분할 정복법을 이용한 Haskell GC 조정 시간 개선

안형준[1], 변석우[2], 우균[3]
[1,3]부산대학교 전기전자컴퓨터공학과
[2]경성대학교 컴퓨터공학과
[3]LG전자 스마트 제어 센터
e-mail:hyungjun@pusan.ac.kr, swbyun@ks.ac.kr, woogyun@pusan.ac.kr

Improving Haskell GC-Tuning Time Using Divide and Conquer

Hyungjun An[1], Sugwoo Byun[2], Gyun Woo[3]
[1,3]Dept. of Electrical and Computer Engineering, Pusan National University
[2]School of Computer Science & Engineering, Kyungsung University
[3]Smart Control Center of LG Electronics

요 약

단일 코어 프로세스의 성능 향상은 전력 소모, 발열 등의 이유로 한계에 달했다. 이에 대한 대안으로 멀티 코어가 등장했으며 매니 코어 기술에 대한 연구가 활발히 진행 중에 있다. 이렇듯 멀티 코어 환경이 보편화됨에 따라 병렬 프로그래밍의 중요성이 더욱 커졌다. 한편, 순수 함수형 언어 Haskell은 부수효과가 없고 다양한 병렬화 도구를 지원함으로써 다가오는 병렬 프로그래밍 시대에 적합한 언어라 할 수 있다. 이때 Haskell 병렬 프로그램의 성능은 메모리 재사용(Garbage Collection) 시간에 큰 영향을 받는다. 그래서 Haskell 병렬 프로그램의 성능 향상, 분석을 위한 메모리 프로파일링 도구가 필요하다. 이미 Haskell이 제공하는 메모리 프로파일링 도구로 ghc-gc-tune이 있지만 실행 속도 측면에서 개선이 필요하다. 본 연구에서는 분할 정복법을 이용해서 매 단계마다 탐색 영역을 4분의 1로 줄이도록 ghc-gc-tune을 개선했다. 개선된 ghc-gc-tune을 극대 독립 집합 프로그램과 K-means 프로그램에 적용한 결과, 평균 98%의 정확도로 실행 시간을 평균 7.78배 단축했다.

1. 서 론

단일 코어 프로세서의 성능 향상은 한계에 달했다. 전력 소모가 심해지면서 발열 억제 등 시스템 운영비용이 증가하기 때문이다. 이에 대한 대안으로 등장한 것이 멀티 코어인데, 멀티 코어는 하나의 집적 회로에 여러 개의 CPU가 집적되어있는 것을 말한다. 그러나 멀티 코어 역시 공유 메모리 관리 등의 문제로 현재 성능 향상이 한계에 달했다. 그래서 오늘날에는 수백 내지는 수천 개의 코어를 하나의 CPU에 집적하는 매니 코어 기술이 각광받고 있다[1].

여러 개의 코어 또는 CPU 사용이 일반화됨에 따라 이를 효과적으로 활용하기 위한 병렬 프로그래밍 연구도 활발히 진행되고 있다. 한편, 순수 함수형 언어인 Haskell은 앞으로 도래할 병렬 프로그래밍 시대에 적합한 언어라고 할 수 있다[2]. 부수 효과가 없기 때문에 의존성을 고려할 필요가 없고 다양한 병렬화 도구를 지원하기 때문이다.

그러나 성능 면에서는 단점도 존재하는데, 사용 코어 수를 늘려가면서 Haskell 병렬 프로그램의 성능을 측정해 보면 속도 향상이 고르지 못하다. 이는 코어가 늘어날수록 Haskell의 메모리 재사용(GC: Garbage Collection) 오버헤드가 늘어나기 때문이다[3]. 한편, Haskell은 GC 환경을 사용자가 실행 옵션으로 지정할 수 있는데 GC환경의 예로는 GC에 사용되는 힙 메모리 크기 등이다. GC 환경을 조절하면 고르지 못한 Haskell 병렬 프로그램의 속도 향상을 개선할 수 있다. 그래서 Haskell 병렬 프로그램의 성능 향상 또는 분석을 위해서는 프로그램의 최적 GC 환경을 알아내는 것이 중요하다.

