KIPS Tr. Comp. and Comm. Sys.
Vol.6, No.9 pp.377~384 pISSN: 2287-5891

https://doi.org/10.3745/KTCCS.2017.6.9.377

Improving Haskell GC-Tuning Time Using Divide-and-Conquer

Hyungjun An[†] · Hwamok Kim[†] · Xiao Liu[††] · Yeoneo Kim[†††] · Sugwoo Byun[††††] · Gyun Woo[†††††]

ABSTRACT

The performance improvement of a single core processor has reached its limit since the circuit density cannot be increased any longer due to overheating. Therefore, the multicore and manycore architectures have emerged as viable approaches and parallel programming becomes more important. Haskell, a purely functional language, is getting popular in this situation since it naturally supports parallel programming owing to its beneficial features including the implicit parallelism in evaluating expressions and the monadic tools supporting parallel constructs. However, the performance of Haskell parallel programs is strongly influenced by the performance of the run-time system including the garbage collector. Though a memory profiling tool namely GC-tune has been suggested, we need a more systematic way to use this tool. Since GC-tune finds the optimal memory size by executing the target program with all the different possible GC options, the GC-tuning time takes too long. This paper suggests a basic divide-and-conquer method to reduce the number of GC-tune executions by reducing the search area by one-quarter for every searching step. Applying this method to two parallel programs, a maximally independent set and a K-means programs, the memory tuning time is reduced by 7.78 times with accuracy 98% on average.

Keywords : Haskell, Parallel Programming, Generational GC, Divide-and-Conquer, GC-Tune

분할 정복법을 이용한 Haskell GC 조정 시간 개선

안 형 준[†] · 김 화 목[†] · 류 샤 오[††] · 김 연 어[†††] · 변 석 우[††††] · 우 균[†††††]

요 약

발열 때문에 더이상 회로 집적도를 높일 수 없기 때문에 단일 코어 프로세서의 성능 향상은 한계에 달했다. 그래서 코어를 여러 개 사용하는 멀티 코어, 매니 코어 형태의 프로세서가 등장했으며 병렬 프로그래밍이 중요해졌다. 이러한 상황에서 병렬 프로그래밍에 여러 장점이 있는 순수 함수형 언어 Haskell이 주목받고 있다. Haskell은 식 계산 방식에서 이미 병렬성이 내재되어 있으며 병렬 구조를 지원하는 모나드 도구를 제공한다. 그런데 Haskell 병렬 프로그램의 성능은 메모리 재사용 시스템을 포함한 실행시간 시스템에 큰 영향을 받는다. 이미 Haskell이 제공하는 메모리 프로파일링 도구로 GC-tune이 있지만, GC-tune은 가능한 모든 GC 옵션에 대해 프로그램 실행 시간을 반복 측정하기 때문에 GC 조정 시간이 너무 오래 걸린다. 그래서 본 연구에서는 기본적인 분할 정복법을 이용해서 GC-tune의 탐색 영역을 매 단계마다 4분의 1로 줄이는 방법을 제안한다. 제안하는 방법을 두 가지 병렬 프로그램(극대 독립 집합 프로그램과 K-평균 프로그램)에 적용한 결과, 평균 98%의 정확도로 실행 시간을 평균 7.78배 단축시켰다.

키워드 : Haskell, 병렬 프로그래밍, 세대별 GC, 분할 정복법, GC-Tune

1. 서 론

단일 코어 프로세서의 성능 향상은 한계에 다다른 것으로 생각된다. 회로 집적도를 더욱 높였다가는 코어가 발열을 견디지 못하고 녹아버리기 때문이다[1]. 집적도 향상에 대한 대안으로 등장한 것이 코어를 여러 개 사용하는 멀티 코어[2], 매니 코어[3] 프로세서다. 오늘날에는 개인용 컴퓨터는 물론이고 휴대전화에도 멀티코어 프로세서가 사용될 정도로 코어를 여러 개 사용하는 기술이 보편화되었다.

