# Feliss: Flexible distributed computing framework with light-weight checkpointing

Takuya Araki*† Kazuyo Narita*† and Hiroshi Tamano†
*Cloud System Research Laboratories, †Knowledge Discovery Research Laboratories
NEC Corporation, Japan
t-araki@dc.jp.nec.com, k-narita@ct.jp.nec.com, h-tamano@bx.jp.nec.com

***Abstract*—Current distributed computing frameworks, such as MapReduce and Spark, allow programmers to use only limited operations defined by the framework. Because of this restriction, algorithms that do not fit with the framework cannot be efficiently expressed. This restriction arose from the need of fault-tolerance. That is, these frameworks recover lost data by re-computing them from available data when a fault occurs. To ensure this mechanism works correctly, only operations provided by the system can be used. On the other hand, there is another fault-tolerance method called checkpointing. Since it achieves fault-tolerance by saving memory contents, there is no such limitation to operations. However, the cost of saving a memory image is high. To overcome this trade-off, we propose a light-weight checkpointing method called continuation-based checkpointing, which enables low overhead fault-tolerance without any restriction. It saves only the information that is necessary for restarting, which significantly reduces the cost of checkpointing. We implemented a distributed computing framework called Feliss by using our continuation-based checkpointing method, which includes an improved MapReduce without the above restriction and a message passing interface (MPI) subset. We evaluated Feliss with various applications and showed that order-of-magnitude speedup can be attained with applications that cannot be expressed efficiently with current frameworks.**

## I. Introduction

It is becoming common to collect data from sensors, the Web, business transactions, and so on, and analyze these "Big Data" to produce useful information.

To analyze these data, Hadoop/MapReduce is commonly used. This works efficiently if the program fits well with MapReduce. Otherwise, the performance is very poor.

This is due to the restriction of the framework. Since MapReduce only provides limited operations (e.g. map and reduce), multiple MapReduce functions must be combined to implement practical algorithms. MapReduce basically requires its input and output to be on HDFS, which is a distributed file system provided by Hadoop. Therefore, these combined MapReduce functions must communicate through HDFS file I/O, for example. It is difficult to bypass the file I/O because the dataflow is predefined and access to other distributed data from map or reduce is not allowed. Because of this restriction, machine learning and data mining algorithms, which require iterative computation, are known to run slowly with MapReduce [1].

To solve this problem, various frameworks have been proposed. Spark [1] is seen as one of the most promising frameworks. Spark can execute machine learning algorithms much faster than MapReduce by avoiding such file I/O. Spark works on a distributed data structure called resilient distributed dataset (RDD), on which operations such as map, reduce, join, and filter are provided.

Though Spark is much improved from MapReduce, operations that can be used are still limited, which makes it difficult to efficiently implement certain types of algorithms. An example of such an algorithm is "Gaussian elimination", which iteratively updates only a part of a matrix. Spark does not provide an operation that modifies an RDD, because RDDs are immutable; a new RDD needs to be created to update the previous one. Therefore, updating a part of a matrix requires copying the entire matrix including the non-updated part, which prevents efficient implementation.

Such restriction of MapReduce and Spark stems from their re-computing based implementation of fault-tolerance. In the case of MapReduce, the system re-executes map or reduce functions on part of the input file to recover lost data if a fault occurs. In the case of Spark, the system remembers the list of operations applied to RDDs and re-computes the lost part from where the data remains. RDDs should be immutable to make this mechanism work correctly. In both cases, programmers should not use distributed data other than what is defined by the system (i.e. RDD and HDFS). That is, if such user-defined data are not fault-tolerant, re-computation cannot recover them. If the data are modified during computation, re-computation modifies them again, which results in an inconsistent execution state.

On the other hand, there is another well known mechanism to achieve fault-tolerance: checkpointing. With checkpointing, the execution state is saved periodically as a "checkpoint". If a fault occurs, the saved checkpoint is restored and execution is resumed. Therefore, there is no restriction of operations when using checkpointing.

