Data-Aware Task Dispatching for Batch Queuing System

Xieming Li and Osamu Tatebe

***Abstract*—This paper describes a scheduling method focusing on exploiting the local access of a nonuniform storage-access file system. In our approach, the file access cost is calculated and combined with the CPU load average into a comprehensive value, which will be used as the standard for scheduling. We evaluated our approach in comparison with the original Torque scheduler using three benchmarks: thput-gfpio, readgf, and BLAST benchmarks. For thput-gfpio, the read throughput showed a 3.59 times boost, whereas for readgf, the total execution time was reduced to about 1/10th of the original value. Finally, using the BLAST benchmark, the total execution time was reduced by 33%.**

***Index Terms*—Batch queuing system, data-aware task scheduling, file locality, Gfarm, nonuniform storage-access (NUSA), scheduling.**

I. Introduction

SIMULATION technology has become one of the most important branches for scientific computations in energy physics, genomics, astronomy, and other fields. As the granularity of simulations is increasing, the demand for handling larger data sets is growing accordingly [1], [2]. Such requirement is met by a parallel file system that bundles multiple storage devices and achieves a high I/O performance through simultaneous access. File systems such as GPFS [3], Lustre [4], pNFS [5], and PVFS [6] require dedicated storage nodes connected to compute nodes using a storage area network. The performance in accessing files on such file systems can be considered rather "uniform" because each access has to travel through the network (Fig. 1, left). However, this approach requires a high-bandwidth network between the compute and storage nodes, which could incur a relatively high cost, especially for a large-scale cluster.

In contrast, file systems such as Gfarm [7] and Google File System [8], which federate local file systems on compute nodes, have been proposed. In this type of file system, the access performance can be considered "nonuniform" because the compute node can now access the files on its local drive as well as files on the remote node through the network (Fig. 1, right).

In addition, in a large-scale computing environment, it is rare for a single task to occupy all computing resources. More often, multiple jobs are executed in parallel, where a task scheduler is utilized to control the usage of hardware resources such as the CPU cycles, memory, and disk space. Some widely deployed task schedulers are used, including Torque [9], Condor [10], Platform Load Sharing Facility (LSF) [11], [12], and Open Grid Scheduler [13]. These schedulers are suitable for a uniform file system mainly because there is no need to consider the file location at the task dispatch. The access performance is nearly identical regardless of which compute node the task is dispatched to. However, such schedulers may not be ideal for a nonuniform storage-access file system because, if a task is dispatched without consideration of the file allocation, it might be assigned to a node where the file cannot be accessed locally, thereby leading to a drop in performance. In this paper, we describe a scheduling strategy for a nonuniform storage-access file system that emphasizes the exploitation of high-performance local access. In our approach, the cost of a file access is calculated and combined with the CPU load average into a comprehensive value. This value will then be used as the standard for scheduling.

The contributions of this paper are as follows.

1) A design of the data-aware dispatch (DAD) algorithm, which is applicable for various nonuniform storage-access file systems, is proposed.
2) DAD is implemented based on the Torque scheduler.
3) DAD is evaluated using three benchmarks and shows a significant improvement.

The rest of this paper is organized as follows. Section II introduces previous research related to the present topic. Section III provides an overview of Gfarm and the Torque scheduler from a task-scheduling perspective. Next, Section IV describes the design of data-aware task scheduling for Gfarm. Section V then presents an evaluation based on thput-gfpio, readgf, and BLAST benchmarks. Finally, we provide some concluding remarks in Section VI.

II. Related Research

Scheduling algorithms have been widely studied from various perspectives. In this section, we focus on those works emphasizing file allocation.

Schedulers for Hadoop/MapReduce [14]: The default FIFO scheduler of MapReduce tries to schedule map tasks on the

Manuscript received February 1, 2015; revised June 4, 2015; accepted July 31, 2015. Date of publication September 22, 2015; date of current version June 26, 2017. This work was supported by JST CREST, "System Software for Post Petascale Data Intensive Science" and "EBD: Extreme Big Data–Convergence of Big Data and HPC for Yottabyte Processing."

X. Li is with the Graduate School of Systems and Information Engineering, University of Tsukuba, Tsukuba 305-8577, Japan, and also with the Core Research for Evolutional Science and Technology, Japan Science and Technology Agency, Kawaguchi 332-0012, Japan (e-mail: risyomei@hpcs.cs.tsukuba.ac.jp).

O. Tatebe is with the Faculty of Engineering, Information and Systems, University of Tsukuba, Tsukuba 305-8577, Japan, and also with the Core Research for Evolutional Science and Technology, Japan Science and Technology Agency, Kawaguchi 332-0012, Japan (e-mail: tatebe@cs.tsukuba.ac.jp).

