Journal of Theoretical and Applied Information Technology
30th September 2017. Vol.95. No.18

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

AN APPROACH TO IMPROVING THE SCALABILITY OF PARALLEL HASKELL PROGRAMS

[1]HWAMOK KIM, [1]JOOWON BYUN, [2]SUGWOO BYUN, [3]GYUN WOO

[1,3]Dept. of Electrical and Computer Engineering, Pusan National University, Busan 46241, South Korea

[2]Dept. of Computer Engineering, Kyungsung University, Busan 48434, South Korea

[3]Smart Control Center of LG Electronics, Busan 46241, South Korea

E-mail: [1,3]{hwamok, joowon, woogyun}@pusan.ac.kr, [2]swbyun@ks.ac.kr

ABSTRACT

Though the performance of computer hardware is increasing owing to the many-core architectures, the software counterpart is lack of the proportional throughput. In this situation, 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 on both the space and the time for the execution of programs. This paper justifies this conjecture using the specific tuning tool, namely GC-Tune. We have tuned the behavior of the garbage collector in the executions of three large-scale parallel programs: the K-means, a maximal independent set, and plagiarism detection programs. As a result, the scalabilities of the programs have been improved by 38%, 21%, and 7%, respectively; the fluctuation ranges are also narrowed down by 45%, 30%, and 58%, respectively, compared to the original execution of the programs without any tuning. This result implies that GC-tuning can be an effective method to promote the scalability of parallel Haskell programs. In results, the execution time of parallel programs can also be much accurately estimated.

Keywords: *Parallel Programming, Haskell, Garbage Collection, GC-Tune, K-means, Maximal Independent Set, Plagiarism Detection*

1. INTRODUCTION

Though the multi-core or many-core processors are getting popular recently, the software counterpart hardly takes up the technology of the hardware [1]. 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 promotes the implicit parallelism [2].

Despite the many advantages of parallel programming, 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 [3].

This bad scalability can be a significant problem for real-time applications since the execution time cannot be predictable for many-core environment. The response-time predictability is a valuable criterion especially for real-time applications. According to our experiment, the execution time for many-core environment can be even worse than that of the less-core environment. Practically, the bad scalability voids the advantage of many-core resources in this sense.

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

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

a supplementary tool of the Glasgow Haskell Compiler (GHC). This tool enables the 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 three programs (a parallel K-means, a maximal independent set, and a plagiarism detection programs) as the available number of cores is increased.

The structure of this paper is as follows. Section 2 briefly introduces Haskell, GC-Tune, and Eval Monad as related work. Section 3 describes the scalability problem and the method taken to tackle this problem. Section 4, 5, and 6 describe the experimental results on the executions of the K-means, the maximal independent set, and the plagiarism detection programs, respectively. Section 6 discusses on the results. Section 7 describe concludes. Finally, Section 8 addresses some future directions of this research.

2. RELATED WORK

2.1 Features of Haskell

Haskell is a purely functional language in which the whole program is written as a sequence of equations much like mathematical declarations [4, 5]. Since Haskell has no side-effects, the purity of Haskell expressions guarantees that the function result is solely determined by the arguments. Therefore, Haskell expressions have implicit parallelism. However, we need some impure features to handle the input and output facilities.

Haskell provides monads to handle the impure features such as I/O operations with side effects [6]. A monad is a special type to ensure that a set of operations to be executed sequentially. This implies that the result of the previous computation can be handled by the next computation sequentially. Actually, the monad a class of types including IO monads for input and output and ST monads for a storage with states. Further, a new monadic type can be defined by the programmer as needed.

Haskell also provides lightweight threads. Many programming languages provide lightweight threads to increase concurrency. Typical examples are Haskell, Go, and Erlang [7]. In particular, Haskell lightweight threads are more suitable for parallelism because of the small overhead of context switching [8].

2.2 GC-Tuning Tool

The GHC provides a powerful memory tuning tool called GC-Tune, suggesting the best memory size for the execution of a program [9]. GHC supports multiple-stage garbage collectors executed on the virtual machine, and GC-Tune calculates the best heap sizes for the executions of a Haskell program. GC-Tune even shows the tuning result graphically to reveal the memory consumption tendency.