Haskell 프로그램의 최적 GC 환경을 알려주는 메모리 프로파일링 도구로 ghc-gc-tune이 있다. 그러나 기존 ghc-gc-tune은 가능한 모든 GC 옵션에 대해 프로그램 실행 시간을 측정한 뒤 결과를 알려주기 때문에 실행시간이 느리다.

이러한 기존 ghc-gc-tune의 단점을 개선하기 위해 분할 정복법을 적용해보려 한다. 이때 분할 정복법의 여러 형태 중에서도 이진 검색 방법을 응용하여 매 단계마다 탐색 영역을 4분의 1로 줄이고자 한다.

본 논문의 구조는 다음과 같다. 2장에서는 관련 연구로 Haskell과 세대별 GC 기법 그리고 Haskell 메모리 프로파일링 도구인 ghc-gc-tune을 소개한다. 3장에서는 기존 ghc-gc-tune 문제점을 지적하고 이를 개선하기 위한 방법을 제시한다. 4장에서는 개선된 ghc-gc-tune의 성능을 측정하고 5장에서 토의를 한 뒤 끝으로 6장에서 결론을 맺는다.

2. 관련 연구

2.1. Haskell

Haskell은 1990년에 만들어진 순수 함수형 언어로 당시 산재해있던 함수형 언어들을 통합하고자 했던 학자들이 위원회를 조직하여 만든 언어다. Haskell의 장점으로는 쉬운 병렬 프로그래밍을 들 수 있다. 실제로 Haskell은 부수 효과가 없어서 병렬 프로그래밍 시 흔히 발생하는 의존성 문제가 적다. 그러나 Haskell 병렬 프로그래밍도 한 가지 문제가 있다. 사용 코어 수를 증가시켜가며 Haskell 병렬 프로그램을 실행해보면 프로그램의 속도 향상이 일정하지 않기 때문이다. 예시로 K-means 알고리즘을 구현한 Haskell 병렬 프로그램의 속도 향상 그래프는 그림 1과 같다. 그림 1을 보면 사용 코어가 1에서 6까지 늘어남에 따라 속도가 향상되지만, 그 이후부터는 출렁이는 것을 알 수 있다.

(그림 1) K-means 알고리즘을 구현한 Haskell 병렬 프로그램의 속도 향상

2.2. 세대별 GC

Haskell은 기본적으로 세대별 GC 기법을 이용해서 메모리를 관리한다[4]. 세대별 GC 기법은 '최근에 만들어진 객체일수록 일찍 없어진다.'라는 가설을 기반으로 한다. 세대별 GC의 기본적인 동작은 다음과 같다. 먼저, 생성되는 객체를 그 시기에 따라 다른 힙 메모리 공간에 위치시킨다. 이때 각 영역은 '세대'라는 표현으로 구분되며 생성 시기가 짧은 세대를 그렇지 않은 세대보다 어린 세대라고 표현한다. 그 다음, 시간이 지남에 따라 어린 세대를 대상으로 세대 이동 또는 객체 소멸 등의 작업을 진행한다.

세대별 GC 기법은 메모리 공간을 나누는 개수 또는 각 메모리 공간의 크기를 조절하는 등 다양한 변화를 줄 수 있다. Haskell에서는 사용자가 직접 GC 옵션을 지정하는 형태로 이러한 변화를 줄 수 있는데, 예를 들어 'prog'라는 프로그램이 있다고 하면, '`./prog +RTS -A16384 -H1048576 -RTS`'와 같은 형태로 GC 환경을 지정할 수 있다. 이때, `+RTS`, `-RTS`는 그 사이의 옵션이 실행 환경을 지정하는 옵션임을 명시하는 옵션이다. 그리고 '`-A`'는 어린 세대가 사용할 힙 메모리 용량, '`-H`'는 전체 세대가 사용할 힙 메모리 용량을 지정하는 옵션이다.

