Evaluation of Asynchronous MPI Communication in Map-Reduce System on the K Computer

Motohiko Matsuda  
RIKEN AICS  
Kobe, Japan  
m-matsuda@riken.jp

Shinichiro Takizawa  
RIKEN AICS  
Kobe, Japan  
shinichiro.takizawa@riken.jp

Naoya Maruyama  
RIKEN AICS  
Kobe, Japan  
nmaruyama@riken.jp

ABSTRACT

KMR is a map-reduce library designed to exploit the capacity and capability of supercomputers. To match the supercomputer environment, KMR runs on memory, communicates by MPI, and utilizes CPU cores by multithreading. In such a design, shuffling communication is implemented by MPI_Alltoallv. However, since many map-reduce systems have demonstrated to exploit overlaps of communication and computation in the literature, it is necessary to assess overlapping in KMR. For this purpose, two new shuffling modes are introduced in addition to the normal all-to-all mode. In the send-at-add mode, messages are sent when the buffer is full during mapping and reducing. In the fast-notice mode, a lightweight event notification is performed alongside the send-at-add mode messages. We chose TPC-H as a benchmark with a shuffle-heavy workload, and benchmarked with 1024 nodes on the K Computer. The results show only a slight improvement, at most 15% reduction of the execution time. Although the fast-notice mode achieves precise polling timing, it slightly degrades the performance. Analyzing the breakdown of completions of the send-at-add mode messages reveals that the mapping/reducing operation are quickly finished before the messages are exchanged. That is, there is a little chance of overlapping for TPC-H queries, per se. We conclude that the benefit of overlapping is marginal in the current MPI environment.

Categories and Subject Descriptors

D.1.3 [Software]: Concurrent Programming

1. INTRODUCTION

The map-reduce model, first publicized by Google’s MapReduce [3], is successful at abstracting the typical massive data processing in the cloud/cluster environment. However, similar data processing can be found in supercomputer applications as well, such as in the post-processing of scientific simulations or preparations for visualization. More over, some Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

EuroMPI/ASIA ’14, September 9-12 2014, Kyoto, Japan  
Copyright 2014 ACM 978-1-4503-2875-3/14/09 ...$15.00.  
http://dx.doi.org/10.1145/2642769.2642800

scientific computations are well suited to the map-reduce model. Typical examples are ensemble simulations and parameter surveys, which run many simulations, collect partial results, and generate new parameter sets. However, the underlying computing systems have very different characteristics from the cloud/cluster environment. Hardware is much more reliable, network is much faster, disks are not locally attached, and vast CPU resources are available but only for the relatively short duration of a job submission. Additionally, the user environment is very different and even the data processing needs to be high-performance. For example, Java is not officially available and a Fortran interface is mandatory.

To help scientists run data processing in an abstract way on supercomputers, we are developing KMR map-reduce library to provide similar abstraction, but with a more suitable implementation for supercomputers [6, 8]. KMR runs on top of MPI, and has some similarity to the precursor of MR-MPI [10]. MR-MPI also runs on top of MPI, but nevertheless targets the cloud/cluster environment. It processes large amounts of data using backing-stores of disks and is even single-threaded. KMR, however, is designed to exploit the capacity and capability of supercomputers, and it runs on memory and the mappers and reducers are multithreaded.

KMR runs on MPI, and because of this design, it inevitably works bulk-synchronously. The steps of map-reduce programs, that is, mapping, shuffling, and reducing are repeated virtually in lock-steps. Executions are synchronized at shuffling, which is essentially all-to-all collective communication, although there is no need for explicit synchronization. Whereas data processing applications in map-reduce are preferably performed by asynchronous communication, the model of MPI hinders its applicability.

The design of KMR naturally chose MPI_Alltoallv to implement shuffling, where a choice that was made at an early stage of its development. In contrast, in Hadoop or similar map-reduce systems, the data generated during mapping/reducing is sent on-the-fly, and thus, overlaps of communication and computation are inherent. Moreover, extensions have been proposed and demonstrated to improve communication performance in the literature [1, 2]. However, KMR cannot. Hence, in this study, we investigate whether overlapping by asynchronous communication can improve performance. This issue is not so clearly understood because there are obstacles to asynchronous communication in MPI, whereas shuffling by blocking collective communication is very efficient. Shuffling implemented by point-to-point com-
munication may suffer from buffer management and polling overhead. Our results in this paper show that the benefit does exist but is marginal. Moreover, it is important to evaluate the potential benefits of overlapping that are not restricted by the current MPI definitions and implementations. It is because the cost of message polling in large-scale supercomputers is expected to be reduced in the future. To address this, a lightweight event notification using RDMA (Remote DMA) was implemented in this evaluation.