However, checkpointing has the following problems. In the case of system level checkpointing, the entire memory of a process is saved as an execution state. It requires a large amount of time to save memory contents. This high overhead is the reason MapReduce and Spark do not use checkpointing. In the case of application level checkpointing, a programmer explicitly specifies which variables to save as an execution state. This takes significantly less time than system level checkpointing because only the information

that is necessary to restart is saved. However, specifying every variable to save is a burdensome for programmers and prone to error; checkpointing does not work correctly if a programmer forgets to save variables that are needed to restart.

To overcome this trade-off between the existing fault-tolerance methods, we propose a light-weight checkpointing method called continuation-based checkpointing, which enables low overhead fault-tolerance without any restriction. It is a kind of application level checkpointing method, but programmers do not have to explicitly specify which variables to save. Instead, programmers write a program so that the rest of the computation is occasionally converted to a data structure, called a continuation, and stored into a pool.

Fig. 1 shows the execution flow of our continuation-based checkpointing method. First, a program puts an initial continuation into the pool. The system picks a continuation up from the pool and executes it. During execution, new continuations are stored into the pool. By repeating this process, the entire program is executed.

When the system takes a checkpoint, it stops picking up a continuation after finishing the execution of a continuation (it is assumed that the execution time of one continuation is shorter than the checkpoint interval time). Then, the continuation pool is "serialized" and saved. Serialization means converting data structures that contain pointers into contiguous data. By saving the continuation pool, all information that is needed to restart the program can be saved because a continuation contains pointers to variables that are needed to execute it; these variables are serialized together when the continuation is serialized. In other words, the system only saves the information that is needed to restart the program, which is much smaller than that of system level checkpointing.

We developed a distributed computing framework called Feliss using our continuation-based checkpointing method.

We also implemented an improved MapReduce and a message passing interface (MPI) subset within Feliss. The improved MapReduce implementation does not have the restriction that Hadoop MapReduce has; it does not require the input and output to be on HDFS, and access to arbitrary distributed data during the MapReduce execution is allowed. Therefore, algorithms that cannot be implemented efficiently with Hadoop MapReduce can be executed efficiently. In addition, implementing MapReduce and MPI on the same framework makes it possible to write a program that uses both APIs. This is useful because it enables us to write a program that executes a machine learning algorithm written in MPI after preprocessing raw data with MapReduce.

## II. Continuation-Based Checkpointing Method

In this section, we explain our continuation-based checkpointing method in detail including its implementation written in C++ on Linux.

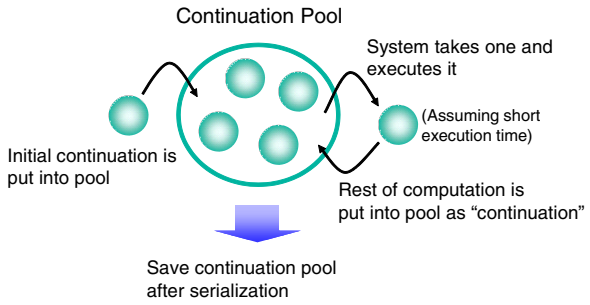


Figure 1. Continuation-Based Checkpointing

Conceptually, a continuation is a data structure that represents an execution state, which includes the data and the code required to run it. The actual implementation of the continuation is just a data structure that holds the pointer to the function and its arguments. Therefore, the overhead of creating a continuation is limited.

As described in Section I, a program needs to be written so that the rest of the computation is occasionally converted to a continuation and stored into a pool in order to use continuation-based checkpointing.

For example, functions normally written as

```
int sample(int a)
  {int b = f(a); return g(b);}
int g(int b) {return h(b);}
```

can be separated into continuations as follows:

```
void sample(int a, int *r) {
  int b = f(a);
  put_pool(make_cont(g,b,r));
}
void g(int b, int *r) {*r = h(b);}
```

In this example, the function `g` is stored into the continuation pool. Here, `make_cont` creates a continuation by taking the pointer to the function and its arguments; `put_pool` stores the given continuation into the continuation pool. A continuation cannot return a value; therefore, the value is returned through a pointer given as an argument. The function `sample` can return from the function after executing `put_pool`, without executing `g`. This is when the system can take a checkpoint. Then, the system picks up the continuation of `g` and executes it.

