Proceedings of IEEE International Conference on Applied System Innovation 2018
IEEE ICASI 2018- Meen, Prior & Lam (Eds)

SAM: A Haskell Parallel Programming Model for Many-Core Systems

Yeoneo Kim[1], Junseok Cheon[1], Xiao Liu[1], Sugwoo Byun[2], Gyun Woo[3]

[1,3]Department of Electrical and Computer Engineering, Pusan National University, Busan, Korea
[2]Department of Computer Engineering, Kyungsung University, Busan, Korea
[3]Smart Control Center of LG Electronics, Pusan National University, Busan, Korea
{yeoneo, jscheon, liuxiao, woogyun}@pusan.ac.kr, [2]swbyun@ks.ac.kr

Abstract

Since the multicore CPU has been released, the parallel programming method has become a significant issue to programmers. For parallel programming, functional languages such as Haskell are especially attractive since they reduce the data dependency hindering parallelism. One of the popular parallel programming models of Haskell is Cloud Haskell. Though it is an effective model for the manycore environments, it is quite difficult to use compared with other models. This paper proposes an efficient Haskell programming model for the manycore system named SAM. SAM takes advantages of the outstanding performance of Cloud Haskell and overcomes its disadvantages. To present the performance of SAM, we developed four different applications for the experiment. According to the experimental result, the compilation time of using SAM increased 5%, and execution time reduced 0.2%, respectively, compared with Cloud Haskell. In addition, the code size of using SAM reduced to 86% on the parallel part.

Key words: Actor Model, Manycore, Haskell, Cloud Haskell

Introduction

With the popularization of multicore CPU, it is difficult to improve the program performance using the similar method adopted for the single core. To solve such problems, the parallel programming technique is attracting interest recently. Further, the multi-core environment is rapidly replaced by manycore since the number of cores is the crucial factor for the performance. However, the conventional imperative programming languages such as C and Java have some limitations on parallel programming. In this situation, the functional programming languages become popular as viable alternatives for those languages since they reduce the data dependency hindering parallelism.

Haskell, one of the popular functional programming languages, is wildly recognized as a suitable language for parallel programming. Since Haskell provides various models for parallel programming, we can choose the appropriate one for different situations. Cloud Haskell, a programming model based on actor model, is used in the distributed environment. However, it also shows a scalable performance in the manycore environment based on the shared memory. The reason for this scalability is because the VMs (Virtual Machines) of Haskell in Cloud Haskell are distributed, resulting the GC (Garbage Collection) overhead is also distributed [1].

However, comparing with other parallel models of Haskell, Cloud Haskell has several inconveniences when using it. The code size of the parallel code in Cloud Haskell is getting much larger than that in Eval monad, which is relatively common in the parallel programming. Also, using Cloud Haskell is way too complicated not only in a remote environment but also in local which is not efficient.

This paper proposes a model named SAM (Simple Actor Model) to solve such inconveniences and limitations of Cloud Haskell. SAM tackles this problem assuming the shared memory environment for the manycore system. With this assumption, much of the bookkeeping code can be reduced or even eliminated in SAM. Further, using Template Haskell guarantees the same performance of the SAM code to that of Cloud Haskell counterpart even for tiny-sized code. In this paper, we use to simplify the complicated progress of using Cloud Haskell. SAM can easily implement parallel programming in manycore environment and guarantee scalability.

The rest of this paper is organized as follows. Section 2 introduces the related work such as Cloud Haskell and Template Haskell. Section 3 shows the problems of using Cloud Haskell in manycore environment. Section 4 presents SAM and explains how it can solve such problems. Section 5 verifies the performance of SAM by comparing it with Cloud Haskell. The limitations of the current version of SAM is discussed in Section 6. Section 7 presents future work and concludes.

Related Work

A. Cloud Haskell

Cloud Haskell is developed based on the actor model of Erlang for the Haskell platform of cloud computing service provided by Well-typed [2,3]. Actor model allows the actors communicate with each other through sending messages with their own memory space. Since there is no shared memory among actors, interaction only through direct asynchronous message passing with no restriction on message arrival order [4]. This feature is utilized by Cloud Haskell to guarantee scalability. In addition, Haskell provides more lightweight thread than Erlang which is more efficient for performance.

Cloud Haskell is separated as two parts: the frontend based on Process monad for execution in distributed environment, and the backend for network connection. Especially, the backend supports 3 modes for different network types: simplelocalnet mode can be used when it runs in same network, p2p mode can be used in shared network, and if it is used in VMs such as Microsoft Azure, there is also an Azure library is

ISBN 978-1-5386-4342-6

supported. Due to the manycore environment is based on shared memory, the simplelocalnet mode for same network can be used for programming. In this case, a master/slave approach is used for program development.

