구현되어 있다. 그림 3은 시그널(signal)과 시그널 함수의 타입이다. 시그널 타입은 시간에 따라 변하는 값 이며, 시그널 함수는 시그널을 입력으로 받아 시그널을 출력으로 내는 함수이다.

```
Signal a :: Time -> a
SF a b : Signal a -> Signal b
```

그림 3 시그널 함수

FRP 프로그램은 각 기능을 담당하는 시그널 함수로 구성된 단위 모듈을 합성하여 메인 프로그램을 구성하며, 애로우 프레임워크 기반의 컴비네이터(combinator) 라이브러리를 이용하여 시그널 함수를 합성할 수 있다. 그림 4는 FRP의 대표적인 컴비네이터 라이브러리인 순차합성 연산자(>>>), 병렬합성 연산자(***)와 일반 함수를 시그널 함수 영역으로의 전환 연산자(arr)이다.

그림 4 시그널 함수 합성을 위한 컴비네이터

병렬합성 연산자로 합성된 2개 이상의 시그널 함수에 대해 최종 계산되는 튜플(tuple) 타입의 출력 시그널에 2절에서 소개한 rseq와 같은 명시적 병렬처리 연산자와의 합성을 통해 실행 시스템에서 병렬 계산이 가능하다. 그림 5는 병렬 합성된 시그널 함수와 병렬 실행을 위한 시그널 함수의 합성에 대한 구성을 나타낸 그림이다.

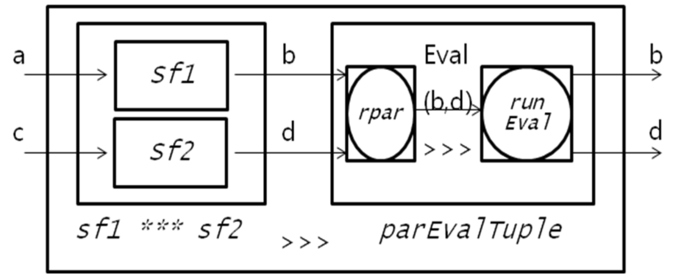

그림 5 시그널 함수의 병렬처리 구성

시그널 함수 sf1과 sf2는 병렬합성 연산자(***)를 이용하여 하나의 시그널 함수로 합성할 수 있으며, parEvalTuple 시그널 함수와의 순차 합성으로 기존 병렬합성 연산자도 동일한 튜플 타입의 결과를 낼 수 있다. parEvalTuple 시그널 함수는 그림 6과 같으며, 튜플 타입의 입력에 대해 rpar로 병렬 계산해야 할 대상을 명시하고 runEval을 적용하는 시그널 함수로 정의된다. parEvalList의 경우 튜플과 동일하며 리스트 요소에 대해 병렬 계산을 명시하고 있다.

```
parEvalTuple :: SF (a,b) (a,b)
parEvalTuple
  = arr(\(a,b) ->
        do { a' <- rpar a;
              b' <- rpar b;
             return (a',b') })
      >>> arr runEval

parEvalList :: SF [a] [a]
parEvalList
  = arr(\xs -> (parList rpar) xs)
      >>> arr runEval
```

그림 6 시그널 함수의 병렬처리

4. 실험

본 실험에서는 각기 다른 입력 속도에 따른 거리를 구하고 거리의 누적 값을 계산하는 5000개의 시그널 함수를 정의하고 1 사이클(cycle) 샘플링 주기에 실행 시간을 측정한다. 그림 7은 실험 프로그램으로 5000개의 시그널 함수를 리스트 타입 입/출력 형태의 시그널 함수로 합성하였고, 3절에서 정의한 parEvalList 시그널 함수를 이용하여 병렬 계산을 구현하였다.

```
parSplitList :: [SF a b] -> SF [a] [b]
…

spdToDist :: SF Double Double
spdToDist = integral
   >>> arr (\n -> foldr (+) 0 . [1..n])

spdToDist5000 :: SF [Double] [Double]
spdToDist5000 = parSplitList sfList
 where sfList = take 5000 (repeat
spdToDist)

mainSf :: SF [Double] [Double]
mainSf = spdToDist5000 >>> parEvalList
```

그림 7 실험 프로그램

실험 환경은 Intel core i5 1.8ghz (4core) 환경의 프로세서와 MAC OSX 10.10 운영체제하에 GHC 7.6.3 컴파일러로 구성되며, 컴파일 이후 동일한 바이너리 파일을 대상으로 수행 시 옵션을 통해 코어 수(core thread)를 조절하여 실험을 진행하였다.

