Parallel Discrete-Event Simulation on Data Processing Engines

Kazuyuki Shudo, Yuya Kato*, Takahiro Sugino†, and Masatoshi Hanai‡
Tokyo Institute of Technology

***Abstract*—Development of a decent parallel simulator is challenging work. It should achieve enough performance, scalability and fault tolerance. Our proposal is utilizing general-purpose data processing engines such as MapReduce implementations for parallel simulation. Widely used and mature engines take away a large part of the development effort and support scalability and fault tolerance. We demonstrate that a parallel discrete-event simulator can be implemented on such engines, Apache Hadoop and Apache Spark, by modeling message passing of distributed systems on MapReduce key-value processing model. Implemented simulators could handle 10^8 nodes with 10 computers. Preliminary evaluation showed that our Spark-based simulator is about 20 times as fast as an existing simulator thanks to Time Warp.**

***Index Terms*—parallel discrete-event simulation, Time Warp, peer-to-peer, data processing engine, MapReduce**

I. Introduction

Parallel simulation with multiple computers enables large-scale and possibly fast simulation beyond a single computer. There have been number of efforts to develop parallel simulators and they have achieved a certain amount of success in each application area. Development of a decent parallel simulator is challenging work. Desirable properties of such a parallel simulator include enough performance, scalability and fault tolerance. Scalability is rather easy to achieve, but it is not easy to achieve enough performance comparable even with a single computer and existing parallel simulators are hardly fault-tolerant. Furthermore, each parallel simulator duplicates development effort to achieve those desirable properties.

Our proposal is utilizing general-purpose data processing engines such as MapReduce implementations for parallel simulation. They can drive thousands of computers and provide fault-tolerance features. We can leave a large part of difficulties of the development caused by distributed nature to those engines. In addition, well-known engines are expected to be mature as software because they are used worldwide not only for simulation purposes.

We demonstrate that such simulator construction is possible by our implementation. Our discrete-event simulators run on data processing engines, Apache Hadoop [1] and Apache Spark [2], that support MapReduce [3] programming and processing model. The simulators implement message passing between nodes in distributed systems, that we modeled on key-value processing model of MapReduce. Experiments

* Present affiliation is Cygames, Inc.
† Present affiliation is Digital Arts Inc.
‡ Present affiliation is Nanyang Technological University

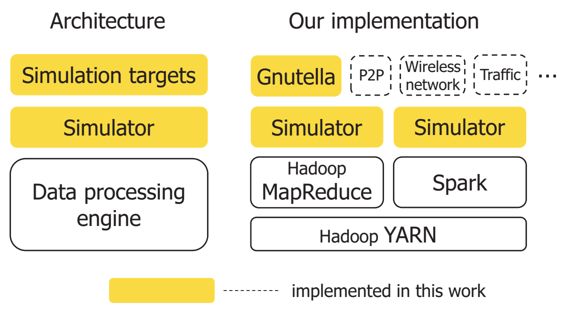


Fig. 1. A simulator architecture utilizing data processing engines.

confirm that the simulators process a communication-intensive distributed system, Gnutella [4], that performs flooding of a search message over nodes. The simulators could handle 10^8 nodes with 10 computers though they are not optimistic simulation.

Optimistic simulation with Time Warp [5] requires a larger amount of memory or storage to preserve simulation events and anti-messages than non-optimistic methods. Suppression of memory and storage consumption improves scalability of simulation targets. It should be especially effective for memory-based data processing engines such as Spark. Two techniques, Moving Time Window (MTW) [6] and Adaptive Time Warp (ATW) [7] could reduce the number of messages that reside on memory at the same time in return for an increase of simulation time.

Optimistic simulation contributed performance of our simulators. Our Spark-based simulator achieved high performance that is about 20 times of an existing simulator though the evaluation was very preliminary.

This paper is organized as follows. Section II reviews the present situation of parallel simulators. Section III describes our proposal and confirms its feasibility based on implementation and experiments. Section IV introduces Time Warp, an optimistic synchronization protocol into our simulators and reports experimental results with the message number reduction techniques.

II. Related work

There have been number of parallel simulators developed for their own simulation targets. In most cases communication between computers or CPU cores is accomplished by a simulator itself using shared memory, BSD socket API or a message passing library such as MPI. Overlay Weaver [8], [9] and dPeerSim [10] are peer-to-peer protocol simulators, that support BSD socket-based distributed execution. Overlay Weaver is a real-time simulator, that can also work as a discrete-event simulator in non-parallel execution. dPeerSim is a discrete-event simulator, that implements Null Message algorithm [11], a non-optimistic synchronizaiton protocol. ns-3 [12] is a discrete-event network simulator, that can run on multiple computers communicating with MPI. It implements non-optimistic synchronization protocols with MPI collective operations and the Null Message algorithm. dSUMO [13] is a distributed version of SUMO [14], a traffic simulator. It is based on BSD socket API. Above-mentioned simulators are not fault-tolerant and, in the first place, there are few parallel simulators that are tolerant of computer failures. There are MPI implementations supporting fault-tolerance techniques. For example, Open MPI [15] supports checkpoint and restart. But it is still an open problem to investigate how much a fault-tolerant MPI contributes to a fault-tolerant parallel simulator. A fault-tolerant MPI alone cannot make a parallel simulator fault-tolerant of computer failures. One of the reasons is that not all the states of a simulator are managed by MPI. A simulator may store the states on storage, not only on memory, and the checkpoint tecnique does not save them.