이렇듯 여러 개의 코어 또는 CPU를 사용하는 환경이 일반화됨에 따라 이를 효과적으로 활용하기 위한 병렬 프로그래밍 연구도 활발히 진행되고 있다. 한편, 순수 함수형 언어인 Haskell은 병렬 프로그래밍에 매우 적합한 언어라고 할 수 있다[4]. 부수 효과가 없기 때문에 의존성을 고려할 필요

※ 이 논문은 2017년도 정부(과학기술정보통신부)의 재원으로 정보통신기술진흥센터의 지원을 받아 수행된 연구임(No.B0101-17-0644, 매니코어 기반 초고성능 스케일러블 OS 기초연구).
† 준 회 원 : 부산대학교 전기전자컴퓨터공학과 석사과정
†† 준 회 원 : 부산대학교 전기전자컴퓨터공학과 석·박사통합과정
††† 준 회 원 : 부산대학교 전기전자컴퓨터공학과 박사과정
†††† 비 회 원 : 경성대학교 컴퓨터공학부 교수
††††† 종신회원 : 부산대학교 전기전자컴퓨터공학과 교수
Manuscript Received : July 21, 2017
Accepted : August 2, 2017
* Corresponding Author : Gyun Woo(woogyun@pusan.ac.kr)

가 없고 코딩 편의성을 위한 다양한 병렬화 도구를 지원하기 때문이다.

물론 Haskell 병렬화 도구 말고도 쉬운 병렬 프로그래밍을 제공하는 다른 도구들도 있다. 대표적인 예가 OpenMP이다. 그러나 OpenMP는 컴파일러 지시문에 의존하기 때문에 내부의존성이 없는 반복문에서만 사용가능하다는 단점이 있다. 반면에 Haskell의 병렬화 도구는 높은 수준의 병렬화를 적은 코딩양으로 해낼 수 있다는 장점이 있다.

그러나 성능 면에서는 단점도 존재하는데, 사용 코어 수를 늘려가면서 Haskell 병렬 프로그램의 성능을 측정해보면 속도 향상이 고르지 못하다는 것을 확인할 수 있다. 이는 사용 코어 수가 늘어날수록 Haskell의 메모리 재사용(GC: Garbage Collection) 오버헤드가 늘어나기 때문이다[5]. 한편, Haskell은 GC 환경을 사용자가 실행 옵션으로 조정할 수 있는데 구체적으로 GC에 사용되는 힙 메모리 크기 등을 지정할 수 있다. 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 GC-tune[1]이 있다. 그런데 기존 GC-tune은 가능한 모든 GC 옵션에 대해 프로그램 실행 시간을 측정한 뒤 결과를 알려주기 때문에 실행시간이 느리다. 게다가 사용 가능한 GC 옵션이 늘어나면 이에 비례해서 프로그램 실행 시간 측정횟수가 늘어난다. 예를 들어 프로그램 실행 환경의 변화로 사용 가능한 GC 옵션 조합이 10개에서 100개로 10배 늘어났다면 실행 시간 측정횟수도 10회에서 100회로 10배 늘어난다.

이 논문에서는 이러한 기존 GC-tune의 단점을 개선하기 위해 분할 정복법을 적용하고자 한다[6, 7]. 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 GC-tune을 소개한다. 3장에서는 기존 GC-tune의 문제점을 지적하고 이를 개선하기 위한 방법을 제시한다. 4장에서는 개선된 GC-tune의 성능을 측정하고 5장에서 토의를 한 뒤, 끝으로 6장에서 결론을 맺는다.

2. 관련 연구

이 장에서는 Haskell 병렬 프로그램의 최적 GC 옵션을 찾는 것이 왜 중요한지를 논한다. 먼저 Haskell 병렬 프로그래밍의 장단점을 소개한다. 그리고 Haskell 병렬 프로그램 성능 저하의 주된 원인으로 추정되는 GC 환경에 대한 이해를 위해 Haskell의 GC 방법인 세대별 GC 기법을 소개한다. 또한, 본 연구에서 개선할 Haskell 메모리 프로파일링 도구인 GC-tune을 소개한다.

2.1 Haskell

Haskell은 1990년에 만들어진 순수 함수형 언어로 무료로 사용할 수 있는 표준 지연 함수형 언어를 만들고자 학자들이 위원회를 조직하여 만든 언어다. Haskell의 장점으로는 쉬운 병렬 프로그래밍을 들 수 있다. 실제로 Haskell은 부수효과가 없어서 병렬 프로그래밍 시 흔히 발생하는 의존성 문제가 적다.

그러나 Haskell 병렬 프로그래밍에도 한 가지 문제가 있다. 사용 코어 수를 증가시켜가며 Haskell 병렬 프로그램을 실행해보면 프로그램의 속도 향상이 일정하지 않기 때문이다. 이후 실험에서 사용할 K-means 알고리즘을 구현한 Haskell 병렬 프로그램을 예시로 들겠다. 이 프로그램은 Ubuntu 14.04.1 버전의 32코어(CPU Opteron 6272 2개), 96GB RAM 환경에서 실행되었으며 약 120만 개의 2차원 점을 입력값으로 사용한다. 이렇게 측정된 K-means 알고리즘을 구현한 Haskell 병렬 프로그램의 속도 향상 그래프는 Fig. 1과 같다. Fig. 1을 보면 사용 코어 수가 1에서 6까지 늘어나는 동안에는 속도가 향상되지만, 그 이후부터는 속도 증가 그래프가 출렁이는 것을 알 수 있다.

