매니코어 환경을 위한 공유 메모리 기반의 Haskell 병렬 프로그래밍 모델

김연어[O1], 천준석[1], 김민성[1], 왕인서[1], 변석우[2], 우균[1]
[1]부산대학교 정보융합공학과, [2]경성대학교 소프트웨어학과
[1]{yeoneo, jscheon, msungkim, inseowang, woogyun}@pusan.ac.kr, [2]swbyun@ks.ac.kr

A Shared-Memory based Haskell Parallel Programming Model for the Manycore Environments

Yeoneo Kim[O1], Junseok Cheon[1], Minsung Kim[1], Inseo Wang[1], Sugwoo Byun[2], Gyun Woo[1,3]
[1]Dep. of Information Convergence Engineering, Pusan National University,
[2]School of Computer Science & Engineering, Kyungsung University

요 약

최근 프로그래밍 언어에서는 CPU 코어 수 증가에 따라 병렬 프로그래밍에 대한 관심이 높아지고 있다. 다양한 프로그래밍 언어 중 함수형 언어인 Haskell은 매니코어 환경에 적합한 언어로 알려져 있다. 이 논문에서는 매니코어 환경에 적합한 병렬 프로그래밍 모델인 SAM을 공유 메모리를 이용하여 개선하는 방법을 소개하고자 한다. 이를 위해 이 논문에서는 기존 SAM의 장단점을 살펴보고 공유 메모리를 이용해 단점을 개선하는 방법을 소개한다. 그리고 이를 기반으로 공유 메모리 기반의 SAM인 SAM_{Shm}을 설계하고 실험을 통해 SAM_{Shm}의 성능을 분석해 보았다. 그 결과 SAM_{Shm}의 확장성이 SAM_{Soc} 대비 약 40.67%, SAM_{STM} 대비 약 25.54% 향상된 것을 확인하였다.

1. 서 론

최근 CPU 시장은 인텔과 AMD의 점유율 확보 전쟁이 발발하면서 코어 수를 늘리는 경쟁이 다시금 이루어지고 있다. 이 덕분에 최근에는 서버에서 사용하는 CPU의 경우 단일 CPU의 코어 수가 64코어까지 지원하는 모델도 나오고 있다. 즉 일반 사용자도 매니코어를 사용하는 환경에 가까워지고 있다.

이러한 추세 때문에 최근 프로그래밍 언어에서도 병렬 프로그래밍에 대한 관심이 높아지고 있다. 이는 최근에 개발된 프로그래밍 언어의 경향을 살펴보아도 알 수 있는데, Go나 Rust, Julia와 같은 언어는 언어 수준에서 별도의 병렬 프로그래밍 방식을 제공하고 있다. 이러한 프로그래밍 언어에서는 효과적인 병렬화를 위해 경량화된 쓰레드(lightweight thread)를 사용하거나 변수의 상태 변경을 막는(immutable) 방법을 이용하고 있다.

이러한 방법은 이미 함수형 언어에서 널리 사용되고 있던 방법이기 때문에 함수형 언어는 병렬화에 적합하다고 볼 수 있다. 순수한 함수형 언어는 상태를 저장한다는 개념이 없기 때문에 부수 효과(side effect)가 없어 계산 순서가 정해져 있지 않다. 그렇기 때문에 각 계산을 병렬로 수행하여 문제가 발생하지 않는다. 이러한 특징 때문에 최근에는 함수형 언어를 이용한 병렬 프로그래밍도 관심을 받고 있다.

이 논문에서 다루고자 하는 다양한 함수형 언어 중 Haskell은 특히 병렬화에 적합한 프로그래밍 언어로 알려져 있다. Haskell은 함수의 순수성을 유지하면서도 모나드를 이용하여 입출력과 같은 부수 효과를 분리하고 있기 때문에, 효과적인 병렬 프로그래밍이 가능하다. 더욱이 Haskell은 다양한 병렬 프로그래밍 모델을 제공하고 있기 때문에 상황에 적합한 병렬 프로그램을 선택하여 사용할 수 있다[1]. 이러한 특징 때문에 Haskell은 매니코어 환경에서도 효과적으로 프로그래밍이 가능하다.

이 논문에서는 Haskell의 다양한 프로그래밍 모델 중 SAM(simple actor model)[2]의 성능을 개선하고자 한다. SAM은 액터 모델[3] 기반의 병렬 프로그래밍 모델로 매니코어 환경을 위해 개발된 라이브러리이다. 기존 SAM은 메시지 전달 채널로 소켓이나 STM(software transactional memory)을 사용하고 있지만 소켓 생성 오버헤드 등의 문제가 여전히 남아있다. 이 논문에서는 이러한 문제를 해결하기 위해 리눅스의 공유 메모리(shared memory)를 이용한 방법인 SAM_{Shm}을 제안한다.

