SSAM: A Haskell Parallel Programming STM Based Simple Actor Model

Y Kim[1*], J Cheon[1*], T Hur[1*], S Byun[2*] and G Woo[1*,3]

[1]Dep. of Electrical and Computer Engineering, Pusan National University,
[2]Dep. of Computer Science, Kyungsung University
[3]Smart Control Center of LG Electronics

*Email : {yeoneo, jscheon, vhxpffltm, woogyun}@pusan.ac.kr, swbyun@ks.ac.kr

Abstract. As computers with manycore architecture are being widely spread, parallel programming becomes a pending issue. While parallel programming has been a challenging issue, Haskell is known to be of the best available one. The immutability and declarative nature of Haskell suit well the need for parallelism. In our previous work, we implemented the SAM, a parallel programming model based on Haskell. In this paper, we propose the SSAM, an updated version of SAM, by solving the performance overhead of SAM socket. We discuss the architecture of SSAM and its major functions and show the 14% scalability upgrade in a performance test.

1. Introduction

Since the release of AMD Ryzen, the number of CPU cores has been being increased again. Since then, parallel programming becomes an important pending issue. However, the more complex the application, the more difficult parallel programming becomes. But functional programming language does not have a data dependency. So, functional languages are getting attention in parallel programming.

While data parallelism is rather easier to handle, it is well-known that control parallelism is a challenging issue. It is difficult to do parallel-programming with usual imperative programming, which requires reasoning about the dependency of data structure and the control flow. In this matter, the purely functional programming like Haskell is known to be of the best solutions, since it is not needed to do this kind of reasoning. Pure functional programming uses immutable data structures, and it is declarative, which means that users do not describe the computation order. In this sense, pure functional programming is said to have inherent parallelism.

Haskell is considered to be a good parallel programming language. It possesses not only features of pure-functionality, but also the programming environment for parallel computation. Haskell provides the lightweight thread that shows quite a little overhead for context switching which is appropriate for parallel programming [1]. In addition, Haskell provides a variety of parallel programming models, we can choose the appropriate one for different situations.

In our previous work, we implemented the SAM, a parallel programming model based on Haskell. SAM is a suitable parallel programming model in a manycore environment and it can be easily used for parallel programming. But SAM has a problem with low performance due to socket overhead. To solve this problem, we proposed SSAM which is an STM-based SAM, which reduces the communication overhead using STM (software transactional memory).

The rest of this paper is organized as follows. Section 2 introduces the related works such as Haskell parallel programming models. Section 3 introduces the socket overhead problem in SAM. Section 4 introduces the architecture of SSAM to solved socket overhead. Section 5 evaluates the performance of SSAM by comparing it with Eval monad and SAM. And Section 6 we discussion for experimental results. Finally, Section 7 presents future work and concludes.

2. Related work

As a pure functional programming language, Haskell based on a mechanism lazy evaluation. Therefore, users do not describe the evaluation order in Haskell. In case that the user wants to do specify the evaluation order, they should use monads. In other words, except for monads, functions can be executed in parallel without considering control flow. Moreover, Haskell uses the lightweight thread that is more advantageous for parallelism.

The approach of using the Eval monad is parallelizing the control flow of the program based on shared memory. Eval monad uses a strategy for determining computation order [3]. The communication among threads in the Eval monad is implemented by a mutable variable named `MVar`. Using the Eval monad, parallelism is applied simply by changing the higher-order function to a parallel higher-order function. But Eval monad has a problem which is a Scalability decrease as the number of cores increases.

The approach of Cloud Haskell is a Haskell platform for cloud computing provided by Well-typed [4]. This model is developed based on the actor model [5] of Erlang. In the actor model, each actor has its own memory space. And each actor communicates via a message. Therefore, the actor model has no shared memory between actors. So, the actor model has a high concurrency. However, Cloud Haskell has a disadvantage that is more difficult than other parallel programming models.

The approach of SAM is a suitable parallel programming model in manycore environments. SAM is a programming model based on Cloud Haskell for easy parallel programming. But SAM has a problem that is small cores have low performance due to socket overhead.