Parallel programming languages support development of a parallel simulator. XAXIS [16] is a multi-agent simulation middleware written in X10 [17], a parallel programming language. Megaffic [18] is a traffic simulation model on XAXIS. Their synchronization protocol is time-step-based as well as other traffic simulators such as SUMO. Simulator developers receive benefit from such a language and runtimes. They can utilize higher-level abstractions such as PGAS for parallel execution instead of low-level communication APIs. An implemented simulator runs on multiple runtimes including shared memory and MPI, that X10 supports. This approach is promising and different from our approach as follows. Performance and efficiency of parallel execution mainly depend on simulator developers while they depend on a data processing engine in our approach. Pros and cons of this approach are possibility of high-performance and difficulty in achieving high-performance. Fault-tolerance is another problem. This approach does not generally provide it due to difficulty of taking a snapshot of running programs though there have been attempts [19]. In contrast, our approach benefits from fault-tolerance features of a data processing engine. On a data processing engine, a state of a simulation target is always data and an engine reexecutes part of the simulation lost by a computer failure.

There have been attempts to run simulation on a data processing engine and even model a simulation target on MapReduce programming and processing model [20], [21], [22], [23]. Our contribution that differentiates our work from them include parallel discrete-event simulation and optimistic synchronization required to achieve high performance.

In our previous work [24], optimistic parallel simulation of peer-to-peer systems succeeded. Time Warp enabled optimistic parallel discrete-event simulation. In the work we developed a simulator from scratch in C++ language and MPI. It is not based on a data processing engine.

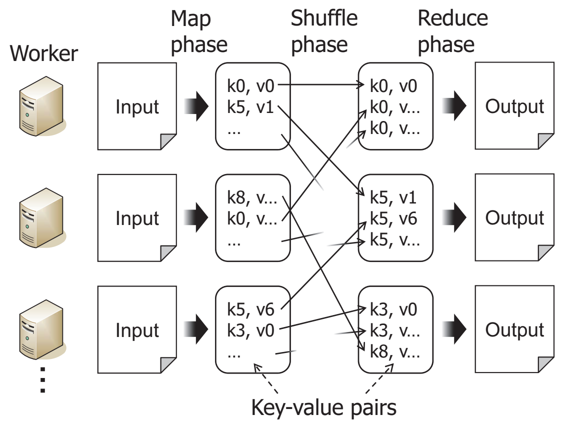


Fig. 2. Programming and processing model of MapReduce.

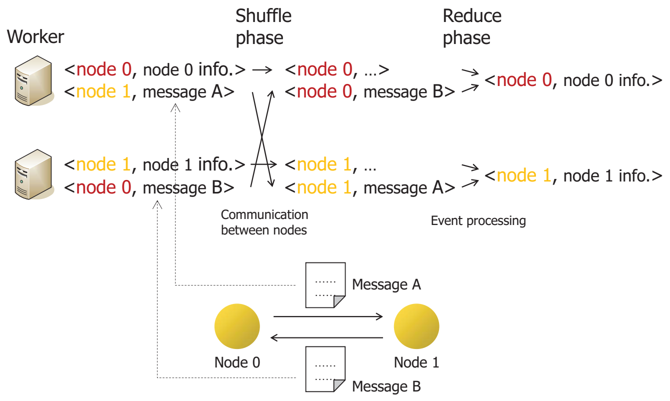


Fig. 3. A model of peer-to-peer systems on MapReduce model.

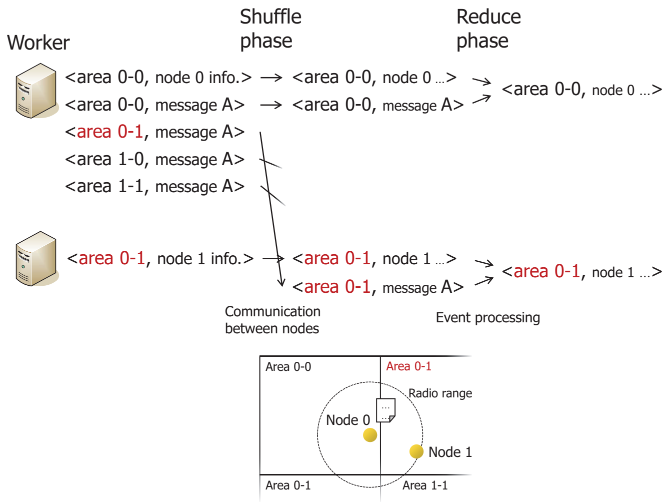


Fig. 4. A model of wireless networks on MapReduce model.

III. Parallel simulation on data processing engines

The left part of Figure 1 is an architecture of a simulator utilizing data processing engines. A simulator is hosted by a data processing engine, and processes a simulation target.

