

Analysis of the Parallel Programming Models in Haskell for Many-Core Systems

Xiao Liu¹, Yeoneo Kim¹, Junseok Cheon¹, Sugwoo Byun², Gyun Woo³

^{1,3}Department of Electrical and Computer Engineering, Pusan National University, Busan, Korea

²Department of Computer Engineering, Kyungsung University, Busan, Korea

³Smart Control Center of LG Electronics, Pusan National University, Busan, Korea
{liuxiao, yeoneo, jscheon, woogyun}@pusan.ac.kr, ²swbyun@ks.ac.kr

Abstract

The Moore's law has reached its limitation since the integration and economic issues of CPU. Therefore, the trend of chip design is moving to the increase of the number of cores rather than stressing the density of circuits. Consequently, the parallel programming is attracting interests. In this trend, functional languages are getting popular for parallel programming since they have inherent parallelism. This paper aims to compare and analyze two Haskell programming models for many-core environment. We developed applications based on a Haskell parallel programming model named Eval monad and Cloud Haskell, respectively to compare the performance of them. We test the application on both 32 cores and 120 cores CPU. The experimental result shows that on 32 cores, the performances are similar, but on the 120 cores, Cloud Haskell performs 32% faster on run-time, and 123% better on scalability. This result implies that Cloud Haskell is more appropriate for a large number of cores than Eval monad, and the latter is more suitable for simple parallelism involving just tens of cores.

Key words: Parallel Programming, Eval Monad, Cloud Haskell, Many-core

Introduction

The development of CPU has changed since the Moore's law reached the limitation that caused by the integration and economic issues. Hence, the trend has changed from improving the performance of single core to increasing the number of cores of CPU. Such change also ended the age of improving software performance by improving the performance of CPU. Consequently, it is necessary to use parallel programming method for using the performance of computers more efficiently. On the other hand, since the imperative programming is way too complicated on parallel programming, recently the functional programming languages are attracting interests.

Haskell as a pure functional programming language is known for being advantageous on parallel programming. Since Haskell has no side effect, the computation sequence is not specified which is efficient for parallel programming. Moreover, compares with Erlang that claims good performance on parallel programming, Haskell provides the lightweight thread that shows quite a little overhead for context switching which is more appropriate for parallel programming [1]. Besides, Haskell also supports lots of parallel programming models those can be used for different situations.

This paper aims to verify whether Haskell can still perform efficient performance in manycore environment. For the experiment, we developed applications based on two parallel programming models. The applications we developed are a K-means program and a plagiarism detection program, both of which are requiring a large amount of data, and they are appropriate for performance evaluation. After executing the applications on both 32 cores and 120 cores environments, we compare the performance and analyze the features of Haskell in manycore environment.

This paper is organized as follows. Section 2 as related works presents the features of using Haskell for parallel programming. Section 3 introduces how to develop programs based on Eval monad and Cloud Haskell, respectively. Section 4 analyzes the experimental result of the performance of Haskell in the manycore environment. Section 5 discusses the unusual points occurred in the experiment. Section 6 summarizes future work and concludes.

Related Work

As a pure functional programming language, Haskell based on a mechanism named lazy evaluation. The purity of Haskell guarantee it has no side effect hence the functions can be independently executed even run them in a same time. Therefore, such purity makes Haskell naturally own parallelism and advantageous for parallel programming. Such purity also causes a non-specified computation sequence of Haskell functions. Except for monad, the pure function and monad type are separated in Haskell. In other words, the pure functions in Haskell don't consider the control flow which can be considered as parallel computing, and monad can implement the sequential programming as well.

Moreover, since Haskell uses lightweight thread that is more advantageous for parallelism. Haskell provides thread at different levels for corresponding uses. The parallelism of Haskell is demonstrated in Fig. 1 [2].

As shown in Fig. 1, the spark will be generated when a parallel request happened by using `par`. The spark will be transformed to one of green threads named Haskell thread if a real computation applied on it. Since the thread in Haskell is more lightweight than other threads in other languages, it has the feature of less overhead occurrence than others when switching context. The parallelism of Haskell is implemented by passing the Haskell threads to real OS threads. This approach of parallelism in Haskell is caused by lazy evaluation. Such feature can reduce unnecessary computations and only the necessary computations are handled. Hence, Haskell uses concept such as spark to avoid passing unnecessary

the performances of them. There are 2 applications used in the experiment which are plagiarism detection program and K-means program, respectively. The K-means program of Eval monad version is referred from the Simon Marlow's book [3], and its Cloud Haskell version is modified based on it to fit the Cloud Haskell environment. The plagiarism detection is developed based on the work proposed by Ji [7]. All the developed applications are executed on both of 32 cores and 120 cores many-core environments, respectively.

