SEMem: Deployment of MPI-Based In-Memory Storage for Hadoop on Supercomputers

Thanh-Chung Dao(✉) and Shigeru Chiba

Graduate School of Information Science and Technology,
The University of Tokyo, Tokyo, Japan
chung@csg.ci.i.u-tokyo.ac.jp, chiba@acm.org

Abstract. This paper reports our experiments to compare various deployment strategies of memcached-like in-memory storage for Hadoop on supercomputers, where each node often does not have a local disk but shares a slow central disk. For the experiments, we developed our own memcached-like file system, named *SEMem*, for Hadoop. Since SEMem was designed for supercomputers, it uses MPI for communication. SEMem is configurable to adopt various deployment strategies and our experiments revealed that a good deployment strategy was allocating some nodes that work only for in-memory storage but do not directly perform map-reduce computation.

1 Introduction

This research is motivated by the challenges of running data-intensive frameworks, such as Hadoop [15], Spark [16], and Flink [5] on supercomputers to meet their design. A supercomputer is a big machine consisting of thousands of high-performance nodes that are connected to each other through high-speed network. Not only running efficiently compute-intensive but also data-intensive workloads, for example, clustering and classification in machine learning and graph processing, are challenges on supercomputers.

In today's top supercomputers, each compute node does not have a local disk or is equipped only with a small size solid state drive (SSD) due to its relative costs and high failure rate. It makes writing and reading intermediate data during job execution (Hadoop/Spark) become a bottleneck since those data must be stored on the distributed central disk whose access time is considered slower five orders of magnitude than one of main memory [9]. To solve that problem, a natural approach is using in-memory intermediate data storage in order to avoid spilling to disk. We call that approach as *memcached-like* style [7] since in-memory storage could be at either a local or remote node.

How to deploy memcached-like file systems on supercomputers is not an easily answerable question and combination of memcached-like in-memory storage and Hadoop is not studied well as far as we know. A feature of supercomputers is that the number of compute nodes is often huge and supercomputer users submit a job to a job queue in order to request a number of nodes they need. When

F.F. Rivera et al. (Eds.): Euro-Par 2017, LNCS 10417, pp. 442–454, 2017.
DOI: 10.1007/978-3-319-64203-1_32

compute nodes are given to users, should we deploy in-memory storage on every node? should we allocate dedicated nodes from the given nodes for in-memory usage only? or should we allow using memory of remote nodes?

In this paper, we answer that research question by designing experiments of different in-memory storage deployment strategies for figuring out which one achieves a good performance when running data-intensive MapReduce applications on supercomputers. For experiments, we have designed our own memcached-like file system, named *SEMem*. Since Hadoop and Spark are good choices to run data-intensive applications even on supercomputers in aspects of productivity and maturity, we integrated our SEMem with the implementation of Hadoop. The deployment of SEMem is easily configurable and the intra-communication is through MPI, which is the de facto networking protocol on supercomputers. We examined the following deployment strategies:

1. SEMem is deployed as RAM disks where data is stored only in local memory.
2. SEMem is deployed on every node and data can be stored in remote memory.
3. SEMem is deployed only on dedicated nodes that are used only for storage.

Note that the original memcached software [11] is not a file system, does not support Hadoop directly, and uses TCP socket for data exchange. That is the reason why we developed SEMem from scratch.

Our experimental results reveal that allocating a group of nodes used only to store data in memory shows a good result in data-intensive applications with 7–10% improvement in comparison with deploying the memcached-like file system on every computation node. The benefit has been achieved in the context in which the ratio of the dedicated nodes to the total nodes is small (less than 12%). Note that there is no computation task running on those dedicated nodes. No computation task on the dedicated nodes might give a chance of allocating bigger memory space to in-memory storage and makes those nodes less busy.

Moreover, our experiments reveal that supercomputer centers can consider in-memory storage (e.g. SEMem) instead of installing SSD storage for flexible hardware resource configuration and supercomputer users can choose in-memory storage as an alternative of SSD. Compared to the central disk and SSD storage on TSUBAME supercomputer [13], our experimental results show that SEMem reduces execution time by 25% and 5%, respectively, on average when data size is bigger than 128 GB. In this experiment, SEMem runs on dedicated nodes and for fairness of this experiment, all three configurations use the same total number of nodes. In case of SEMem, some nodes are assigned to only storing data, whereas they are used for computation in the central-disk and SSD configurations. That means SEMem uses a smaller number of computation nodes.

