ISSN 2383-6318(Print) / ISSN 2383-6326(Online)
KIISE Transactions on Computing Practices, Vol. 23, No. 8, pp. 459-465, 2017. 8
https://doi.org/10.5626/KTCP.2017.23.8.459

GC-Tune을 이용한 Haskell 병렬 프로그램의 성능 조정

(Tuning the Performance of Haskell Parallel Programs Using GC-Tune)

김 화 목 † 안 형 준 † 변 석 우 †† 우 균 †††
(Hwamok Kim) (Hyungjun An) (Sugwoo Byun) (Gyun Woo)

요 약 매니코어 기술에 힘입어 컴퓨터 하드웨어의 성능이 향상되고 있지만 그에 비례한 소프트웨어 성능 증가는 다소 미미한 실정이다. 함수형 언어는 병렬 프로그램의 성능을 향상시키는 대안 중 하나이다. 이러한 언어는 부수효과가 없는 순수한 수식을 통해 내재된 병렬성을 지원하기 때문이다. 함수형 언어인 Haskell은 모나드를 기반으로 하는 다양하고 쉬운 병렬 구조를 제공하기 때문에 병렬 프로그래밍에서 널리 사용된다. 하지만 Haskell로 작성된 병렬 프로그램의 성능 확장성은 코어 수가 증가함에 따라 변동이 큰 경향이 있다. 이는 프로그램 실행에 있어 가비지 컬렉션이 공간과 시간에 모두 영향을 미치는데 Haskell은 이러한 가비지 컬렉션을 사용하는 가상머신 위에서 실행되기 때문이라고 추정된다. 따라서 본 논문에서는 GC-Tune이라는 메모리 튜닝 도구를 사용하여 이 추정이 맞는지 검증하고 Haskell 병렬 프로그램의 성능 확장성을 높이는 방법을 모색한다. 병렬 Haskell 표절 검사 프로그램을 대상으로 실험한 결과 성능 확장성이 향상되었다. 특히 메모리 튜닝을 하지 않은 프로그램에 비해 속도 향상의 변동 범위가 39% 감소하였다.

키워드: 매니코어, 병렬 프로그래밍, 하스켈, 가비지 컬렉션, GC 튜너, 표절검사 프로그램

Abstract Although the performance of computer hardware is increasing due to the development of manycore technologies, software lacking a proportional increase in throughput. Functional languages can be a viable alternative to improve the performance of parallel programs since such languages have an inherent parallelism in evaluating pure expressions without side-effects. Specifically, Haskell is notably popular for parallel programming because it provides easy-to-use parallel constructs based on monads. However, the scalability of parallel programs in Haskell tends to fluctuate as the number of cores increases, and the garbage collector is suspected to be the source of this fluctuations because it affects both the space and the time needed to execute the programs. This paper uses the tuning tool, GC-Tune, to improve the scalability of the performance. Our experiment was conducted with a parallel plagiarism detection program, and the scalability improved. Specifically, the fluctuation range of the speed-up was narrowed down by 39% compared to the original execution of the program without any tuning.

Keywords: manycore, parallel programming, Haskell, garbage collection, GC-Tune, plagiarism detection program

· 이 논문은 2017년도 정부(과학기술정보통신부)의 재원으로 정보통신기술진흥센터의 지원을 받아 수행된 연구임(No.B0101-17-0644,매니코어 기반 초고성능 스케일러블 OS 기초연구)
· 이 논문은 제43회 동계학술발표회에서 'GC-Tune을 이용한 Haskell 병렬 프로그램의 성능 조정'의 제목으로 발표된 논문을 확장한 것임

† 학생회원 : 부산대학교 전기전자컴퓨터공학과
hwamok@pusan.ac.kr
hyungjun@pusan.ac.kr
†† 종신회원 : 경성대학교 컴퓨터공학과 교수
swbyun@ks.ac.kr
††† 종신회원 : 부산대학교 전기전자컴퓨터공학과 교수
LG전자 스마트 제어 센터
(Pusan Nat'l Univ.)
woogyun@pusan.ac.kr
(Corresponding author임)

논문접수 : 2017년 3월 6일
(Received 6 March 2017)
논문수정 : 2017년 5월 31일
(Revised 31 May 2017)
심사완료 : 2017년 5월 31일
(Accepted 31 May 2017)

정보과학회 컴퓨팅의 실제 논문지 제23권 제8호(2017. 8)

1. 서 론

하드웨어 기술의 발전에 힘입어 고성능 프로세스가 늘어나고 있다. 그리고 추세는 단일 코어의 성능 향상이 아니라 코어의 수를 늘려가는 방향으로 바뀌고 있다. 프로세스는 사람과 비교하면 일종의 두뇌이다. 따라서 코어의 수가 많은 프로세스의 경우 동시에 독립적인 연산을 통해 병렬로 데이터를 처리할 수 있으며 궁극적으로 한 번에 컴퓨터가 처리할 수 있는 데이터의 양이 늘어난다[1].