A typical garbage collecting algorithm adopted in Haskell is a generational one [10, 11, 12]. The generational garbage collector manages the heap memory in multiple sections assigned for different generations of objects. Also, the sizes of them can be set using the run-time options: '–H' for the total size of heap memory and '–A' for that of the young generations. GC-Tune fine tunes these options to run the target program to optimize the speed, and provides the user with the best options.

2.3 Parallelization with the Eval Monad

The Eval monad provides a parallel programming facilities suitable for a shared memory model. It defines several parallel evaluation strategies, and these strategies can be applied to function expressions to make parallel operations. The parallelized expressions change into sparks. The sparks in Haskell are the computational units that can be executed in parallel. The Haskell sparks are much fine-grained units smaller than lightweight threads as shown in Fig 1.

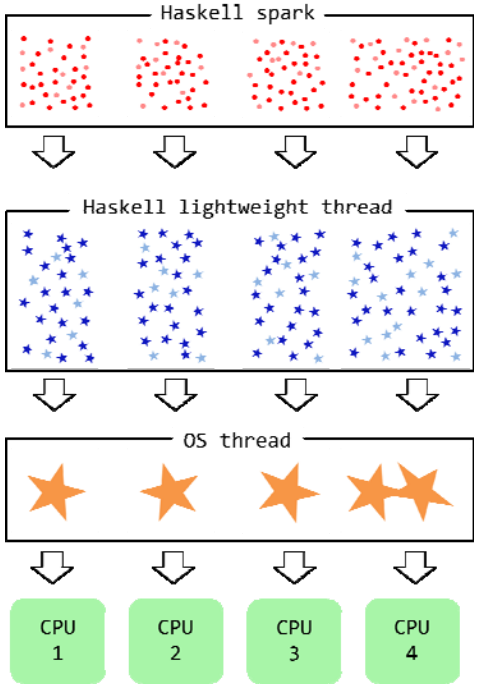

Figure 1: Haskell spark conversion process. The fine-grained Haskell sparks in the spark pool are converted into coarse-grained Haskell lightweight threads.

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

Haskell threads are under control of the run-time system.

Fig. 1 shows the conversion process until the sparks are delivered to the CPU. The sparks are generated by evaluation strategies and delivered to the spark pool. And then, they are changed into lightweight threads under the control of the RTS (run-time system) during execution, or processed by garbage collection if they do not need to be evaluated. The lightweight threads generated from sparks are passed to the OS threads if available, and executed by the CPU.

3. THE PROBLEM AND THE METHOD

To take a quick grasp of the problem, let us examine the graph shown in Fig. 2. The graph shows the speedup of a parallel K-means program in Haskell. From one to five cores, the speedup is also scaled up. But above six cores, the speedup is not observed proportional to the number of cores. For 31 cores, specifically, the speedup is even worse than that of five cores.

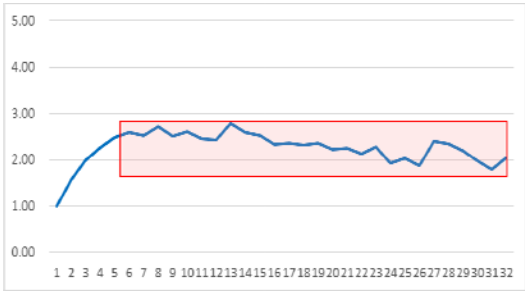

Figure 2: The scalability problem found in a parallel K-means program written in Haskell. The boxed area shows the fluctuation of speed-up.

The reason for this bad scalability is presumed to be in the RTS, including the garbage collector, of the virtual machine since it is not specialized for the parallel execution of programs [13]. Even though multiple threads are running, there is a single run-time system. This implies that the overhead due to RTS can be a source of this bad scalability especially for the many-core environment.

In particular, Haskell's garbage collection may not suitable for parallel programs [14]. Haskell uses generational garbage collection as a garbage collection method. This is based on the hypothesis that most of the generated objects will die prematurely. The generated objects are placed in two or more physically separated areas within the heap memory and are divided into several generations according to time or size.