In the context in which the ratio of the dedicated nodes to the total nodes is small, better performance from using fewer compute nodes demonstrates that Hadoop aims for data organization rather than computation.

2 Motivation

2.1 Running Hadoop MapReduce on Supercomputers

In today's top supercomputers, it is typical that there is no local disk on each compute node, but SSD is sometimes equipped [13]. Local disk on each node can be a point of failure and when it happens, it is difficult to fix. Instead of local disk, there is a shared storage called *central disk*. Disk I/O from and to the central disk is slow. SSD can be integrated in each compute node. Note that SSD might be a temporary storage where data will be deleted after job execution.

When running Hadoop MapReduce on supercomputers, writing/reading intermediate data is a performance bottleneck since it must be stored on the central disk. There are two main phases in Hadoop MapReduce workflow: mapping and reducing. Mapping tasks generate intermediate data that is written to a node's local disk and deleted after being sent to reducers. On supercomputers, the central disk is used to store that intermediate data instead of local disks. Reducing tasks fetch data from nodes where intermediate data is available and execute the reducing function. Figure 1 shows execution timeline of tera-sort application on TSUBAME supercomputer consisting of 256 mapping tasks and 128 reducing tasks. It reveals that in mapping tasks, writing time of intermediate data to the central disk (red color) is relatively long in comparison with the total execution time. In the figure, mapping tasks have shorter execution time than reducing ones. Using a local SSD helps improve writing/reading performance, but it is not always available.

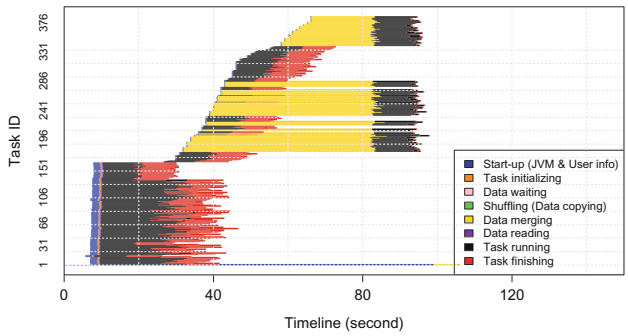


Fig. 1. Tera-sort running on TSUBAME supercomputer using its central disk: shorter running tasks are mappers, the longer ones are reducers.

2.2 In-Memory Approach and Deployment

In-memory storage is a natural approach that helps avoid disk spilling by keeping intermediate data in memory [4,12,16]. We call that approach as memcached-like style [7] since in-memory storage could be at either a local or remote node.

Memcached [11] is a distributed memory cache software that is used widely in web applications to speed up database accessing. A typical deployment of Memcached is installing its daemon on dedicated nodes used only for in-memory storage.

How to deploy memcached-like file systems on supercomputers is not an easily answerable question, even on commodity or off-the-shelf clusters, such as laboratory clusters and Amazon EC2. In this paper, we focus only on the supercomputer environment since the modern (or future) cloud systems [3] look like supercomputers. A feature of supercomputers is that the number of compute nodes are often huge and supercomputer users submit a job to a job queue in order to request a number of nodes they need. When compute nodes are given to users, should we deploy in-memory storage on every node? should we allocate dedicated nodes from the given nodes for in-memory usage only? or should we allow using memory of remote nodes? Moreover, combination of memcached-like storage and Hadoop is not studied well as far as we know.

3 Experiment Design

We answer the above research question by designing experiments of different in-memory storage deployment strategies for figuring out which one achieves a good performance when running data-intensive MapReduce applications on supercomputers. For experiments, we have designed our own memcached-like file system named SEMem. We examined the following deployment strategies:

RamDisk: SEMem is deployed as RAM disks where data is stored only in local memory. Memory size on each node is limited, data might not fit into memory. When out of memory happens, the task must be stopped or restarted on other nodes. For large-scale datasets, this approach is not feasible. Our expectation is this deployment strategy is fast in small-scale datasets.