이를 위해 먼저, 매니코어 환경에 적합한 병렬 프로그래밍 모델인 SAM이 어떤 모델인지를 살펴본다. 그리고 3절에서는 SAM_{Shm}을 어떻게 설계하였는지를 소개하고 구현에 사용된 일부 함수를 소개하고자 한다. 4절에서는 SAM_{Shm}을 이용해 기존 SAM과 성능을 비교하고자 한다. 이후 5절에서는 실험 내용에 대해 논의하고 6절에서 결론을 맺는다.

2. 관련 연구

이 절에서는 이 논문에서 대상으로 하는 SAM에 대해 살펴보고자 한다. SAM은 액터 모델을 기반으로 하는 Haskell 병렬 프로그래밍 모델로, 매니코어 환경에 적합한 모델이다[2]. 기존 Haskell 병렬 프로그래밍 모델은 성능은 우수하지만 프로그래밍이 어렵거나, 프로그래밍이 간단하지만 GC(garbage collection) 등 부담으로 인해 성능 향상이 낮은 문제가 있었다[4]. SAM은 이러한 문제점을 해결해 매니코어 환경에서 쉽게 병렬 프로그

*이 논문은 2021년도 정부(과학기술정보통신부)의 재원으로 정보통신기획평가원의 지원을 받아 수행된 연구임(No.2014-3-00035, 매니코어 기반 초고성능 스케일러블 OS 기초연구 (차세대 OS 기초연구센터)).
*교신 저자: 우균(부산대학교, woogyun@pusan.ac.kr).

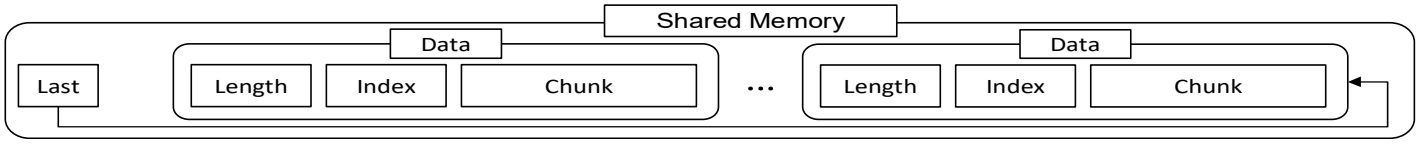

그림 2 SAM_{Shm}의 데이터 저장 구조

래밍이 가능하면서 우수한 성능을 보장하는 병렬 프로그래밍 모델이다.

SAM을 이용한 병렬 프로그래밍은 Haskell에서 많이 사용되는 고차 함수(highorder function)를 이용한 방법이다. 그렇기 때문에 기존 프로그램도 쉽게 병렬로 변경할 수 있다. 이 과정을 간단한 프로그램 예로 설명하면, `map`을 사용한 Haskell 프로그램을 병렬로 변경하기 위해서는 `map` 함수를 `mCoreMap` 함수로 변경해 주고 몇 가지 코드만 추가해 주면 해당 프로그램은 병렬로 수행 가능하다.

SAM은 총 두 가지 메시지 채널을 제공하고 있는데, 소켓과 STM을 이용하는 채널이다. 소켓을 사용하는 SAM은 SAM_{Soc}라는 이름으로 불리며 가장 처음 개발된 모델이다[2]. 이 모델은 액터 노드 간의 메시지 전달에 소켓 통신을 이용하는 방법으로 각 액터는 OS 프로세스 수준으로 실행된다.

두 번째로 STM을 사용하는 SAM은 SAM_{STM}으로서[5], 기존 SAM의 소켓 오버헤드를 줄이기 위해 제안된 모델이다. SAM_{STM}은 SAM_{Soc}와 달리 메시지 전달 채널로 소켓이 아닌 STM을 이용하는 방법이다. 그리고 이 모델의 각 액터는 STM 데이터 공유 문제로 Haskell 쓰레드 수준으로 실행된다.

SAM_{Soc}와 SAM_{STM}은 각각 장단점이 존재한다. SAM_{Soc}의 경우 소켓 오버헤드라는 단점이 존재하지만 계속 코어 수가 늘어나더라도 실행 속도가 지속적으로 상승하는 특성이 있다. SAM_{STM}의 경우 메시지 전달에 오버헤드가 없다는 장점이 있지만, Haskell STM의 특성상 OS 프로세스 간 공유가 불가능하기 때문에 Haskell 쓰레드를 사용하여 코어가 늘어남에 따른 실행 속도 상승에 제한이 있다. 이처럼 기존의 SAM은 장단점 존재하기 때문에 상황에 맞춰 적합한 모델을 사용해야 한다.