Though idea of the generational garbage collector is good, the control of it in the parallel setting can be unpredictable. All the objects in the generational garbage collector are centrally managed. Further, the garbage collector is not processed in parallel. Therefore, the overhead during run-time may greatly affect the throughput of parallel processing.

To justify this presumption, we applied memory tuning using GC-Tune. Using GC-Tune, we can reduce the effect of the garbage collector. Reducing the fluctuation of the speedup, the scalability can be improved hopefully.

Sections 4, 5, and 6 show the results of the experiment. The executional environment for the parallel programs is Ubuntu (14.04.1 LTS) on top of two 16-core CPU (Opteron 6272), 32 cores in total, with 96GB memory.

The experimental method is as follows. First, the performance is measured without GC-tuning. Next, the performance is measured using the largest possible heap memory. Finally, the performance is measured with GC-tuning. And the scalability and the fluctuation range in the speedup graph is measured for comparison.

4. K-MEANS PERFORMANCE ANALYSIS

4.1 Performance analysis of K-means without GC-tuning

The program used for the first 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 [15, 16]. Since the clustering can be performed in parallel, K-means is a typical data-parallel program. In our experiments, 1.2 million random points are given to generate five clusters. The result of the first execution without GC-tuning is shown in Fig. 3 and Table 1 (Without Tuning).

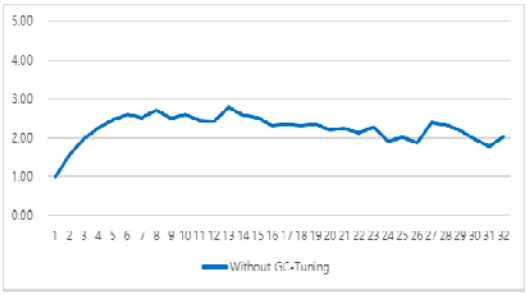

Figure 3: Run-time speedup of K-means without tuning. The graph shows that the speed-up is unstable when the number of cores is greater than five.

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

As noted before, the speedup of the parallel program is not observable for the cores more than five as shown in Fig. 3, which implies that there is no scalability above five. Even worse, the speed-up is not predictable. For the next step, additional experiments will be performed to validate this presumption.

4.2 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. 4 and Table 1 (Maximum Memory). 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.'

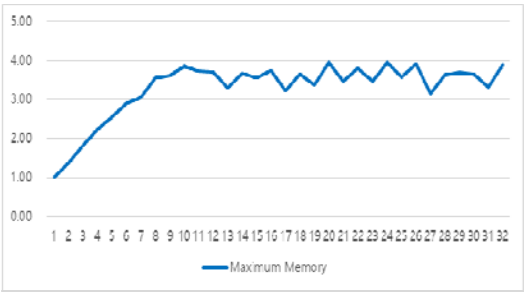

Figure 4: Run-time speedup of K-means with maximum possible memory. The fluctuation of the speed-up is deferred until the number of cores is ten, but it is getting even worse when the number of cores is more than ten.

Comparing the columns of Table 1 (Without Tuning and Maximum Memory), the garbage collection time does not differ much. However, it is confirmed that the runtime scarcely decreases when the maximum possible memory is set. Fig. 4 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.

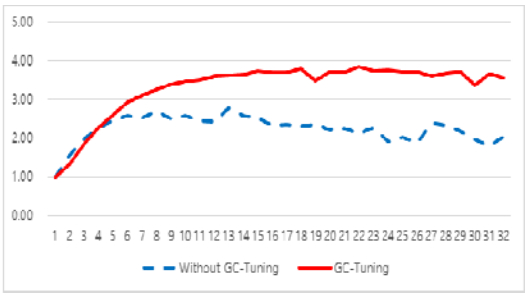

Figure 5: Run-time speedup of K-means without tuning and with GC-tuning. The GC-tuned execution shows much smooth speed-up change, and the speed-up remains stable until the number of cores is 18.