Every Node: SEMem is deployed on every computation node and data can be stored in remote memory. We call nodes that are responsible for running applications *computation nodes*. Computation nodes are used to run a job's tasks, e.g. Mapping or Reducing tasks in Hadoop MapReduce. The SEMem daemon is run on all computation nodes and helps share a node's in-memory storage with other nodes. This strategy solves the problem of out of memory when data size is bigger than a node's in-memory storage size.

Dedicated Nodes: SEMem is deployed only on dedicated nodes that are used only for storage. We allocate a group of nodes that is only used to store data in their memory. There is no computation task running on that group of nodes. We call those nodes *memory nodes* and their memory *external memory*. The SEMem daemon is run only on each memory nodes. In this deployment strategy, we trade computation resource for data storage by using memory nodes only for keeping data. A question is that whether this approach is a complete waste of resources or not. No computation task on memory nodes gives a chance of allocating bigger memory space to SEMem and makes the memory nodes less busy.

SEMem Architecture: In this section, we describe our own in-memory storage called SEMem. SEMem is easy to be configured to change the deployment strategy of in-memory storage. SEMem is designed to tightly integrate with Hadoop MapReduce framework.

In original Hadoop MapReduce workflow (Fig. 2a), intermediate data generated by mapping tasks is written to a node's local disk. In case of supercomputer, instead of local disks, the central disk or SSD is used to store that data. Figure 2a shows how map output is copied from mappers to reducers. At the mapping side, map output is buffered in memory, and then when its size reaches buffer size, it is spilled to the local disk. The spilled files are merged into a final map output at the end of the mapping phase. There is a *shuffle server* (thread) running on each computation node waiting for fetching requests from reducers. When a request comes, the shuffle server will look up map output metadata, read again those data from the local disk, and send back to the reducer.

In the RamDisk deployment (Fig. 2b), map output is stored totally in memory to avoid writing to and reading from the local disk. Map output is kept in memory instead of spilling to the local disk when the buffer is full. After merging, map output is sent to the shuffle server. A memory space is created on each shuffle server to keep map output in memory. When a fetching request comes, the shuffle server looks up the memory space and is able to send back immediately to the reducer. The size of the memory space on each shuffle server is limited, so out

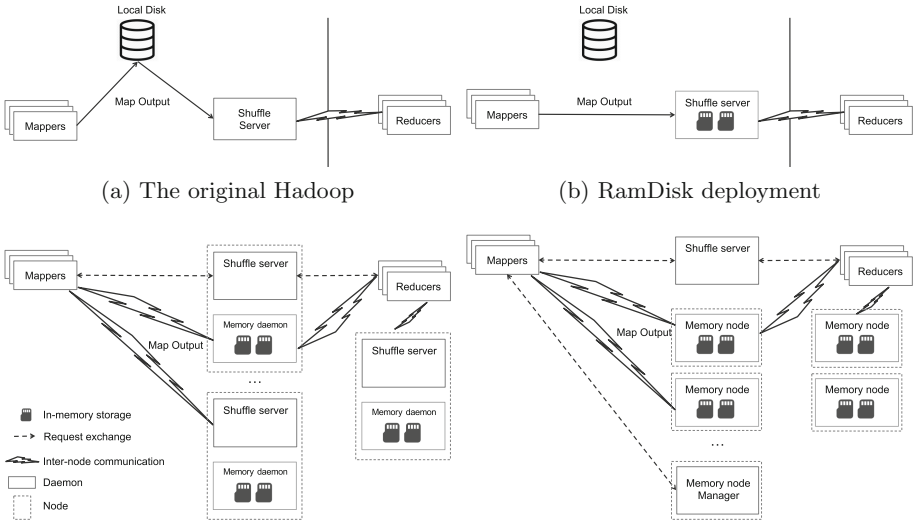


Fig. 2. SEMem configuration for each deployment strategy: a shuffle server is integrated into NodeManager of Hadoop framework. Memory daemon (or node) is a separate process running on each node of the every-node and dedicated-node strategies and responsible for in-memory storage. In the every-node strategy, the memory daemon and shuffle server are available on the same node.

of memory might happen during data fetching. For example, suppose that the memory space size is 8 GB and there are 20 map tasks running on the node. If each input data is 512 MB and running tera-sort application, the total size of map output is 10 GB. Those data cannot fit into the memory space. Although the map output can be deleted right after it is sent to the corresponding reducer, that out of memory error can happen at any time.