3. Performance Issue of SAM

In this section, we introduce the problems of SAM. SAM has an advantage that high performance in manycore environments. However, for a small core environment, SAM is slower than other parallel programming models [6]. This problem occurred since it uses socket communication in the local node environment. The cause of this problem can be seen in Figure 1.

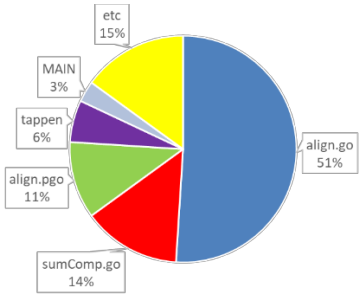

Figure 1. Analysis of the execution time of the source code plagiarism detection program.

Figure 1 analyzes the result of the execution time of the source code plagiarism detection program developed through SAM. The `align` function is an implementation of an alignment algorithm. It consumes more than half of the execution time. Next, the execution time of many functions is `sumComp`. `sumComp` is a function that receives a calculated value from slave nodes. But `sumComp` is not a function with many operations in SAM. Hence, with further analysis of this function, we figure out that this problem occurs from the socket communication function.

4. Design of SSAM

In this section, we introduce our method called SSAM, which is an STM based SAM. STM is a method that uses the concept of transaction used in the database [7]. This method can allow lock-free based programming in the shared memory program. Moreover, Haskell STM is known to be effective because it supports GHC. So, we use an STM instead of socket communication for the improved SAM. Rest part of this section, we introduce the architecture of SSAM and core functions of SSAM.

4.1. Architecture of SSAM

In this chapter, we introduce the architecture of SSAM. SSAM architecture is similar to SAM. SSAM is organized of frontend which is a set of interfaces and backend which is a set of real data communication. And the architecture of SSAM is illustrated in Figure 2.

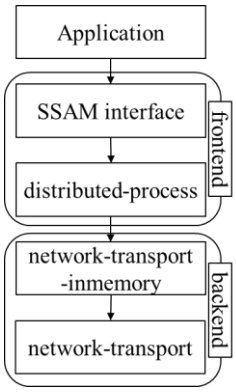

Figure 2. Architecture of SSAM.

Figure 2 shows the architecture of SSAM. An application may use SSAM by using the SSAM interface at the frontend. The SSAM interface provides map-style parallel functions and auxiliary functions for map function. And at compile-time, the interface functions are changed to Cloud Haskell code using Template Haskell. Next, the generated code is serialized for function transfer using the distributed-process [8].

Next data is communicated using the `Transport` function at the backend. SAM transfers data using functions of the network-transport-TCP [9] library. But SSAM uses the network-transport-in-memory [10] library function to use STM. Both libraries commonly create a `Transport` with the `createTransport` function. And the network-transport [11] library processes the data passed through the `Transport`.

4.2. Major functions of SSAM

In this chapter, we introduce major functions of SSAM interface. SSAM interface organized by four main functions. And the data flow between these functions is shown in Figure 3.

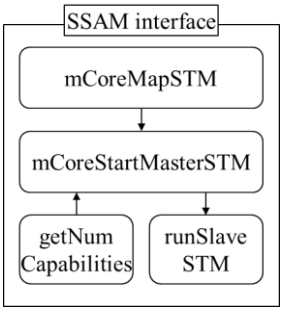

Figure 3. Inter-function data flow in SSAM interface.

As can be seen in Figure 3, SSAM begins to flow in `mCoreMapSTM`. To use SSAM, the programmer only needs to know `mCoreMapSTM` function; other functions are internal functions in SSAM. At first, the type of `mCoreMapSTM` is as follows:

```
mCoreMapSTM :: Lift a => Name -> [a] -> ExpQ
```

`mCoreMapSTM` takes 3 arguments as input. The first argument is the function will be applied to the input data. This argument is similar to the first argument in map function which is defined as `f :: a-> b`. The last argument is the input data for the function in the first argument. This argument defined as Lift type which is a feature of Template Haskell. The actual input data will be changed as a Serializable type. Next, the function name and data are then passed to the `mCoreStartMasterSTM` function. The type of `mCoreStartMasterSTM` is as follow.

```
mCoreStartMasterSTM :: :: Serializable a => LocalNode -> [NodeId] ->
                                ([NodeId] -> MVar a -> Process ()) -> IO(a)
```