In this example, only one continuation is created in the function and stored into the pool, but storing multiple continuations in one function is allowed. In this case, the execution order of these continuations is not defined.

The execution time of the continuation does not have to be very small; it only needs to be only small enough compared to the checkpoint interval time. Therefore, a programmer does not have to divide a program into many

small continuations. If the execution time of a continuation is large, the system waits for the continuation to finish before taking a checkpoint, which results in a larger checkpoint interval time than expected.

Continuations must be serializable. We used the serialization library provided by Boost [2], which can serialize primitive types and containers provided by the standard library by default. To serialize a user-defined class, a programmer needs to add a piece of code into the class.

The serialization library properly serializes data structures that have pointers in them. That is, if an object is pointed to by multiple objects, the information is stored in the serialization result. When the serialized data are deserialized, such a relationship is recovered.

On the other hand, the serialization library cannot serialize the pointer to a function. Thus, we implemented the serialization of the pointer to a function by using the "symbol name" of the function, which can be obtained as a string using a function called `dladdr` provided by the OS. The pointer to the function is retrieved by calling `dlsym` with the symbol name as an argument.

With the above implementation, the execution state can be saved by saving the continuation pool after serialization, which can save all the data that can be traced from the continuations.

As described above, programmers need to modify programs if they want to use continuation-based checkpointing. In this sense, this method is semi-transparent. However, the modification is small enough and easy to implement. In addition, if a programmer only uses an API, such as MapReduce, he/she does not need to be aware of the checkpointing layer; it is hidden by the MapReduce implementation and he/she only needs to provide map and reduce functions, etc.

## III. Design and Implementation of Feliss

We designed and implemented a distributed computing framework called Feliss by using our continuation-based checkpointing method. In this section, we describe Feliss by explaining components that compose Feliss.

### A. IDL-less RPC

We used remote procedure call (RPC) as the primitive of distributed execution, which is a mechanism to call a function on a remote server. Sending and receiving data between servers can also be done with RPC.

Remote procedure call is a traditional mechanism, but existing RPC requires a programmer to write an interface description in IDL. It is compiled by an IDL compiler to produce stub codes. A client and a server program are written using the stub.

Since such a procedure is cumbersome, we designed and implemented a new RPC system that does not require IDL.

With this system, RPC can be written like:

```
int r = rpc(node, foo, 1, 2);
```

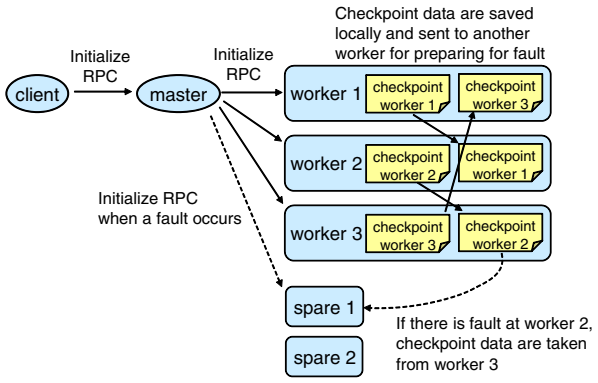


Figure 2. Overview of distributed checkpointing

Here, "`rpc`" is a template function that takes as its arguments the remote server (`node`), the pointer to the calling function (`foo`), and their arguments (`1` and `2`). Static type information of C++ is kept in this RPC; type mismatch between the callee and the caller causes a compilation error.

To make such RPC possible, the client needs to tell the server which function to call. For this purpose, we used the symbol name of the function just like the serialization of a function pointer in our continuation-based checkpointing method. Arguments of the function are serialized by Boost and sent to the server together with the symbol name of the function.