한편, 김화목의 연구[3]에 따르면 Haskell 병렬 프로그램의 저조한 속도 증가 원인은 GC 실행 환경 때문이라고 추정되었다. 추정 근거는 Haskell 병렬 프로그램을 실행할 때 GC 옵션에 따라 바뀌는 실행 시간이었다. 이처럼 GC 옵션이 병렬 프로그램의 성능에 영향을 주기 때문에 상황별로 최적의 GC 옵션을 알아내는 것은 중요하다. 그리고 이 과정에서 Haskell 프로그램의 최적 GC 옵션을 찾아주는 도구로 ghc-gc-tune이 있다.

2.3. ghc-gc-tune

ghc-gc-tune은 GHC에서 제공하는 메모리 프로파일링 도구이다[5]. ghc-gc-tune은 가능한 모든 경우의 '`-A`'옵션과 '`-H`'옵션에 대한 프로그램 실행 시간을 측정해서 그 중 가장 최적이라 판단되는 5개의 GC 실행 환경을 추천해준다. 이때 '`-A`'옵션과 '`-H`'옵션의 메모리 지정은 시중에 판매되는 메모리 용량이 그러하듯이 2의 거듭제곱 표현이 가능한 수로 한정한다. 예를 들어 '`-A`'옵션이 1MB부터 8MB까지 가능하고 '`-H`'옵션이 8MB에서 32MB까지 가능하다면 '`-A`'옵션의 경우의 수는 1MB, 2MB, 4MB, 8MB로 4개, '`-H`'옵션의 경우의 수는 8MB, 16MB, 32MB로 3개가 되어 '`-A`', '`-H`'옵션의 조합은 총 3×4인 12가지가 된다.

3. GC-Tune의 개선

3.1. 기존 GC-Tune

GC-Tune은 가능한 모든 경우의 '`-A`', '`-H`'옵션을 조합해서 실행시간을 측정하기 때문에 항상 확실한 최적 GC 환경을 추천해준다는 장점이 있다. 그러나 '`-A`', '`-H`'옵션 조합의 수가 늘어나면 그에 따라 실행시간 측정 횟수 역시 증가한다는 단점도 있다.

이후 실험에 사용할 극대 독립 집합을 구하는 Haskell 프로그램을 예로 들도록 하겠다. 기존 GC-Tune이 이 프로그램의 최적 GC 환경을 구하기 위해 측정한 실행시간은 표 1과 같다. 표 1을 보면 알 수 있듯이 기존 GC-Tune은 최적 GC 환경을 구하기 위해 총 16×11=176번의 실행 시간 측정이 필요했다. 문제는 176번 측정한 실행 시간을 더해보면 약 54초가 나오는데 실행 시간이 0.5초도 되지 않는 프로그램의 최적 환경을 구하기 위해 그것의 100배 이상에 달하는 시간이 필요하다는 것이다.

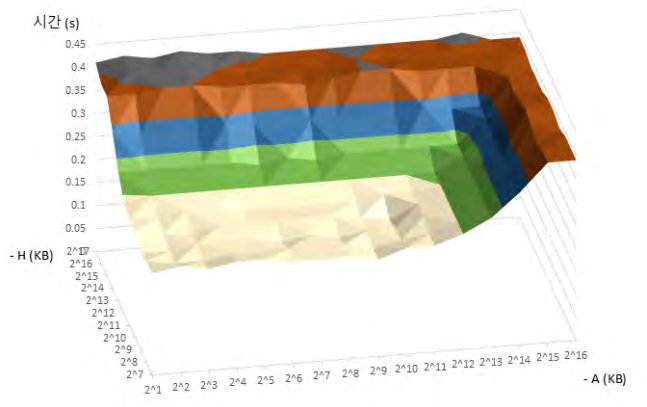
<표 1> 극대 독립 집합을 구하는 Haskell 프로그램의 최적 GC 환경을 구하기 위해 GC-Tune이 측정한 실행 시간

(옵션 단위: KB, 시간 단위: 초)