The right part of Figure 1 shows the structure of our implementation of the architecture. We implemented two simulators and each of the two runs on Apache Hadoop [1] and Apache Spark [2]. Our approach does not depend on a specific data processing engine. We demonstrate it by the two separated implementations. Hadoop is a software suite that implements Google's infrastructure for storing and processing a large amount of data. It includes Hadoop MapReduce as its primary data processing engine. Spark is another data processing engine developed to be faster than Hadoop MapReduce. One of the reasons we chose them is their results in scalability. Hadoop could drive a computer cluster with 4500 computers [25]. Spark has been successfully launched on a cluster with more then 4000 cores [26]. Another reason is their fault tolerance feature. Both of them reexecute a failed task on another computer. Note that a task is part of a data processing job and each task is assigned to a computer.

The programming and processing model of Hadoop MapReduce is MapReduce [3]. Spark also supports the model. It is necessary for a simulation target to be represented in a model that a data processing engine supports. Otherwise the simulation target cannot be processed in parallel. In our case a simulation target has to be represented in the MapReduce model though investigating a better model is part of future work.

Figure 2 depicts the MapReduce model. It does not assume computers, that called worker, have shared memory. A data item is a key-value pair. In map phase, a user-supplied map function generates key-value pairs from input data. Shuffle phase collects key-value pairs with the same key onto the same worker. The shuffle phase involves communication between workers in general. In reduce phase, a user-supplied reduce function processes the collected key-value pairs to generate output data. An iteration that consists of the three phases can repeat.

Our primary simulation target is distributed systems, especially peer-to-peer systems because the motivation of this research is difficulty in development of a parallel simulator for peer-to-peer systems [8], [9]. In a peer-to-peer system, a large number of nodes communicate each other by sending and receiving messages. A certain part of distributed systems are just alike. We could model peer-to-peer systems on the MapReduce model and Figure 3 shows the designed peer-to-peer model. A node and a message are represented by a key-value pair whose key is a node identifier (ID). A node ID can be concrete such as an IP address or abstract such as a number on a structured overlay network. Value part of a key-value pair is node information or a message body. Shuffle phase "delivers" a message to a node by this design. In case that a message key-value pair and a node key-value pair have the same node ID as its key, they are collected onto the same worker and the worker processes message reception by the node. Communication latency and error on a simulation target are simulated in a sending or receiving process based on network environment information that is shared by all the workers. A sending node embed send time into value part of a message, or calculates receive time and embed it. A receiving node adopts the embedded receive time, or calculates receive time based on the embedded send time.

Peer-to-peer systems are not the only simulation target that the MapReduce model can support. Figure 4 shows a model of wireless networks on the MapReduce model. A two- or three-dimensional Euclidean space is divided into partitions and the partitions have their IDs. Note that it is not necessary for a partition to be a square and elaborated partitioning leads to good load balance. Key part of a key-value pair is a partition ID for message delivery. Value part is node information or a message. A sending node calculates which partitions possibly receive the message and generates message key-value pairs for the partitions. In shuffle phase, a node and messages to the partition where the node resides are collected onto the same worker. The worker processes message reception. Car traffic and pedestrian traffic can also be modeled in this manner.

A simulation requires a simulation scenario. It describes events including node joins, leaves and query issues in case of peer-to-peer simulation. Our simulators convert a scenario into key-value pairs representing those events in advance.

We selected discrete-event simulation to implement while time-step-based simulation can run on the MapReduce model. Discrete-event simulation enables precise handling of simulated time but its performance was our concern. There is possibility that it processes very limited number of events such as message reception in an iteration of MapReduce even with bounded lag technique [27]. But we expected that an optimistic synchronization protocol would increase the number and the performance. It worked as we expected and contributed performance of our simulators as shown in Section IV-C.

TABLE I
EXPERIMENTAL SETUP.

OS	Ubuntu 14.04.2 LTS with Linux 3.16.0
CPU	2.40 GHz Xeon E5620 $\times$ 2
Network	Gigabit Ethernet
Apache Hadoop	2.7.1
Apache Spark	1.4.1
Scala	2.11.6
Java Virtual Machine	Java SE 8 Update 45
`yarn.nodemanager .resource.memory-mb`	`30000`

TABLE II
HADOOP MAPREDUCE PARAMETERS.

`mapreduce.map.memory.mb`	`2048`
`mapreduce.map.java.opts`	`-Xmx1500`
`mapreduce.reduce.memory.mb`	`3072`
`mapreduce.reduce.java.opts`	`-Xmx2500`

TABLE III
SPARK PARAMETERS.

`spark.executor.instances`	`19`
`spark.executor.memory`	`12G`
`spark.executor.cores`	`4`
`spark.driver.memory`	`5G`
`spark.serializer`	`org.apache.spark .serializer .JavaSerializer`
`spark.shuffle.manager`	`sort`
`spark.task.cpus`	`1`

A. *Comparison among data processing engines*

Our simulators have their own implementations of the peer-to-peer model, in which a large number of nodes communicating each other. On top of the peer-to-peer model we implemented Gnutella [4], a peer-to-peer system, as a simulation target to evaluate feasibility, scalability and performance of our approach. Gnutella is a query flooding-based protocol. In a Gnutella network a node issues a query to search target objects such as a movie content and a query is flooded over the network with the maximum number of hops, that is 7 in default.