하지만 병렬 하드웨어의 성능 발전과 비교하면 병렬 소프트웨어의 성능은 하드웨어 성능을 따라가지 못하고 있다. 소프트웨어 작성 자체가 복잡한 탓도 있겠지만, 병렬 소프트웨어의 경우에는 작성 기법이 지금까지 사용해 왔던 순차적 명령형 작성 기법과 다소 다르기 때문이다. 함수형 언어는 이에 대한 대안으로 떠오르고 있다.

특히 순수 함수형 언어(pure functional language)인 Haskell은 병렬 프로그래밍에 적합한 대안으로 떠오르고 있다[2]. Haskell은 부수효과(side effect)가 없어 출력 값은 입력에 의해서만 결정된다. 따라서 입력이 같은 여러 개의 함수를 병렬로 처리해도 결과가 변하지 않는다. 이는 Haskell이 기존의 절차형 언어와 달리 병렬성이 내재되어 있음을 뜻한다.

또한 Haskell은 STM(software transactional memory)[3], Cloud Haskell[4], HdpH(Haskell distributed parallel Haskell) 등 다양한 병렬화 도구를 지원하고 가벼운 스레드(lightweight thread)를 제공하여 병렬화에 필요한 비용이 적어 병렬 프로그래밍에 적합하다.

기존에 작성된 Haskell 병렬 프로그램을 실행한 결과는 코어 수에 따라 균일한 성능 확장성이 보장되지 못한다. 여기서 성능 확장성이란 사용하는 코어에 비례한 데이터 처리량 또는 처리 속도를 의미한다. 확장성 문제는 비단 Haskell만의 문제는 아니다. 작성되는 병렬 프로그램의 알고리즘 설계의 문제일 수도 있고 다른 관리 오버헤드 때문일 수도 있다. 하지만 Haskell의 경우에는 프로그램의 공간과 시간에 영향을 주는 가비지 컬렉션(GC: garbage collection) 때문인 것으로 생각된다.

이 추정이 맞는지 확인하기 위해 전형적인 Haskell 병렬 프로그램의 수행 성능과 GC 영향을 측정하고자 한다. 구체적으로 메모리 튜너인 GHC GC-Tune을 사용하여 이를 확인하고 병렬 Haskell 프로그램의 성능 확장성을 높이는 방법에 관해 연구하고자 한다. GHC GC-Tune은 GHC(Glasgow Haskell Compiler)에서 제공하는 메모리 프로파일링 도구이다. 특히 GC에 많은 영향을 받는 Haskell의 단점을 보완하기 위해 사용자가 요구하는 코어에 맞는 최적의 GC 옵션을 자동으로 찾아준다. 사용자는 이러한 옵션을 참고하여 코어별 최적의 프로그램 실행환경을 구성할 수 있다[5].

본 논문의 구성은 다음과 같다. 2장에서는 관련 연구로 순수 함수형 언어인 Haskell과 세대별 GC 그리고 메모리 튜너인 GHC GC-Tune에 대하여 설명한다. 3장에서는 Haskell로 작성된 표절검사 프로그램을 GHC GC-Tune을 사용하지 않고 실행한 결과를 그래프와 함께 설명한다. 다음으로 4장에서는 GHC GC-Tune을 사용하여 실행한 결과를 그래프와 함께 설명한다. 5장에서는 앞 장에서의 실험 결과에 대해 토의한 후 마지막으로 6장에서 결론을 맺는다.

2. 관련 연구

2.1 순수 함수형 언어 Haskell

순수 함수형 언어인 Haskell은 프로그램 전체가 수학 분야의 함수와 같다. 함수의 결과 값은 오직 인자로 넘겨진 입력 값에 의해서만 결정되는 순수성이 보장된다. 따라서 제어 흐름이 단순하고 실행 순서나 같은 입력 때문에 결과가 달라지는 부수효과가 없어 병렬성이 내재되어 있다.