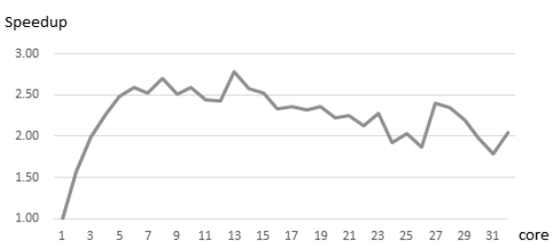


Fig. 1. Speedup of Haskell Parallel Programs Implementing K-means Algorithm

한편, Fig. 1에서 프로그램의 속도 향상이 3을 넘지 못하는 것은 크게 세 가지 이유 때문이다. 첫 번째는 최초 입력값을 읽어오는데 걸리는 시간 때문이고, 두 번째는 K-means 알고리즘의 각 단계마다 클러스터를 재조정하는 과정을 병렬화할 수 없기 때문이다. 마지막으로 세 번째는 프로그램이 K-means 알고리즘의 진행 과정을 화면에 출력하는 형태로 구현되어 있어 화면 출력에 시간을 소모하기 때문이다.

2.2 세대별 GC

Haskell은 기본적으로 세대별 GC 기법을 이용해서 메모리를 관리한다[8]. 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 저장한다. 이때 각 영역은 '세대'라는 표현으로 구분되며 생성 시기가 짧은 세대를 그렇지 않은 세대보

1) GHC(Glasgow Haskell Compiler)에서 제공하는 도구로서 2017년 7월 현재 최신 버전은 ghc-gc-tune-0.3이다.

다 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, 프로그램 실행 시 '`./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, Haskell 병렬 프로그램 확장성 연구[5]에 따르면 Haskell 병렬 프로그램의 저조한 속도 증가 원인은 GC 실행 환경 때문이라고 추정된다. 추정 근거는 Haskell 병렬 프로그램을 실행할 때 GC 옵션에 따라 바뀌는 실행 시간이었다. 이처럼 GC 옵션이 Haskell 병렬 프로그램의 성능에 영향을 주기 때문에 최적의 GC 옵션을 알아내는 것은 중요하다. 그러나 최적 GC 옵션을 구하기 위해 모든 GC 옵션 값에 대해서 실행 시간을 측정하는 것은 무리가 있다. 그러므로 빠른 시간 안에 차선값(suboptimal point)을 구하는 것이 중요하다. 한편, Haskell에는 GC-tune이라는 최적 GC 옵션을 찾아주는 기존 도구가 있다.

2.3 GC-Tune

GC-tune은 GHC에서 제공하는 메모리 프로파일링 도구이다[9]. GC-tune은 가능한 모든 경우의 '`-A`' 옵션과 '`-H`' 옵션에 대한 프로그램 실행 시간을 측정해서 그 중 가장 최적이라 판단되는 5개의 GC 실행 환경을 추천해 준다. 이때 '`-A`' 옵션과 '`-H`' 옵션의 값은 2의 거듭제곱 수로 한정된다. 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 4×3인 12가지가 된다.

3. GC-Tune의 개선

이 장에서는 기존 GC-tune의 문제점을 지적하고 어떻게 하면 이를 개선할 수 있는지에 대해 논한다. 구체적으로는 GC-tune 개선을 위해 설계한 분할정복법 알고리즘을 그림과 코드를 이용해서 설명한다.

3.1 기존 GC-tune의 문제점

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.

Table 1. To Obtain the Optimal GC Environment for the Maximal Independent Set Program, the Execution Time Measured by the Original GC-tune

(Option Unit: KB, Time Unit: s)

-H / -A	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

이후 실험에 사용할 극대 독립 집합[10]을 구하는 프로그램을 예로 들도록 하겠다. 기존 GC-tune이 이 프로그램의 최적 GC 환경을 구하기 위해 측정한 실행시간은 Table 1과 같다. Table 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.

3.2 분할정복법 적용

이 절에서는 기존 GC-tune의 문제점을 어떻게 개선할 수 있는지를 논한다. 본 연구에서는 GC-tune 개선에 분할 정복법을 사용했다. 분할 정복법을 택한 이유는 Fig. 2를 통해 설명할 수 있다. Fig. 2는 Table 1의 결과를 표면형 그래프로 나타낸 것이다. Fig. 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.