In this paper, we investigate the benefit of overlapping communication and computation in map-reduce in the supercomputer environment. In the following, Section 2 introduces KMR, Section 3 describes the asynchronous communication implementation, and then Section 4 explains the TPC-H benchmark and details the experimental results. The following Sections 5 and 6 review related work and give concluding remarks, respectively.

2. MAP-REDUCE ON THE K COMPUTER

2.1 Design Overview

KMR is a map-reduce library designed to exploit the capacity and capability of the K Computer [6, 8]. When designing KMR, we decided to run KMR on memory, because it is prohibitively slow to access external storage constantly. In its on-memory design, the problem size is limited by the available memory size. However, this limit is a parameter controlled by the number of nodes of a job. Although the allocation of larger jobs looks unusual at first glance, requesting the number of nodes to fit the problem size is a very rational solution that does not waste computing resources with slow accesses to external storage.

On the K Computer, 1,000 nodes are the norm, and up to 36,000 nodes can be used without restriction. Even larger jobs with over 80,000 nodes can be run on occasions, scheduled a week per month. Thus, we have a wide range of choice. For small jobs, 1,000 nodes with 16 TB of memory is adequate. The capacity can be expanded to 1 PB, which matches the amount of a fairly large storage system. In other words, problems whose the data size exceeds this volume are not appropriate for supercomputers. They should be run elsewhere in the cloud/cluster environment, where it is often acceptable to be limited by the throughput of the storage systems.

The mapping and reducing in KMR are multithreaded, and the map and reduce functions are invoked by OMP threads. It can be turned off by option, when the map and reduce functions are multithreaded by themselves. Additionally, the sorting, frequently used inside of shuffling and reducing and used to order the final results, is multithreaded. The OMP task directives are applied to quicksort to make it multithreaded.

2.2 Basic Map-Reduce

The map-reduce model works on a data structure called key-values and consists of three main operations. A mapping operation converts each key-value independently. A reducing operation aggregates the key-values by selecting ones that have the same keys, and may generate new key-values. A shuffling operation performs communication gathering the same keys in advance for reducing. Whereas most map-reduce systems provide shuffling as an implicit operation, KMR provides shuffling as a separate primitive operation, where it shares the design with MR-MPI [10]. Having shuffling as a separate operation enables the local combiner, which reduces only key-values locally available in each node, to be performed without defining it as an additional operation. Although locality is abstracted out in the map-reduce model, keeping locality in mind is important in practice for efficiency. It becomes obvious by having a separate shuffling operation.

As implemented on memory, the mapping, shuffling, and reducing operations can be treated as programming constructs that are quickly executed. As such, many utility operations in KMR are implemented as a combination of map-reduce operations. Even the sorting is implemented by map-reduce operations with the help of a local sorter that only sorts keys within a node. This programming style is a kind of data-parallel programming in some languages, but it is only provided as a library.

2.3 Other Supports

KMR also provides other utility functionality in addition to basic map-reduce, including command pipelining under map-reduce, spawning single or MPI processes, aggregate file reading, and check-pointing to tolerate some faults. Pipelining, or streaming, feeds the output of external programs to mapping/reducing or vice versa through Unix pipes. Similarly, spawning allows external programs to run as map functions on the spawned processes. KMR supports mapping in the master-slave style via spawning. Aggregate file reading helps improve the access speed for large files. The file system of the K Computer is FEFS (Fujitsu Exabyte File System), an extension of Lustre File-System, that consists of over 5000 disk units. At this size, the disk units naturally shape locality groups to compute nodes, and aggregate file reading exploits this locality. Check-pointing provides automatic saving and restoring of key-value data. As the error rate of the K Computer is very low, however, it can be used to carry on long-running map-reduce jobs without fear of an abrupt termination caused by the expiration of a time allotment.