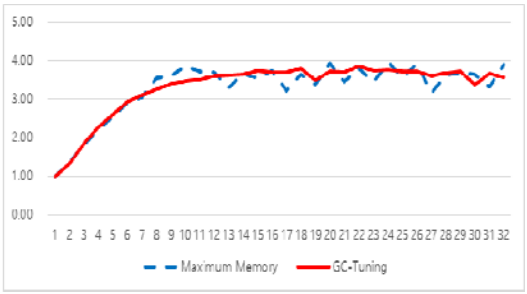

Figure 6: Run-time speedup of K-means with maximum possible memory and with GC-tuning. The GC-tuned result is more stable than the maximum heap case, which implies that the speed-up is more predictable.

The third column of Table 1 (With GC-Tuning) shows the average execution time of ten executions with the best heap size calculated by GC-Tune. This result is compared with the previous results of Table 1 (Without Tuning and Maximum Memory). As shown Fig. 5 and 6, the fluctuation is disappeared without losing the scalability.

The scalability of the speedup of the GC-tuned executions is improved by 38% compared with that of those without GC-tuning as shown in Fig. 5. The range of fluctuation of the GC-tuned executions is compared with that of executions with the maximum possible heap memory (Fig. 6) resulting that the range is reduced by 45%.

One notable finding from Fig. 6 is that there is a speed-up plateau more than 17 cores. In this plateau it seems that the overhead for maintaining the many-core compensates the performance gain owing to many-cores.

Even in the plateau section, the fluctuation of the speedup is much reduced. Though some fluctuations are still found (in 19 and 30 cores), the fluctuation range is mostly reduced in overall. In summary, GC-tuning contributes even in the plat-

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

eau section, resulting a better predictability for the execution time of the program.

5. MAXIMAL INDEPENDENT SET ANALYSIS

5.1 Performance analysis of Maximal Independent Set without GC-tuning

The program for the second experiment is a parallel maximal independent set program in Haskell. In graph theory, an independent set is a set of non-contiguous nodes. A maximal independent set is a set that is not a subset of another independent set. The maximal independent set program finds all independent sets in parallel [17, 18, 19]. In order to test this code, we assigned a graph with ten nodes as an input data. The experimental environment is the same as test of K-means. The result of the first execution without GC-tuning is shown in Fig. 7 and Table 2 (Without Tuning).

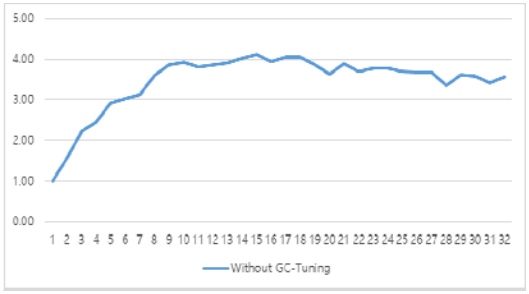

Figure 7: Run-time speedup of Maximal Independent Set without tuning

As shown in Fig. 7, the speedup of the parallel program is not observable for the cores more than ten, which implies that there is no scalability when the number of cores is above ten. As shown in Table. 2 (Without GC-Tuning), garbage collection time was measured more than K-means program. It is estimated that the number of objects allocated and deallocated is larger than that of K-means program.

Just as the case in Section 4, the reason for this bad scalability is presumed to be the overhead due to the garbage collector. For the next step, additional experiments will be performed to validate this presumption. Specifically, the garbage collection will be optimized and the result will be compared.

5.2 Performance Analysis of Maximal Independent Set with GC-tuning

To estimate the maximum scalability, the maximum possible memory is set for the second execution. The maximum possible memory is set to be same to that of the previous experiment. The result of which is shown in Fig. 8 and Table 2 (Maximum Memory).

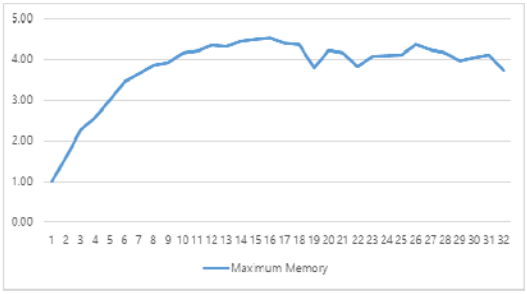

Figure 8: Run-time speedup of Maximal Independent Set with maximum possible memory

Comparing Table 2 (Without Tuning and Maximum Memory) shows the difference of the garbage collection time. This implies that memory tuning can reduce not only the garbage collection time but also the execution time. Fig. 8 shows that the scalability is improved up to fifteen cores, but unexpected fluctuation occurred for the large number of cores more than fifteen. As a final experiment, the execution time with fine tuning of the garbage collection is measured.