Digital Object Identifier 10.1109/JSYST.2015.2471850

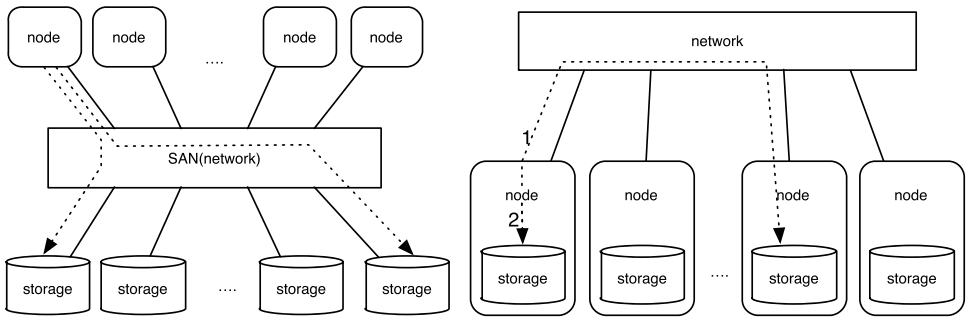

Fig. 1. Uniform (left) and nonuniform (right) storage-access file system. For a uniform system, each access from a compute node has to travel through the network and has a similar performance. For a nonuniform system: some of the accesses are completed locally (2), whereas other accesses go through the network (1), where the performance might differ.

machine that contains the needed file. If a failure occurs, the tasks will be dispatched to nodes near the file. Because tasks can be assigned either locally or remotely, the file system used can be considered a nonuniform storage-access file system.

There are many approaches to improve the default FIFO scheduler from the perspective of file locality [15]–[20]. These works depend on the unique characteristics of the Map/Reduce programming model, making them inappropriate for general-purpose nonuniform storage-access file systems.

Delay scheduling (DS) [15] was designed to tackle the conflict between locality and fairness. In [15], when a job is to be scheduled next according to the fairness but cannot be launched locally, it waits for a small amount of time and allows other jobs to launch instead. The authors argue that the tasks are likely to run locally with little compromise in fairness. The base assumptions of this approach are the following: 1) each job has a nearly identical execution time, and it will finish relatively quickly, and 2) there will always be a local node with a file(s) that the job requires. If 1) fails, delayed jobs may have to wait a relatively long time, and if 2) fails, the jobs will not find a suitable node at all.

Wang *et al.* presented a locality and energy-aware scheduling method [16] that takes advantage of file locality. They defined a method to evaluate the energy efficiency and developed an algorithm to maximize the efficiency when considering both the locality and energy. Their approach is based on the assumption that each file has a fixed number of replicas. Although this is the basic characteristic of the MapReduce model, it may not hold true for other general-purpose nonuniform storage-access file systems.

Data-Aware Scheduler for Grid and Distributed File System: In [21] and [22], Stork scheduler, a data-aware scheduler for a grid, was proposed. Stork scheduler manages the in/out staging of files across a grid and arranges the storage space at the destination. The sender and receiver might not share a name space. In [23], a scheduling method for balancing the workload when considering the locality, network state, and current workload is presented. In this approach, files are divided into multiple blocks of the same size and distributed and replicated across the nodes. Because each block size is identical, the execution time and load impact of local and remote tasks can be determined. The estimated execution time and load are then used to balance the workload. Because certain nonuniform storage-access file systems such as Gfarm do not divide files into blocks, the file sizes may differ from each other completely, and the execution time of each task can therefore not be predicted before the execution, which make this an inapplicable approach.

Scheduling Methods for NUMA: In [24]–[27], methods exploiting the locality to gain a better loop and OpenMP performance are described. The meaning of the term "locality" in these works differs significantly from the one used in our context. In these works, locality refers to the advantages of accessing local memory, whereas in our work, it refers to the merit of accessing data.

In [28], a hardware scheduling approach, where the scheduler is provided with task's data requirements, was proposed. This approach exploits the information of data placement and usage history to calculate the affinity for each CPU core and makes the scheduling decision. However, this approach depends on a specific programming language and model, which makes it inapplicable for batch queuing systems.

Data-Aware Scheduling LSF Plugin for Gfarm: The methods in [29] and [30] are the most relevant approaches that we are currently aware of. The LSF Gfarm plugin tries to take advantage of the effective local file access in Gfarm but emphasizes the manipulation of replicas. In these methods, the tasks are dispatched to a compute node with the required data locally, and the workload is distributed by creating a replica on a new host (such that a task can be dispatched to it). The authors proposed two approaches to optimize the creation of a replica: 1) a method for selecting the best node to create the replica when considering the source, destination, and network loads, and 2) a method for categorizing the jobs to make sure that the time and performance will not be wasted when creating the replica.