또한, Haskell은 병렬화 성능이 뛰어나다고 알려진 Erlang보다 더 가벼운 경량화 스레드를 제공하여 문맥교환(context switch)의 오버헤드가 무척 작다. 게다가 정적 타입 시스템을 사용하고 있어 프로그램 실행 중 오류로 인한 비정상적 종료 확률이 낮다[6]. 이외에도 비교적 간결하고 짧은 시간에 프로그램을 작성할 수 있고 다양한 병렬 프로그래밍 모델을 지원하기 때문에 기존의 순차적 형식으로 작성된 프로그램에 병렬화를 부여하기가 쉽다[7].

2.2 세대별 GC

GC는 객체 생성에 할당된 메모리 영역 가운데 더 이상 사용할 수 없게 된 영역 혹은 참조되지 않는 영역을 탐지하여 자동으로 메모리를 해제하는 기법이다. Java[8], Javascript, C#, Haskell 등의 언어는 처음부터 GC가 자동으로 지원되게 설계되었다. 따라서 이러한 환경에서는 프로그래머가 할당한 메모리 영역의 전체를 수동으로 관리할 필요가 없다.

GC는 메모리 누수, 해제된 메모리의 이중 해제, 이미 해제된 메모리에 접근하는 버그 등 다양한 버그들을 줄이거나 막을 수 있는 장점이 있다. GC가 동작하기 위해서는 전체 프로그램이 일정 시간 정지해야 하는데 프로그램 성능이 중요한 병렬 프로그램의 경우에 이러한 특성은 치명적인 단점으로 작용한다. 또한, GC가 일어나는 시점과 점유 시간을 프로그래머가 예측하기 어렵고 객체의 메모리가 해제되는 시점을 알기 어려운 점도 존재한다.

Haskell의 대표적인 GC 기법인 세대별 GC는 대부분의 생성된 객체들은 일찍 소멸한다는 가설을 바탕으로

동작한다. Haskell의 객체는 실제로 75~95%가 어린 세대의 상태로 소멸하며 단지 5% 정도만이 오래 살아남는다[9]. 생성된 객체는 힙 메모리 내에 물리적으로 분리된 두 개 이상 별개의 영역에 배치되며 시간 또는 크기에 따라 여러 세대로 나누어서 관리된다. GC은 여러 세대 중 가장 어린 객체들이 존재하는 세대에서 집중적으로 일어난다. 따라서 최소한의 동작으로 최대한 많은 메모리 공간을 회수할 수 있는 장점이 있으며 GC 시간 또한 절약된다[10].

어린 세대들을 위한 GC는 비교적 짧은 시간 안에 수행되며 수집 간격이 일정하고 충분하다면 전체 프로그램에 큰 영향을 미치지 않는다. 하지만 오래된 세대에 대해 GC를 수행해야 할 때도 있는데 이는 프로그래머가 힙 메모리 공간을 모두 사용하였거나 어린 세대만 GC를 수행해서 회수되는 메모리 공간이 부족할 것으로 예상되는 경우이다. 따라서 최악의 경우에는 GC에 수행되는 예상시간이 많이 늘어날 수 있다. Haskell 병렬 프로그램은 코어 수에 따른 확장성이 보장되지 않는 단점이 있는데 세대별 GC의 이러한 단점 때문이라고 추정된다.

2.3 GHC GC-Tune

GHC GC-Tune은 GHC에서 제공하는 메모리 프로파일링 도구이다. GHC는 가상머신 위에서 여러 가지의 GC를 사용하고 있으며 대표적인 GC는 세대별 GC이다[11,12]. GHC는 실행 시에 GC 옵션으로 '-A' 옵션과 '-H' 옵션을 부여할 수 있는데 '-A'는 GC에 사용되는 전체 힙 메모리 중에 어린 세대들을 위한 힙 메모리를 설정하는 옵션이며 '-H' 옵션은 전체 힙 메모리를 설정하는 옵션이다. GHC GC-Tune은 GC가 사용하는 메모리를 미세하게 조정하여 프로그램을 실행한다. 그리고 실행 시간을 분석하여 프로그램에 따라 좋은 성능을 내는 코어별 최적의 '-A', -H' 메모리 옵션을 찾아 사용자에게 알려준다.

3. 표절검사 프로그램 성능 분석

실험에 사용한 Haskell 프로그램은 표절 검사 프로그램으로서 SoVAC(software verification and analysis center)에서 사용되고 있는 DNA 기반의 검사 기법을 이용하여 작성된 프로그램이다. SoVAC에서 DNA는 프로그램 파싱 도중 발생하는 토큰을 특별한 형태로 변형한 정보로서 이를 이용하여 표절 여부를 판단한다[13,14].