In Every-node deployment (Fig. 2c), SEMem daemon (called memory daemon) is run on the same node with the shuffle server. In contrast to RamDisk where data is stored at a shuffle server, data can be kept at any node's in-memory storage (memory daemon). Round-robin data affinity is used to store in-memory data since it makes data more distributed and helps speed up data fetching later. This kind of data affinity does not use the local memory for storage as long as possible since unbalanced intermediate data might be a performance bottleneck.

Dedicated-node deployment (Fig. 2d) is to create a group of nodes (called *memory nodes*) used only for storing map output and there is no map or reduce task running on those nodes. On supercomputer environment, those nodes are requested on the same job as the nodes used to run Hadoop applications. In Fig. 2d, the group of memory nodes consists of one node manager (master) and slave nodes (just called memory nodes). Node manager is responsible for data placement and monitoring memory left on each memory node. Since there is no task running on memory nodes, the amount of memory allocated can be bigger.

In comparison with RamDisk design, data exchange among nodes of SEMem in Every-node and Dedicated-node is more complex. First, the mapper requests its shuffle server to send map output. If the shuffle server still has empty space, data is sent then. If the shuffle server's memory space is full, the mapper requests the memory node manager to send map output. The node manager will find a relevant place and send back to the mapper. Then, the mapper starts sending map output to the specified memory node. When finishing sending map output, the mapper also informs the shuffle server about the location of map output on memory nodes.

What is the best strategy for storing map output in external memory? Should we distribute map output to as many memory nodes as possible or just store on a memory node and when it is full, the next memory node will be used? The former helps increase fetching throughput since reducers can request to more memory nodes. The latter is useful if map output is packaged and sent once to a reducer. In the current design, we have implemented a round robin scheduler on the memory node manager. Data will be distributed on all memory nodes.

Communication Protocol: SEMem uses MPI as communication protocol, which is the de facto protocol on supercomputers and exploits underlaying hardware such as remote DMA. This is a reason why we developed SEMem from scratch since Memcahced uses TCP/IP. SEMem uses the direct buffer memory of Java and MPI binding for Java [14] included in OpenMPI 1.7.5. It also uses our HPC-Reuse framework [6] to avoid MPI-Spawn or other methods to keep MPI connections available when a new process starts. To multiplex non-blocking com-

munication, SEMem runs a dedicated thread on every node to handle it. Note that our supercomputers do not support *MPI_THREAD_MULTIPLE* mode.

Storage Size: Storage size of SEMem depends on the number of memory nodes. To waste less CPU resources on memory node, it is necessary to minimize the number of memory nodes, although out of memory must not happen. In our current design, the number of memory nodes is estimated roughly based on the size of input data. Our assumption is that total size of map output is equal to one of input data. Therefore, the number of memory nodes chosen satisfies the condition that the total size of memory space on shuffle servers and the memory nodes must be bigger than the size of input data. It might not achieve the best performance of data fetching, but it helps prevent out of memory error. That error is more serious if it happens since the whole task will be restarted.

Minimizing Changes in Hadoop: Keeping the original source code (e.g. Hadoop or Spark) unchanged as much as possible is important to increase SEMem's capability in order to be integrated easily to other different data-intensive frameworks. In our implementation of Hadoop integration, Hadoop source code is changed with the below ratio:

- Line of code/total of Hadoop: 443/1,851,473
- Number of classes/total of Hadoop: 8/35,142

4 Experimental Results

In this section, first we compare three in-memory deployment strategies including *RamDisk*, *Every-node*, and *Dedicated-node* that are described in Sect. 3. We use three workloads in Puma benchmark suit [1]: WordCount, InvertedIndex, and SequenceCount. Second, we evaluate how fast SEMem (using the dedicated-node deployment strategy) is in comparison with the central-disk-based and SSD-based storage. There are three test configurations in this experiment: *Central-disk*, *SSD*, and *SEMem* storages. Tera-sort application is chosen for comparison since the size of intermediate data is big enough to show the bottleneck of the central disk.