3. ASYNCHRONOUS COMMUNICATION

3.1 Shuffling Communication

Shuffling implemented by blocking collective communication MPI_Alltoallv does not incur a high performance penalty, because all nodes ultimately have to synchronize at the beginning of reducing that comes after shuffling. Reducing in map-reduce can only be started after all data items have been gathered according to the keys. This is because the reduction operation takes a whole array of key-values with the same keys, that means reducing is not defined as a commutative associative binary operation. Thus, shuffling as blocking collective communication does not introduce extra synchronization and fits well into the bulk-synchronous design of KMR.

3.2 Shuffling during Mapping/Reducing

Since shuffling is a separate operation in KMR, the possibility of overlaps has not been considered so far. This is partly because shuffling by blocking collective communication in MPI is very efficient. We refer here to the normal mode of shuffling by MPI_Alltoallv as the all-to-all mode.
A new send-at-add mode is implemented, that buffers key-value data and sends it as a message when the buffer becomes full. Key-values in KMR are stored in a KMR_KVS structure. To implement the send-at-add mode, a variant of KMR_KVS is implemented. It is modified to check the buffer at the addition of a new key-value performed by kmr_add_kv to determine if a message needs to be sent. The sends are posted by MPI_Isend, and the receives for incoming messages are posted by MPI_Irecv. They are periodically polled for by MPI_Testsome or MPI_Waitsome, depending on the need for completion. Since the polling should neither be too often nor too seldom, the polling by MPI_Testsome is performed when messages are sent. This has been found to be a fairly good heuristics.

The buffering size, which determines when the temporarily stored data is sent as a message, is an important parameter. In the benchmark, it is 64 KB. When the generated data amounts to this value, a message is created and sent. The amount of 64 KB was chosen because it is 1/1024 of 64 MB, the unit size of memory allocation in the normal all-to-all mode. Since the buffering is for each node in the send-at-add mode and there are 1024 nodes in the benchmark, the buffering size was chosen to be 1/1024.

3.3 Lightweight Event Notification

As supercomputers increase in scale, frequent polling becomes prohibitive. The appropriate polling frequency naturally depends on the problem, varying by the rate of data generation. More importantly, it is a remote-side condition that cannot be known without communication. The K Computer has an RDMA (Remote DMA) capable network system, and data transfer in MPI is implemented by RDMA. Furthermore, the MPI system on the K Computer is equipped with an extension to directly issue RDMA operations. This capability is exploited here to inform the remote side about a message arrival event.

The fast-notice mode is a modification to the send-at-add mode. The event notification by RDMA works by setting a flag in memory alongside of sending a message. Checking a flag in memory is cheap, and this allows frequent checking for incoming messages. Since the flag is only a hint and the true message arrivals are checked by the MPI primitives, a simple one word RDMA put operation suffices.

Note that the problem here is the cost of polling, rather than the well-known progress issue. Progress mainly concerns of a latency, while polling cost concerns computation time. Polling cost basically increases proportionally to the number of MPI processes because the request objects are created and handled for the processes.

3.4 Stream Communication in MPI

The send-at-add mode is equivalent to implementing a channel of a slow stream over MPI communication. In the implementation, the messages are sent one-by-one for each destination node, where a new message can be posted by MPI_Isend only after the previous one has been completed by MPI_Testsome. This method is parsimonious with the usage of MPI communication, but it is necessary to avoid overwhelming the MPI system and keep it from running out of resources caused by the unrestricted issues of sends. Bounding the resource usage is important when the number of nodes becomes large. Although this parsimonious implementation may limit channel utilization, there are many concurrent channels (1023 for each node in the 1024 node benchmark), and thus, it should be enough. In addition, as pointed out by Sur et al. [12], an RDMA operation is suitable for overlapping communication without incurring the progress problem, and thus, we assume the underlying MPI implementation appropriately achieves ceaseless communication.

4. EVALUATION

4.1 TPC-H Benchmark

The TPC-H benchmark [4, 7, 14] is used as a shuffle-heavy workload in our evaluation. It consists of SQL queries for the decision support system (DSS) of a simulated commercial company. It has a fair amount of communication and computation, and is used to benchmark the map-reduce framework [5]. Since TPC-H was only chosen for a workload, the evaluation does not follow the rules of the benchmark.