실험은 Ubuntu 14.04.1 버전의 32코어(CPU Opteron 6272 2개), 96GB RAM 환경에서 진행되었다. 표절 검사 대상으로 사용한 프로그램은 2012년 부산대학교 객체지향 프로그래밍 수업에서 과제로 제출된 Java 코드 82개를 이용하였다.

먼저 첫 번째 실험으로 GC-Tune을 사용하지 않고실행한 결과를 분석한다. 표/그림 1은 GC-Tune을 사용하지 않고 코어별로 각각 10번 반복실행을 통해 평균 성능을 계산한 결과이다.

표 1 GC-Tune 사용 전 표절검사 프로그램 실행시간 및 스피드업

Table 1 Run-time speedup of Plagiarism Detection without GC-Tune

measure : sec

CORE	MUT	GC	TOTAL	SPEED UP	△ SPEED UP
1	60.61	1.22	61.84	1.00	0.00
					...
12	9.31	1.39	10.72	5.77	-0.25
13	8.48	1.76	10.36	5.97	0.20
14	9.18	1.41	10.66	5.80	-0.17
15	9.28	1.67	11.01	5.62	-0.18
16	8.15	1.42	9.71	6.37	0.75
17	8.91	1.51	10.79	5.73	-0.64
18	9.56	3.52	13.16	4.70	-1.03
19	9.18	3.98	13.30	4.65	-0.05
20	9.01	2.36	11.46	5.39	0.74
21	9.36	2.37	11.82	5.23	-0.16
22	9.35	2.76	12.20	5.07	-0.17
23	9.44	2.46	12.05	5.13	0.06
24	10.05	4.43	14.90	4.15	-0.98
25	9.60	2.25	11.94	5.18	1.03
26	10.52	3.32	14.03	4.41	-0.77
27	8.54	2.29	10.96	5.64	1.23
28	8.79	2.61	11.52	5.37	-0.27
29	8.77	2.63	11.51	5.37	0.00
30	10.31	5.41	15.92	3.88	-1.49
31	9.41	5.10	14.73	4.20	0.31
32	10.03	6.21	16.37	3.78	-0.42

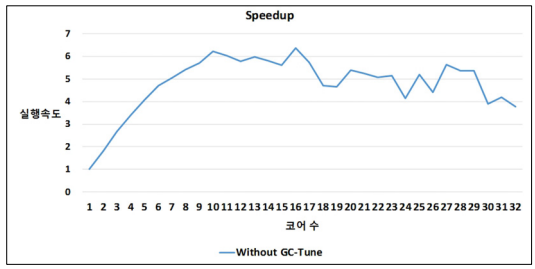

그림 1 GC-Tune 사용 전 평균 표절검사 프로그램 스피드업 그래프

Fig. 1 Average of Run-time Speedup of Plagiarism Detection without GC-Tune

본 논문의 모든 표는 좌측 열부터 코어 수, 프로그램 초기화와 GC 시간을 제외한 실행시간, GC에 소요된 시간, 프로그램 총 실행시간, 실행속도, 그리고 직전 결과 대비 속도 상승 및 감소량을 나타내었다. 실험결과는 소

수점 두 번째 자리까지 출력하였다.

표/그림 1과 같이 11개의 코어까지는 확장성이 보장된 결과를 볼 수 있지만, 그 외 개수의 코어에서는 실행 속도 그래프 상에서 변동 폭이 큰 것을 볼 수 있다. 특히 12, 15, 18, 19, 22, 24, 26, 31, 30, 32개의 코어를 사용한 부분에서는 변동 폭이 크게 요동치는 모습을 확인할 수 있었다. 그 중에서도 30, 31, 32코어에서는 다른 코어에 비해 GC에 많은 비용을 소비하는 것을 확인할 수 있었다. 다음 4 장의 실험으로는 GC-Tune을 이용하여 GC가 Haskell 병렬 프로그램에 미치는 영향을 확인하는 실험을 하고자 한다.

4. GC-Tune을 이용한 프로그램 성능 분석

실험환경은 3장과 같으며 3장에서 먼저 추려낸 10가지의 코어 환경을 바탕으로 GC-Tune을 사용하여 실험을 진행하였다. 먼저 3장에서 확인한 변동 폭이 큰 10가지의 코어 환경에 대한 최적의 메모리 옵션을 얻기 위해 GC-Tune을 사용하였다.