Our experiments are conducted on Fujitsu FX10 supercomputer at The University of Tokyo [8] and TSUBAME supercomputer at Tokyo Institute of Technology [13]. A FX10 node is equipped with SPARC64 IXfx 1.848 GHz (16 cores) and 32 GB memory. FX10 nodes are connected with each other through Tofu interconnection [2]. The maximum throughput of that interconnection is 80 Gbps. A TSUBAME node has less cores (Intel Xeon X5670 2.93 GHz - 12 cores) and 54 GB memory. Each node is connected with Infiniband device Grid Director 4700. Another feature of TSUBAME is that 120 GB of SSD storage is available on each node.

SEMem has been implemented using Java since it helps integrate more easily with Hadoop, Spark, or Flink frameworks. We use Hadoop v2.2.0 for experiments, but any Hadoop version *2.x* can be integrated with SEMem. Spark can also use SEMem as an intermediate storage. Our Hadoop cluster consists of one resource manager (master) and 31 slave nodes. Each slave node can run at most four MapTasks or ReduceTasks simultaneously and maximum heap size of a JVM is 4096 MB of memory. Block size of Hadoop Distributed File System (HDFS) is 256 MB. In our experiments, all input data is stored in HDFS running on the central disk.

4.1 Deployment Strategies

This experiment is aimed to show which deployment strategy of in-memory storage shows a better performance in data-intensive applications. We compare three configurations on Fujitsu FX10 supercomputer: RamDisk, Every-node, and Dedicated-node. Moreover, for comparison of in-memory storage and spilling data to the central disk, we compare three strategies with another configuration called *Central-disk*. The largest size of input data is 120 GB. The number of reducers is set to 128 for all datasets. Note that running time is 0 (zero) denoting that out of memory happens and the corresponding job is stopped.

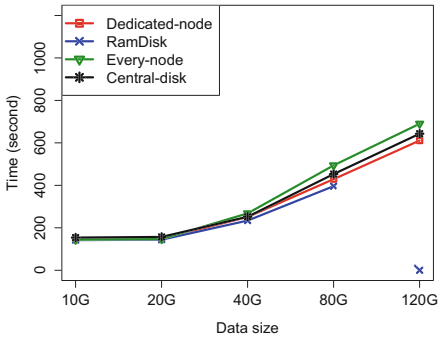
Regarding fairness of this experiment, all three configurations use the same total number of nodes. In case of dedicated-node, some nodes are assigned to only storing data, but they are used for computation as well in Ramdisk and Every-node. In this experiment, while we use 32 nodes for computation and 4 nodes for SEMem storage, 36 nodes are allocated for computation in Ramdisk and every-node configurations.

Figure 3 reveals that RamDisk is the fastest strategy, but out of memory happens when data size is bigger than 120 GB. Dedicated-node shows a better performance (10% on average) than Every-node when data size is bigger than 40 GB. Compared with central-disk configuration, Dedicated-node is always faster, especially 32% improvement in SequenceCount workload, whereas Every-node is slower in WordCount and InvertedIndex, but faster in SequenceCount 26% on average.

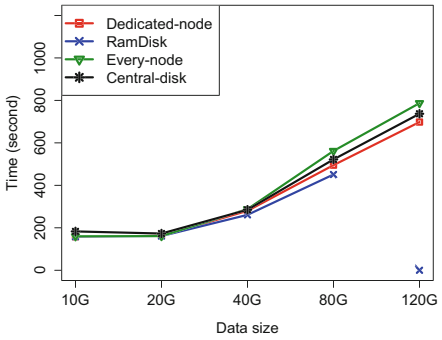
RamDisk is the fastest deployment strategy since data is stored at local memory and no inter-node communication. However, due to uneven distribution of input data, intermediate data is not fit into memory at some nodes. That is why out of memory happens. In WordCount and InvertedIndex workloads, Every-node is slower than central-disk configuration since more communication is required and computation nodes might be busier when SEMem daemon is run. By contrast, central-disk is the slowest configuration in SequenceCount because size of intermediate data is larger.