Table 1 shows the number of records in the eight tables of the 1 TB TPC-H dataset. TPC-H provides a generator for the table data, enabling benchmarking with arbitrary problem sizes. In the evaluation, the 1 TB dataset, that takes about a minute to finish on the K Computer, is adequate and used for benchmarking.

While SQL defines many datatypes, they were translated to only the three datatypes in the evaluation: long integers, double floats, and variable-length strings. To represent the records, we defined an n-ary tuple structure which can store the above datatypes as byte sequences in the key or value part of a key-value.

The five queries listed in Table 2 were selected out of the 22 queries defined in TPC-H for this evaluation. The choice was basically random, but they were selected considering

<table>
<thead>
<tr>
<th>Query</th>
<th>Description and Schedule of Joins</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q7</td>
<td>Volume Shipping ((L \bowtie (C \bowtie O)) \bowtie ((N \bowtie N) \bowtie S)))</td>
</tr>
<tr>
<td>Q9</td>
<td>Product Type Profit Measure ((L \bowtie O) \bowtie ((N \bowtie S) \bowtie (P \bowtie PS))))</td>
</tr>
<tr>
<td>Q10</td>
<td>Returned Item Reporting ((C \bowtie N) \bowtie (L \bowtie O)))</td>
</tr>
<tr>
<td>Q13</td>
<td>Customer Distribution (C \bowtie O)</td>
</tr>
<tr>
<td>Q21</td>
<td>Suppliers Who Kept Orders Waiting ((L \bowtie ((L \bowtie (N \bowtie S)) \bowtie O)))</td>
</tr>
</tbody>
</table>

(\(\bowtie\) indicates a join)

<table>
<thead>
<tr>
<th>Table</th>
<th>Records</th>
</tr>
</thead>
<tbody>
<tr>
<td>R (region.tbl)</td>
<td>5</td>
</tr>
<tr>
<td>N (nation.tbl)</td>
<td>25</td>
</tr>
<tr>
<td>S (supplier.tbl)</td>
<td>10,000,000</td>
</tr>
<tr>
<td>C (customer.tbl)</td>
<td>150,000,000</td>
</tr>
<tr>
<td>P (part.tbl)</td>
<td>200,000,000</td>
</tr>
<tr>
<td>PS (partsupp.tbl)</td>
<td>800,000,000</td>
</tr>
<tr>
<td>O (orders.tbl)</td>
<td>1,500,000,000</td>
</tr>
<tr>
<td>L (lineitem.tbl)</td>
<td>5,999,989,709</td>
</tr>
</tbody>
</table>

Table 1: Number of records of the 1 TB dataset of TPC-H.
the characteristics analysis given in [15] although the criteria are for a micro-processor design and are not relevant to the workings of map-reduce. Q9 was selected, because it is explained in detail and is said to be one of the most complex queries in [5]. The queries were parameterized, and we used the parameters given for the validation runs in TPC-H.

Since KMR does not have an SQL front-end, the queries were converted to map-reduce programs by hand. The translations were straightforward. Selections were converted to mapping and joins were converted to reducing. The scheduling of query decomposition is also listed in Table 2. The decomposition shows the ordering of applications of joins. For example, Q10 is decomposed to make a join between the tables, first for C and N, second for L and O, and then between the results of the first and the second joins. The decomposition of Q9 is taken to be the same as that given in [5]. The only strategy/heuristics used in the translations was to prefer the first column as the candidate of a join. This corresponds to the common strategy of choosing the primary keys, although the tables of this benchmark do not have the primary keys. The choice of keys should have no significant impact on the performance, because the TPC-H benchmark simulates data analysis. The joins mostly result in fewer records, except for a few trivial cases, for example, when four records result from joining two by two records.

The joins were implemented in an ordinary way as in other map-reduce systems. The value parts of the key-values are tagged as either the left or the right corresponding to which arguments to the join. Then, the tables are then shuffled and merged as a direct sum. The key-values with matching keys are reduced by first discerning the left and the right then making a direct product of the two sets.