`mCoreStartMasterSTM` is a modification `startMasterManyCore` function in SAM. This function is a modification of the `startMaster` functions used by Cloud Haskell and SAM. This function can use map-style programming because `MVar` receives from slave nodes data. And in this function, `getNumCapabiliies` is used to get the number of slaves. The function `getNumCapabiliies` gets the number of threads entered through the RTS option. In SSAM, the slaves are executed as much as the thread value passed from the RTS. And SSAM used `runSlaveSTM` function for executing slaves. The type of `runSlavesSTM` is as follows:

```
runSlaveSTM :: Int -> Transport -> [NodeId]
```

`runSlavesSTM` function is execution function for slave node. The first argument is the number of slaves to execute. And the second argument is the transport of the master node to be used for creating slaves. `runSlavesSTM` internally creates a slave thread using `forkIO` and returns the created `[nodeId]`. `nodeId` is passed a function to be executed later through `spawn`. Figure 4 below is an example of SSAM program.

```
 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 <- $(mCoreMapSTM 'slaveJob) input
18:      afterFunc ret
```

Figure 4. Example of SSAM Program.

Figure 4 is a simple program to add `1` for every element in a list which contains `1` to `100`. The whole program is similar to SAM. The SAM difference is that the map style function on line 17 is different. So, SSAM can use existing SAM code with little change.

5. Performance Test

In this section, we compare the performance of SSAM. SSAM is a proposed model to solve the low performance of SAM in a small-numbered manycore environment. So, experiments should be performed in a small-numbered manycore environment. In this paper, we used 32 cores environment and the experiment consists of follows. The CPU is 2 AMD Opteron 6272, RAM space is 96GB, and we use 256GB SSD as a disk, the operating system we use is Ubuntu 18.0.4 LTS, the compiler we use is GHC 8.4.3. The experimental program is a source code plagiarism detection program and input data is 108 pieces of Java source code. And the comparison targets the SAM and Eval monads. The result of the scalability experiment is shown in Figure 5.

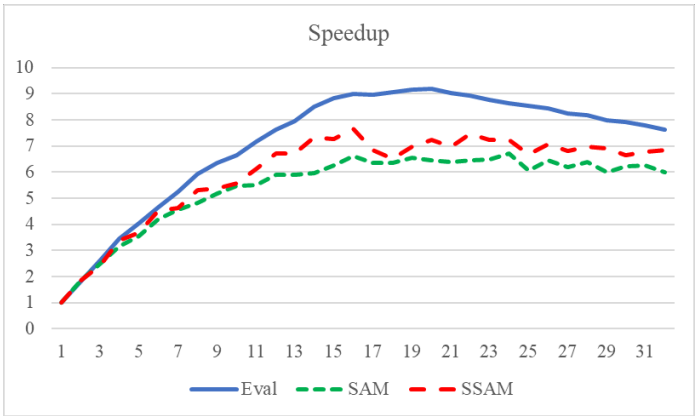

Figure 5. Comparison of scalability.

Looking at the results in Figure 5, we can see that the Eval monad has the highest scalability, followed by SSAM. The result also shows that SAM has the lowest scalability among the three models. This result is due to the experiment in the 32-core environment. In the 120-core environment, more than 50 cores will improve the performance of SAM [7]. Looking at the experimental result, the Eval monad scalability shows 7.63. And SAM scalability shows 5.99, SSAM scalability shows 6.84. This shows that SSAM is slower than the Eval monad, but about 14% more scalable than SAM.

6. Discussion

The performance tests in section 5 showed that SSAM performs better than SAM in the small-numbered manycore environments. While the performance of SSAM is lower than the Eval monad, it has two advantages.