GC-Tune은 해당 코어 환경에서 힙 메모리를 미세하게 조정하여 실행한다. 예를 들어 19 코어 환경의 실행 과정 일부는 그림 2와 같다. 다음으로 표 2는 3장에서 추려낸 10가지 코어 환경을 대상으로 GC-Tune을 실행한 결과이다.

그림 2에서와 같이 '-A', '-H' 옵션은 병렬 프로그램의 실행시간과 GC 시간에 모두 영향을 미치는 것을 확인할 수 있다. 특히 실행시간 보다는 GC 시간에 많은 영향을 미치는 것으로 미루어 보아 병렬 Haskell 프로그램의 성능 확장성 문제의 원인은 GC임을 알 수 있다.

또한 '-A', '-H' 옵션이 일종의 상관관계를 이루는 것을 확인할 수 있는데 이는 병렬 프로그램이 다양한 알고리즘과 병렬 모델을 사용하는 만큼 본 논문과 같이

```
program +RTS -A16384 -H1048576 -RTS ./82/ +RTS -N19
<<GCs 594994, peak 218, MUT 21.921s, GC 158.524s>>
program +RTS -A16384 -H2097152 -RTS ./82/ +RTS -N19
<<GCs 581263, peak 214, MUT 21.836s, GC 150.912s>>
...
program +RTS -A2097152 -H1073741824 -RTS ./82/ +RTS -N19
<<GCs 3742, peak 6939, MUT 8.013s, GC 41.001s>>
praogram +RTS -A4194304 -H1048576 -RTS ./82/ +RTS -N19
<<GCs 2412, peak 329, MUT 7.727s, GC 41.480s>>
...
program +RTS -A134217728 -H1073741824 -RTS ./82/ +RTS -N19
<<GCs 36, peak 2623, MUT 9.354s, GC 1.483s>>
program +RTS -A268435456 -H1048576 -RTS ./82/ +RTS -N19
<<GCs 18, peak 5102, MUT 11.730s, GC 1.255s>>
```

그림 2 코어 19 환경의 GC-Tune 실행 과정
Fig. 2 GC-Tune execution process of a 19-core environment

표 2 GC-Tune 사용 후 최적 메모리 옵션
Table 2 Memory optimization options obtained using GC-Tune

measure : kbyte

CORE	-A(Young Space)	-H(Full Heap Size)
12	134,217,728	536,870,912
15	134,217,728	268,435,456
18	134,217,728	536,870,912
19	134,217,728	1,073,741,824
22	536,870,912	536,870,912
24	268,435,456	1,073,741,824
26	134,217,728	536,870,912
30	134,217,728	268,435,456
31	134,217,728	268,435,456
32	134,217,728	268,435,456

표 3 GC-Tune 사용 후 표절검사 프로그램 실행시간 및 스피드업
Table 3 Run-time Speedup of Plagiarism Detection with GC-Tune

measure : sec

CORE	MUT	GC	TOTAL	SPEED UP	△ SPEED UP
1	60.61	1.22	61.84	1.00	0.00
					...
12	8.29	1.37	9.72	6.37	0.35
13	8.48	1.76	10.36	5.97	-0.40
14	9.18	1.41	10.66	5.80	-0.17
15	8.43	1.44	9.93	6.23	0.43
16	8.15	1.42	9.71	6.37	0.14
17	8.91	1.51	10.79	5.73	-0.64
18	8.89	1.55	10.51	5.88	0.15
19	9.35	1.48	10.91	5.67	-0.21
20	9.01	2.36	11.46	5.39	-0.28
21	9.36	2.37	11.82	5.23	-0.16
22	9.39	1.52	11.00	5.62	0.39
23	9.44	2.46	12.05	5.13	-0.49
24	9.48	1.84	11.42	5.42	0.29
25	9.60	2.25	11.94	5.18	-0.24
26	8.79	1.70	10.59	5.84	0.66
27	8.54	2.29	10.96	5.64	-0.20
28	8.79	2.61	11.52	5.37	-0.27
29	8.77	2.63	11.51	5.37	0.00
30	8.42	1.50	10.04	6.16	0.79
31	8.36	1.62	10.10	6.13	-0.04
32	8.51	1.63	10.26	6.03	-0.10

특정 프로그램으로 실행한 결과로는 이 두 옵션의 정확한 상관관계를 분석하기는 어렵다. 표 3은 GC-Tune 사용 후 얻은 메모리 최적 옵션을 바탕으로 한 번 실행한 코어별 결과이다.