Making a replica is indeed an effective way to exploit the effective local access of Gfarm. However, the LSF Gfarm plugin lacks the ability to deal with a "fake data-intensive job" that refers to a large data set but only accesses a small part of it. For example, Blastp and Blastx from the BLAST benchmark refer to the same data set and have similar execution times, which cannot be differentiated using this method. Executing Blastp locally and remotely shows an insignificant difference of 0.02%, whereas the difference in execution time for Blastx is about 17.28%. Moreover, because a task is forced to run locally using the LSF Gfarm plugin, vacant compute nodes will not be usable until the required data are replicated to them, which may occasionally be a waste of resources.

III. Technical Background

In this section, the details of the Torque Resource Manager and Gfarm File System are briefly summarized, with the former being used as the basis of our implementation and the latter having certain key features utilized in the proposed work.

A. Torque Resource Manager

The Torque Resource Manager [9] is an open-source product derived from the original PBS project. It consists of three main components, i.e., pbs_server, pbs_sched, and pbs_mom.

pbs_mom resides on execution hosts, as is responsible for controlling the job.
pbs_sched accepts commands from pbs_server, talks with pbs_mom to maintain the latest information of the execution hosts, and makes scheduling decisions.
pbs_server is the central part of the entire system, accepts tasks from clients, initiates scheduling on pbs_sched, and monitors the jobs on pbs_mom.

In this paper, the main modification is made to pbs_sched, whereas pbs_mom remains completely untouched. In addition, the pbs_server and qsub commands are minimally changed to pass information regarding the referenced data of a task. We expanded the system using the command-line option −g, allowing the user to specify a set of files through the `qsub` command as follows:

$$\texttt{qsub}\ -\texttt{g}\ \texttt{file1}, \texttt{file2}, \ldots \texttt{task.sh}.$$

Alternatively, the file set can be specified through the following directive in the batch file:

$$\#\texttt{PBS}\ -\texttt{g}\ \texttt{file1}, \texttt{file2}, \ldots$$

This information will be sent to pbs_sched and used to determine the allocation of tasks.

B. Gfarm File System

The Gfarm file system is a globally distributed file system used to share data and support distributed data-intensive computing [7]. Instead of setting a dedicated storage node, it federates the local file systems of compute nodes and manages them using a single namespace. The Gfarm file system has multiple instances of gfsd as the storage daemon, which access the local file system, and a master–slave gfmd as the metadata daemon, which manages the file metadata including the hierarchical namespace, file properties, directory structure, and replica information.

Each compute node in Gfarm can access its local storage using gfsd and its remote storage using gfsd on other nodes via the network (see Fig. 2). Gfarm does not divide one file into multiple nodes, but it achieves a scalable performance by duplicating the replica in several nodes. Gfarm can be accessed through the Command Line Interface, API, and FUSE [31] through gfarm2fs.

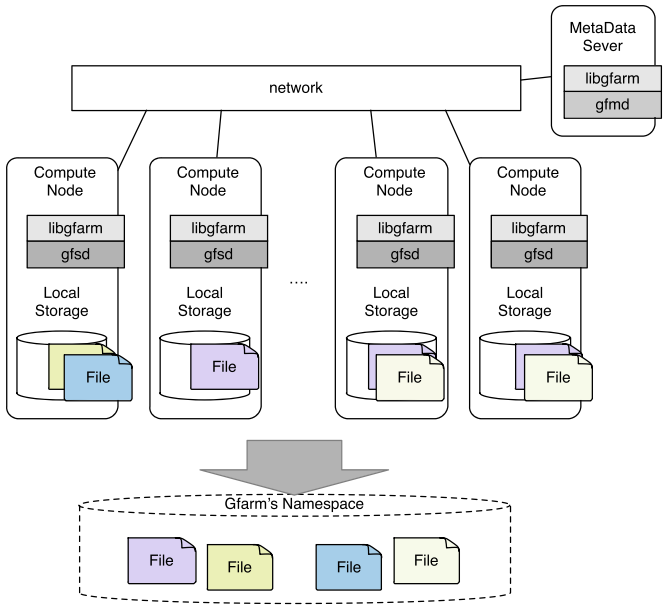

Fig. 2. Basic structure of Gfarm.