GC 옵션과 실행 시간의 관계가 오목함수 형태를 띠는 이유는 Haskell 세대별 GC의 특징 때문이다. '`-A`' 옵션을 예로

들자면 '-A' 옵션 값은 어린 세대가 사용할 힙 메모리 용량이다. 만약 이 값이 너무 작으면 크기가 큰 객체가 생성되었고 이 객체가 오랫동안 살아남아야 할 경우 성능이 떨어지게 된다. '-A' 옵션 값이 너무 큰 것도 성능을 떨어뜨린다. 어린 세대의 메모리 재사용 시 탐색할 영역이 넓어지기 때문이다.

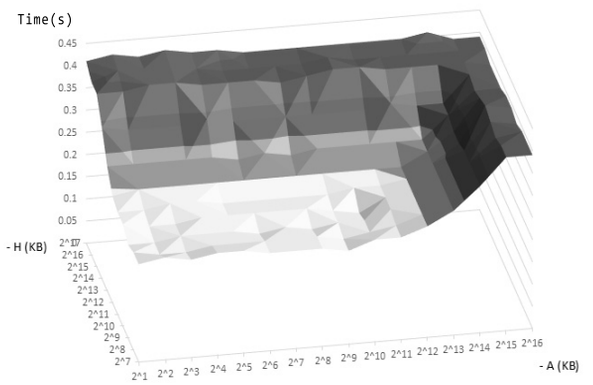


Fig. 2. Surface Type Graph for the Execution Time Data in Table 1

이러한 사실에 기초해서 GC-tune을 개선하는 분할정복 알고리즘을 설계하였다. 설계된 알고리즘은 이진 검색과 유사하며 구체적인 과정은 다음과 같다. 먼저 초기 탐색 영역을 설정하기 위해 가능한 '-A', '-H' 옵션의 최솟값, 최댓값을 구한다. 그러면 각 옵션마다 최소, 최대의 두 가지 경우가 존재하므로 가능한 조합은 총 4개이며 이를 초기 탐색 영역으로 설정한다. Table 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.

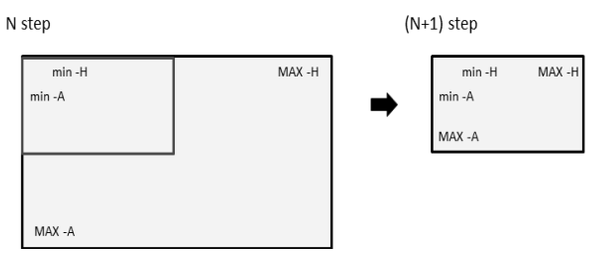


Fig. 3. The Reducing Search Area Process of the Divide-and-Conquer Algorithm for GC-tune Improvement

Fig. 3은 본 논문에서 제시하는 알고리즘의 동작 중 탐색 영역을 줄이는 과정을 도식화한 것이다. 이때 N 단계에서 가장 실행 속도가 빠른 경우는 '-A', '-H' 옵션이 모두 최소일 때로 가정했다. Fig. 3의 왼쪽 사각형은 N 단계의 탐색 영역을 나타낸다. N 단계에서 '-A', '-H' 옵션이 최소일 때 가장 실행 속도가 빨랐다고 가정했으므로 N+1 단계에서는 Fig. 3의 오른쪽 사각형과 같이 N 단계의 왼쪽 위 모서리를 기준으로 크기를 4분의 1로 줄인 탐색 영역이 사용된다.

3.3 분할정복법 알고리즘 구현

이 절에서는 앞서 제시한 알고리즘을 구현한 Haskell 코드를 설명한다. 먼저 구체적인 구현 코드 설명에 앞서 구현 코드의 이해를 돕기 위해 개선한 GC-tune이 사용하는 타입을 소개한다. 개선된 GC-tune이 사용하는 대표적인 타입은 다음과 같다.

```
type FileInfo  = (FilePath, [String], Int)
type Opt       = (Int64, Int64, Maybe Int64)
type ValidOpts = ([Int64], [Int64], Maybe Int64)
type OptI      = (Int, Int)
```

`FileInfo` 타입은 GC-tune의 측정 대상이 되는 프로그램의 정보를 뜻하는 타입이다. 순서쌍 타입이며 측정될 프로그램의 경로, 실행 인자, 사용할 코어 수로 구성된다. `Opt` 타입은 GC 옵션을 나타내는 타입이다. `Int64` 타입 두 개와 `Maybe Int64` 타입 하나로 이뤄진 순서쌍이며 왼쪽부터 '-A' 옵션값, '-H' 옵션값, 시스템에서 사용 가능한 최대 힙 메모리 용량을 뜻한다.