A. The 32 Cores Environment

We execute K-means program and plagiarism detection program on 32 cores environment and measure their performances in this subsection. The experiment consists of follows. The CPU is 2 AMD Opteron 6272, RAM space is 96GB, and we use 256GB SSD as disk, the operating system we use is Ubuntu 14.04.1 LTS. The input for K-means is composed by 0.32 million random points. And the input for plagiarism detector is 108 pieces of Java source code collected from undergraduate students in their course. The execution time and scalability of the 2 programs on 32 cores are demonstrated in Fig. 2 and Fig. 3, respectively.

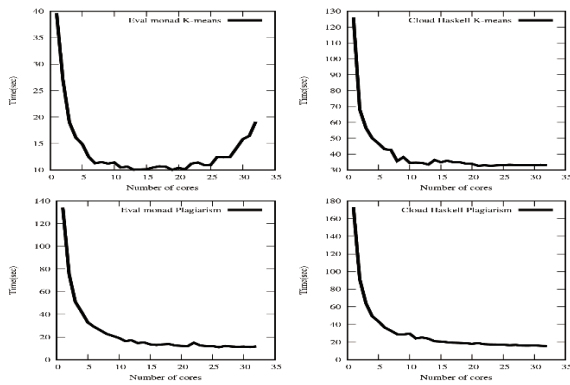


Fig. 2 The execution time of the 2 applications in 32 cores with 2 different versions. The applications are Eval monad K-means, Eval monad plagiarism detector, Cloud Haskell K-means, and Cloud Haskell plagiarism detector.

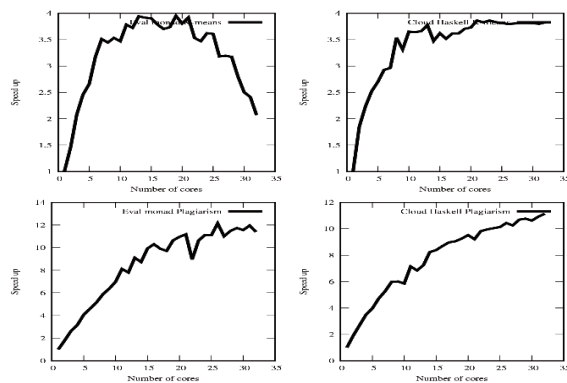


Fig. 3 The scalability of the 2 applications in 32 cores with 2 different versions. The applications are Eval monad K-means, Eval monad plagiarism detector, Cloud Haskell K-means, and Cloud Haskell plagiarism detector.

According to the experiment result in Fig. 2, in the case of less cores the Eval monad performs shorter execution time. However, with the increasing of cores, the difference of

execution time between two applications is reducing. On the other hand, the scalability of the applications based on Eval monad resents significant changes in Fig. 3. The scalability turns to decrease with the growth of the number of cores in the case of executing K-means program.

B. The 120 Cores Environment

We execute K-means program and plagiarism detection program on 120 cores environment and measure their performances in this subsection. The experiment consists of follows. The CPU is 8 Intel Xeon E78870 v2, RAM space is 792GB, and we use 1TB SSD as disk, the operating system we use is Ubuntu 15.04. Since the hardware for 120 cores is better than the case of 32 cores, we also increase the size of input data for both applications. We use 15 millions of random points as input for K-means, and use 250 pieces of Java source code as input for plagiarism detector. The execution time and scalability of the applications are illustrated in Fig. 4 and Fig. 5, respectively.

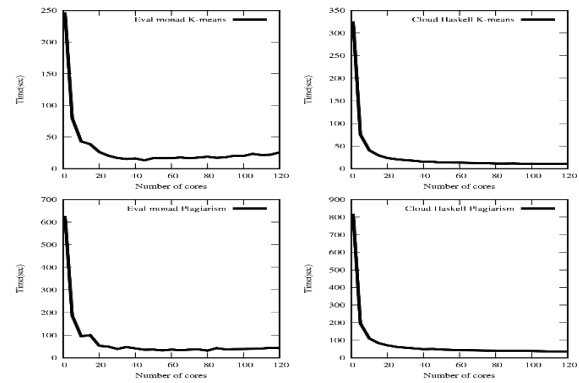


Fig. 4 The execution time of the 2 applications in 120 cores with 2 different versions. The applications are Eval monad K-means, Eval monad plagiarism detector, Cloud Haskell K-means, and Cloud Haskell plagiarism detector.

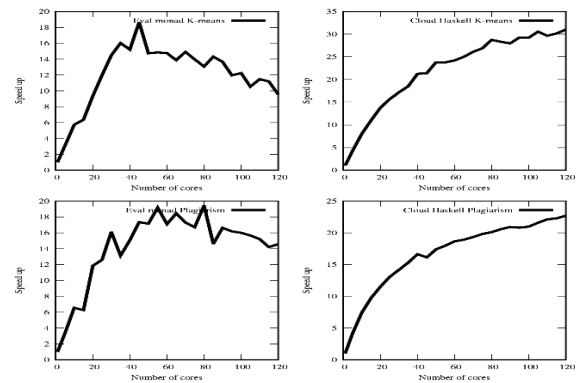


Fig. 5 The scalability of the 2 applications in 120 cores with 2 different versions. The applications are Eval monad K-means, Eval monad plagiarism detector, Cloud Haskell K-means, and Cloud Haskell plagiarism detector.

