An Approach to Improve the Scalability of Parallel Haskell Programs

Hwamok Kim[1*], Hyungjun An[2], Sugwoo Byun[3], and GyunWoo[4]

[1, 2, 4]Department of Electrical Computer Engineering, Pusan National University, Busan, Korea
[3]Department of Computer Engineering, Kyungsung University, Busan, Korea
[Email :{hwamok, hyungjun, woogyun}@pnu.edu[1,2,4], swbyun@ks.ac.kr[3]]

Abstract

Though the performance of computer hardware is increasing owing to the many cores, the software counterpart is lacking of the proportional throughput. Functional languages can be one of the alternatives to promote the performance of parallel programs since those languages have an inherent parallelism in evaluating pure expressions without side-effects. Specifically, Haskell is notably popular in 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 is getting increased. The garbage collector is suspected to be the source of this fluctuation because it affects both on the space and the time for the execution of programs. This paper justifies that it truly is using the specific tuning tool, namely GC-Tune. We have tuned the behavior of the garbage collector in the executions of a large-scale parallel K-means program. As a result, the scalability has been improved by 71% and the fluctuation range is narrowed down by 45% compared to the original execution of the program without any tuning.

Index Terms: Parallel Programming, Haskell, Garbage Collection, GC-Tune, K-means

I. INTRODUCTION

Though the multi-core or many-core processors are getting popular recently, the software counterpart hardly takes up the technology of the hardware. It is apparent that the parallel programming is mostly appropriate to take the full advantage of the multi-core, but it is extremely hard to write parallel programs especially in imperative languages. Functional languages such as Haskell are considered as alternatives to parallel programming since their pure functional nature promote the implicit parallelism.

However, the experiments on typical parallel Haskell programs indicate that the scalability of the parallel executions of them is not proportional to the number of available cores; the speed-up graph even shows unexpected fluctuations particularly when the number of cores is getting higher. This result may be a natural consequence since the parallel Haskell programs are executed on top of the run-time system supporting the pure functional nature of the language excluding any side-effects. Therefore, the run-time system including the garbage collector is suspected to be the reason of this bad scalability.

This paper proposes a method to improve the scalability of the executions of parallel Haskell programs as the number of cores is getting larger. Specifically, we used the tool called GC-Tune, which is an auxiliary tool of the Glasgow Haskell Compiler (GHC). This tool enables a fine tuning of the sizes of several sections of the heap memory, which is the target of the garbage collection. We performed several experiments on the scalability of the executions of a parallel K-means program as the available number of cores is getting increased.

The structure of this paper is as follows. Section 2 briefly introduces Haskell and GC-Tune as related work. Section 3 describes the tuning procedure and the experimental result on tuning the garbage collector parameters of the executions of K-means. Section 4 discusses the experimental results and Section 5 concludes.

II. RELATED WORK

A. *Features of Haskell*

Haskell is a pure functional language in which the whole program is written as a sequence of equations much like mathematical declarations. The purity guarantees that the function result is solely determined by the arguments. Therefore, the expressions have implicit parallelism. Additionally, Haskell provides parallel constructs using Eval monads for easy parallelization of the existing imperative sequential programs [1].

B. *GC-Tuning Tool*

The GHC provides a powerful tuning tool called GC-Tune suggesting the best memory size for the execution of a program [2]. GHC supports multiple-stage garbage collectors executed on the virtual machine, and a typical algorithm for the garbage collection is a generational one [3-4]. The generational garbage collector manages the heap memory in multiple generations of objects. And the sizes of them can be set using the compiler options: H for the total size of heap memory and A for that of the young generations. GC-Tune calculates the best heap sizes for execution of a Haskell program.

III. K-MEANS PERFORMANCE ANALYSIS

A. *Performance analysis of K-means without GC-tuning*

The program used for the experiment is a parallel K-means program in Haskell. The K-means is a well-known algorithm combining a large set of randomly given two-dimensional points into several clusters [5]. In our experiments 1.2 million random points are given to generate five clusters. The experiment environmental is Ubuntu (14.04.1 LTS) on top of two 16-core CPU (Opteron 6272), 32 cores in total, with 96GB memory. The result of the first execution without GC-tuning is shown in Fig. 1 and Table 1.

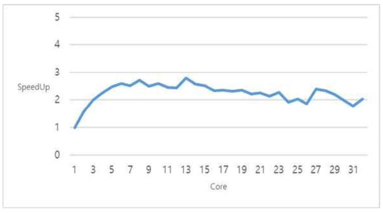

Fig. 1. Run-time speedup of K-means without tuning

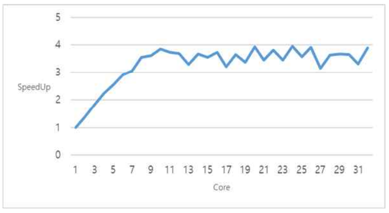

Fig. 2. Run-time speedup of K-means with maximum possible memory

Table 1.

Run-time speedup of K-means without tuning

Cores	Time	Speedup
5	18.33	0.40
10	17.51	0.39
15	18.00	0.40
20	20.47	0.45
25	22.35	0.49
30	23.04	0.51

Table 2.