All the experiments in this paper are conducted on a YARN cluster with one master computer running YARN's ResourceManager and 10 worker computers running YARN's NodeManager. All the 11 computers have the same setup shown in Table I. Table II and III show parameters for Hadoop MapReduce and Spark.

We measured execution time of Gnutella simulation with three network topologies as follows.

- Two-dimensional mesh
- Complex network generated by BA model where $m = 1$
- Complex network generated by BA model where $m = 2$

These three have different degree distributions. A two-dimensional mesh (2D mesh network) represents a Gnutella network in which almost all the nodes have connections to 4 neighbor nodes. This topology corresponds to the early days of Gnutella network, where the average number of neighbors is 3.4 [28]. In a real Gnutella network a part of nodes have an Internet access with higher bandwidth and they establish a larger number of connections to other nodes. Complex networks generated by Barabási-Albert model (BA network) [29] are scale-free and they represent such Gnutella networks.

In this section, in 2D mesh networks 20% of nodes issue a query each. In BA networks 100 queries are issued. The source node of a query is randomly chosen and the search target is another node randomly chosen. In all the following experiments there is one target object on a entire Gnutella network. The maximum number of hops (`TTL`) is 7, that is the default number of Gnutella protocol.

Figure 5, 6 and 7 show execution time of Gnutella simulation with 2D mesh, BA network ($m = 1$) and BA network ($m = 2$). In these experiments we did not adopt Time Warp [5], an optimistic synchronization protocol while Section IV premises it. Non-optimistic simulation can process a very limited number of message reception events at a time because it can process only the earliest events and a little more even with bounded lag technique [27]. In the worst case just one message reception is processed in an iteration of MapReduce over an entire data processing engine. But in these experiments it is not the case because communication patterns are simplified as follows. All the latest message reception events happen simultaneously and they are processed in an iteration because all the queries are issued simultaneously and the simulated communication latency is constant. More realistic simulation with precise time management requires optimistic simulation to process enough number of events in parallel.

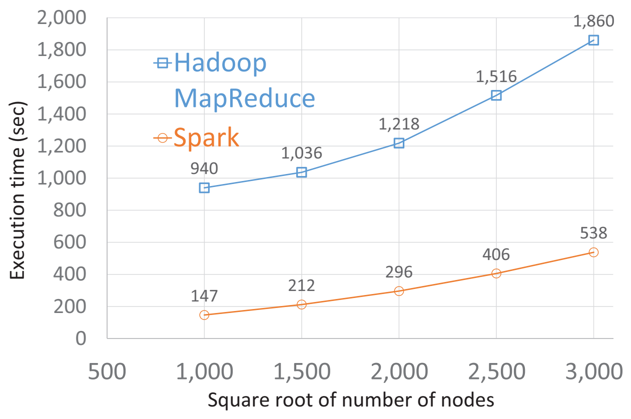


Fig. 5. Execution time of simulation of Gnutalla on a two-dimensional mesh network.

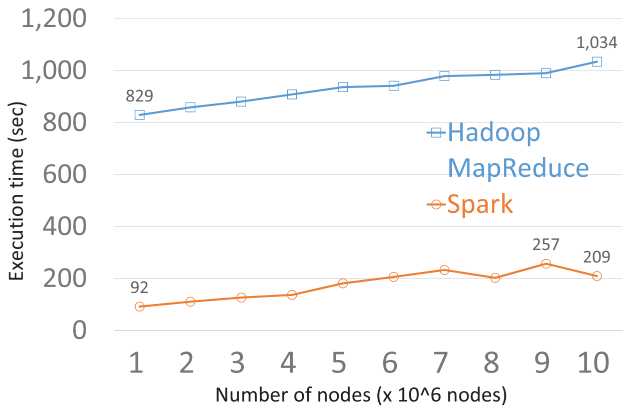


Fig. 6. Execution time of simulation of Gnutalla on a BA model ($m = 1$) complex network.

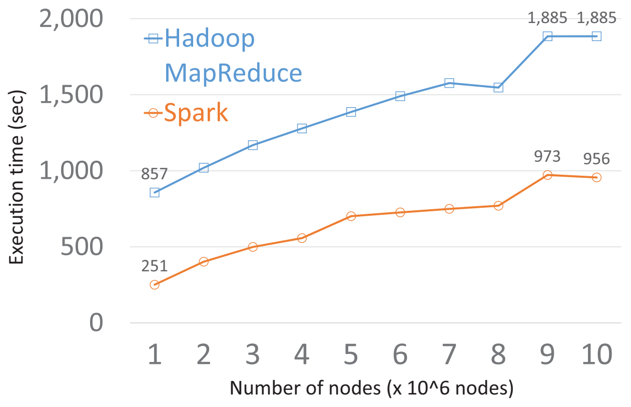


Fig. 7. Execution time of simulation of Gnutella on a BA model ($m = 2$) complex network.