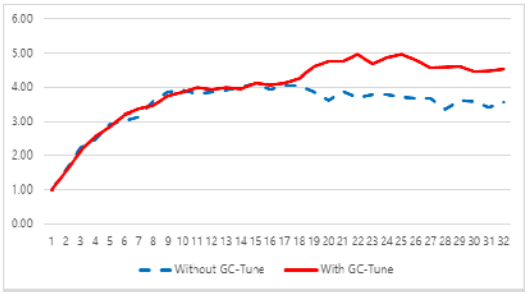

Figure 9: Run-time speedup of Maximal Independent Set without tuning and with GC-tuning. The GC-tuned result shows more speed-up over 18 cores.

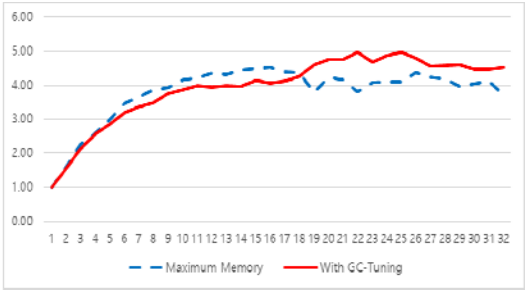

Figure 10: Run-time speedup of Maximal Independent Set with maximum possible memory and with GC-tuning. When the number of cores is more than 18, the

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

GC-tuned result shows more speed-up than the maximum possible memory case.

The third column of Table 2 (With GC-Tuning) shows the average execution time of ten executions with the best heap size calculated by GC-Tune. This result is compared with the previous results of Table 2 (Without Tuning and Maximum Memory). As shown Fig. 9 and 10, the fluctuation is also disappeared without losing the scalability.

The scalability of the speedup of the GC-tuned executions is improved by 21% compared to that of those without GC-tuning as shown in Fig. 9. The range of fluctuations of the GC-tuned executions is compared with that of executions with the maximum possible heap memory (Fig. 10) resulting that the fluctuation range is reduced by 45%.

6. PLAGIARISM DETECTION ANALYSIS

6.1 Performance analysis of Plagiarism Detection without GC-tuning

The program for the third experiment is a parallel plagiarism detection program in Haskell, which was created using DNA-based testing techniques used in SoVAC (software verification and analysis center). SoVAC treats a sequence of special tokens as the program DNA. The program DNA is generated during program parsing, which is used to determine whether or not it is plagiarized [20, 21, 22, 23].

The target programs used for the plagiarism test are 82 Java codes submitted for an assignment in the Object-Oriented Programming class held in 2012 at Pusan National University. The experimental environment is the same as test of K-means. The result of the first execution without GC-tuning is shown in Fig. 11 and Table 3 (Without Tuning).

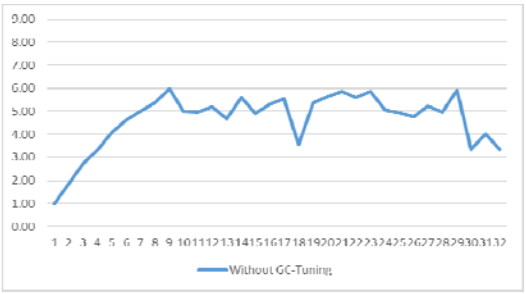

Figure 11: Run-time speedup of Plagiarism Detection without tuning. The speed-up graph is most unpredictable from the three test programs.

As shown in Fig. 11, the speedup of the parallel program is not observable for the cores more than nine, which implies that there is no scalability when the number of cores is above nine. As shown in Table. 3 (Without GC-Tuning), the amount of time spent in garbage collection increases with the number of cores. Also, the performance fluctuation is large. The reason for this bad scalability is presumed to be same as the previous experiments.