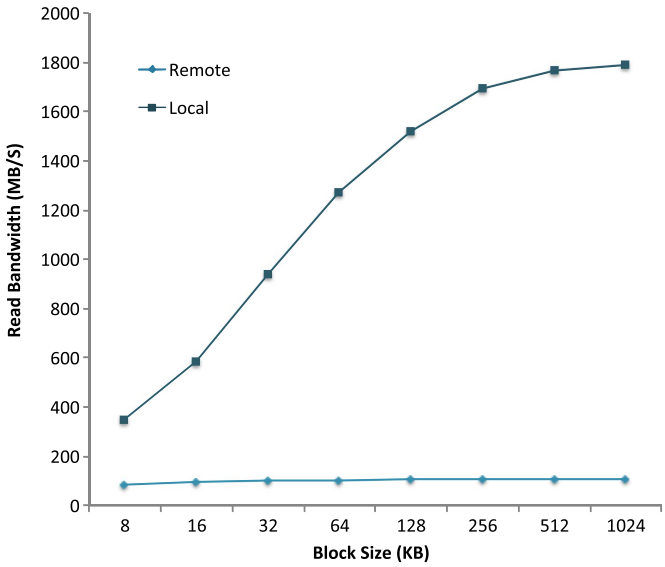

Fig. 3. Performance variation of Gfarm.

C. Nonuniformity of the Gfarm File System

As described previously, a compute node in Gfarm can access a file both locally and remotely, which makes Gfarm a nonuniform storage-access file system. The difference in performance is discussed in this section. We evaluated the throughput of Gfarm using the thput-gfpio benchmark, which comes with the Gfarm package, the results of which are shown in Fig. 3.

For this evaluation, the file size was set to 512 MB, and 1-Gb/s Ethernet was used. As shown in Fig. 3, the throughput of the local access dominates the remote access for all chunk sizes. Therefore, it is safe to conclude that utilizing local access is preferable under normal situations.

IV. Data-Aware Task Dispatch

As discussed in the previous section, most of the existing works depend on some particular constraints, such as a fixed number of replicas or a fixed task size. However, these assumptions do not hold true for some general-purpose nonuniform storage-access file systems such as Gfarm. In this paper, the data-aware task dispatch (DAD) is proposed to exploit local access regardless of those conditions. We implemented DAD on the Gfarm file system and Torque scheduler, but it can be easily implemented on other systems like Hadoop/MapReduce.

A. File Locality and Score

The traditional scheduler takes the CPU load average as the primary factor when selecting a compute node for a specific task. In contrast, DAD introduces $fileLocality$, a parameter that indicates the difficulty of accessing the data set, and combines it with load average as a comprehensive $Score$ to determine the most suitable node. These two parameters are described in detail in the following.

The $fileLocality(t,h)$, which indicates the difficulty of accessing the data set referenced by task t when it is dispatched to a specific compute node h, is defined as follows:

$$fileLocality(t,h) = \left[\sum_{y=1}^{n} locality(f_y,h) \Big/ \sum_{y=1}^{n} sizeof(f_y) + 1\right] \Big/ 2$$

$$locality(f_i,h) = \begin{cases} -sizeof(f_i) & \text{if } on(f_i,h) \\ sizeof(f_i) & \text{other} \end{cases} \quad (1)$$

where the $locality(f_i,h)$ is a value determined by the size of file f_i and whether compute node h has a replica of f_i. If one of the replicas of f_i is on h, the cost of accessing it will be smaller, and therefore, the file size of f_i will be subtracted to make the "cost" smaller and vice versa. The $fileLocality(t,h)$ is the normalized sum of the $locality(f_y,h)$ ranges [0, 1].

The comprehensive $Score$ can be calculated in advance using the $fileLocality$ as follows:

$$Score(t,h) = fileLocality(t,h) \times \beta + load(h) \times (1-\beta) \quad (0 \le \beta \le 1). \quad (2)$$

The load average $load$ and $fileLocality$ are unified into $Score$ using parameter β. Here, β is a modifier used to adjust the strength of DAD. When $\beta = 1$, the scheduler will ignore the CPU load at dispatch. Although there should be a method for acquiring the optimal value of β, we leave this for future work and only show the effectiveness of this particular parameter.

$Score$ can now be used to judge whether a host is desirable for a job execution in the exact way in which the load average is used in a CPU-focused scheduler, with consideration of both the CPU load and the file locality. The symbols used in this section are listed in Table I.

TABLE I
Symbols Used

Symbol	Meaning
t	Task to dispatch
f_i	ith file referenced by task t
h	Execution node with slots available
$on(f_i,h)$	File f_i is on execution node h
$locality(f_i,h)$	Access cost from node h to file f_i
$fileLocality(t,h)$	Difficulty of accessing the dataset when task t is dispatched to node h
$load(h)$	Normalized CPU load average of node h

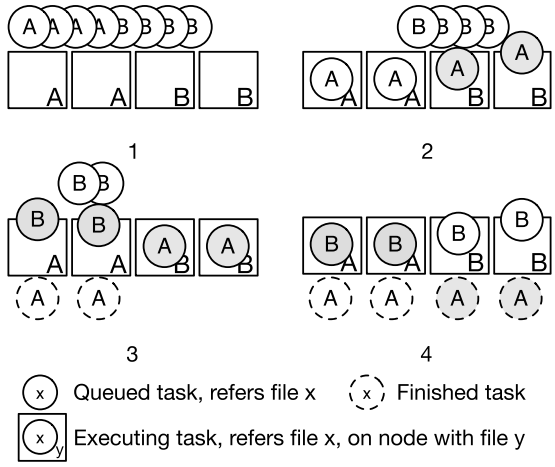

Fig. 4. Task order causing performance degradation.