3. SAM_{Shm} 설계

이 절에서는 2절에서 소개한 SAM의 단점을 해결한 모델인 SAM_{Shm}을 소개하고자 한다. 관련 연구에서 살펴본 바와 같이 기존 SAM은 소켓 버전의 경우 소켓을 사용한 오버헤드가 문제이고 STM 버전은 Haskell 쓰레드를 사용하여 문제가 발생하고 있다. 그렇기 때문에 이 논문에서는 실행되는 액터의 단위를 OS 프로세스 수준으로 실행하면서 메시지 전달을 공유 메모리를 이용하는 방법을 제안하고자 한다.

이를 위해 우선 SAM_{Shm}에서 어떤 방식으로 액터 간에 메시지를 전달하는지를 살펴본다. SAM_{Shm}은 OS 프로세스를 이용해 실행되기 때문에 OS 프로세스 사이에 데이터를 주고받을 수 있는 수단이 필요하다. 리눅스에서는 다양한 프로세스 간 통신 수단(IPC: inter-process communication)을 제공하고 있는데, 이 논문에서는 그중 가장 속도가 빠르다고 알려진 공유 메모리를 사용하고자 한다.

이 논문에서 사용하고자 하는 공유 메모리는 리눅스 커널에 공유할 메모리 공간을 요청하여 필요한 메모리 공간을 생성하고 이를 여러 프로세스가 같이 사용하는 방법이다. 이 방법을 사용하기 위해서는 C 언어에서 제공하는 API를 이용해야 한다. 그래서 SAM_{Shm}에서는 Haskell의 FFI(foreign function interface)를 이용하여 공유 메모리를 할당하는 함수인 `initShmHaskell`과 공유 메모리 읽기 함수인 `getShmHaskell`, 쓰기 함수인 `insertShmHaskell` 함수를 작성하였다. SAM_{Shm}은 이 함수를 이용해 그림 1과 같은 구조로 동작하게 된다.

그림 1은 SAM_{Shm}의 실행 구조인데, SAM_{Shm}에서는 크게 세 영역에서 데이터를 주고 받게 된다. 우선 Haskell의 마스터 영역에서 데이터를 생성한 뒤 C 영역의 메모리 관리자에게 데이터를 전달한다. 그리고 전달받은 데이터를 커널 영역의 공유 메모리에 저장한 뒤 `forkProcess` 함수를 이용하여 Haskell 슬레이브 프로그램을 실행한다. 슬레이브 프로그램은 메모리 관리자를 통해 지정된 영역의 데이터를 가져와 필요한 함수를 적용한 뒤 다시 메모리 관리자를 통해 데이터를 저장한다. 이후 모든 슬레이브의 동작이 완료된다면 마스터 영역에서 데이터를 취합한다.

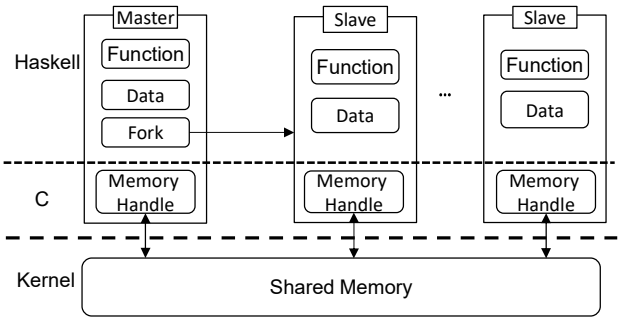

그림 1 SAM_{Shm}의 실행 구조

그리고 SAM_{Shm}은 공유 메모리를 사용하여 발생하는 불필요한 잠금(lock)을 줄이기 위해 공유 메모리에 저장되는 데이터 구조를 설계하였다. 여러 프로세스가 공유 메모리를 쓰는 경우 데이터의 무결성을 유지하기 위해 동시에 여러 프로세스가 접근하지 못하도록 해야 한다. 그렇기 때문에 SAM_{Shm}은 그림 2와 같이 데이터 구조를 활용하고 있다.

그림 2의 데이터 구조를 살펴보면 저장되는 데이터는 크게 두 가지로 구분된다. 첫 번째로 `Last`는 현재 저장된 영역의 마지막 위치를 가리키는 값으로서 데이터를 효과적으로 쓰기 위해 관리되는 값이다. Haskell에서 `insertShmHaskell` 함수를 이용해 데이터를 저장하게 되면 `Last`의 값을 참조하여 맨 마지막에 데이터를 저장하게 된다. 이 값을 유지하고 있으면 모든 쓰기에 대해 잠금을 잡지 않고 `Last` 값을 갱신할 때만 잠금을 잡으면 되기 때문에 불필요한 잠금을 줄일 수 있다.