`ValidOpts` 타입은 GC-tune이 프로그램의 실행 시간을 측정할 때 사용 가능한 GC 옵션을 전부 나타내는 타입이다. 순서 쌍으로 구현되어 있으며 왼쪽부터 가능한 모든 '-A' 옵션의 리스트, 가능한 모든 '-H' 옵션의 리스트, 시스템에서 사용 가능한 최대 힙 메모리 용량을 뜻한다.

끝으로 `OptI`는 `ValidOpts` 타입의 '-A', '-H' 옵션 리스트의 원소에 접근하기 위한 첨자의 순서쌍이다. 왼쪽부터 '-A' 옵션 리스트에 접근하기 위한 첨자, '-H' 옵션 리스트에 접근하기 위한 첨자이다. 리스트의 첨자이기 때문에 리스트의 `n`번째 원소에 접근하기 위한 첨자는 `n`보다 `1` 작은 `n-1`이다.

```
1  findBestOpt :: FileInfo -> ValidOpts -> IO Opt
2  findBestOpt fileInfo validOpts = do
3    let
4      (optAs, optHs, _) = validOpts
5      maxAi  = length optAs - 1
6      maxHi  = length optHs - 1
7      initOptIs =
8          [(0,0), (0,maxHi), (maxAi,0), (maxAi,maxHi)]
9
10   bestOpt <- loop fileInfo initOptIs validOpts
11   return bestOpt
12 where
13   loop ...
```

Fig. 4. `findBestOpt`, a Core Function of Improved GC-tune which Find Optimal GC Options

본격적으로 개선된 GC-tune의 코드를 살펴보자. 개선된 GC-tune의 구현 코드 중 3.2절에서 설계한 알고리즘이 반영된 핵심 함수는 `findBestOpt`이다. `findBestOpt` 함수는 측

정 대상 정보, 사용 가능한 모든 GC 옵션을 인자로 받아서 최적 GC 옵션을 반환한다. 이러한 `findBestOpt` 함수의 구현 코드는 Fig. 4와 같다.

`findBestOpt`가 가장 먼저 하는 동작은 3.2절 알고리즘 설계에서 언급한 초기 탐색 영역 설정이다. Fig. 4의 3~8번 줄이 이에 해당하며 사용 가능한 '`-A`', '`-H`' 옵션의 최솟값, 최댓값을 조합하여 설정한다. 이렇게 길이 4인 GC 옵션 첨자의 리스트 `initOptIs`가 만들어진다. 이후 10~11번 줄과 같이 `findBestOpt` 함수는 지역적으로 정의된 `loop` 함수를 호출해서 최적 GC 옵션을 찾아낸다.

`findBestOpt` 함수 내부에 지역적으로 정의된 `loop` 함수의 구현 코드는 Fig. 5와 같다. `loop` 함수는 3.2절에서 설계한 알고리즘의 반복을 담당하는 함수다. `loop` 함수는 `findBestOpt`의 인자였던 측정 대상 정보, 사용 가능한 모든 GC 옵션 외에 추가로 탐색 영역을 인자로 받는다. 그리고 재귀적으로 호출되면서 탐색 영역을 변화시키는데 탐색 영역 변화가 없으면 재귀호출을 끝낸다.

구현 코드에 대한 자세한 설명에 앞서 `loop` 함수의 동작은 크게 세 단계로 나눌 수 있다. 순서대로 실행 시간 측정, 탐색 영역 재설정, 반복 결정인데 Fig. 5의 3~7번 줄이 실행 시간을 측정하는 단계이다. 9~10번 줄은 탐색 영역을 재설정하는 단계이고 12~14번 줄이 반복 여부를 결정하는 단계이다.

```
1  loop :: FileInfo -> [OptI] -> ValidOpts -> IO Opt
2  loop fileInfo optIs validOpts = do
3    let (optAs, optHs, maxmem) = validOpts
4    res <- forM optIs $ \(a_i, h_i) -> do
5      let opt = (optAs!!a_i, optHs!!h_i, maxmem)
6      Just s <- runGHCProgram fileInfo opt
7      return (totalTime s, (a_i, h_i))
8
9    let (time, (a_i, h_i)) = minimum res
10       nextOptIs = map (mid (a_i, h_i)) optIs
11
12   bestOpt <- case (nextOptIs /= optIs) of
13        True -> loop fileInfo nextOptIs validOpts
14        _    -> return (optAs!!a_i, optHs!!h_i, maxmem)
15   return bestOpt
16   where
17     mid ...
```

Fig. 5. Locally Defined `loop` Functions Inside `findBestOpt`

먼저 실행 시간을 측정하는 부분을 살펴보면 4번 줄에서 `loop` 함수의 두 번째 인자인 탐색 영역 `optIs`에 대해 `forM` 반복문을 통한 원소 접근이 시작된다. 탐색 영역 `optIs`의 각 원소는 사용 가능한 GC 옵션 리스트에 접근하기 위한 첨자를 뜻하므로 5번 줄과 같이 리스트 원소 접근 연산자 '`!!`'를 이용해서 실제 GC 옵션으로 바꿀 수 있다. 이렇게 만들어진 GC 옵션은 6번 줄과 같이 `runGHCProgram` 함수의 인자로서 실행 시간 측정에 사용된다.