B. DS for Data-Aware Scheduling

We found that the order of the tasks might cause a drastic degradation in the performance. An example of this is shown in Fig. 4, in which there are eight tasks in a queue, with four requiring file A and the other four requiring file B. Because there are two nodes for each referenced file, the ideal case is for each node to be dispatched with two local tasks. This can be achieved by the arranging tasks as AABBAABB. However, if the tasks come in the order of AAAABBBB, after the first two tasks are dispatched (phase 2 in Fig. 4), the following two tasks will be dispatched to the two available nodes remaining without a needed file. Finally, half of the tasks has to access the file remotely, which will cause a significant drop in performance for data-intensive tasks.

We applied DS [15] to alleviate this issue. DS is a simple idea for a scheduler to achieve locality in the Hadoop file system. In DS, when a task is to be dispatched according to the scheduling policy but has no local node, instead of being executed immediately, it waits for a few slots so that it can be executed locally.

C. Implementation of DAD

We chose Gfarm for the underlying file system and implemented DAD by modifying the naive FIFO scheduler included

in the Torque package. In this section, some of the details for implementing DAD are discussed.

Communication Cost: DAD needs to know the files a job refers to and the nodes where those files reside. The information regarding the replica is managed by the metadata server of Gfarm, i.e., gfmd. Therefore, the scheduler has to communicate with gfmd before making a decision. Because only one file can be queried each time, a task refers to many files, thereby requiring communication multiple times. We exploit a hash table to store such information and reduce the amount of communication with gfmd. Similarly, the file size is also hashed to avoid redundant communication.

Redundant Wait Time: An issue may arise when naively implementing DS for data-aware dispatching on Gfarm. The original DS was designed for the Hadoop file system in which each task can find a node with all of the access files. However, in Gfarm, a task can refer to multiple files, and each compute node might hold only a small portion of such files. Therefore, it is possible that neither of the compute nodes satisfies the standard of "local." In this case, waiting for a local node would be a waste of time. DAD will judge whether it is necessary for a job to wait for the next available slot. If a job is not local to either of the compute nodes, it will not be delayed.

Scheduling by Queue: When submitting two groups of tasks, where both groups refer to the same file sets, it is more preferable to submit them to two different queues because, when all of the tasks are submitted to a single queue, if the first task of a group cannot find a local node, the following tasks in the same group are not likely to find a local node as well. The time used in calculating the $Score$ for these nodes will be wasted. On the other hand, when the two groups of tasks are submitted to different queues, where all tasks in the same queue refer to the same file set, the scheduler can judge only the first task in the queue and skip the rest.

In summary, the pseudocode of DAD is given in Algorithm 1. The algorithm starts at DATAAWAREDISPATCH, and it first calculates the $fileLocality$ of each execution host by calling GETFILELOCALITY and calculates the $Score$ using (2). It then finds a free node with the lowest $Score$. If the $Score$ of this node is smaller than a predefined local threshold $lShold$, the node is selected as a local execution node. If the $Score$ exceeds $lShold$ and no other nodes have a $Score$ smaller than $lShold$, the node is selected as a remote execution node. Otherwise, no nodes are selected, and the task will be delayed until the wait time exceeds the wait limit $wLimit$.

Algorithm 1 Data-Aware Dispatching

function GETFILESIZE($filePath$)
Communicate with **gfmd** and get file size of $filePath$, then store the result to hash table;
end function
function GETFILEEXIST ($filePath, nodeName$)
Communicate with **gfmd** and return if $filePath$ is on $nodeName$, then store the result to hash table;
end function
function GETFILELOCALITY($job, nodeName$)
    $fileMatch, fileMisMatch, fileTotalSize \leftarrow 0$
    for f in $job.fileUsed[]$ **do**
        $size \leftarrow$ GETFILESIZE(f)
        if GETFILEEXIST($f, nodeName$) **then**
            $fileMatch \leftarrow fileMatch + size$
        else
            $fileMisMatch \leftarrow fileMisMatch + size$
        end if
    end for
    return $((-fileMatch + fileMisMatch)/(fileTotalSize \times 2)) + (1/2)$
end function
function DATAAWAREDISPATCH(job)
    $lShold \leftarrow$ local threshold;
    $wLimit \leftarrow$ max delay time;
    $minScore \leftarrow$ FLOAT_MAX;
    $pNode, gNode \leftarrow NULL$; //possible and good node.
    $ifWait \leftarrow false$; //if need to wait for a good node
    for each execution node h **do**
        $Locality \leftarrow$ GETFILELOCALITY(job, h);
        $Score \leftarrow Locality \times \beta + loadAverage \times (1 - \beta)$;
        if h is not free $\wedge Score < lShold$
            $ifWait \leftarrow true$;
        end if
        if h is free $\wedge Score < minScore$ **then**
            $pNode \leftarrow h$;
            $minScore \leftarrow Score$;
            if $Score < lShold$ **then**
                $gNode \leftarrow h$;
            end if
        end if
    end for
    if $pNode = NULL$ **then**
        return $NULL$;
    end if
    if $gNode \neq NULL$
        return $gNode$;
    end if
    if $job.wTime < wLimit \wedge ifWait = true$ **then**
        $job.wTime++$;
        return $NULL$;
    end if
    return $pNode$;
end function