3장에서의 실험결과인 표 1과 비교하면 GC-Tune을 사용한 코어별 결과의 실행시간과 GC 시간이 대부분

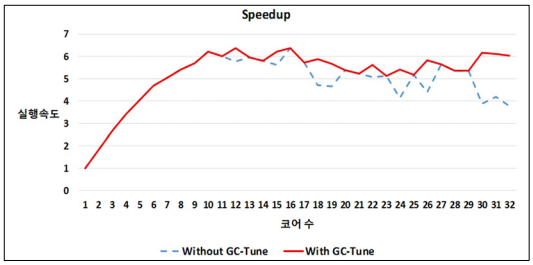

그림 3 GC-Tune 사용 전 & 사용 후 표절검사 프로그램 스피드업 그래프

Fig. 3 Speedup of Plagiarism Detection without GC-Tune & with GC-Tune

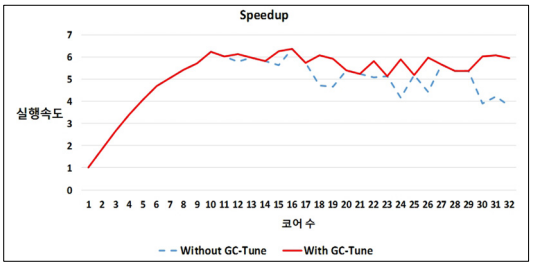

그림 4 GC-Tune 사용 전 & 사용 후 표절검사 프로그램 평균 스피드업 그래프

Fig. 4 Average of Speedup of Plagiarism Detection without GC-Tune & with GC-Tune

감소한 것을 확인할 수 있었다. 특히 표 1의 30, 31, 32 코어 실행 결과와 비교하면 GC에 소비하는 시간이 대폭 줄어든 결과를 확인할 수 있었다. 그림 3은 첫 번째 실험과 두 번째 실험의 결과를 비교한 그래프이다.

GC-Tune을 사용하지 않은 결과에 비해 전체 성능이 향상됨을 확인할 수 있었다. 이는 메모리 튜닝이 GC의 오버헤드를 줄이는 데 효과가 있다는 것을 의미한다.

표 4 GC-Tune 사용 후 평균 표절검사 프로그램 실행 시간 및 스피드업

Table 4 Average of Run-time Speedup of Plagiarism Detection with GC-Tune

measure : sec

CORE	MUT	GC	TOTAL	SPEED UP	△ SPEED UP
1	60.61	1.22	61.84	1.00	0.00
					...
12	8.67	1.38	10.11	6.12	0.10
13	8.48	1.76	10.36	5.97	-0.15
14	9.18	1.41	10.66	5.80	-0.17
15	8.33	1.49	9.90	6.25	0.45
16	8.15	1.42	9.71	6.37	0.12
17	8.91	1.51	10.79	5.73	-0.64
18	8.54	1.56	10.18	6.07	0.34
19	8.90	1.47	10.45	5.92	-0.16
20	9.01	2.36	11.46	5.39	-0.52
21	9.36	2.37	11.82	5.23	-0.16
22	8.85	1.68	10.64	5.81	0.58
23	9.44	2.46	12.05	5.13	-0.68
24	8.82	1.60	10.51	5.88	0.75
25	9.60	2.25	11.94	5.18	-0.70
26	8.63	1.62	10.37	5.96	0.78
27	8.54	2.29	10.96	5.64	-0.32
28	8.79	2.61	11.52	5.37	-0.27
29	8.77	2.63	11.51	5.37	0.00
30	8.59	1.50	10.26	6.03	0.66
31	8.34	1.57	10.18	6.07	0.04
32	8.47	1.68	10.39	5.95	-0.12

표 4는 본 장에서 GC-Tune을 이용하여 얻은 결과를 바탕으로 첫 번째 실험과 동일하게 각각 10번의 반복실행을 통해 평균 성능을 계산한 결과이고 그림 4는 GC-Tune을 사용하지 않은 첫 번째 실험 결과와 반복실행을 통해 얻은 결과를 비교하기 위한 그래프이다.

그림 4에서 GC-Tune을 사용해 10번 반복한 결과 그래프가 GC-Tune을 사용하지 않은 그래프보다 전체적인 성능 향상을 보였다. 또한 실행 속도가 감소하는 부분만을 이용하여 변동 폭의 차이를 계산해 본 결과 GC-Tune을 사용하지 않은 그래프보다 약 39% 변동 폭이 감소한 것을 확인할 수 있었다. 따라서 GC-Tune을 사용하면 성능 그래프 변동 폭이 어느 정도 감소됨을 알 수 있다.