-A \ -H	2^7	2^8	2^9	2^{10}	2^{11}	2^{12}	2^{13}	2^{14}	2^{15}	2^{16}	2^{17}
2^1	0.22	0.22	0.21	0.23	0.23	0.24	0.28	0.34	0.39	0.39	0.41
2^2	0.23	0.21	0.21	0.22	0.23	0.25	0.28	0.33	0.39	0.41	0.42
2^3	0.22	0.21	0.22	0.22	0.24	0.25	0.28	0.33	0.40	0.41	0.41
2^4	0.23	0.22	0.22	0.23	0.24	0.25	0.28	0.33	0.40	0.41	0.42
2^5	0.23	0.22	0.22	0.23	0.23	0.25	0.28	0.34	0.39	0.40	0.41
2^6	0.22	0.22	0.21	0.23	0.23	0.25	0.27	0.33	0.39	0.39	0.41
2^7	0.22	0.22	0.22	0.23	0.23	0.25	0.28	0.33	0.39	0.39	0.40
2^8	0.22	0.22	0.21	0.23	0.23	0.25	0.27	0.33	0.38	0.39	0.40
2^9	0.21	0.21	0.22	0.22	0.23	0.25	0.28	0.33	0.39	0.39	0.40
2^{10}	0.23	0.22	0.22	0.21	0.24	0.25	0.28	0.33	0.40	0.40	0.40
2^{11}	0.23	0.23	0.24	0.24	0.24	0.25	0.28	0.33	0.40	0.39	0.40
2^{12}	0.25	0.25	0.25	0.25	0.25	0.25	0.28	0.34	0.39	0.39	0.40
2^{13}	0.28	0.28	0.28	0.28	0.28	0.28	0.28	0.33	0.39	0.39	0.40
2^{14}	0.33	0.33	0.33	0.33	0.32	0.32	0.33	0.33	0.39	0.39	0.41
2^{15}	0.39	0.39	0.38	0.39	0.40	0.39	0.38	0.39	0.39	0.39	0.39
2^{16}	0.39	0.39	0.39	0.38	0.39	0.39	0.38	0.38	0.38	0.39	0.39

3.2. 분할 정복법 적용

이 절에서는 분할 정복법을 응용해서 지나치게 많은 실행 시간 측정을 필요로 하는 기존 GC-Tune의 단점을 개선하려 한다. GC-Tune 개선을 위한 방법으로 분할 정복법을 택한 이유는 표 1을 표면형 그래프로 바꾼 그림 2를 통해 알 수 있다. 그림 2를 보면 GC 옵션과 실행 시간의 관계가 오목함수와 유사한 형태를 띠고 있다. 이는 서로 다른 두 GC 환경 g1, g2에 대해 g1에서의 프로그램 실행 시간이 g2보다 빠르다면 최적 GC 환경은 g2보다는 g1과 유사할 확률이 높다는 것을 의미한다.

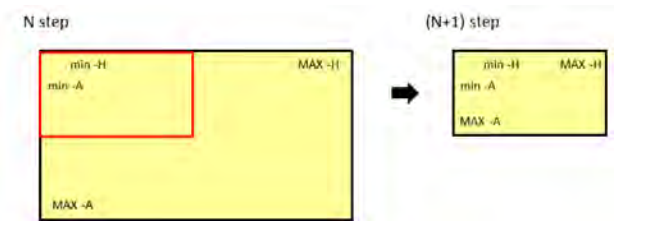


(그림 2) 표 1의 표면형 그래프

이러한 사실에 기초해서 GC-Tune을 개선하는 분할정복 알고리즘을 설계하였다. 설계된 알고리즘은 이진 검색과 유사하며 구체적인 과정은 다음과 같다.

먼저 가능한 '-A', '-H' 옵션의 최소, 최댓값을 구한다. 그러면 각 옵션마다 최소, 최대의 두 가지 경우가 존재하므로 가능한 조합은 총 4개이다. 표 1의 네 귀퉁이가 이에 해당한다. 다음으로 총 4개의 조합에 대해 그중 가장 실행 시간이 빠른 옵션 조합을 구한 뒤 이 조합을 기준으로 탐색 영역을 4분의 1로 줄인다. 이 과정을 탐색영역을 더 이상 줄일 수 없을 때까지 반복한다.