V. EVALUATION

DAD can be applied to various task schedulers by modifying its dispatch phase. In our experiment, we implemented DAD on the Torque scheduler and compared it to the original to see the performance gain as evaluated through three benchmarks, i.e., thput-gfpio, readgf, and BLAST benchmark [32]. These benchmarks are explained in detail in the following. We ran the experiment using a cluster at the University of Tsukuba. The specifications of the machine and software used are listed in Table II.

TABLE II
EVALUATION SPECIFICATION

Hardware	
CPU	Intel Xeon E5-2665 (2.4GHz) x2
HDD	146GB (6Gbps) x4 with 1GB NVRAM
Memory	64GB (8GB 1600MHz) x8
Network	BCM5720 Gigabit Ethernet
Software	
Gfarm	2.5.8.1
Torque	4.2.6.1
BLAST	2.2.29+
BLAST Benchmark	BLAST+ Benchmark Suite 2013
Settings	
Compute Node	4
Meta Data Node	1

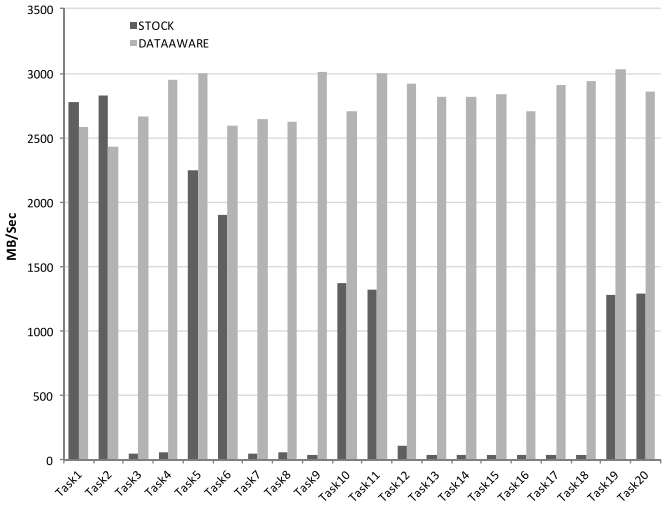

Fig. 5. Evaluation result using thput-gfpio.

A. thput-gfpio

The thput-gfpio benchmark is included in the Gfarm file system package and is used for evaluating the read, write, and copy performance of Gfarm. This benchmark is utilized to show the I/O performance gain made by DAD. In our experiment, 20 different 1-GB files were generated beforehand and distributed evenly to four compute nodes. In addition, 20 thput-gfpio tasks were submitted to read each file accordingly. This experiment shows the boost in I/O speed brought about by DAD. The results are shown in Fig. 5. Only eight tasks had a relatively high throughput when using the stock Torque scheduler and were tasks accidentally dispatched to the local node. Five of these tasks suffered from a degraded performance with a throughput of around 1400 MB/s because some of the remote tasks were in contention for I/O resources. The remaining tasks had to access the required file through the network, which caused the throughput to drop drastically. In contrast, using DAD, tasks were dispatched to the local node and showed a high performance level. Finally, the average throughput of a task dispatched by the Torque scheduler is 780.11 MB/s, whereas that dispatched by DAD is 2803.61 MB/s, which is a 3.59 times increase in throughput.