4.2 Benchmark Settings

The number of nodes used in the evaluation was 1024, and the dataset size was 1 TB as mentioned above. The runs were ordinary ones, that is, runs did not request a specific topology to the allocation of nodes, nor specify options. The only exception was the setting of the environment variable XOS_MM_L_ARENA_FREE=2, that disables returning memory pages to the pool in the OS kernel at deallocation. Nodes in a job are allocated exclusively and hence returning memory pages is useless. However, the effect of this option was not evaluated.

The evaluation excludes the time to read the tables from the files. Since the 1 TB dataset is small enough for 1024 nodes, the files were first loaded and kept in memory during benchmarking. However, it does include the time to scan the text data and generate records of table rows. This is mainly because we were only interested in on-memory performance, and as a result, we decided to run the benchmark on the micro-queue on the K Computer, which is for program development and has a short turnaround time, but is restricted in disk access throughput. Additionally, by reducing the running time, a set of benchmarks may be run in a single job, which in effect avoids disturbance caused by the allocation of the nodes. In this benchmark, the table files were split into the number of nodes (1024) prior to the runs. As a note, reading 1 TB files took about 30–60 seconds.

4.3 Experimental Results

Figure 1 shows the relative performance of the methods, and Table 3 shows their time in seconds. The label All-to-all refers to shuffling by MPI_Alltoallv, which is our baseline case. The labels Send-at-Add and Fast-Notice refer to the cases communicating during mapping or reducing. In the Fast-Notice case, it uses the lightweight event notification using RDMA to tell a remote node about a new message. The measurements were repeated three times, and the figures are the best of all runs.

The results show some improvements, but the reductions in time are 15% at most and marginal. The lightweight event notification degrades the performance slightly, although it handles receives of messages with more precise timing (shown in the following analysis). The reason for this degradation is that the preciseness of the notification causes MPI_Testsome to lose chances to complete multiple requests at once, and forces requests be completed one-by-one. The polling time by MPI_Testsome takes approximately 20 μsec on average in the benchmark, when it is called with requests typically populated by 1023 sends and 1023 receives and it does not hit any requests. The cost of 20 μsec is relatively large compared with the computation speed.

Table 4 shows the time taken for shuffling in the all-to-all mode in seconds. To measure the time, the program was modified by inserting barriers at the beginning and end of shuffling to exclude the wait time due to load imbalance. Q13 has relatively little shuffling, about a second, while Q7 and Q21 take about 5 seconds, and Q9 and Q10 take about 10 seconds. The time to join N x N in Q7 was not measured because table N is small and known to be local.

Figure 2 shows the breakdown of percentage of receive completions: those completed during mapping/reducing, those completed after mapping/reducing but before the results were needed, and those waited for in MPI_Waitsome when the results were needed. The bars labeled by QN” indicate cases of Send-at-Add and the bars labeled by QN” cases of Fast-Notice. The count of receives inside mapping/reducing is the most important metric. From the graph, we can see that only a fraction of messages were received inside mapping/reducing (less than 3%). The fast-notice has shown
and hence our runs did not follow the benchmark rules.

that we are not interested in the TPC-H benchmark itself, larger problem size, but uses many more processors. Note results because their performance figures are for runs with external storage on a smaller system with a smaller problem size. Our benchmark runs in significantly shorter time for a external storage on a smaller system with a smaller problem size. The precursor of map-reduce systems in MPI is MR-MPI [10]. We confirmed that MR-MPI runs comparably well to KMR when the use of external storage is in effect disabled by setting the size of on-memory buffers to appropriately large. However, the lack of thread support makes the use of CPU cores the user’s responsibility. With the use of external storage, it is very likely that communication and computation will be overlapped in MR-MPI, because the storage systems typically have a throughput that is in orders of magnitude smaller, and the speed of mapping and reducing restricted by this causes some CPU cores to idle. To attain sufficient overlapping, however, some method would be needed to poll both of I/O and communication simultaneously and efficiently, and this could be challenging. The lightweight event notification by RDMA could be useful.

Phoenix [11] is an map-reduce system for shared memory machines. The usage of threads is similar, but communication is implicit and the scale of the system is limited to several CPU cores. There are many reports on Hadoop-based database processing including above mentioned Hive [13] and Pig [9]. Map-Reduce-Merge [16] optimizes a direct sum operation to merge two sets of key-values for joins, and it was evaluated using TPC-H. ConMR [17] exploits the reuse of the databases by concurrently running query processing.