표 1 병렬화 여부와 코어 수에 따른 실행 시간

병렬여부	코어수	실행 시간 (ms)	비율
비 병렬화	1	1506	100%
병렬화	2	944	62.70%
	3	874	58.03%
	4	833	55.35%

표 1은 하스켈 런타임 시스템에서 병렬 여부와 코어 수 옵션에 따른 실험 프로그램의 샘플링 주기에 대한 평균 실행 시간이다. FRP에서 샘플링 주기란 전체 시그널 함수가 한번 실행되어 계산되는 시간으로 시간이 짧을수록 계산이 빠르게 된 것으로 볼 수 있다. 실험 결과 비 병렬화 프로그램보다 병렬화 된 프로그램이 평균 수행시간이 짧음을 알 수 있다. 코어 수가 늘어감에 따라 평균 수행 시간이 적어져 성능 향상이 있음을 확인할 수 있었으며, 4개의 코어 기준으로 단일 코어 대비 최대 44% 향상이 있는 것을 확인할 수 있다.

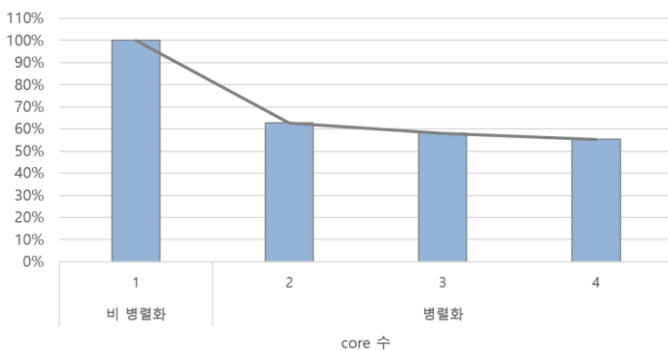

그림 8 병렬화 여부와 코어 수에 따른 실행시간 비율

5. 결론

FRP는 하스켈에 내장된 도메인 특화 언어(DSL)이며 애로우와 같은 강력한 타입 시스템을 기반으로 모듈화 및 합성이 용이하다. 본 논문에서는 병렬 처리를 위한 parEvalTuple과 같은 시그널 함수를 정의하고 애로우 컴비네이터를 이용하여 합성하는 방법을 제안했다. FRP 프로그램의 구현은 기존과 동일하게 유지되고 복잡한 구현 없이 합성을 통해 쉽게 병렬 처리로 확장됨을 확인할 수 있다. 또한 실험을 통해 멀티코어 실행 환경에서는 최대 44%만큼의 성능 향상을 확인할 수 있다.

향후 FRP를 다양한 응용 분야에 적용하기 위한 연구가 필요하다. 그 중 FRP를 활용할 수 있는 분야인 매니코어(Many Core) 시스템 실행 환경에 따른 성능과 관련하여 추가 연구가 필요하며, 함수형 언어를 기반으로 하는 FRP가 병렬 처리에 적합한 언어[6]가 되기를 기대한다.

ACKNOWLEDGMENT

본 연구는 미래창조과학부의 SW컴퓨팅산업원천기술개발 사업의 일환으로 수행하였음(B0101-15-0644, 매니코어 기반 초고성능 스케일러블 OS 기초연구).
*교신 저자 : 우균(부산대학교, woogyun@pusan.ac.kr).

참 고 문 헌

[1] Elliott, Conal, and Paul Hudak. "Functional reactive animation." ACM SIGPLAN Notices. Vol. 32. No. 8. ACM, 1997.

[2] P. Hudak, A. Courtney, H. Nilsson, and J. Peterson. "Arrows, robots, and functional reactive programming." in Summer School on Advanced Functional Programming 2002. Oxford University. vol. 2638 of Lecture Notes in Computer Science. pp. 159-187. Springer-Verlag, 2003.

[3] Yampa Homepage. https://wiki.haskell.org/Yampa.

[4] Simon Marlow. "Parallel and Concurrent Programming in Haskell." pp 16. O'REILLY. 2013.

[5] J. Hughes. "Generalising monads to arrows." Science of Computer Programming, vol. 37. pp. 67~111. May 2000.

[6] 김진미, 변석우, 김강호, 정진환, 고광원, 차승준, 정성인. "매니코어 시대를 대비하는 Haskell 병렬 프로그래밍 동향". ETRI 전자통신동향분석, 29권 제5호. pp 167-175. 2014년 10월.