### B. Fault-Tolerance with Continuation-based Checkpointing

In Section II, we described our continuation-based checkpointing method on a single server. In this section, we explain distributed checkpointing using our method.

For distributed checkpointing, the following servers are specified at initialization: a master, workers that actually do the computation, and spare servers that are used when a fault occurs. Fig. 2 gives the overview of the distributed checkpointing system. A client program first connects to the master and initializes RPC. Then the master connects to the workers and initializes RPC.

A program creates a continuation and specifies it as the first continuation. The master puts it into the continuation pool of one of the workers, which starts the program execution. During the execution of the continuations, some continuations are put into other workers' continuation pools with RPC, which results in distributed execution.

The master periodically takes checkpoints at specified intervals. When a checkpoint is taken, the master stops execution of all the workers. That is, after finishing currently executing continuation, all the workers stop picking up continuations from the continuation pool.

After confirming that all the workers have stopped, the master orders the workers to take a checkpoint. Each worker serializes the continuation pool and saves it onto the local

disk. The serialized data are also sent to another worker to prepare for server fault. After confirming that all the workers finished taking a checkpoint, the master restarts the workers.

If there is a long running continuation on a worker, other workers must wait for that worker to stop before starting checkpointing. Asynchronous checkpointing, such as the Chandy-Lamport algorithm can solve this problem. Implementing such an algorithm is for future work.

The master periodically checks if the workers are running correctly. If a worker is not working, the master restarts all the workers from the checkpoint. More specifically, the master stops all the worker processes and restarts them; they read previously taken checkpoint data and continue execution from that state. The checkpoint data are read from the local disk if the worker server is still working. Otherwise, a spare server is used instead; it reads the checkpoint data from the worker that has a copy of it.

We assume that a fault does not occur at the master. This assumption is reasonable because the number of workers is much larger than that of the master, which is one, and the fault probability is proportional to the number of servers. This assumption is adopted by most distributed frameworks such as Hadoop and Spark.

### C. Variable Wrapper and Utility Functions

If a server restarts from a checkpoint, addresses of the variables may change when they are recovered from checkpoint data. This is not a problem within a server because the pointers that point to the variables are also changed according to their addresses during the deserialization process. On the other hand, it becomes problematic when a programmer wants a pointer that points to another server's variable.

To solve this problem, we introduce "variable wrapper", which wraps variables. A variable wrapper works as an ID of a pointer; there is a mapping table between the ID and the pointer in each server. Since the table exists within a server, the mapping can be maintained correctly after restart. Programmers can use the ID as the pointer to another server's variable.

We also used the variable wrapper for other purposes. The first one is synchronization. The execution time of a continuation must be short enough. However, if the execution is blocked by synchronization, it might become too long. To solve this problem, the system provides a synchronization mechanism that does not block execution.

With this mechanism, the variable wrapper works like a "future". That is, the variable wrapper has two states: a pointer is set or not set to it. To read the pointer of the variable wrapper, a continuation is specified; it is executed after the pointer is set to the variable wrapper. This enables synchronization without blocking.

Another functionality added to the variable wrapper is saving the contents of the pointer onto the local disk after serialization, which is used to support out-of-core computation. The data saved on the disk are copied to another worker together with checkpoint data when a checkpoint is taken.

The system provides broadcast and reduction functions with a tree-based communication algorithm, which uses variable wrappers as remote pointers.

In addition, the system provides bounded/unbounded queues, which also require synchronization. This synchronization is implemented using a continuation similarly as mentioned above.

### D. Improved MapReduce and MPI

Programmers can write distributed fault-tolerant programs with the functionalities explained above: RPC, continuations, and utility functions together with variable wrappers. However, these functionalities are rather primitive. Therefore, we provided MapReduce on top of the checkpointing layer.

The MapReduce we implemented was improved so that the restriction of Hadoop/MapReduce is removed; it does not require the input and output to be on HDFS, and access to arbitrary distributed data is allowed.