6. CONCLUDING REMARKS

The effectiveness of shuffling during mapping and reducing, which makes use of overlaps of communication and computation in the cloud/cluster environment, is marginal, with at most 15% of reduction of execution time in a map-reduce system with on-memory operations. The results are largely because the operations of selections and joins in the TPC-H benchmark took a relatively short time, and the chances of overlaps do not exist in the first place. To make this result independent from a particular implementation of MPI and a map-reduce system, we have implemented a lightweight event notification that makes the timing of polling more precise and reduces the frequency of wasted polling. Although the mechanism was shown to achieve its purpose, it degrades the performance.

As a result of this benchmark, the default shuffling mode is set to use MPI_Alltoallv in KMR. This is because the effect is limited, even in a shuffle-heavy workload such as TPC-H, and the overlaps are not needed in non-shuffle-heavy workloads. In this regard, the non-blocking collectives of MPI-3.x do not improve the performance, because message contents are generated on-the-fly. However, we anticipate returning to this issue, when the underlying communication

Table 4: Shuffle time in seconds.

<table>
<thead>
<tr>
<th>Queries</th>
<th>(L ( \bowtie_0 ) (C ( \bowtie_1 ) O))</th>
<th>(N ( \bowtie_2 ) (N ( \bowtie_3 ) S) ( \bowtie_4 ) O)</th>
<th>( \bowtie_0 )</th>
<th>( \bowtie_1 )</th>
<th>( \bowtie_2 )</th>
<th>( \bowtie_3 )</th>
<th>( \bowtie_4 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q7</td>
<td>2.102</td>
<td>1.023</td>
<td>1.681</td>
<td>-</td>
<td>0.065</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Q9</td>
<td>10.513</td>
<td>4.988</td>
<td>0.097</td>
<td>0.466</td>
<td>0.999</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Q10</td>
<td>1.454</td>
<td>11.994</td>
<td>0.949</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Q13</td>
<td>C ( \bowtie_0 ) O</td>
<td></td>
<td>1.112</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Q21</td>
<td>2.572</td>
<td>3.714</td>
<td>2.581</td>
<td>0.026</td>
<td>0.556</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 2: Breakdown of completion of receives (%).

improvements, but is still limited to 20%. Most of the messages are waited for when finishing the communication. This indicates that the mapping/reducing operations in TPC-H are quickly finished compared with the communication. This is because the selections and joins of TPC-H are not computationally complex.

Note that the breakdown only shows the number of receives and does not reflect the actual wait time. Even if the number of the receives waited for is small, it could potentially have a large impact on the execution time. Unfortunately, the breakdown does not satisfactorily explain the performance. Although Q9 should be as good as Q10, given the breakdown, but the performance is not so. We checked that the performance is also very dependent on the block sizes of buffers. This is naturally because, as noted above, only a few messages are handled in mapping/reducing, and thus increasing or decreasing the number of messages matters.

5. RELATED WORK

Yuntao benchmarked Hive with TPC-H in [4], and Jie et al. benchmarked Pig (and Hive) with TPC-H in [7]. Both papers focus on the optimization of SQL queries in their environments. The results cannot be compared with our results because their performance figures are for runs with external storage on a smaller system with a smaller problem size. Our benchmark runs in significantly shorter time for a larger problem size, but uses many more processors. Note that we are not interested in the TPC-H benchmark itself, and hence our runs did not follow the benchmark rules.

As a result of this benchmark, the default shuffling mode is set to use MPI_Alltoallv in KMR. This is because the effect is limited, even in a shuffle-heavy workload such as TPC-H, and the overlaps are not needed in non-shuffle-heavy workloads. In this regard, the non-blocking collectives of MPI-3.x do not improve the performance, because message contents are generated on-the-fly. However, we anticipate returning to this issue, when the underlying communication
layers will have much lower overhead such as the ones based on remote get/put operations for PGAS languages.

Acknowledgments
Part of these results were obtained using the K Computer at RIKEN AICS.

7. REFERENCES