그림 3은 본 논문에서 제시하는 알고리즘의 동작을 도식화한 것이다. 이때 N번째 단계에서 가장 실행 속도가 빠른 경우는 '-A', '-H' 옵션이 모두 최소일 때로 가정했다.



(그림 3) 본 논문이 제시하는 알고리즘 동작의 도식화

앞서 제시한 알고리즘을 이용해서 개선한 GC-Tune으로 표 1에 사용된 극대 독립 집합을 구하는 Haskell 프로그램의 최적 GC 환경을 다시 구해보았다. 이 과정에서 단계별로 측정된 실행 시간은 표 2와 같다. 각 단계별로 4번의 실행 시간 측정이 필요하고 6단계를 반복했으므로 총 24번의 실행 시간 측정이 진행되었고 약 6초가 걸렸다. 이는 176번의 실행이 필요하고 총 54초가 걸린 개선 전에 비해 실행 시간 면에서는 9배, 측정 횟수 면에서는 7배가 개선됐음을 뜻한다.

<표 2> 개선된 GC-Tune이 극대 독립 집합을 구하는 Haskell 프로그램의 최적 GC 환경을 구하기 위해 측정한 실행 시간

Step	Time(s)			
1	0.23	0.41	0.40	0.39
2	0.23	0.25	0.25	0.22
3	0.23	0.22	0.22	0.22
4	0.22	0.21	0.22	0.21
5	0.22	0.22	0.22	0.22
6	0.22	0.21	0.22	0.22

4. 실 험

본 장에서는 3장을 통해 개선한 GC-Tune과 개선 전의 GC-Tune을 정확도와 실행 시간의 관점에서 비교 실험한다. 이때 정확도를 비교의 척도로 사용한 이유는 개선된 GC-Tune이 탐색 영역을 좁히는 과정에서 최적 GC 옵션을 놓칠 수 있기 때문이다. 개선된 GC-Tune이 최적 GC 옵션을 놓치게 되는 경우는 GC 옵션에 따른 실행시간 그래프가 완벽한 오목 함수가 아니기 때문에 발생한다.

4.1. 실험 환경 및 방법

실험은 Ubuntu 14.04.1 버전의 32코어(CPU Opteron 6272 2개), 96GB RAM 환경에서 진행되었다. 개선 전후 비교를 위해 GC-Tune의 입력으로 사용될 Haskell 병렬 프로그램은 크게 두 가지로 극대 독립 집합 알고리즘과 K-means 알고리즘이다. 이때 두 프로그램 모두 병렬 프로그램이기 때문에 각각 실험환경이 허락하는 1에서 32코어까지의 경우 모두를 GC-Tune의 입력으로 사용한다.

개선된 GC-Tune의 정확도는 식 1과 같이 계산한다. 이때 G는 개선 전 GC-Tune이 측정한 실행 시간의 집합이고 G'은 개선 후 GC-Tune이 측정한 실행 시간의 집합이다.

$$\frac{\mathrm{Max}(G)-\min(G')}{\mathrm{Max}(G)-\min(G)} \qquad (1)$$

4.2. 실험 결과

먼저 극대 독립 집합 프로그램을 이용해서 GC-Tune의 개선 전후 성능을 측정한 결과는 표 3과 같다. 표 3을 보면 개선 전의 GC-Tune보다 개선 후의 GC-Tune이 월등히 빠르면서도 높은 정확도를 유지함을 알 수 있다. 수

치적으로는 평균 97%의 정확도로 실행 시간을 7.53배 단축한 것이다. K-means 프로그램 역시 극대 독립 집합 프로그램과 유사하게 평균 99%의 정확도로 실행 시간을 8.04배 단축했다. 두 프로그램에 대한 결과를 종합하면 GC-Tune을 개선함으로써 평균 98% 정확도로 7.78배의 실행 시간을 단축시켰음을 알 수 있다.