The first advantage is easier code migration. In a small-numbered manycore environment, the SSAM and Eval monads show good performance as shown in the experiment. But as cores increase, the performance of the SAM improves. In this case, SSAM can be run with SAM simply by replacing the map function. Firstly, SSAM is easier for code migration. In the environment of small-numbered cores, both the SSAM and Eval monads show good performance, as shown in the performance test. Since the performance of SAM improves as the number of cores is increased, we can run the SSAM simply by replacing the map function.

Secondly, we can apply explicit parallelism. As Eval monad is semi-automatic in parallelism, it may not be executed in parallel, depending on the execution state. To solve this problem, the developer needs to analyze the runtime state. With explicit parallelism in the SSAM, it guarantees a reliable parallel performance.

7. Conclusion

In this paper, we proposed and implemented the SSAM which solves the overhead problem of SAM socket. For inter-process communication, the SSAM uses STM instead of the socket. The SSAM also can be used existing SAM code with a few changes. In this paper, we introduced the SSAM architecture and the major functions of the SSAM interface. We also compared the performance of the existing parallel model, Eval monad and SAM through experiment. Experimental results show that SSAM is about 14% more scalable than SAM in a small-numbered manycore environment.

Future work remains to improve the slave execution of SSAM. The current version of SSAM uses Haskell threads to perform parallel tasks. However, Haskell threads cause a GC problem because it uses a single VM [12]. To solve this problem, we will study how to use the STM library among OS processes.

8. Acknowledgement

This work was supported by Institute of Information & communications Technology Planning & Evaluation (IITP) grant funded by the Korea government (MSIT) (No. 2014-3-00035, Research on High Performance and Scalable Manycore Operating System)

References

[1] S. Marlow, S.P. Jones, and S. Singh, 2009, Runtime support for multicore Haskell, *ACM Sigplan Notices*, Vol. **44**, No. 9, pp. 65-78.

[2] Y. Kim, J. Cheon, X. Liu, S. Byun and G. Woo, 2018, SAM: A Haskell parallel programming model for many-core systems, *Proc. 2018 IEEE International Conference on Applied System Invention (ICASI)*, pp. 822-826.

[3] S. Marlow, 2013, *Parallel and Concurrent Programming in Haskell: Techniques for Multicore and Multithreaded Programming*, 1st Ed., O'Reilly Media.

[4] J. Epstein, A. P. Black and S. Peyton-Jones, 2011, Towards Haskell in the cloud, *ACM Sigplan Notices*, Vol. **46**, No. 12, pp. 118-129.

[5] C. Hewitt, P. Bishop, and R. Steiger, 1973, A universal modular actor formalism for artificial intelligence, *Proc. the 3rd international joint conference on Artificial intelligence*, pp. 235-245.

[6] A. Discolo, T. Harris, S. Marlow, S. P. Jones and S. Singh, 2006, Lock free data structures using STM in Haskell," *Int. Symp. Functional and Logic Programming*, pp. 65-80.

[7] X. Liu, Y. Kim, J. Cheon, S. Byun and G. Woo, 2018, Analysis of the Parallel Programming Models in Haskell for Many-Core Systems," *Proc. 2018 IEEE International Conference on Applied System Invention (ICASI)*, pp. 838-841.

[8] D. Coutts, N. Wu and E. d. Vries, distributed-process: Cloud Haskell: Erlang-style concurrency in Haskell, http://hackage.haskell.org/package/distributed-process, (last visit: 2019.11.01).

[9] D. Coutts, N. Wu and E. d. Vries, network-transport-tcp: TCP instantiation of Network.Transport, http://hackage.haskell.org/package/network-transport-tcp, (last visit: 2019.11.01).

[10] D. Coutts, N. Wu, E. d. Vries and A. Vershilov, network-transport-inmemory: In-memory instantiation of Network.Transport, http://hackage.haskell.org/package/network-transport-inmemory, (last visit: 2019.11.01).

[11] D. Coutts, N. Wu and E. d. Vries, network-transport: Network abstraction layer, http://hackage.haskell.org/package/network-transport, (last visit: 2019.11.01).

[12] H. Kim, J. Byun, S. Byun and G. Woo, 2017, An approach to improving the scalability of parallel Haskell programs," *Journal of Theoretical & Applied Information Technology*, Vol. **95**, No. 18, pp. 4826-4835.