Run-time speedup of K-means with maximum possible memory

Cores	Time	Speedup
5	22.31	2.55
10	14.74	3.86
15	16.00	3.56
20	14.44	3.94
25	15.88	3.58
30	15.61	3.64

Table 3.

Run-time speedup of K-means with GC-tuning

Cores	Time	Speedup
5	21.69	2.61
10	16.35	3.46
15	15.12	3.75
20	15.23	3.72
25	15.24	3.72
30	16.75	3.38

As shown in Fig. 1, the speedup of the parallel program is not observable for the cores more than five, which implies that there is no scalability above five. For 30 cores, the execution time is even more than that of 5 cores. The reason for this bad scalability is presumed to be in the run-time system (RTS), including the garbage collector, of the virtual machine since it is not specialized for the parallel execution of programs. For a next step, additional experiments will be performed to validate this presumption.

B. Performance analysis of K-means with GC-tuning

To estimate the maximum scalability, the maximum possible memory is set for the second execution, the result of which is shown in Fig. 2 and Table 2. The size of the maximum possible heap memory is 268,435,456 bytes, which can be set using run-time option ‘H.’ And the size of memory for the young generation is set to the half of the maximum size using option ‘A.’

Fig. 2 shows that the scalability is improved up to ten cores, but unexpected fluctuation occurred for the large number of cores more than ten. The RTS is highly suspected to be the cause of this fluctuation. As a final experiment, the execution time with fine tuning of the garbage collection is measured.

Table 3 shows the average execution time of ten executions with the best heap size calculated by GC-Tune. This result is compared with that of Table 1 and 2 resulting Fig. 3 and 4. As shown Fig. 3 and 4, the fluctuations are disappeared without losing the scalability. As a result, it was confirmed that the cause the bad scalability is the RTS. The scalability of the speedup of the GC-tuned executions is improved by 71% compared with that of those without GC-tuning as shown in Fig. 3.

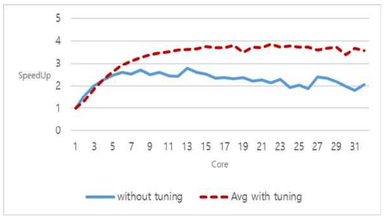

Fig. 3. Run-time speedup of K-means without tuning & with GC-tuning

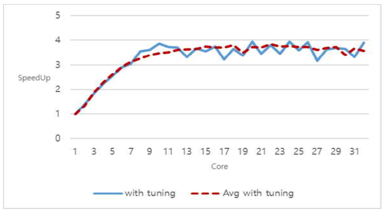

Fig. 4. Run-time speedup of K-means with maximum possible memory & with GC-tuning

Concerning the fluctuations, the GC-tuned executions still show slight fluctuations. The range of fluctuations of the GC-tuned executions is compared with that of executions with the maximum possible heap memory (Fig. 4), resulting that the range is reduced by 45%.

IV. DISCUSSION

The experimental results in Section 3 shows that the RTS greatly affects the executional behavior of parallel Haskell programs. We tried to find the best heap sizes for the number of cores involved in the parallel executions, but there seems no certain rule for determining the heap sizes. However, the scalability of the parallel executions can be improved using GC-tune.

Tuning the sizes of heap sections is one way to improve the performance of parallel Haskell programs, but the RTS itself eventually should be re-organized to cope with many cores to get the most performance of parallel executions. In the meanwhile, GC-tuning can be an effective way to improve the scalability of parallel Haskell programs.

V. CONCLUSION

The reason of the bad scalability of parallel Haskell programs is presumed to be due to the run-time system, particularly the garbage collector. With this assumption, this paper presents several experiments on tuning the garbage collection for the execution of a parallel K-means program, resulting the improvement of scalability by 71%. The fluctuation range of speed-ups is also observed to be narrowed down by 45% compare to the execution without tuning. This result indicates that the run-time system of Haskell should be adjusted to take the full advantage of massively parallel many core systems. In the meanwhile, tuning the run-time system including the garbage collection is found much helpful to promote the scalability of parallel Haskell programs. Developing an automatic tuning method for parallel Haskell programs can be considered as a future work.

ACKNOWLEDGMENTS

This work was supported by Institute for Information & communications Technology Promotion (IITP) grant funded by the Korea government (MSIP) (No. B0101-16-0644, Research on High Performance and Scalable Manycore Operating System).

REFERENCES

[1] S Marlow, *Parallel and Concurrent Programming in Haskell*, O'Reilly Media, 2013.

[2] D. Stewart, ghc-gc-tunes, [Online]. Available:http//hackage.haskell.org/package/ghc-gc-tune. (downloaded 2016. Oct. 13)

[3] S. Marlow, et al. "Parallel generational-copying garbage collection with a block-structured heap," In *Proceedings of the 7th international symposium on Memory management*. pp.11-20, 2008.

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

[5] J. MacQueen, "Some methods for classification and analysis of multivariate observations," In *Proceedings of the fifth Berkeley symposium on mathematical statistics and probability.* pp.281-297, 1967.