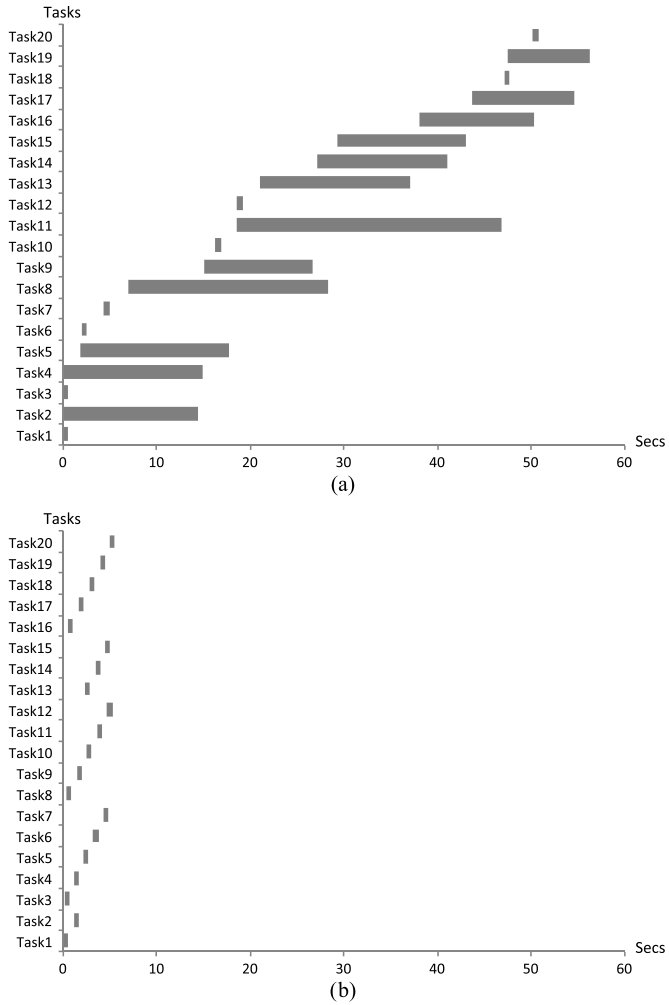

Fig. 6. Evaluation results from readgf: The makespan for 20 readgf tasks was 56.38 s using the stock Torque scheduler and 5.63 s with DAD. (a) Stock Torque scheduler. (b) DAD.

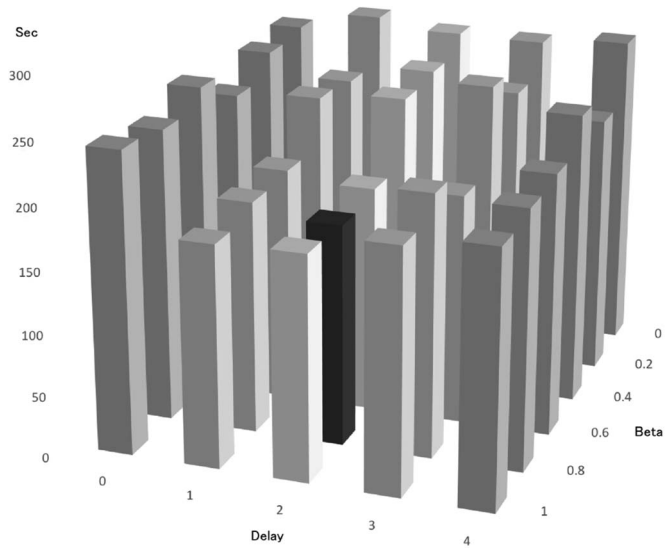

Fig. 7. Evaluation of the BLAST benchmark.

B. readgf

An increase in throughput performance can be clearly observed in the previous benchmark, but the behavior of the tasks remained unknown. The readgf benchmark was developed to reveal detailed task-scheduling information such as the start

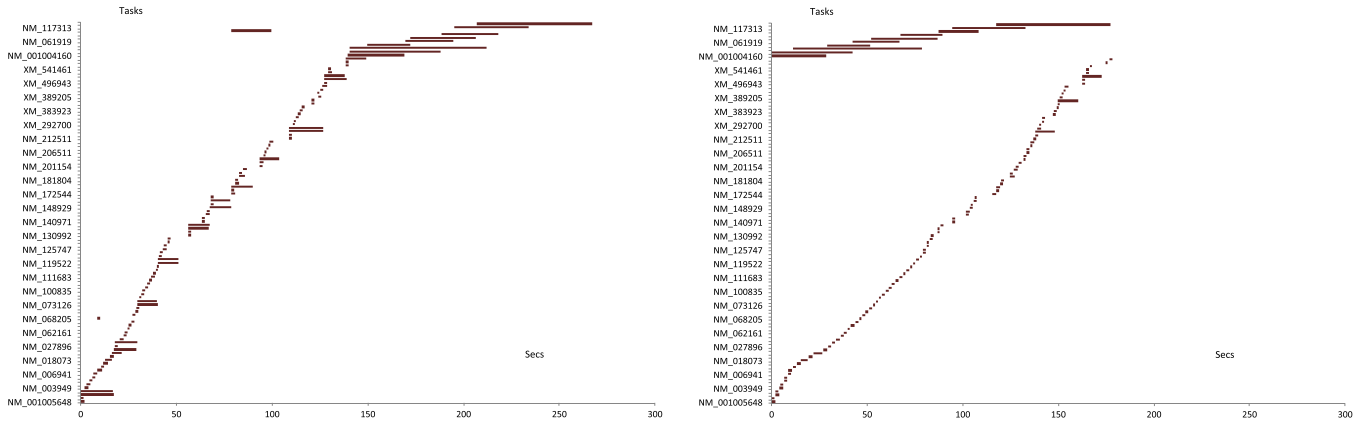

Fig. 8. Gantt chart of the original Torque scheduler (left) and DAD (right) ($\beta = 0.8$, and $Delay = 2$).