`runGHCProgram` 함수는 Haskell 프로그램의 실행 시간을 측정하는 함수로 프로그램의 GC 사용 시간, 전체 실행 시간 등 다양한 정보를 측정하여 반환한다. 7번 줄의 `totalTime` 함수는 `runGHCProgram` 함수의 반환 값에서 전체 실행 시간만을 추출하는 함수다.

여기까지 진행되면 탐색 영역의 각 원소에 대한 실행 시간 측정이 끝난다. 9번 줄부터의 동작은 구체적인 예를 통해 살펴보자. 탐색영역 `optIs`가 `[(0, 0), (0, 4), (4, 0), (4, 4)]`라 하자. 그리고 `optIs`의 각 원소에 해당하는 GC 옵션으로 프로그램을 실행했을 때 측정된 시간이 순서대로 `1.1`, `2.0`, `3.2`, `4.7`초라고 하자. 그러면 3 ~ 7번 줄을 통해 만들어지는 실행 시간 측정 결과는 `[(1.1, (0, 0)), (2.0, (0, 4)), (3.2, (4, 0)), (4.7, (4, 4))]`로 표현할 수 있다. Fig. 5의 `res` 변수가 이에 해당한다. 9번 줄에서는 이렇게 측정된 결과에서 실행 시간이 최소인 옵션을 찾게 된다. 앞서 소개한 예에서는 `(1.1, (0, 0))`이 된다.

10번 줄에서는 9번 줄에서 찾은 옵션을 이용해서 탐색 영역을 재설정한다. 이때 탐색 영역을 재설정하는 방법은 기존 탐색영역의 원소를 9번 줄에서 찾은 옵션과의 중점으로 대체하는 것이다. 앞서 소개한 예로 설명하자면 이전 탐색 영역 `[(0, 0), (0, 4), (4, 0), (4, 4)]`와 현재 최적 옵션 `(0, 0)`에 대해 `(0, 0)`과 `(0, 0)`의 중점은 `(0, 0)`, `(0, 4)`와 `(0, 0)`의 중점은 `(0, 2)`, 같은 방법으로 나머지 옵션에 대해서도 중점을 계산하면 새로운 탐색 영역 `[(0, 0), (0, 2), (2, 0), (2, 2)]`를 만들어낼 수 있다. 이렇게 탐색 영역을 재설정하면 2차원 평면 관점에서 탐색 영역의 크기가 이전 단계보다 4분의 1로 줄어들게 된다.

다음으로 12~14번 줄은 재귀 호출의 종료를 결정하는 코드다. 12번 줄에서는 새로 구한 탐색 영역이 이전의 탐색 영역과 같은지를 판별한다. 만약 두 영역이 다르다면 13번 줄과 같이 `loop` 함수를 재귀 호출한다. 그렇지 않고 같다면 즉 탐색 영역의 변화가 없다면 14번 줄과 같이 현재의 최적 옵션을 최종 최적 옵션으로 보고 재귀 호출을 종료한다.

Table 2. The Measured Time by Improved GC-tune to Obtain the Optimal GC Environment of the Maximal Independent Set Program

Step	-A(KB) / -H(KB) / Time(s)			
	Min-A Min-H	Min-A Max-H	Max-A Min-H	Max-A Max-H
1	$2^{1}/2^{7}$/ 0.23	$2^{1}/2^{17}$/ 0.41	$2^{16}/2^{7}$/ 0.40	$2^{16}/2^{17}$/ 0.39
2	$2^{1}/2^{7}$/ 0.23	$2^{1}/2^{12}$/ 0.25	$2^{8}/2^{7}$/ 0.25	$2^{8}/2^{12}$/ 0.22
3	$2^{4}/2^{9}$/ 0.23	$2^{4}/2^{12}$/ 0.22	$2^{8}/2^{9}$/ 0.22	$2^{8}/2^{12}$/ 0.22
4	$2^{4}/2^{10}$/ 0.22	$2^{4}/2^{12}$/ 0.21	$2^{6}/2^{10}$/ 0.22	$2^{6}/2^{12}$/ 0.21
5	$2^{4}/2^{11}$/ 0.22	$2^{4}/2^{12}$/ 0.22	$2^{5}/2^{11}$/ 0.22	$2^{5}/2^{12}$/ 0.22
6	$2^{4}/2^{11}$/ 0.22	$2^{4}/2^{11}$/ 0.21	$2^{4}/2^{11}$/ 0.22	$2^{4}/2^{11}$/ 0.22