4.2 SEMem (Dedicated-Node) vs. Central-Disk (HDD) and SSD

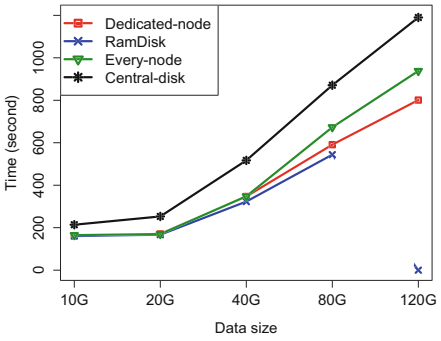
This experiment is aimed to show how fast SEMem is in comparison with the central-disk-based and SSD-based approaches. We compare three test configu-



(a) WordCount



(b) InvertedIndex



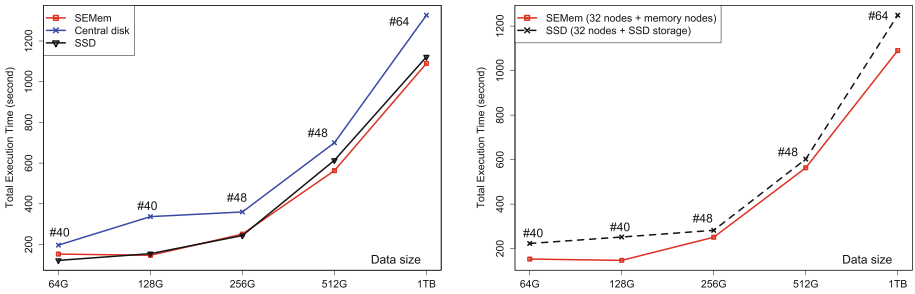
(c) SequenceCount

Fig. 3. Running time on different deployment strategies: zero at 120 GB at RamDisk denotes that out of memory happens and the corresponding job is stopped.

rations on TSUBAME: the central-disk-based Hadoop, SSD-based Hadoop, and our approach (SEMem). Tera-sort is used for comparison and input data is generated by tera-gen. The largest size of input data is 1 Terabyte. We run each experiment of a data size twice and calculate the average execution time. The number of reducers is fixed to 128 for all data sizes. The number of nodes (#) is denoted in the figure.

In this experiment, SEMem runs on dedicated nodes and for fairness of this experiment, all three configurations use the same total number of nodes. In case of SEMem, some nodes are assigned to only storing data, whereas those nodes are used for computation in the central-disk and SSD configurations. That means SEMem uses a smaller number of computation nodes.

Figure 4a shows that as the size of input data increases, SEMem is faster than central-disk-based storage, and close to SSD-based storage. Compared to central disk and SSD storage, SEMem reduces execution time by 20% and 5%, respectively, on average when data size is bigger than 512 GB. SSD storage has better performance when data size is less than 256 GB.



(a) SEMem vs. Central disk and SSD (b) Choosing SSD or compute nodes?

Fig. 4. SEMem vs. central disk and SSD

Choosing SSD or SEMem storage? at the supercomputer centers, when they design a new supercomputer, a question is that whether each node should be equipped with a SSD storage or not. SSD storage is considered difficult for maintenance and it can be a point of failure. For example, data on SSD storage of TSUBAME must be deleted manually after each job. If they do not want to install SSD, a question is whether SEMem can be an alternative. Moreover, if SSD is available and also become a paid resource on supercomputers, whether supercomputer users should buy SSD storage or choose to increase number of computation nodes in order to create SEMem. We design an experiment to answer those questions. This experiment is conducted on TSUBAME.