Other than that, the MapReduce API is similar to that of Hadoop/MapReduce and Google's original MapReduce [3]; map, reduce, combine, and partition functions are specified as the arguments of the `mapreduce` function. These user-provided functions are function objects, which are instances of classes that have `operator()` as a member function.

Such function objects can have states within them. The states can be pointers; access to other data from these function objects is allowed. This enables more flexible and faster execution than with the traditional MapReduce implementations.

Our MapReduce implementation uses the queues explained above for input and output of MapReduce. Multiple MapReduce functions can be connected through the queues. One MapReduce can pass the data through memory to the other. In addition, the latter MapReduce can begin execution before the former MapReduce completely finishes. Therefore, in such a case, our implementation runs faster than traditional implementations.

In one MapReduce, the map-processing and the reduce-processing parts are also connected through queues. Sorted map results are sent to the reduce part, and the reduce part merge-sorts results from multiple maps, which results in so called shuffle phase.

Since the reduce part cannot start before the map part completely finishes, the map results might not fit in memory. Such data are spilled out to the local disk using the variable wrapper described above. The spilled map results are sorted with an external sort algorithm in the shuffle phase.

Hadoop/HDFS can be used as input and output. To use HDFS as input, the system provides a function that reads chunks of a file from HDFS and puts them into the input queues of MapReduce. Load balancing is done here similar

to the original MapReduce. It is also aware of the locality of the file; it puts local chunks as far as possible into the queues.

We also implemented an MPI subset on top of the checkpointing layer, which enables us to use existing MPI libraries and algorithms written in MPI.

To use MPI, the continuation of the top level function that uses MPI needs to be sent to all the workers. In the function, a programmer can write an MPI program as usual. To use both MapReduce and MPI together, a programmer can just call the `mapreduce` function together. They can communicate through pointers to variables, variable wrappers, queues, and so on.

## IV. Evaluation

To show the effectiveness of the proposed method and framework, we conducted the following evaluations. The servers used for the evaluation had Xeon 2GHz quad-core CPUs and 12GB of RAM, which were connected through Gigabit Ethernet. We used a maximum of 18 servers.

### A. Checkpointing

To confirm the effectiveness of our continuation-based checkpointing method, we compared its performance with Berkeley Lab Checkpoint/Restart (BLCR) [4], which realizes checkpointing by storing the entire memory image of a process onto a disk.

Fig. 3 shows the evaluation results. The program used for the evaluation holds `std::map<string, string>` as data, which stores 20 million elements. We evaluated their checkpointing times and checkpoint file sizes using one server. Feliss could take a checkpoint 5.3 times faster than BLCR. The checkpoint file size was 5.6 times smaller than that of BLCR, which contributed to the speed of the checkpoint.

One reason for this is that `std::map` uses a data structure called a red-black tree, which uses many pointers internally. Because of this memory overhead, it requires much more memory than the actually stored data size. Serialization transforms such data structures to contain only the actually stored data, which significantly reduced the data size. Data analysis applications usually require such complex data structures, which is different from traditional HPC programs that commonly use dense data structures such as an array.

Next, we evaluated the checkpointing and restarting times of applications on Feliss. The applications used for evaluation were: word count (WC), logistic regression (LR), LU decomposition (LU), latent semantic analysis (LSA) [5], and latent Dirichlet allocation (LDA) [6].

WC uses MapReduce. LU is implemented with Gaussian elimination using MPI. LSA uses both MapReduce and MPI together with Parallel ARPACK [7] to realize singular value