개선한 GC-tune으로 Table 1에 사용된 극대 독립 집합을 구하는 Haskell 프로그램의 최적 GC 환경을 다시 구해보았다. 이 과정에서 단계별로 측정된 실행 시간은 Table 2

와 같다. 각 단계별로 4번의 실행 시간 측정이 필요하고 6단계를 반복했으므로 총 24번 실행 시간을 측정하였다. 그리고 전체 실행 시간은 약 6초이다. 이는 176번의 실행 시간 측정이 필요하고 총 54초가 걸린 개선 전에 비해 실행 시간 면에서는 9배, 측정 횟수 면에서는 7배가 개선되었음을 뜻한다.

4. 실 험

이 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 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 개선 전 실행시간의 집합 G와 개선 후 실행시간의 집합 G'으로 계산할 수 있다. 정확도 $Prec(G, G')$은 Equation (1)과 같이 계산한다. 이때 함수 max와 min은 각 인수 집합에서 최대 원소와 최소 원소를 돌려주는 함수이다.

$$Prec(G, G') = \frac{\max(G) - \min(G')}{\max(G) - \min(G)} \quad (1)$$

단순히 G와 G'의 최솟값을 비교할 수도 있다. 그러나 본 연구에서는 G의 최댓값과의 차를 비교하였다. 왜냐하면 G 또는 G'의 최솟값은 얼마만큼 빠른 옵션을 찾았느냐에 대한 척도인데 최악의 실행시간에 따라 이를 다르게 해석해야 하기 때문이다. 예를 들어 $\min(G) = 2$, $\min(G') = 3$, $\max(G) = 4$인 프로그램이 있다고 하자. 그러면 최악의 실행시간 대비 개선 전의 GC-tune은 2초 빠른 최적 실행 시간을 찾았고 개선 후 GC-tune은 1초 빠른 최적 실행 시간을 찾은 것이 된다. 즉, 개선 전 GC-tune이 개선 후 GC-tune보다 실행 시간을 두 배 단축한 것이다. 그래서 이 경우 아무리 측정에 필요한 시간이 짧았어도 개선 후의 GC-tune 성능을 긍정적으로 평가하기 어렵다.

그러나 $\max(G) = 10000$이었다면 다른 해석을 할 수 있다. 이 경우 개선 전 GC-tune은 9,998초 빠른 최적 실행 시간을 찾았고 개선 후 GC-tune은 9,997초 빠른 최적 실행 시간을 찾은 것이 된다. 때문에 $\max(G) = 4$인 경우에 비해서 최적 실행 시간 차이인 1초를 큰 차이라고 볼 수 없다. 즉, 똑같은 최적 실행 시간 차이라도 최악의 실행 시간이 얼마였느냐에 따라 다른 해석이 필요하다.

4.2 실험 결과

먼저 극대 독립 집합 프로그램을 이용해서 GC-tune의 개선 전후 성능을 측정한 결과는 Table 3과 같다. Table 3을 보면 개선 전의 GC-tune보다 개선 후의 GC-tune이 월등히 빠르면서도 높은 정확도를 유지함을 알 수 있다. 수치적으로는 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 실행 시간을 7.78배 단축시켰음을 알 수 있다.

Table 3. Comparison of GC-tune Performance Before and After Improvement about Maximal Independent Set Program

Core	Total Time(s)		Accuracy of Improved GC-Tune(%)	Improvement Degree of Total Time(times)
	Original	Improved		
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 Table 3에서 코어 1 경우를 보면 개선된 GC-tune의 총 실행 시간이 5.88인데 실제로 코어 1일 때 극대 독립 집합 프로그램의 최적 시간은 0.21이다. 최적 GC 환경을 알아내기 위해 최적 시간의 28배에 달하는 시간을 사용한 셈이다. 이는 GC-tune이 현업 수준의 프로그래밍에 사용되려면 실행 시간 측정횟수를 더욱 줄여야 함을 뜻한다.