Spark showed better performance than Hadoop MapReduce in all the experiments as it is designed to be. The minimum execution times of Spark and Hadoop MapReduce are about 90 second and more than 800 second as shown in Figure 6. The overhead of Hadoop MapReduce is certainly due to heavy processes between iterations, that include write-out and read-in on a distributed filesystem, and reinvocation of Java Virtual Machines. Spark removed them. Increase of execution time along scale of a simulated network is also larger with Hadoop

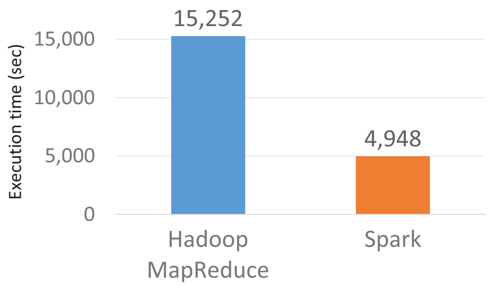


Fig. 8. Execution time of simulation of Gnutalla with a hundred million (10^8) nodes on 10 computers.

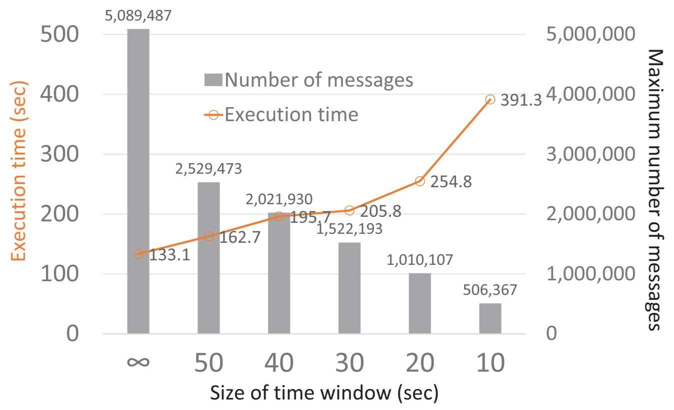


Fig. 9. Execution time and maximum number of messages with MTW and lazy cancellation.

MapReduce than Spark.

Our approach benefits from improvement of data processing engines from Hadoop MapReduce to Spark as the experiments showed. More efficient and fast engines bring about further performance improvement. Spark for The Machine (Spark4TM) [30] is an example of such coming engines. It achieved from 3.8 to 5.8 times performance of the original Spark in basic operations by utilizing a shared memory pool. It is possible for simulators developed from scratch to achieve higher performance than our approach. But such a simulator requires a large amount of development work and it is more difficult to be mature as software due to a limited number of users compared with the data processing engines.

B. Scalability evaluation

We confirmed that our simulators could process a hundred million (10^8) nodes with 10 computers. The network topology is BA network ($m = 2$). Figure 8 shows the execution times. The number of nodes 10^8 is enough to simulate today's running peer-to-peer systems. The largest one of them has a little less than 10^7 nodes [31], [32] to the best of our knowledge. Our target is the magnitude of 10^{10} because it is the number of Internet-connected things in near future [33]. It is still our challenge but realistic because a data processing engine could run on thousands of computers [25].

dPeerSim could simulate 5.75×10^6 nodes on a computer with 1.5 GB of memory and 84×10^6 nodes on 16 computers [10]. Our number 10^8 nodes on 10 computers is comparable with it though dPeerSim simulated Chord, not Gnutella.

IV. Otimistic parallel simulation

Most data processing engines adapt bulk-synchronous parallel (BSP) [34] or its more-restricted subset such as MapReduce as its processing model. In our models of simulation targets, a simulated communication between nodes, agents or partitions requires a real communication between workers. In BSP or its subset, inter-worker communication happens only synchronously as shuffle phase of MapReduce.

Without an optimistic synchronization protocol, a BSP-based data processing engine can process only the earliest events on a synchronization. In the worst case, an iteration of MapReduce processes only one message reception event over an entire engine with number of workers. Bounded lag technique [27] increases the number of events per a synchronization but it is expected not to be able to fulfill a massive number of parallel workers on an engine. Our target is thousands of workers. Therefore our simulators implement an optimistic synhronization protocol Time Warp [5].

Time Warp requires a simulator to keep data for rollbacks. Such data include a snapshot of the simulated system or a log of changes of the simulated system. Note that we adopted taking a snapshot. A simulator also has to keep anti-messages to notify other nodes of a rollback. A simulator stores them on memory or disks.

Spark is generally faster than Hadoop MapReduce as shown in Section III. One of the reasons is that it keeps data on memory from map phase to reduce phase while Hadoop MapReduce writes them down to disks. The capacity of memory is limited compared with disks. It is important for simulating a large-scale system to suppress the amount of data that reside on memory at the same time.

We test two categories of data suppressing techniques. The technique in the first category is lazy cancellation [35]. It suppresses rollbacks on other nodes by suppressing wasteful sendings of anti-messages. The only cost of the technique is judgement of the need for anti-message sendings. It is a completely local process within a node and the cost is negligible. Techniques in the second category are Moving Time Window (MTW) [6] and Adaptive Time Warp (ATW) [7]. They involve a trade-off between suppression of the data and other factors. The techniques reduce the number of anti-messages by limiting time window for event processing. Instead the number of iterations to complete a simulation increases. MTW uses a fixed size of time window and ATW adjusts the size dynamically. ATW can adjust other parameters such as the number of messages a node processes once. But in this paper it adjusts the size of time window as well as MTW.