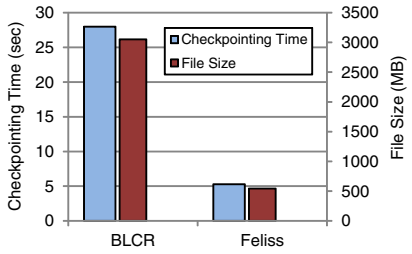


Figure 3. Comparison with BLCR

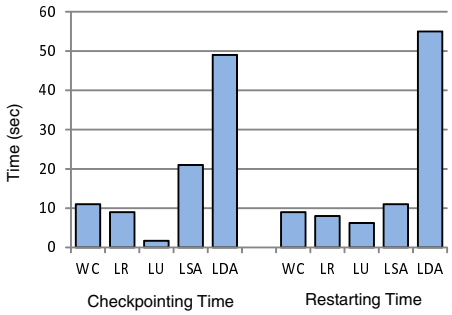


Figure 4. Checkpointing and Restarting Times of Applications

decomposition that is needed in LSA. Feliss has such advantage that it can utilize existing high quality library written in MPI, which is difficult with Spark or Hadoop. LDA uses multiple MapReduce functions, which are efficiently connected as described in Section III-D.

The input data of WC, LSA, and LDA was English Wikipedia text of about 8 GB. The input data of LR is 20 GB of vectors. The input data of LU is a matrix of size 4320 x 4320.

We measured (1) the execution time without checkpointing, (2) execution time with one checkpointing, and (3) execution time with one checkpointing and a fault; one of the processes was killed just after taking the checkpoint, and the program restarted from the checkpoint after that. We used (2) - (1) as checkpointing time, and (3) - (2) as restarting time. We evaluated them on 18 servers (72 CPUs). Fig. 4 shows the evaluation results.

Both checkpointing and restarting times were small enough on real applications. The checkpointing and restarting times of LDA were larger than those of the other applications because LDA uses more memory than the others.

### B. Performance of Feliss

To determine the performance of Feliss, we first evaluated its scalability. Fig. 5 shows the execution speed of the above applications with various numbers of CPUs. It shows relative speed normalized by the speed with 1 CPU. The checkpoint was taken once during the execution. The input data were the same as in the previous evaluation.

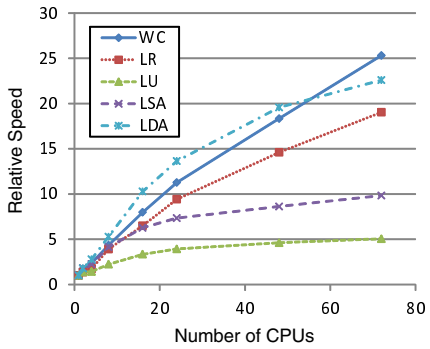


Figure 5. Scalability of Feliss

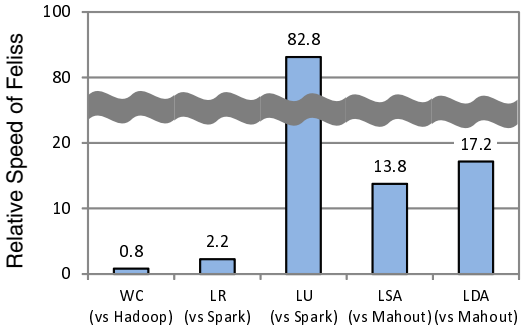


Figure 6. Comparison with Other Frameworks

The evaluation results show good scalability of Feliss with WC, LR, and LDA. The scalability with LU and LSA was lower than the other applications. This is because network communication became bottlenecked during the execution of LU and singular value decomposition used in LSA. We believe that scalability can be improved by using a faster interconnect such as 10 Gigabit Ethernet or InfiniBand.

We then executed the applications written in other frameworks, and compared their performance with Feliss. We executed WC on Hadoop (0.20.2), LR and LU on Spark (0.6.0 and 0.7.2 respectively), and LSA and LDA on Mahout (0.7), which is a machine learning framework implemented on top of Hadoop. Fig. 6 shows the relative speed of Feliss compared with these frameworks.

The input data of WC, LR, LU, and LSA were the same as the previous evaluation. For the input data of LDA, we used part of the Wikipedia text of 250 MB, because the execution speed of Mahout was too slow. One checkpoint was taken during the execution of Feliss. We evaluated these applications on 18 servers (72 CPUs).

With WC, Hadoop was 20% faster than Feliss. Hadoop is well optimized for such kind of applications.