6.2 Performance Analysis of Plagiarism Detection with GC-tuning

To estimate the maximum scalability, the maximum possible memory is set for the second execution. The maximum possible memory is set to be same to that of the first experiment. The result of which is shown in Fig. 12 and Table 3 (Maximum Memory).

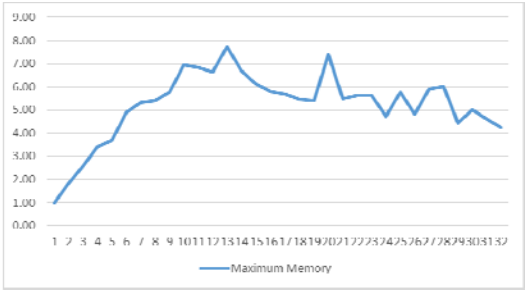

Figure 12: Run-time speedup of Plagiarism Detection with maximum possible memory. This shows the similar result to non-tuned case, though the speed-up gets much better than that.

Comparing the columns of Table 3 (Without Tuning and Maximum Memory), garbage collection time and runtime have been reduced significantly. Fig. 12 shows that the scalability is improved up to thirteen cores, but unexpected fluctuation occurred for the large number of cores more than thirteen. As a final experiment, the execution time with fine tuning of the garbage collection is measured.

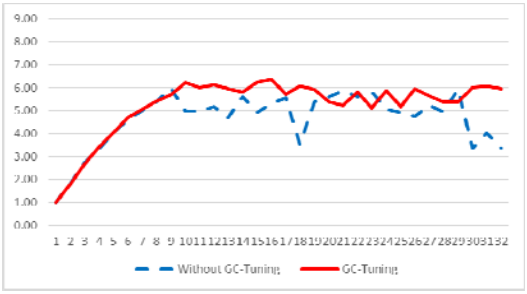

Figure 13: Run-time speedup of Plagiarism Detection without tuning and with GC-tuning. The fluctuation range of GC-tuned graph is smoother than that of non-

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

tuned one, though the fluctuation still occurs when the number of cores is over ten.

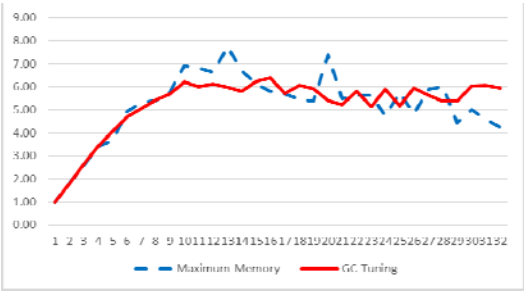

Figure 14: Run-time speedup of Plagiarism Detection with maximum possible memory and with GC-tuning. The fluctuation range of GC-tuned graph is smoother than that of maximum possible memory.

The third column of Table 3 (With GC-Tuning) shows the average execution time of ten executions with the best heap size calculated by GC-Tune. This result is compared with the previous results of Table 3 (Without Tuning and Maximum Memory). As shown Fig. 13 and 14, the fluctuation is disappeared without losing the scalability.

The scalability of the speedup of the GC-tuned executions is improved by 7% compared with that of those without GC-tuning as shown in Fig. 13. The range of fluctuation of the GC-tuned executions is compared with that of executions with the maximum possible heap memory (Fig. 14) resulting that the range is reduced by 58%.

According to the results from Sections 4, 5, and 6, it was confirmed that the cause the bad scalability is the RTS, specifically the garbage collector. Thus, using GC-Tune to resize the heap section is revealed to be an effective way to improve the performance of parallel Haskell programs.

7. DISCUSSION

The experimental results in Sections 4, 5, and 6 show 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.

The improvement of the scalability depends of the characteristics of parallel programs. Using GC-tune, we found that all the parallel programs used in the experiments (K-means, maximal independent set, and plagiarism detection programs) showed some improvements in scalability. However, the improvement ratios are different depending on the parallel programs. Specifically, the plagiarism detection program has a relatively low scalability of 7% compared to other parallel programs.

Also, the experimental results show that performance is not proportional to the size of the heap memory available at the start of the program execution. The performance of setting the maximum heap memory was often not as good as the result of GC-tuning. Therefore, if the programmer wants to increase the parallel program's scalability, one needs to set the memory optimized for the core.