B. Template Haskell

Template Haskell is a Haskell meta programming tool that can keep type safety [5]. It is used for generating large size code and developing EDSLs (Embedded Domain Specific Languages) [6].

An understanding on Quasi-quote is necessary since it is a significant grammar in Template Haskell. Quasi-quote allows syntax like "`[|…|]`" which "…" can be replaced by Haskell or Expression syntax. Quasi-quote uses `Q Exp` type which means in this step it is not an expression yet. It is necessary to use "`$`" operator to transform Quasi-quote to expression. "`$`" operator can transform the `Q Exp` typed Quasi-quote into real expression in compile time.

We can use Quasi-quote to make variable function liked functions that decide the type in compile time. With some modification, it is also available to develop variable arguments. This paper attempts to reduce the unnecessary Haskell code in Cloud Haskell by using the features of Template Haskell.

Limitation of Cloud Haskell in Manycore System

It is appropriate to use distributed architecture such as Cloud Haskell in manycore environment since it can avoid GC problem. However, there are 3 issues on using Cloud Haskell in shared memory based manycore environment.

The first issue is it is mandatory to handle all the detailed actions for every node in the master/slave structure when using Cloud Haskell. This is because Cloud Haskell focuses on concurrency rather than parallelism. This feature can be advantageous when the structure of a parallel program is complicated. However, since most of parallel programs are not so complicated, even some well written programs are separated as several small size parts, on the contrary, it can be disadvantageous. Also, this issue of using Cloud Haskell can cause an increment of code size rather than using Eval monad.

The second issue is related to the `startMaster` function which is used for activating the master node in Cloud Haskell. Since the `startMaster` function takes backend with master node information and executing function as input, and returns `IO ()` typed value at last, it is difficult when trying to use the returned value in the middle of the execution in Cloud Haskell.

The third issue is the executions are separated for master node and slave node in Cloud Haskell. This is because Cloud Haskell based on a distributed memory structure and before executing any program the specified number of slave node programs and master node should be executed respectively. This issue also causes the mandatory execution of slave node even in a local environment.

Moreover, there are other inconveniences using Cloud Haskell even in a local node. For example, it is necessary to find out all the network port numbers for every executing programs. The reason is the communication among processes is occurred through network sockets even in a local node. Hence, it is inconvenient since the sockets can't be overlapped, it is necessary to manually find idle port and execute it when every time run programs in Cloud Haskell.

Design of SAM

This section introduces how the SAM be designs to solve the problems in Cloud Haskell. We also present how to use SAM. The architecture of SAM is illustrated as Figure 1.

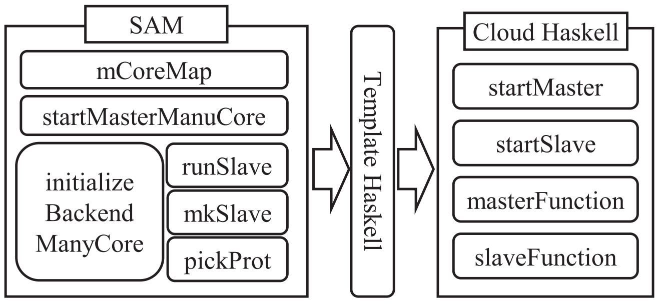

Fig. 1 Design of SAM. The main idea of SAM. The programs developed based on SAM can be transformed to Cloud Haskell programs by using Template Haskell.

SAM provides map styled programming feature to solve the problems in Cloud Haskell. The functions in Figure 1 are used to eventually transform Template Haskell to Cloud Haskell.

A. Design of SAM

The most important function for programmers in SAM is `mCoreMap`. Other functions in SAM are built for internal executions therefore programmers can easily use SAM by only understanding how to use `mCoreMap` function. This function is defined as follows.

```
mCoreMap :: Lift a => Name -> Int -> [a] -> ExpQ
```

`mCoreMap` takes 3 arguments as input. The first argument is the function will be applied on the input data. This argument is similar with the first argument in map function which is defined as `f :: a-> b`. The difference between `mCoreMap` and map is the former only is able to take functions transformed by `mkSlave` function. `mkSlave` is a function to transform the functions which only take 1 argument for SAM. The disadvantage of `mkSlave` is it only can take the functions that defined in top-level. The second argument is the number of parallel processes will be executed. This argument specifies how many processes are used for the corresponding function as a number, the performance can be changed by changing this number. The last argument is the input data for the function in first argument. This argument defined as `Lift` type which is a feature of Template Haskell. The actual input data will be changed as `Serializable` type. The first issue in Cloud Haskell can be solved by using `mCoreMap`.