다음으로 `Data`는 실제 저장되는 값을 위한 구조이다. `Data` 구조는 다시 세 가지로 구분되는데, 우선 `Length`는 저장될 값의 크기를 의미한다. SAM_{Shm}은 공유 메모리에 데이터를 `ByteString` 형태로 저장하기 때문에 각 데이터의 크기는 가변적이다. 그렇기 때문에 첫 번째로 값의 길이를 저장한다. 그리고 `Index`는 해당 데이터가 몇 번째 Haskell 리스트에서 온 값인지를 저장하는 영역이다. 이 값은 함수를 병렬로 수행하면 실행 순서가 일정하지 않기 때문에 발생할 수 있는 문제를 해결하기 위한 값으로 추후 마스터 노드가 `Index` 영역의 값을 참조하여 원래 순서대로 데이터를 조합하는 데 사용된다. 마지막으로 `Chunk`는 `ByteString` 형태로 변경된 데이터 문자열이 저장되는 영역이다.

4. 실험

이 절에서는 3절에서 설계한 SAM_{Shm}의 성능을 살펴보고자 한다. 이 논문에서는 이를 위해 기존 SAM 병렬 프로그래밍 모델인 소켓 버전의 SAM인 SAM_{Soc}와 STM 버전의 SAM인 SAM_{STM}을

비교 대상으로 이용하였다. 그리고 실험에 사용한 응용 프로그램은 Java 소스 코드를 대상으로 표절 여부를 판단하는 프로그램을 이용하였다.

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. 실험 데이터로는 Java 코드 108개를 대상으로 코드 간의 유사도 계산을 수행하였다. 그리고 이를 실행하기 위한 실행 옵션으로 SAM_{Soc}와 SAM_{Shm}은 별다른 옵션을 추가하지 않았지만, SAM_{STM}은 GC 문제를 해결하기 위해 별도로 메모리 옵션인 `-A128M -H512M`를 추가한 상태로 실행하여 실험을 진행하였다. 위와 같은 실행 환경을 이용해 세 병렬 프로그래밍 모델을 실행한 결과는 표 1, 그림 3과 같다.

표 1 SAM_{Shm}의 실행 시간 비교 결과

코어 수	SAM_{Soc}		SAM_{STM}		SAM_{Shm}	
	실행시간	확장성	실행시간	확장성	실행시간	확장성
1	166.88	1.00	144.67	1.00	161.74	1.00
4	52.17	3.20	39.82	3.63	45.79	3.53
8	34.05	4.90	24.06	6.01	27.73	5.83
12	28.27	5.90	19.97	7.25	21.62	7.48
16	24.12	6.92	16.61	8.71	18.62	8.69
20	23.81	7.01	15.58	9.28	17.67	9.15
24	22.54	7.40	16.46	8.79	16.47	9.82
28	21.46	7.78	16.74	8.64	15.49	10.44
32	21.09	7.91	16.31	8.87	14.53	11.13

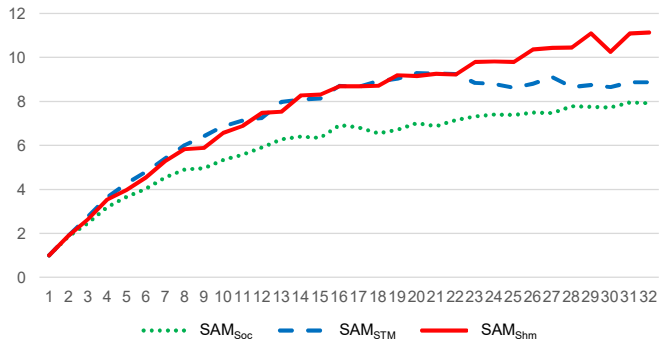

그림 3 SAM_{Shm}의 성능 비교 결과

표 1은 이 논문에서 제안한 SAM_{Shm}의 실행 시간을 측정한 결과의 일부이며, 그림 3은 표 1의 데이터를 이용해 확장성을 비교한 결과이다. 우선 SAM_{Soc}는 단일 코어에서 166.88초의 실행 시간이 소요되었으며, 32코어에서 21.09초가 소요되어 확장성은 7.91로 계산되었다. 그리고 SAM_{STM}은 단일 코어에서 144.66초의 실행 시간이 소요되었으며, 32코어에서 16.31초가 소요되어 확장성은 8.87으로 계산되었다. 마지막으로 SAM_{Shm}은 단일 코어에서 161.74초의 실행 시간이 소요되었으며, 32코어에서 14.53초가 소요되어 확장성은 11.13으로 계산되었다. 즉, SAM_{Shm}은 SAM_{Soc} 대비 약 40.67% 확장성이 높아졌으며, SAM_{STM} 대비 약 25.45% 확장성이 높아진 것으로 나타났다.