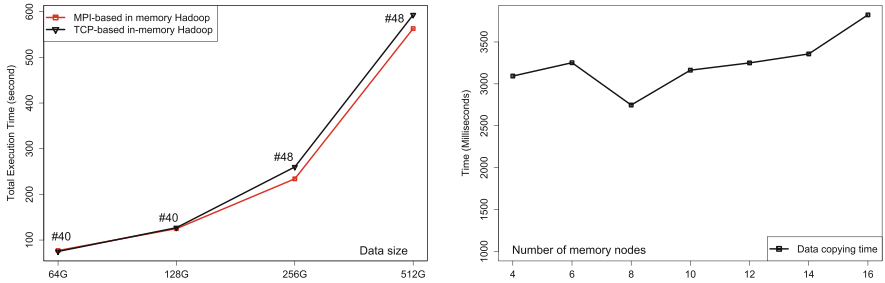
We keep the same number of computation nodes for both SEMem and SSD test configurations. We assume that SEMem and SSD is two types of resources that users can choose. SEMem has a defined number of nodes used for storage. Tera-sort is used for comparison and maximum input data is 1 Terabyte.

Figure 4b reveals that SEMem is always faster than SSD storage, especially 41% improvement at 128 GB of input data. When input size is 1 Terabyte, the improvement is 13% in comparison with SSD storage, but SEMem helps decrease the total execution time (1249 s) by 159 s. The figure shows that SEMem is feasible to become an alternative of SSD storage.

4.3 Communication Protocol

This experiment is aimed to check how fast our MPI implementation on SEMem is in comparison with TCP communication and whether the improvement achieved by using SEMem comes mainly from in-memory storing or fast MPI communication. We run the original Hadoop in memory and compare it with SEMem-based Hadoop. The original Hadoop uses TCP communication for data exchange between reducers and shuffle servers. This comparison is fair because SEMem is an in-memory storage, but uses MPI for data exchange. Tera-sort workload is used for comparison and the number of nodes (#) is denoted in the figure.

Figure 5a shows there is no difference when the size of input data is smaller than 128 GB. However, MPI-based Hadoop is faster 10% and 5%, respectively, when input size is 256 and 512 GB. Compared with Fig. 4a at the data size of 256 GB, MPI communication contributes less than 20% to the performance improvement by using SEMem. This experiment has proved that the main source of improvement (80%) comes from in-memory storing on memory nodes of SEMem.



(a) MPI-based vs. TCP-based in-memory Hadoop MapReduce

(b) Average data copying time when changing the number of memory nodes

Fig. 5. Evaluation of implementation issues

4.4 Storage Size of SEMem

This experiment is designed to measure performance impacts of storage size. SEMem's memory capacity can be configured based on number of memory nodes. We have conducted an experiment by keeping the same number of computation nodes and changing number of memory nodes. The purpose of this experiment is to check whether our current approach of finding the number of memory nodes is effective or not. We always run tera-sort application on 32 computation nodes. The number of memory nodes ranges from 4 to 16. The size of input data is 64 GB generated using tera-gen application. According to our estimation, required number of memory nodes should be 4–8 nodes. We measure data copying time (between shuffle servers and memory nodes and reducers) rather than total execution time. Figure 5b shows that when number of memory nodes is 8, data is copied fastest. However, if increasing number of memory nodes from 8 to 16, copying time is slower due to complexity in node management.

5 Related Work

There are several proposals of using in-memory storage in Hadoop, but they did not clearly describe and evaluate deployment strategies including location of in-memory instances and storage size. M3R [12] is an in-memory Hadoop engine

implemented using X10 programming language and M3R instances running on each node is responsible for in-memory storage by providing a shared heap-state. Although X10 supports where data is stored through *places* and *activities* operators, but the paper did not mention it explicitly and also have any evaluation. HaLoop [4] provides caching preferences, such as reducer input and output cache and mapper input cache, but intermediate data is only shared on the same node between jobs and deployment strategies are not relevant in this context. Spark [16] is a data-intensive framework and uses in-memory storage to improve performance compared with Hadoop. It proposed a programming model based on Resilient Distributed Dataset (RDD) and intermediate data is built and generated from RDDs. It is possible to choose a location for a RDD through *preferredLocations()* operator, but there was no evaluation of RDD deployment in the paper. Moreover, the supercomputer context makes our contribution unique.

Memcached software [11] is close to our implementation of SEMem since it is a distributed caching system. The combination of Hadoop and Memcached is a study topic. However, to the best our knowledge, there is no study on using Memcached to store intermediate data implicitly in Hadoop MapReduce workflow. We also evaluated advantages of that combination SEMem and Hadoop. Memcached uses TCP-based communication and RDMA-based Memcached [10] is an extension that speeds up internode communication by using RDMA. In our SEMem, we use MPI for communication among memory nodes. RDMA could be also enabled automatically in MPI communication on supercomputers.