The type of an internal function in SAM named `initializeBackendManyCore` is defined as follows.

```
initializeBackendManyCore :: HostName -> ServiceName ->
                                 RemoteTable -> IO Backend
```

ISBN 978-1-5386-4342-6

The `initializeBackendManyCore` function is developed by partially modifying a function supported in Cloud Haskell named `initializeBacked`. Although `initializeBackendManCore` is basically same with `initializeBackend`, it is different when handling a failed socket initialization event during the function execution. The original function only prints error and ends the program. However, the modified one can finds other idle socket (port) and tries to regenerate Backend. SAM provides a function named `pickPort` for finding idle port. The `pickPort` function is developed based on Linux that can find and return the unused ports, hence it can be used not only in `innitializeBackendManyCore` function but also in functions such as `mCoreMap` and `runSlave`. The `initializeBackendManyCore` function is different with the existed function and related to `startMasterManyCore` function which is defined as follows.

```
startMasterMnayCore :: Serializable a => Backend -> Int ->
                       ([NodeId] -> MVar a -> Process()) ->
                       IO (a)
```

The `startMasterManyCore` function is modified based on existing `startMaster` for SAM. The existing function returns `IO ()` which is difficult to find out the middle result. However, the `startMasterManyCore` in SAM uses `MVar` that is available to find out the data which is executed in master node. Also, this function takes the number of slaves as input and executes the slaves in the function. To implement this feature, `startMasterManyCore` uses the `forkProcess` function that is defined in unix library [6]. Therefore, the second and third issues can be solved by using `startMasterManyCore`.

C. Use of SAM

This subsection introduces how to use SAM. The steps of using SAM are same as follows.

1. Importing ManyCores package.
2. Defining slave execution function by "`(mkSlave '{function name}) :: ProcessId -> Process ()`".
3. Registering the function defined at 2 in the remote table (`remotable['{function name}]`)
4. Executing function as `$(mCoreMap '{function name}) {number of slave} {input data}`

The programs based on same import `ManyCores` packages at first. The step 2 is for defining the function executed in slave node. It is necessary to register the function defined at step 2 in a remote table. After previous 3 steps, it is available to call `mCoreMap` function as step 4. A SAM program uses the 4 steps is demonstrated as Figure 2.

The Figure 2 is a simple program to add 1 for every element in a list which contains 1 to 100. The code before 11th line will not change whatever the parallel model is used. The code from 19th line is for using SAM. The `slaveJob` function and `remotable` function are for running program in slave nodes. The `main` function can execute the program as map type.

```
 1:  {-# LANGUAGE TemplateHaskell #-}
 2:  import ManyCores
 3:
 4:  input = [1 .. 100] :: [Int]
 5:
 6:  incr :: Int -> Int
 7:  incr x = x + 1
 8:
 9:  afterFunc :: [Int] -> IO ()
10:  afterFunc xs = print $ sum xs
11:  ----------------------------------
12:  slaveJob :: ProcessId -> Process()
13:  slaveJob = $(mkSlave 'incr)
14:  remotable ['slaveJob]
15:
16:  main = do
17:     ret <- $(mCoreMap 'slaveJob) 80 3 input
18:     afterFunc ret
```

Fig. 2 Example of SAM Program. Add 1 for every element in a list that composed by integers from 1 to 100.

Experiment of SAM

This section measures the performance of applications developed based on SAM. We use 4 kinds of applications through Cloud Haskell and SAM. The applications are prime number counter, word counter, Sudoku, and plagiarism detector, respectively. We referred content in Simon Marlow's book [7] for developing the Sudoku application. This paper is aimed to measure the reduction of code size, compile time, and execution time between Cloud Haskell and SAM.

A. Comparison of Code Size

This subsection measures and verifies the reduction of code size between using Cloud Haskell and SAM. We developed 4 different applications on SAM Cloud Haskell and compared the LoC of them. The result of comparison is showed in Table 1.

TABLE I
Comparison Cloud Haskell and SAM LoC

Category		Cloud Haskell	SAM	Rate
Prime Number	Full code	60	32	50.00
	Parallel portion	38	5	86.84
Word Count	Full code	82	47	45.12
	Parallel portion	49	5	89.80
Sudoku	Full code	140	105	26.43
	Parallel portion	39	5	87.18
Plagiarism Detection	Full code	602	529	12.79
	Parallel portion	51	10	80.39
Average	Full code	221	176	20.48
	Parallel portion	45	9	86.11

Table 1 shows the comparison result of applications developed based on Cloud Haskell and SAM, respectively.