5. 고찰

이 절에서는 4절에서 진행한 실험에 대해 논의하고 SAM_{Shm}의 한계점에 대해 논의하고자 한다. 우선 실험 결과를 살펴보면 SAM_{Shm}은 다른 SAM보다 성능이 우수한 것으로 나타났다. 이는 확장성만 우수한 것이 아니라 물리적으로 수행되는 시간도 SAM_{Soc} 대비 약 7초가량 빠르고, SAM_{STM} 대비 약 2초가량 빠른 것을 알 수 있다. 하지만 단일 코어 환경에서 실행 시간을 살펴보면 SAM_{STM}보다 실행 시간이 더 오래 걸리는 것을 확인할 수 있다. 이는 소켓을 사용한 SAM_{Soc}보다는 빠르지만 SAM_{STM}보다는 느린 결과이다. 이 문제는 매니코어 관점에서는 문제가 되지 않지만 향후에는 최적화를 통해 해결할 필요가 있다.

또 다른 논의 사항으로는 SAM_{Shm}의 한계점이다. 3절에서 소개한 바와 같이 SAM_{Shm}은 계산할 데이터를 `ByteString` 형태로 변환하여 공유 메모리에 저장하는 방식을 채택하고 있다. 이 때문에 Haskell의 인코딩/디코딩 함수를 이용하여 `ByteString` 형태로 변환하는데, 이 때문에 적용 가능한 데이터 타입에 제한이 있다. Haskell에서 SAM_{Shm}을 사용하기 위해서는 `Binary` 클래스로 표현 가능한 타입만 사용이 가능하며, 이는 `MVar`와 같은 저장 가능한 변수를 데이터로 저장할 수 없음을 의미한다. 또한, 새로운 사용자 정의 타입을 이용하기 위해서는 `Binary` 클래스에 대한 인스턴스를 선언해 줘야 SAM_{Shm}을 적용할 수 있다.

6. 결론

이 논문에서는 기존 SAM의 한계점을 해결한 SAM_{Shm}을 제안하였다. 제안한 SAM_{Shm}은 각 액터를 OS 프로세스로 실행하면서 메시지 전달 채널로 공유 메모리를 사용하는 모델이다. 이 논문에서는 이를 어떻게 설계하였는지를 소개하고, 구현된 SAM_{Shm}을 이용하여 기존 SAM인 SAM_{Soc}와 SAM_{STM}과 성능을 비교하는 실험을 32코어 환경에서 진행하였다. 실험 결과 확장성을 기준으로 SAM_{Soc} 대비 약 40.67%, SAM_{STM} 대비 약 25.45% 성능 향상이 있는 것을 확인할 수 있었다. 즉 SAM_{Shm}이 기존의 SAM의 한계점을 극복하고 개선된 것을 알 수 있다.

향후 연구로는 32코어보다 더 많은 코어의 환경에서 실험을 진행하고자 한다. 이는 기존 SAM_{Soc} 사례를 살펴보아도 코어 수가 더 늘어나면 성능이 바뀌는 경우도 발생할 수 있기 때문에 코어 수를 늘려가며 실험을 진행하고자 한다. 이외에도 고찰에서 살펴본 단일 코어 환경에서 다른 SAM 대비 많은 시간이 소요되는 문제점을 분석하고자 한다.

참 고 문 헌

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

[2] Y. Kim, J. Cheon, X. Liu, S. Byun and G. Woo, "SAM: A Haskell parallel programming model for many-core systems," *In Proceedings of 2018 IEEE International Conference on Applied System Invention (ICASI)*, pp. 822-826, 2018.

[3] 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.

[4] H. Kim, J. Byun, S. Byun and G. Woo, "An approach to improving the scalability of parallel Haskell programs," *Journal of Theoretical & Applied Information Technology*, Vol. 95, No. 18, pp. 4826-4835, 2017.

[5] Y. Kim, J. Cheon, T. Hur, S. Byun and G. Woo, "SSAM: A Haskell Parallel Programming STM Based Simple Actor Model," *Journal of Physics: Conference Series*, Vol. 1566, No. 1, 2020.