In the following experiments, the simulator is the Spark version. Gnutella runs on a 2D mesh network with $1,000 \times 1,000 = 10^6$ nodes. $10,000$ queries are issued by randomly chosen node at random timing during 100 second. The search target is randomly chosen another node as well as Section III.

A. *Moving Time Window*

Figure 9 shows experimental results with MTW. The results are execution times and the maximum numbers of messages that reside on a simulator at the same time. Anti-messages are also counted in addition to normal messages that simulated nodes intentionally send. MTW is enabled with various sizes of time window and lazy cancellation is applied. Infinity of the size of time window means MTW disabled. The case of infinity shows 133.1 second as its execution time, that is a little less than 147 second in Figure 5. It seems to be reasonable because the differences from Figure 5 are less queries and lazy cancellation enabled.

As the size of time window decreases the maximum number of messages decreases. Instead execution time grows as expected. 50 second of time window reduces the maximum number of messages to the half of the case without MTW but the execution time grows only 22% from 133.1 second to 162.7 second. This result suggests that MTW can double the number of nodes if anti-messages are dominant in memory consumption. The actual quantity of the trade-off and merit of it depend on each simulation target but we could confirm that MTW provides a trade-off between simulation scale and execution time.

B. *Adaptive Time Warp*

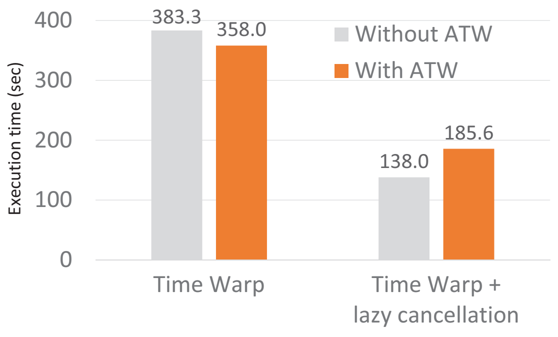
Figure 10 shows experimental results with ATW. Figure 10(a) shows execution times, (b) shows the number of iterations of MapReduce and (c) shows the maximum number of messages that reside on a simulator at the same time. The initial size of time window is 100 second. The size is reduced by half in case the number of messages that reside on a simulator is larger than $3,000,000$, that is a half of the maximum number of plain Time Warp shown in Figure 10(c). Otherwise 10 second is added to the size.

Lazy cancellation improved all the three kinds of numbers. For example, it reduced the execution time from 383.3 second to 138.0 second. Supposing lazy cancellation applied, the maximum number of messages decreased from 5.1×10^6 to 3.7×10^6 at the cost of the execution time, that increased from 138.0 second to 185.6 second. It is the same as MTW and expected. Increase of the number of iterations from 14.2 to 29.2 contributes increase of the execution time.

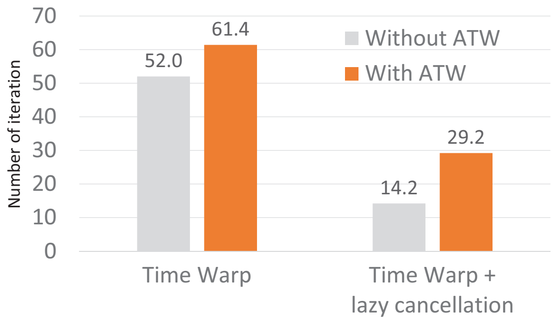
In these experiments ATW refers to the number of messages on an entire simulator to determine the size of time window. But it is possible for ATW to watch the number of messages on each worker. It may enable fine control of memory consumption over every worker. Investigating the best methodology or technique to control memory consumption is still part of open problems.

C. *Preliminary performance evaluation*

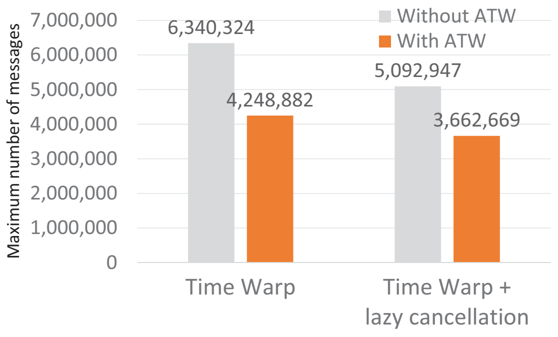
Here we try to evaluate performance of our Spark-based simulator by calculating the number of events processed in a second. We count events only in a simulation target itself and do not count events generated by a simulator such as null messages of Null Message algorithm and anti-messages of Time Warp.



(a) Execution time.



(b) Number of iterations.



(c) Maximum number of messages.

Fig. 10. Execution time, number of iteration and maximum number of messages with ATW.