5. 토 의

본 논문에서는 병렬 Haskell 프로그램의 확장성이 보장되지 않는 이유를 GC가 동작하는 가상머신의 영향이라고 추정하였다. 이를 위해서 병렬 Haskell 표절검사 프로그램을 대상으로 메모리 튜닝 실험을 진행하였다. 3, 4장의 실험 결과 가상머신이 병렬 Haskell 프로그램의 실행에 큰 영향을 미친다는 것을 확인할 수 있었다. 그리고 GC-Tune을 이용한 메모리 튜닝 시 병렬 Haskell 프로그램의 성능이 향상됨을 알 수 있었다.

Haskell의 대표적인 GC 기법인 세대별 GC는 기존의 다른 GC 기법과 달리 최소한의 동작으로 좋은 성능을 낼 수 있는 기법이다. 하지만 2장에서 언급했듯이 최악의 경우 GC에 소요되는 오버헤드는 늘어난다. 이는 성능을 중요시하는 병렬 프로그램에 치명적인 단점이 된다.

이 논문에서의 실험 결과로 미루어 볼 때 GC-Tune을 이용해도 일정 코어 이상의(4절 실험의 경우 18 코어) 성능 향상은 확인할 수 없었다. 그 이유는 프로그램의 특성 때문인 것으로 생각되는데, 프로그램에 따라 다양한 병렬 모델과 알고리즘을 사용하고 있으므로 가능

한 최대 병렬성은 프로그램에 따라 다른 것으로 생각된다. 더욱 정확한 검증을 위해서는 다양한 병렬 프로그램을 대상으로 메모리 튜닝 실험이 필요하다고 생각된다.

궁극적으로 병렬 Haskell 프로그램의 확장성을 극대화하기 위해서는 GC가 병렬 프로그램에 적합하게 수정되어야 한다. GC를 구현하려면 해당 언어의 실행시간 메모리 객체 구조를 파악해야 하므로 이는 단순한 작업이 아니다. 하지만 이는 매니코어 환경에 적합하게 동작하는 병렬 프로그램을 위해 꼭 필요한 과제라고 생각된다.

6. 결 론

이 논문에서는 GC 조정을 통하여 병렬 프로그램의 확장성을 높일 수 있음을 실험을 통해 확인하였다. Haskell로 작성된 병렬 프로그램은 사용하는 코어에 따른 성능 확장성이 보장되지 않는 단점이 있다. 이는 Haskell 병렬 프로그램이 GC를 사용하는 가상머신 위에서 동작하기 때문이라고 생각된다.

먼저 GC-Tune을 사용하지 않고 실험하였다. 그 결과 불규칙적인 변동 폭을 확인할 수 있었다. 다음으로 첫 번째 실험의 결과를 바탕으로 변동 폭이 경우에 대해 GC-Tune을 사용하여 메모리 튜닝을 진행하였다. GC-Tune을 이용하여 얻은 최적의 메모리 옵션을 사용한 실험 결과, GC-Tune을 사용하지 않은 결과에 비해 높아진 확장성과 비교적 안정된 변동 폭을 확인할 수 있었다. 그리고 10번을 반복 실행한 결과, 어느 정도 변동 폭이 줄어든 결과를 확인할 수 있었다. 그리고 실행 속도가 감소하는 부분만을 이용하여 변동 폭의 차이를 구한 결과 GC-Tune을 사용한 결과가 GC-Tune을 사용하지 않은 결과보다 약 39% 개선된 것으로 나타났다.

본 논문은 병렬 프로그램의 성능 확장성 문제의 원인을 확인하기 위해 표절검사라는 특정 프로그램을 사용했다. 따라서 본 연구의 결과가 실질적인 병렬 프로그램 성능 개선에 활용되려면 다양한 실험 결과와 분석이 필요하다. 그에 따른 향후 연구는 다음과 같다.