ISBN 978-1-5386-4342-6

Table 1 contains both LoC comparison results of entire code and parallel part. It is more precise for measuring the reduction rate of source code by removing the unchanging code. For the reduction rate of full code, there are around 20% code are reduced. However, the reduction rate of parallel part is 86%.

B. Comparison of Compile time and Running time

This subsection tries to whether there are meaningful changes of compile time and running time by executing the applications in Cloud Haskell and SAM, respectively. We've proved that the LoC is much reduced by using SAM. However, it would be meaningless if the performance is also reduced by using SAM. Therefore, we also test the performance of compile time and running time SAM and compare them with Cloud Haskell. The comparison is demonstrated in Table 2.

Table 2 shows the result that the running time (seconds) is reduced 0.077 second. This can be treated as a tolerable difference since there is a 0.2% environment bias for every execution. And the average compile time is increased 0.369 second that takes 5% longer than Cloud Haskell. This is caused by SAM using Template Haskell to generate code which is ignorable from the entire compile time.

TABLE II
Comparison Cloud Haskell and SAM compile time and running time

Category		Cloud Haskell	SAM	Rate
Prime Number	Compile time	37.664	37.056	-0.61
	Running time	3.540	3.768	0.23
Word Count	Compile time	32.534	32.692	0.50
	Running time	3.693	4.125	0.43
Sudoku	Compile time	24.602	24.624	0.02
	Running time	6.469	6.492	0.02
Plagiarism Detection	Compile time	41.674	41.796	0.12
	Running time	14.046	14.838	0.79
Average	Compile time	34.119	34.042	0.01
	Running time	6.937	7.306	0.37

C. Experiment Analysis

Similar with using Eval monad, the code size is reduced around 86% by using SAM than Cloud Haskell for the 4 applications. Also, there is few difference of compile time and running time between Cloud Haskell and SAM. Moreover, since it is unnecessary to run slave node separately, it is convenient to use SAM in manycore environment.

Discussion

We discuss the limitations of SAM in this section. There are 2 limitations in the current version of SAM. The first limitation is SAM can't use RTS options. The current version of SAM uses `forkProcess` from unix library to execute slaves. Since this function duplicates the status from running process which will cause the problem that the slave node will duplicate the exactly same RTS option from master node. In this situation, the slave nodes will execute the same threads duplicated from master which can cause performance problems. It is necessary to find out new method for executing new thread.

The second limitation is the function and type are limited when passing them to map. Either function or type have to be inherited from Serializable type. This is because SAM eventually transform code to Cloud Haskell which uses such type. It is necessary to modify the backend of Cloud Haskell for solving this limitation. Moreover, since `mCoreMap` cannot pass the lambda function, it is necessary to declare function at top-level.

Conclusion

This paper proposes a parallel programming model named SAM that is appropriate in the manycore environment. SAM solves the limitations of Cloud Haskell and supports map-styled programming. This paper introduces the functions contained in SAM and presents how to use SAM. According to the performance comparison of SAM and Cloud Haskell using four sample applications, SAM reduced by 86% code in size compared with Cloud Haskell without sacrificing the compile and the execution time.

In the future, we will focus on solving the problems we mentioned in discussion which are the difficulty of using RTS option and limitation on function and data type. We are planning to develop alternatives for `forkProcess`. With this modification, the type limitation on functions and data can be overcome by modifying the structure of Cloud Haskell.

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] H. Kim, et al., "An approach to improving the scalability of parallel Haskell programs," *Journal of Theoretical & Applied Information Technology*, Vol. 95, No. 18, pp. 4826-4835, 2017.
[2] J. Epstein, et al., "Towards Haskell in the cloud," *ACM SIGPLAN Notices*, Vol. 46. No. 12. pp. 118-129, 2011.
[3] J. Armstrong, et al., *Concurrent programming in ERLANG*, Prentice Hall, second edition, 1996.
[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] T. Sheard and S. P. Jones, "Template Meta-programming for Haskell," *In Proceedings of the 2002 ACM SIGPLAN workshop on Haskell*, pp. 1-16, 2002.
[6] Haskell, unix: POSIX functionality, https://hackage.haskell.org/package/unix(last visited: 2018.03.02).
[7] S. Mallow, *Parallel and concurrent programming in Haskell: Techniques for multicore and multithreaded programming*, O'Reilly Media, Inc., 2013.

ISBN 978-1-5386-4342-6