With LR, Feliss was 2.2 times faster than Spark. One of the reasons for this is that Feliss uses an efficient reduction algorithm with tree-based communication, unlike Spark. Since reduction is a predefined operation in Spark, it cannot be modified at the user program level. Another reason is that Feliss is implemented in C++, which is faster than Scala that is used in Spark.

With LU, Feliss was 82.8 times faster than Spark. Feliss was 13.8 times faster with LSA, and 17.2 times faster with LDA than Mahout. These results show that even these common algorithms do not fit well with Spark or Hadoop/MapReduce because of the above mentioned restrictions, but can be expressed efficiently with Feliss.

## V. Related Work

Research on checkpointing has a long history. For system level checkpointing, BLCR [4] has been widely used recently, which works as a kernel module of Linux. Libckpt [8] works as a user-level library, and supports incremental checkpointing, which saves only the changed part after the previous checkpoint. Our current framework does not support incremental checkpointing. We plan to implement this for future work. In Condor [9] checkpointing is implemented as a user-level library. It uses checkpointing to migrate a process for resource management in a distributed computing environment. Applying checkpointing for such a purpose is also for future work.

Transparent application level checkpointing with the help of a compiler was proposed in [10]. It analyzes live variables that should be checkpointed on behalf of a programmer. Its motivation was similar to ours, but they could not reduce the checkpoint size of complex data structures, such as `std::map`, because information of the data structure is needed (e.g. to recover `std::map`, pointer information is not needed). In the case of serialization, a serialization library (or a serialization code provided by a programmer) efficiently serializes such data structures using the information. In addition, the implementation cost of creating a new compiler is much larger than that of library layer implementation such as ours.

Charm++ [11] achieves checkpointing on the programming language layer. Objects of Charm++ are location independent and can migrate to other processors. This characteristic is used for checkpointing. These objects are similar to our continuations. Feliss is different from Charm++ in that Feliss does not require to use a special language.

These checkpointing methods are used together with MPI. For example, Open MPI, which is a production level MPI implementation, supports checkpointing using BLCR [12].

As for data analysis frameworks, there has been many attempts to increase the speed of large-scale machine learning. However, current systems have restrictions.

Haloop [13] and Twister [14] can efficiently execute iterative MapReduce. However, they do not support other execution patterns. Piccolo [15] and Distributed GraphLab [16] are distributed computing frameworks that are specialized to specific data structures such as a table or a graph.

They achieve fault-tolerance by checkpointing these data structures. These frameworks have restriction in which only limited distributed data structures are supported. Sparkler [17] extends Spark to improve the performance of large scale matrix factorization problems. However, it only improves the performance of such kind of problems; the restriction still remains. SystemML [18] compiles a machine learning algorithm written in a domain specific language into a Hadoop/MapReduce program. This provides flexibility to the user, but it exhibits poor performance because of the restriction of Hadoop; if the data size is small, its performance is much worse than R, though it has advantage that it can handle larger data than R. M3R [19] improves its performance by creating a Hadoop/MapReduce-compatible layer that gives up fault-tolerance and out-of-core computation.

Implementation of MapReduce using MPI is proposed in [20]. Using both MapReduce and MPI together may be possible, but it does not support fault-tolerance.

## VI. Conclusion

We proposed a light-weight checkpointing method called continuation-based checkpointing, and used it to implement a flexible distributed computing framework called Feliss. We implemented and evaluated various applications on Feliss and confirmed that it can execute them more efficiently than current frameworks.

Future work includes extension of checkpointing such as asynchronous checkpointing, incremental checkpointing, and using checkpointing for resource management. Completion of MPI implementation is also for future work. We started to use Feliss as a platform of data mining research. We will add other abstractions to Feliss to make it easy to express various data mining algorithms.

## Acknowledgment

This work was performed of the METI R&D of Industrial Science and Technology Frontier Program (Green IT Project) supported by NEDO.

## References

[1] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *NSDI*, 2012.

[2] R. Ramey, "Boost serialization," http://www.boost.org/doc/libs/release/libs/serialization/.