Tuning the sizes of heap sections is one way to improve the performance of parallel Haskell programs, but the RTS itself eventually should be reorganized 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.

8. 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. Haskell's garbage collector, a generational one, seems to cause a lot of overhead especially when the program is executed in parallel, which incurs unstable behavior in the scalability. With this assumption, this paper presents several experiments on tuning the garbage collection for the executions of three parallel programs (the K-means, the maximal independent set, and the plagiarism detection programs), resulting the improvements of scalabilities by 38%, 21%, and 7%, respectively. The fluctuation ranges of speed-ups are also observed to be narrowed down by 45%, 30% and 58% compare to the executions without tuning.

This result indicates that the run-time system of Haskell should be adjusted to take the full advantage of the massively parallel many-core systems. Before the adjustment, tuning the run-time system including the garbage collection can be used and is actually found much effective to promote the scalability of parallel Haskell programs. The results of these experiments also implies that the GC-tuning can be much helpful for estimating the response time of parallel programs.

The contributions of this paper can be summarized as follows:

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

- This paper showed that the scalability problem of parallel Haskell programs can be caused by the generational garbage collector.
- This paper experimentally showed that the scalability of parallel programs can be much improved by GC-tuning.

9. FUTURE WORK

Though this paper suggests that the scalability can be improved by GC-tuning, it does not solved the scalability problem completely. Since the garbage collector itself is not actually adapted for parallel processing, the overall run-time system should be modified to support the parallel executions of programs.

Supporting modularized garbage collector can be one of possible approach. Since the centralized control can be a source of the unpredictability, the distributed modular approach can be adopted by the run-time system especially for the garbage collector. Modifying the run-time system for many-core architectures is an important future work.

ACKNOWLEDGEMENT

This work was supported by Institute for Information & communications Technology Promotion (IITP) grant funded by the Korea government (MSIT) (No.B0101-17-0644, Research on High Performance and Scalable Manycore Operating System).

REFRENCES:

[1] Y. Kim, 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] G. Hutton, *Programming in Haskell*, Cambridge University Press, 2007.

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

[6] S. L. Peyton Jones, Wadler. P, "Imperative functional programming," *Proceedings of the 20th ACM SIGPLAN SIGACT symposium on Principles of programming languages*. ACM, 1993.

[7] J. Armstrong, et al, *Concurrent programming in ERLANG*, 1993.

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

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

[10] 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.

[11] P. M. Sansom, 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.

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

[13] 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.

[14] J. C. Murphy, B. Shivkumar, and L. Ziarek, "Real-time capabilities in functional languages," *Declarative Cyber-Physical Systems (DCPS), CPSWeek Workshop on. IEEE*, 2016.

[15] L. Gidra, G. Thomas, J. Sopena and M. Shapiro, "Assessing the scalability of garbage collectors on many cores," *Proceedings of the 6th Workshop on Programming Languages and Operating Systems. ACM*, 2011.

[16] K. Krishna, M. N. Murty, "Genetic K-means algorithm," *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)*, 29(3), pp.433-439. 1999.

[17] F, Gavril, "Algorithms for minimum coloring, maximum clique, minimum covering by cliques, and maximum independent set of a chordal graph," *SIAM Journal on Computing*, pp.180-187, 1972.

[18] M. Luby, "A simple parallel algorithm for the maximal independent set problem," *SIAM Journal on Computing*, pp.1036-1053, 1986.

[19] N. Alon, L. Babai, and A. Itai, "A fast and simple randomized parallel algorithm for the max-

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

imal independent set problem," *Journal of algorithms*, pp.567-583, 1986

[20] 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.

[21] J. Ji, G. Woo and H. Cho, "A source code linearization technique for detecting plagiarized programs," *ACM SIGCSE BULLETIN*, pp.73-77, 2007.

[22] Y. Kim, Y. Lee and G. Woo, "A Method for Detecting Program Plagiarism Comparing Class Structure Graphs," *JOURNAL OF THE KOREA CONTENTS ASSOCIATION*, 13(11), pp.37-47. 2013.