According to the experiment result in Fig. 4, similar with the case of 32 cores, the execution time of Eval monad is shorter than Cloud Haskell at the first place. However, with the growth of the number of cores, Cloud Haskell is getting faster than

Eval monad. According to the experiment result in Fig. 5, the scalability of Cloud Haskell increases with the growth of the number of cores regularly. However, the scalability of Eval monad changes significantly. The scalability even turns to decrease especially when the number of cores is reaching to 100.

D. Analysis of Experiment Result

After analyzing the experiment result, we noticed that the Eval monad shows better performance in both 32 cores and 120 cores environments when the number of cores is small. However, with the growth of the number of cores, the difference between Eval monad and Cloud Haskell turns to small, the performance of Eval monad even turns to decrease in 120 cores environment on the contrary. Comprehensively, in the environment of 120 cores, the execution time of Cloud Haskell is 32% faster than Eval monad, and the scalability is also 123% better than Eval monad. Moreover, Cloud Haskell performs a growing scalability with the growth of the number of cores, but Eval monad seems like not so much stable.

According to the experiment result, we noticed that it is important to choose appropriate parallel models for different programs for efficiently using the performance of many-core system. It will be more appropriate for using Cloud Haskell in many-core environment if the program is one piece of large scale program. However, it will be more appropriate for using Eval monad in many-core environment if there are several small pieces of programs.

Discussion

This section discusses the issue of the decreasing of Eval monad in many-core environment. To figure out the issue we use profiling option as an input during the Eval monad program execution. The profiling results shows that the GC time of Haskell VM causes such issue. It is necessary to not only reduce the execution time but also reduce the GC time for efficiently improving the performance of VM based parallel programming. We found out that the GC mechanism in Haskell spends similar time during a certain increment of cores, and once the cores are more than a specified number, the GC time increases as well. Although there is a work on solving such problem [8], it still in research progress.

Considering such issue on VM of Haskell, we stand for Cloud Haskell performs better performance than Eval monad. Different with Eval monad that uses only one VM to execute program, Cloud Haskell provides an architecture that runs on several VMs and implements message passing. Consequently, it is more flexible than Eval monad for handling GC problem since GC and data are separated in Cloud Haskell. We confirm Cloud Haskell is more appropriate for many-core environment since it runs at a distributed the Haskell VMs.

Conclusion

In this paper, we present how does Haskell that is known for good at parallel programming execute in the manycore environment. We developed applications based on Eval monad which is a Haskell parallel programming model and Cloud Haskell, respectively. We execute the applications in two different manycore environments. The experimental result shows that Cloud Haskell is 34% faster at execution time and

123% better at scalability in the case of 120 cores. Analyzing the experimental result, we conclude that Cloud Haskell is more efficient than Eval monad when the system resources are fully utilized. By the way, the Eval monad is more appropriate to ensure the parallelism requiring only a few cores for small programs.

In the future, we are planning to solve the GC issue of Haskell that mentioned in discussion. Without this solution, the GC time will be prolonged with the increment of cores. We are going to work on analyzing the RTS (run time system) of GHC to solve this issue.

Acknowledgement

This work was supported by Institute for Information & communications Technology Promotion(IITP) grant funded by the Korea government(MSIT)(No. 2014-0-00035, Research on High Performance and Scalable Manycore Operating System)

References

- [1] S. Marlow, S.P. Jones, and S. Singh, "Runtime support for multicore Haskell," ACM SIGPLAN Notices, Vol. 44. No. 9. pp. 65-78, 2009.
- [2] Nikita Frolov, "GHC heap internals," http://www.cse.chalmers.se/edu/year/2016/course/course/DAT280_Parallel_Functional_Programming/lectures/Frolov14.pdf, 2016.
- [3] S. Mallow, *Parallel and concurrent programming in Haskell: Techniques for multicore and multithreaded programming*, O'Reilly Media, Inc., 2013.
- [4] C. Hewitt, et al., "A universal modular actor formalism for artificial intelligence," In Proceedings of the 3rd international joint conference on Artificial intelligence, pp. 235-245, 1973.
- [5] J. Epstein, et al., "Towards Haskell in the cloud," ACM SIGPLAN Notices. Vol. 46. No. 12. pp. 118-129, 2011.
- [6] J. Armstrong, et al., *Concurrent programming in ERLANG*, Prentice Hall, second edition, 1996.
- [7] J. Ji, et al., "A source code linearization technique for detecting plagiarized programs," ACM SIGCSE Bulletin, Vol. 39, No. 3, pp.73-77, 2007.
- [8] S. Marlow and S. P. Jones, "Multicore garbage collection with local heaps," ACM SIGPLAN Notices, Vol. 46, No. 11, pp.21-32, 2011.