[3] J. Dean and S. Ghemawat, "MapReduce: Simplified data processing on large clusters," in *OSDI*, 2004, pp. 137–149.

[4] Future Technologies Group, "Berkeley Lab Checkpoint/Restart (BLCR)," https://ftg.lbl.gov/projects/CheckpointRestart/.

[5] T. K. Landauer and S. Dumais, "Latent semantic analysis," *Scholarpedia*, vol. 3, no. 11, p. 4356, 2008.

[6] Y. Wang, H. Bai, M. Stanton, W.-Y. Chen, and E. Y. Chang, "PLDA: Parallel latent Dirichlet allocation for large-scale applications," in *AAIM*, 2009, pp. 301–314.

[7] K. J. Maschhoff and D. C. Sorensen, "P_ARPACK: An efficient portable large scale eigenvalue package for distributed memory parallel architectures," in *Third International Workshop on Applied Parallel Computing, Industrial Computation and Optimization*, 1996, pp. 478–486.

[8] J. S. Plank, M. Beck, G. Kingsley, and K. Li, "Libckpt: Transparent checkpointing under Unix," in *Usenix Winter Technical Conference*, 1995, pp. 213–223.

[9] M. Litzkow, T. Tannenbaum, J. Basney, and M. Livny, "Checkpoint and migration of UNIX processes in the Condor distributed processing system," University of Wisconsin - Madison Computer Sciences Department, Tech. Rep. UW-CS-TR-1346, 1997.

[10] I. Cores, G. Rodriguez, M. Martin, and P. Gonz'lez, "Reducing application-level checkpoint file sizes: Towards scalable fault tolerance solutions," in *10th International Symposium on Parallel and Distributed Processing with Applications (ISPA)*, 2012, pp. 371 –378.

[11] G. Zheng, L. Shi, and L. V. Kale, "FTC-Charm++: an in-memory checkpoint-based fault tolerant runtime for Charm++ and MPI," in *CLUSTER*, 2004, pp. 93–103.

[12] J. Hursey, J. Squyres, T. Mattox, and A. Lumsdaine, "The design and implementation of checkpoint/restart process fault tolerance for Open MPI," in *IPDPS*, 2007, pp. 1 –8.

[13] Y. Bu, B. Howe, M. Balazinska, and M. D. Ernst, "HaLoop: efficient iterative data processing on large clusters," *Proc. VLDB Endow.*, vol. 3, no. 1-2, pp. 285–296, 2010.

[14] J. Ekanayake, H. Li, B. Zhang, T. Gunarathne, S.-H. Bae, J. Qiu, and G. Fox, "Twister: A runtime for iterative MapReduce," in *HPDC'10*, 2010, pp. 810–818.

[15] R. Power and J. Li, "Piccolo: Building fast, distributed programs with partitioned tables," in *OSDI*, 2010, pp. 293–306.

[16] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein, "Distributed GraphLab: a framework for machine learning and data mining in the cloud," *Proc. VLDB Endow.*, vol. 5, no. 8, pp. 716–727, 2012.

[17] B. Li, S. Tata, and Y. Sismanis, "Sparkler: supporting large-scale matrix factorization," in *EDBT*, 2013, pp. 625–636.

[18] A. Ghoting, R. Krishnamurthy, E. Pednault, B. Reinwald, V. Sindhwani, S. Tatikonda, Y. Tian, and S. Vaithyanathan, "SystemML: Declarative machine learning on MapReduce," in *ICDE*, 2011, pp. 231–242.

[19] A. Shinnar, D. Cunningham, V. Saraswat, and B. Herta, "M3R: increased performance for in-memory Hadoop jobs," *Proc. VLDB Endow.*, vol. 5, no. 12, pp. 1736–1747, 2012.

[20] S. J. Plimpton and K. D. Devine, "MapReduce in MPI for large-scale graph algorithms," *Parallel Comput.*, vol. 37, no. 9, pp. 610–632, 2011.