최적 GC 옵션을 찾기 위해 프로그램 실행 시간을 2차 곡면에 근사시켜보기도 하였다. Fig. 2와 같이 GC 옵션에 따른 프로그램 실행 시간이 오목 함수 형태를 띠기 때문인데 만약 실제로 GC 옵션에 따른 프로그램 실행 시간이 2차 곡면과 유사하다면 프로그램 실행 시간 측정 횟수를 특정 상수보다 작게 만들 수 있게 된다. 그러나 GC 옵션에 따른 프로그램 실행 시간을 2차 곡면에 근사해본 결과 근사시킨 2차 곡면의 최솟값과 실제 실행 시간의 최솟값 사이에는 큰 차이가 있었으며 이 방법을 사용할 수 없었다.

6. 결 론

본 연구에서는 분할 정복법을 응용해서 Haskell 메모리 프로파일링 도구인 GC-tune의 실행 시간을 개선하는 방법을 제시하였다. 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 횟수는 여전히 늘어나게 된다. 그래서 사용할 수 있는 '`-A`', '`-H`' 옵션 범위가 커지더라도 최적 GC 환경을 구하기 위한 GC-tune의 측정 횟수를 일정 상수보다 작게 만들 수 있는 방법이 필요한데 이는 향후 연구로 생각해 볼 수 있다.

References

[1] Marc Airhart. Hot Chips: Managing Moore's Law, 2015. [Online; accessed 22-may-2017]. URL: https://news.utexas.edu/2015/04/15/hot-chips-managing-moores-law.

[2] L. Chai, Q. Gao, and D. K. Panda. "Understanding the impact of multi-core architecture in cluster computing: A case study with intel dual-core system," *Cluster Computing and the Grid 2007. Seventh IEEE International Symposium on*. IEEE, pp.471-478, 2007.

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

[4] 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.

[5] H. Kim, H. An, S. Byun, and G. Woo, "An Approach to Improve the Scalability of Parallel Haskell Programs," *Proceedings of 2016 International Conference on Computing Convergence and Applications (ICCCA 2016)*, pp.175-178, 2016.

[6] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, "*Introduction to algorithms,*" 3rd ed., MIT press, 2009.

[7] G. Brassard and P. Bratley, "*Fundamentals of algorithmics*," Vol.33. Englewood Cliffs: Prentice Hall, 1996.

[8] 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.

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

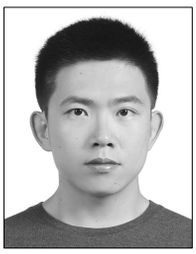
[10] G. Chris and R. Gordon, *"Algebraic Graph Theory,"* Springer New York, 2001.

안 형 준

e-mail : hyungjun@pusan.ac.kr
2015년 연세대학교 수학과(학사)
2015년～현 재 부산대학교 전기전자 컴퓨터공학과 석사과정
관심분야 : 함수형 프로그래밍, 역공학, 병렬 프로그래밍, 사용자 인터페이스

김 화 목

e-mail : hwamok@pusan.ac.kr
2015년 조선대학교 제어계측로봇공학과 (학사)
2015년～현 재 부산대학교 전기전자 컴퓨터공학과 석사과정
관심분야 : 함수형 프로그래밍, 병렬 프로그래밍, 임베디드 시스템, 소프트웨어 교육



Xiao Liu

e-mail : liuxiao@pusan.ac.kr
2014년 부산대학교 정보컴퓨터공학과(학사)
2014년～현 재 부산대학교 전기전자 컴퓨터공학과 석・박사통합과정
관심분야 : Web 프로그래밍, Cloud Computing

김 연 어

e-mail : yeoneo@pusan.ac.kr
2010년 동아대학교 컴퓨터공학과(학사)
2012년 동아대학교 컴퓨터공학과(석사)
2012년～현 재 부산대학교 전기전자 컴퓨터공학과 박사과정
관심분야 : 프로그래밍 분석, 정적 분석, 표절 검사, 함수형 언어

변 석 우

e-mail : swbyun@ks.ac.kr
1980년 숭실대학교 전자계산학과(공학사)
1982년 숭실대학교 전자계산학과(공학석사)
1982년～1999년 ETRI(책임연구원)
1995년 Univ. of East Anglia(영국), Computer Science(박사)
1999년～현 재 경성대학교 컴퓨터공학부 교수
관심분야 : 함수형 프로그래밍, 정형 증명, 프로그래밍 언어

우 균

e-mail : woogyun@pusan.ac.kr
1991년 한국과학기술원 전산학(학사)
1993년 한국과학기술원 전산학(석사)
2000년 한국과학기술원 전산학(박사)
2000년～2004년 동아대학교 컴퓨터공학과 조교수
2004년～현 재 부산대학교 전기전자컴퓨터공학과 교수
관심분야 : 프로그래밍언어 및 컴파일러, 함수형 언어, 프로그램 분석, 프로그램 시각화, 프로그래밍 교육, 한글 프로그래밍 언어