6 Conclusion

In this paper, we have examined in-memory storage deployment strategies including RamDisk, Every-node, and Dedicated-node. For experiments, we have designed our own memcached-like file system called SEMem. Dedicated-node shows a good result in data-intensive applications with 10% improvement in comparison with Every-node strategy. Dedicated-node is motivated by a feature of supercomputers that the number of compute nodes is often huge and supercomputer users can request a number of nodes they need. There is no computation task running on that group of nodes.

SEMem is an easily configurable in-memory storage for different deployment strategies. It is tightly integrated with Hadoop, but Spark can use SEMem as preferred servers in the *preferredLocations()*. The performance of Hadoop MapReduce with SEMem should be the same as Spark since both of them are supporting in-memory storage. In our SEMem, however, users can choose the best in-memory storage deployment strategy for their applications. MPI communication on SEMem is an advantage in comparison with other Memcached-like software.

When we have only a fixed number of nodes, increasing the number of dedicated nodes according to the dataset size should affect the performance. Finding the best ratio of the dedicated nodes to the total nodes is our future work. Moreover, when the dataset does not fit into memory, SEMem needs to be adapted to use the central disk or SSD storage if available.

References

1. Ahmad, F., Lee, S., Thottethodi, M., Vijaykumar, T.: Puma: Purdue MapReduce benchmarks suite (2012)
2. Ajima, Y., Sumimoto, S., Shimizu, T.: Tofu: a 6D mesh/torus interconnect for exascale computers. Computer **11**(42), 36–40 (2009)
3. Amazon Web Services: High Performance Computing (2017). https://aws.amazon.com/hpc/
4. Bu, Y., Howe, B., Balazinska, M., Ernst, M.D.: Haloop: efficient iterative data processing on large clusters. Proc. VLDB Endow. **3**(1–2), 285–296 (2010)
5. Carbone, P., Katsifodimos, A., Ewen, S., Markl, V., Haridi, S., Tzoumas, K.: Apache flink: stream and batch processing in a single engine. Bull. IEEE Comput. Soc. Tech. Committee Data Eng. **36**(4), 28 (2015)
6. Dao, T.C., Chiba, S.: HPC-Reuse: efficient process creation for running MPI and Hadoop MapReduce on supercomputers. In: 2016 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid), pp. 342–345. IEEE (2016)
7. Fitzpatrick, B.: Distributed caching with memcached. Linux J. **2004**(124), 5 (2004)
8. FX10: User's Guide (2015). http://www.cc.u-tokyo.ac.jp/system/fx10/index-e.html
9. He, J., Jagatheesan, A., Gupta, S., Bennett, J., Snavely, A.: Dash: a recipe for a flash-based data intensive supercomputer. In: Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis, pp. 1–11. IEEE Computer Society (2010)
10. Jose, J., Subramoni, H., Luo, M., Zhang, M., Huang, J., Wasi-ur Rahman, M., Islam, N.S., Ouyang, X., Wang, H., Sur, S., et al.: Memcached design on high performance RDMA capable interconnects. In: 2011 International Conference on Parallel Processing, pp. 743–752. IEEE (2011)
11. Memcached: A caching system (2017). https://memcached.org
12. Shinnar, A., Cunningham, D., Saraswat, V., Herta, B.: M3R: increased performance for in-memory Hadoop jobs. Proc. VLDB Endow. **5**(12), 1736–1747 (2012)
13. TSUBAME: User'sGuide (2016). http://tsubame.gsic.titech.ac.jp/en/top
14. Vega-Gisbert, O., Roman, J.E., Squyres, J.M.: Design and implementation of Java bindings in Open MPI (2014)
15. White, T.: Hadoop: The Definitive Guide. O'Reilly Media Inc., Sebastopol (2012)
16. Zaharia, M., Chowdhury, M., Das, T., Dave, A., Ma, J., McCauley, M., Franklin, M.J., Shenker, S., Stoica, I.: Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing. In: Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation, p. 2. USENIX Association (2012)