<표 3> 극대 독립 집합 프로그램을 이용한 GC-Tune의 개선 전후 성능 비교

Core	총 실행 시간(초)		개선 후 정확도(%)	실행 시간 개선도(배)
	개선 전	개선 후		
1	54.05	5.88	100	9.19
2	32.97	3.62	100	9.11
3	26.11	2.92	92	8.94
4	19.93	2.55	90	7.82
5	19.21	2.34	82	8.21
6	18.57	2.35	91	7.90
7	13.48	1.85	100	7.29
8	13.70	1.92	93	7.14
9	14.18	1.80	100	7.88
10	14.47	2.13	100	6.79
11	15.21	1.63	100	9.33
12	15.32	1.96	100	7.82
13	15.38	1.90	93	8.09
14	16.06	2.35	100	6.83
15	16.25	2.29	100	7.10
16	16.86	2.47	98	6.83
17	17.41	2.24	100	7.77
18	17.91	2.64	96	6.78
19	19.09	2.80	97	6.82
20	19.62	2.82	97	6.96
21	19.74	3.54	99	5.58
22	20.90	2.64	100	7.92
23	22.14	3.06	98	7.24
24	22.47	3.22	98	6.98
25	23.44	3.29	99	7.12
26	24.27	3.38	99	7.18
27	25.66	3.42	100	7.50
28	26.40	4.21	98	6.27
29	27.22	3.48	100	7.82
30	28.05	4.32	99	6.49
31	30.05	3.83	99	7.85
32	30.29	3.66	99	8.28

5. 토 의

4장의 실험을 통해 높은 정확도를 유지하면서 GC-Tune의 실행 시간을 단축했음을 보였다. 그러나 실행 시간을 수 배 이상 단축했지만 그렇다고 단축된 실행 시간이 절대적으로 짧은 것은 아니다. 예를 들어 표 3의 core 1 경우를 보면 개선된 GC-Tune의 총 실행 시간이 5.88인데 core 1일 때 극대 독립 집합 프로그램의 최적 시간은 0.21이다. 최적 GC 환경을 알아내기 위해 최적 시간의 28배에 달하는 시간을 사용한 셈이다. 극단적으로는 최적 시간이 1시간인 프로그램이었다면 최적 GC 환경을 알아내기 위해 하루가 넘게 걸릴 수도 있다는 것이다. 이는 GC-Tune이 현업 수준의 프로그래밍에 사용되려면 실행 시간 측정횟수를 더욱 줄여야함을 뜻한다.

6. 결 론

본 연구에서는 분할 정복법을 응용해서 Haskell 메모리 프로파일링 도구인 GC-Tune의 실행 시간을 개선했다. GC-Tune 개선에 사용된 분할 정복법은 이분 검색법의 변형으로 탐색 영역을 단계마다 4분의 1로 줄여나가는 방법을 사용했다. 개선된 GC-Tune의 성능을 Haskell로 만들어진 극대 독립 집합 프로그램과 K-means 프로그램을 이용해서 측정한 결과 개선 전보다 평균 98%의 정확도를 보였고 실행 시간은 평균 7.78배 단축되었다.

그러나 실행 시간을 비약적으로 단축시켰지만 사용할 수 있는 '-A', '-H' 옵션 범위가 커지면 GC-Tune의 측정 횟수가 늘어나는 것은 변함이 없다. 그래서 향후 연구로 사용할 수 있는 '-A', '-H' 옵션 범위가 커지더라도 최적 GC 환경을 구하기 위한 GC-Tune의 측정 횟수를 일정 상수 보다 작게 만들 수 있는 방법을 모색하고자 한다.

ACKNOWLEDGMENT

본 연구는 미래창조과학부의 SW컴퓨팅산업원천기술개발사업의 일환으로 수행하였음(B0101-17-0644, 매니코어 기반 초고성능 스케일러블 OS 기초연구).

*교신 저자 : 우균(부산대학교, woogyun@pusan.ac.kr)

참고문헌

[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] H. Kim, H. An, S. Byun and G. Woo, "An Approach to Improve the Scalability of Parallel Haskell Programs," ICCCA 2016, pp.175-178, 2016.

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

[5] The ghc-gc-tune package, Graph performance of Haskell programs with different GC flags [Internet], https://hackage.haskell.org/package/ghc-gc-tune, 2017.