먼저 다양한 병렬 프로그램을 대상으로 실험을 진행하고자 한다. 병렬 프로그램은 다양한 병렬 모델과 알고리즘을 사용하고 있으므로 프로그램의 성격에 따라 본 연구의 결과가 다양하게 나타날 수 있다. 따라서 다양한 실험 결과는 코어별 프로그램 응답시간을 예측할 수 있는 좋은 자료가 될 것으로 예상한다. 다음으로 GC 옵션('-A', '-H') 사이의 관계를 분석하고자 한다. 4장에서의 GC-Tune을 사용한 결과 두 옵션이 일종의 상관관계를 이루고 있는 것을 확인할 수 있었다. GC-Tune은 미세하게 메모리를 조정하여 프로그램을 실행하는 만큼 많은 시간이 소요된다. 따라서 코어별 GC 옵션을 예측할 수 있다면 많은 시간이 절약될 것으로 예상된다.

References

[1] Y. Kim and S. Kim, "Technology and Trends of High Performance Processors," *Electronics and Telecommunications Trends*, No. 6, pp. 123-136, 2014.

[2] J. Kim, S. Byun, K. Kim, J. Jung, K. Koh, S. Cha, and S. Jung, "Technology Trends of Haskell Parallel Programming in the Manycore Era," *Electronics and Telecommunications Trends*, No. 29, pp. 167-175, 2014.

[3] N. Shavit, and D. Touitou, "Software transactional memory," *Distributed Computing*, pp. 99-116, 1997.

[4] J. Epstein, A. P. Black, and S. P. Jones, "Towards Haskell in the cloud," *ACM SIGPLAN Notices, ACM*, pp. 118-129, 2011.

[5] D. Stewart, GHC GC-Tunes, [Online]. Available: http//hackage.haskell.org/package/GHC GC-Tune. (downloaded 2017. Feb. 13)

[6] S. Marlow, S. P. Jones, and S. Singh, "Runtime support for multicore Haskell," *ACM SIGPLAN Notices, ACM*, 2009.

[7] G. Hutton, Programming in Haskell, Cambridge University Press, 2007.

[8] J. M. Stichnoth, G. Y. Lueh, and M. Cierniak, "Support for garbage collection at every instruction in a Java compiler," *ACM SIGPLAN Notices*, 1999.

[9] P. M. Sansom and S. L. Peyton Jones, "Generational garbage collection for Haskell," *Proc. of the conference on Functional programming languages and computer architecture*, pp. 106-116, 1993.

[10] D. M. Ungar and M. I. Wolczko, "Method and apparatus for generational garbage collection of a heap memory shared by multiple processors," U.S. Patent No. 6, 2001.

[11] S. Marlow, et al., "Parallel generational-copying garbage collection with a block-structured heap," *Proc. of the 7th international symposium on Memory management*, ACM, 2008.

[12] S. Marlow and S. P. Jones, "Multicore garbage collection with local heaps," *ACM SIGPLAN Notices*, 2011.

[13] Y. Kim, J. Cheon, S. Byun and G. Woo, "A Parallel Performance Comparison of Haskell Using a Plagiarism Detection Method," *KCC*, pp. 1724-1726, 2016.

[14] J. Jeong, W. Gyun and H. Cho, "A source code linearization technique for detecting plagiarized programs," *ACM SIGCSE Bulletin*, pp. 73-77, 2007.

김 화 목

2015년 조선대학교 제어계측로봇공학과(학사). 2015년~현재 부산대학교 전기전자컴퓨터공학과 석사과정. 관심분야는 함수형 프로그래밍, 병렬 프로그래밍, 임베디드 시스템, 소프트웨어 교육

안 형 준

2015년 연세대학교 수학과(학사). 2015년~현재 부산대학교 전기전자컴퓨터 공학과 석사과정. 관심분야는 함수형 프로그래밍, 역공학, 병렬 프로그래밍, 사용자 인터페이스

변 석 우

1980년 숭실대학교 전자계산학과(공학사) 1982년 숭실대학교 전자계산학과(공학석사). 1982년~1999년 ETRI(책임연구원) 1995년 Univ. of East Anglia (영국), Computer Science (Ph.D.). 1999년~현재 경성대학교 컴퓨터공학부 교수. 관심분야는 함수형 프로그래밍, 정형 증명, 프로그래밍 언어

우 균

1991년 한국과학기술원 전산학(학사). 1993년 한국과학기술원 전산학(석사). 2000년 한국과학기술원 전산학(박사). 2000년~2004년 동아대학교 컴퓨터공학과 조교수 2004년~현재 부산대학교 전기전자컴퓨터공학과 교수. 관심분야는 프로그래밍언어 및 컴파일러, 함수형 언어, 프로그램 분석, 프로그램 시각화, 프로그래밍 교육, 한글 프로그래밍 언어