In the experiments in Section IV-B, a single query causes 196 message reception events at most because Gnutella runs on a 2D mesh network and the maximum number of hops (`TTL`) is 7. The number of events per query is less than 196 if the query reaches bounds of the mesh. Precise counting concludes that $1,006,992/(196 \times 1,000^2) = 0.5138\%$ of the events decreases by reaching bounds of the mesh. A query reaches its target object at the rate of $113/1,000^2 = 0.0113\%$ at most because a query reaches 113 nodes including the issuing node itself. A query-hit yields a replying message and it can increase the number of events by 7. But the product of the number and the rate is small and we neglect it for a conservative approximation. Consequently, the number of events generated by $10,000$ queries is a little more than $196 \times 10,000 \times (100 - 0.5138)\%$. We can calculate performance of our simulator based on the number and the execution times in Figure 10(a).

dPeerSim processed 320,000 queries on 320,000 nodes running Chord protocol in 1 hour and 6 minutes [10]. Non-distributed PeerSim processed the same situation in 47.46 seconds. The average hop count of Chord with 320,000 nodes is about $1/2 \log_2 320,000 = 9.1439$. Thus the number of message reception events in the situation is 9.1439 hops/query $\times 320,000$ queries. Now we can calculate performance of dPeerSim and PeerSim.

Table IV shows performance of each simulator calculated as described above. The numbers are very preliminary because the simulation targets, performance of a single computer and the number of computers are different for each simulator. Each computer of the computer cluster used in our experiments is equipped with two 2.40 GHz Xeon processors and Gigabit Ethernet, and it was purchased in 2010. Each computer of the copmuter cluster used in dPeerSim and PeerSim experiments has two 3.00 GHz Xeon processors and 2+2 Gbps Myrinet in addition to Gigabit Ethernet, and it started to operate in the end of 2004. They are not different in an order of magnitude.

The performance of our simulator is about 20 times of dPeerSim and about a quarter of PeerSim. It is not very low even compared with non-distributed PeerSim while dPeerSim is slower than PeerSim in 2 orders of magnitude. Note that of course a parallel simulator such as dPeerSim can process a larger scale of system though it is not fast. An optimistic synchronization protocol Time Warp contributed the high performance of our simulator. Without it, the number of events that can be processed in an iteration of MapReduce is very limited and the number of iterations should increase much.

V. Conclusion

We demonstrated that a general-purpose data processing engine can be a solid base of a parallel simulator. Our implementations of the proposed simulator architecture run on widely used engines, Apache Hadoop and Apache Spark. Our simulators could process 10^8 nodes of Gnutella on 10 computers and the number of computers is expected to scale to thousands as the underlying engines scale. Optimistic simulation with Time Warp enabled our Spark-based simulator to achieve high performance, that is comparable even with a non-distributed simulator. Memory consumption, that is a drawback of Time Warp, could be mitigated by MTW and ATW in return for an increase of simulation time.

Future work includes scalability challenge with thousands of computers. It is also required to confirm how fault-tolerance features provided by the data processing engines work. More detailed evaluation should give an improved understanding of the proposed approach. Experiments for such evaluation include comparison with non-optimistic synchronization and MPI-based implementation. Simulating other targets such as car traffic is interesting while traffic simulation has a tendency to be compute-intensive compared with peer-to-peer simulation. Design of a parallel simulator on other models than MapReduce is stimulating. We have implemented message passing of distributed systems on a distributed graph processing system [36].

TABLE IV
Performance comparison of simulators.

Our simulator on Spark	
with lazy cancellation	1.41×10^4 events / second
without lazy cancellation	5.09×10^3
simulating Gnutella	
with 10 computers	
dPeerSim	7.39×10^2
simulating Chord	
with 16 computers	
PeerSim	6.17×10^4
simulating Chord	
with a computer	

Note that these numbers are very preliminary because simulation targets, performance of a single computer and the number of computers for each simulator are different.

Acknowledgments

This work was supported by JSPS KAKENHI Grant Numbers 25700008 and 16K12406. This work was also supported by the New Energy and Industrial Technology Development Organization (NEDO) especially in a study on future directions such as utilization of a shared memory pool.