[23] J. Ji, G. Woo and H. Cho, "An Adaptive Algorithm for Plagiarism Detection in a Controlled Program Source Set," *Journal of KISS: Software and Applications*, 33(12), pp.1090-1102. 2006.

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

Table 1: Run-time and speedup of K-means

CORE	Without Tuning			Maximum Memory			With GC-Tuning		
	GC	Runtime	Speedup	GC	Runtime	Speedup	GC	Runtime	Speedup
1	1.67	45.42	1.00	1.56	56.89	1.00	1.58	56.65	1.00
2	2.70	28.96	1.57	2.30	41.29	1.38	2.32	42.64	1.33
3	2.26	22.81	1.99	2.46	31.29	1.82	2.50	30.58	1.85
					...				
10	2.17	17.51	2.59	2.47	14.74	3.86	3.56	17.41	3.25
11	3.67	18.55	2.45	3.33	15.27	3.73	3.91	16.70	3.39
12	5.06	18.72	2.43	4.06	15.39	3.70	3.91	16.35	3.46
13	5.24	16.32	2.78	5.80	17.22	3.30	6.86	16.12	3.51
14	4.76	17.58	2.58	4.09	15.51	3.67	4.01	15.74	3.60
					...				
30	4.89	23.04	1.97	5.39	15.61	3.64	5.70	16.75	3.38
31	5.79	25.37	1.79	6.46	17.17	3.31	5.85	15.45	3.67
32	4.45	22.17	2.05	4.75	14.60	3.90	5.87	15.88	3.57

Table 2: Run-time and speedup of Maximal Independent Set

CORE	Without Tuning			Maximum Memory			With GC-Tuning		
	GC	Runtime	Speedup	GC	Runtime	Speedup	GC	Runtime	Speedup
1	20.02	94.01	1.00	12.71	102.46	1.00	12.95	102.86	1.00
2	17.86	59.76	1.57	10.66	64.51	1.59	11.66	66.69	1.54
3	12.54	42.25	2.22	9.58	45.17	2.27	10.25	48.42	2.12
					...				
10	10.06	23.97	3.92	8.36	24.66	4.15	9.03	26.63	3.86
11	11.01	24.74	3.80	8.75	24.35	4.21	9.26	25.81	3.98
12	11.05	24.33	3.86	8.69	23.59	4.34	9.36	26.10	3.93
13	11.24	24.04	3.91	8.75	23.73	4.32	9.60	25.79	3.99
14	10.97	23.41	4.02	8.84	23.04	4.45	9.81	25.99	3.96
					...				
30	14.79	26.24	3.58	12.55	25.36	4.04	10.7	22.98	4.45
31	15.81	27.56	3.41	12.33	24.94	4.11	11.63	22.94	4.46
32	15.14	26.36	3.57	15.05	27.54	3.72	11.95	22.65	4.52

Table 3: Run-time speedup of Plagiarism Detection

CORE	Without Tuning			Maximum Memory			With GC-Tuning		
	GC	Runtime	Speedup	GC	Runtime	Speedup	GC	Runtime	Speedup
1	1.23	61.92	1.00	1.22	61.85	1.00	1.34	62.08	1.00
2	2.25	33.75	1.83	2.35	33.85	1.83	2.52	34.09	1.82
3	1.95	22.76	2.72	2.00	23.80	2.60	2.12	22.97	2.70
					...				
10	2.44	12.42	4.99	1.26	8.91	6.94	0.84	9.03	6.88
11	2.40	12.43	4.98	1.30	9.06	6.83	1.49	10.28	6.04
12	1.31	11.93	5.19	1.35	8.32	7.43	1.59	10.03	6.19
13	3.27	13.20	4.69	1.36	8.00	7.73	1.62	9.10	6.82
14	2.43	11.04	5.61	1.30	9.23	6.70	1.58	10.48	5.92
					...				
30	7.23	18.41	3.36	1.93	12.38	5.00	1.80	10.26	6.05
31	6.61	15.38	4.03	3.93	13.44	4.60	1.74	10.28	6.04
32	8.83	18.46	3.35	4.79	14.55	4.25	0.94	9.85	6.30