References

[1] “Apache Hadoop,” http://hadoop.apache.org/.
[2] “Apache Spark,” http://spark.apache.org/.
[3] J. Dean and S. Ghemawat, “MapReduce: Simplified data processing on large clusters,” in *Proc. OSDI’04*, Dec. 2004.
[4] T. Klingberg and R. Manfredi, “Gnutella 0.6,” Jun. 2002, http://rfc-gnutella.sourceforge.net/src/rfc-0_6-draft.html.
[5] D. R. Jefferson, “Virtual time,” *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 7, no. 3, pp. 404–425, Jul. 1985.
[6] L. M. Sokol, D. P. Briscoe, and A. P. Wieland, “MTW: A strategy for scheduling discrete simulation events for concurrent execution,” *Proc. SCS Multiconference on Distributed Simulation*, vol. 19, no. 3, pp. 34–42, Jul. 1988.
[7] K. S. Panesar and R. M. Fujimoto, “Adaptive flow control in Time Warp,” in *Proc. PADS’97*, Jun. 1997, pp. 108–115.
[8] K. Shudo, Y. Tanaka, and S. Sekiguchi, “Overlay Weaver: An overlay construction toolkit,” *Computer Communications (Special Issue on Foundations of Peer-to-Peer Computing)*, vol. 31, no. 2, pp. 402–412, Feb. 2008.
[9] K. Shudo, “Overlay Weaver: An overlay construction toolkit,” http://overlayweaver.sf.net/.
[10] T. T. A. Dinh, M. Lees, G. Theodoropoulos, and R. Minson, “Large scale distributed simulation of p2p networks,” in *Proc. PDP 2008*, Feb. 2008, pp. 499–507.
[11] K. M. Chandy and J. Misra, “Distributed simulation: A case study in the design and verification of distributed programs,” *IEEE Transactions on Software Engineering*, vol. SE-5, no. 5, pp. 440–452, Sep. 1979.
[12] “ns-3,” https://www.nsnam.org/.
[13] Q. Bragard, A. Ventresque, and L. Murphy, “dSUMO: Towards a distributed SUMO,” in *Proc. SUMO2013*, May 2013, pp. 132–146.
[14] M. Behrisch, L. Bieker, J. Erdmann, and D. Krajzweicz, “SUMO - Simulation of Urban MObility: An overview,” in *Proc. SIMUL 2011*, Oct. 2011, pp. 63–68.
[15] R. L. Graham, G. M. Shipman, B. W. Barrett, R. H. Castain, G. Bosilca, and A. Lumsdaine, “Open MPI: A high-performance, heterogeneous mpi,” in *Proc. IEEE Cluster 2006*, Sep. 2006.
[16] T. Suzumura and H. Kanezashi, “Highly scalable X10-based agent simulation platform and its application to large-scale traffic simulation,” in *Proc. IEEE/ACM DS-RT 2012*, Oct. 2012, pp. 243–250.
[17] V. A. Saraswat, V. Sarkar, and C. von Praun, “X10: Concurrent programming for modern architectures,” in *Proc. PPoPP’07*, Mar. 2007, p. 271.
[18] T. Suzumura, S. Kato, T. Imamichi, and M. Takeuchi, “X10-based massive parallel large-scale traffic flow simulation,” in *Proc. X10’12 (in conj. with PLDI’12)*, Jun. 2012.

[19] C. Xie, Z. Hao, and H. Chen, “X10-FT: Transparent fault tolerance for APGAS language and runtime,” *Parallel Computing*, vol. 40, no. 2, pp. 136–156, Feb. 2014.

[20] Y. Liu, Y. Ren, L. Liu, and Z. Li, “A Spark-based parallel simulation approach for repairable system,” in *Proc. RAMS 2016*, Jan. 2016.

[21] T. Yu, M. Dou, and M. Zhu, “A data parallel approach to modelling and simulation of large crowd,” *Cluster Computing*, vol. 18, no. 3, pp. 1307–1316, Sep. 2015.

[22] A. Radenski, “Using MapReduce Streaming for distributed life simulation on the cloud,” in *Proc. ECAL 2013*, Sep. 2013.

[23] G. Pratx and L. Xing, “Monte Carlo simulation of photon migration in a cloud computing environment with MapReduce,” *Journal of Biomedical Optics*, vol. 16, no. 12, p. 125003, Dec. 2011.

[24] M. Hanai and K. Shudo, “Optimistic parallel simulation of very large-scale peer-to-peer systems,” in *Proc. IEEE/ACM DS-RT 2014*, Oct. 2014, pp. 35–42.

[25] “Hadoop Wiki - PoweredBy,” http://wiki.apache.org/hadoop/PoweredBy.

[26] “NTT DATA: Operating Spark clusters at thousands-core scale and use cases for Telco and IoT,” https://databricks.com/blog/2015/05/14/ntt-data-operating-spark-clusters-at-thousands-core-scale-and-use-cases-for-telco-and-iot.html.

[27] B. Lubachevsky, “Bounded lag distributed discrete event simulation,” *Proc. SCS Multiconference on Distributed Simulation*, vol. 19, no. 3, pp. 183–193, 1988.

[28] M. Ripeanu, A. Iamnitchi, and I. Foster, “Mapping the Gnutella network,” *IEEE Internet Computing*, vol. 6, no. 1, pp. 50–57, Jan. 2002.

[29] A.-L. Barabasi and R. Albert, “Emergence of scaling in random networks,” *Science*, vol. 286, no. 5439, pp. 509–512, 1999.

[30] “Spark on Superdome X previews in-memory on The Machine,” http://www.nextplatform.com/2016/04/11/spark-superdome-x-previews-memory-machine/.

[31] K. Jünemann, P. Andelfinger, J. Dinger, and H. Hartenstein, “BitMON: A tool for automated monitoring of the BitTorrent DHT,” in *Proc. IEEE P2P’10*, Aug. 2010.

[32] “BitMon,” https://dsn.tm.kit.edu/english/bitmon.php.

[33] “Gartner says 6.4 billion connected ”Things” will be in use in 2016, up 30 percent from 2015,” Nov. 2015, http://www.gartner.com/newsroom/id/3165317.

[34] L. G. Valiant, “A bridging model for parallel computation,” *Communications of the ACM*, no. 8, pp. 103–111, Aug. 1990.

[35] A. Gafni, “Rollback mechanisms for optimistic distributed simulation systems,” *Proc. SCS Multiconference on Distributed Simulation*, vol. 19, no. 3, pp. 61–67, Jul. 1988.

[36] M. Hanai and K. Shudo, “Simulation of large-scale distributed systems with a distributed graph processing system,” *Tech. Report of IEICE*, vol. 112, no. 173, CPSY2012-27, pp. 109–114, Aug. 2012, (in Japanese, not peer-reviewed).