time, execution length, and makespan. The main workload of readgf is reading files on Gfarm. In this experiment, 20 different 1-GB files were made beforehand and distributed evenly to four compute nodes. In addition, 20 readgf tasks were submitted to read each file accordingly. The results are shown in Fig. 6.

The two subfigures in Fig. 6 are Gantt charts. The x-axis is the timing axis, which records the start and end times of the tasks plotted on the y-axis. As shown in Fig. 6(a), using the FIFO scheduler, only some of the tasks have a significantly short line, which means that the interval between the start and end times was short and that the task was read efficiently from the local node. Conversely, using DAD, as shown in Fig. 6(b), all of the tasks had short lines, indicating that they performed efficient local access.

C. BLAST Benchmark

In the aforementioned section, DAD was evaluated using two benchmarks whose main workload is file access. In this section, DAD is evaluated using the mixed-workload BLAST benchmark. BLAST is an algorithm for comparing primary biological sequence information. The BLAST benchmark [32] simulates the typical workload obtained from an analysis and consists of multiple subbenchmarks. We chose Blastn and Blastx for our benchmark because they refer to different file sets and have a significant difference in performance when executed locally or remotely. Because the two tasks refer to different sets of files, they were submitted to a different queue by Torque, where all tasks in the queue refer to the same set of files. For DAD, the tasks in each queue were dispatched interleavely, and thus, tasks referring to different files were processed in an interleaving manner to avoid excess judgment regarding the DS.

We ran the evaluation using different β and $Delay$ values, and the results are shown in Fig. 7. The total execution time using the original Torque scheduler was 266.9 s, whereas the best case for the DAD scheduler ($\beta = 0.8$, with a $Delay$ of 2, as indicated by the dark bar in Fig. 7) had a total execution time of 178.0 s, which is a 33% time reduction when compared with the original Torque scheduler. The Gantt chart of the original Torque scheduler and the best case for DAD are shown in Fig. 8. One obvious difference is that there is only one clear line for the Torque scheduler but two for DAD. This is because the Torque scheduler sees two queues as one large queue, whereas DAD dispatches the task in each queue once each time (top, Blastx bottom, Blastn). It is also obvious that there are more "local" tasks when using DAD than using the original Torque scheduler when comparing the same tasks in two charts.

To see how β and $Delay$ affect DAD, we fixed one parameter and changed another. Next, we created a chart and overlapped it with the best DAD case, where $\beta = 0.8$ and $Delay = 2$. The results are shown in Fig. 9. First, we fixed the $Delay$ to 2 and changed β to 0.0 and 1.0, corresponding to the graphs in the upper left and upper right, respectively. When β is set to 0, DAD will degrade to a scheduler that does not consider the data location and will therefore have a similar performance as the original Torque scheduler. On the other hand, when β is set to 1, the DAD shows a similar performance as the best case.

Next, when β is fixed to 0.8 and $Delay$ is changed to 0 and 4, the results correspond to the graphs at the bottom-left and bottom-right of Fig. 9, respectively. When $Delay$ is set to zero, DAD does not delay a task when no local nodes are available, thereby incurring a long line on the chart and ending up with a longer total execution time. On the other hand, when $Delay$ is set to 4, tasks have a greater chance to be executed locally. In contrast, for the best case, some longer tasks are executed earlier, resulting in a shorter total execution time.

D. Results and Discussion

The main purpose of DAD is to take advantage of local access in nonuniform storage-access file systems and thus improves the file I/O performance. Because the main workload of thput-gfpio and readgf is file reading, the file placement will have a great impact on the benchmark performance. For the thput-gfpio benchmark, the average throughput was 780.11 MB/s with the Torque scheduler and 2803.61 MB/s with DAD. The same trend can be observed from readgf, where the makespan was 56.38 s with Torque scheduler and 5.63 s with DAD.

The third benchmark, BLAST, has a mixed workload that includes CPU- and IO-intensive tasks, and it was chosen to evaluate the effects of different values of β. When $\beta = 0$, DAD degrades to a normal algorithm that considers only the CPU

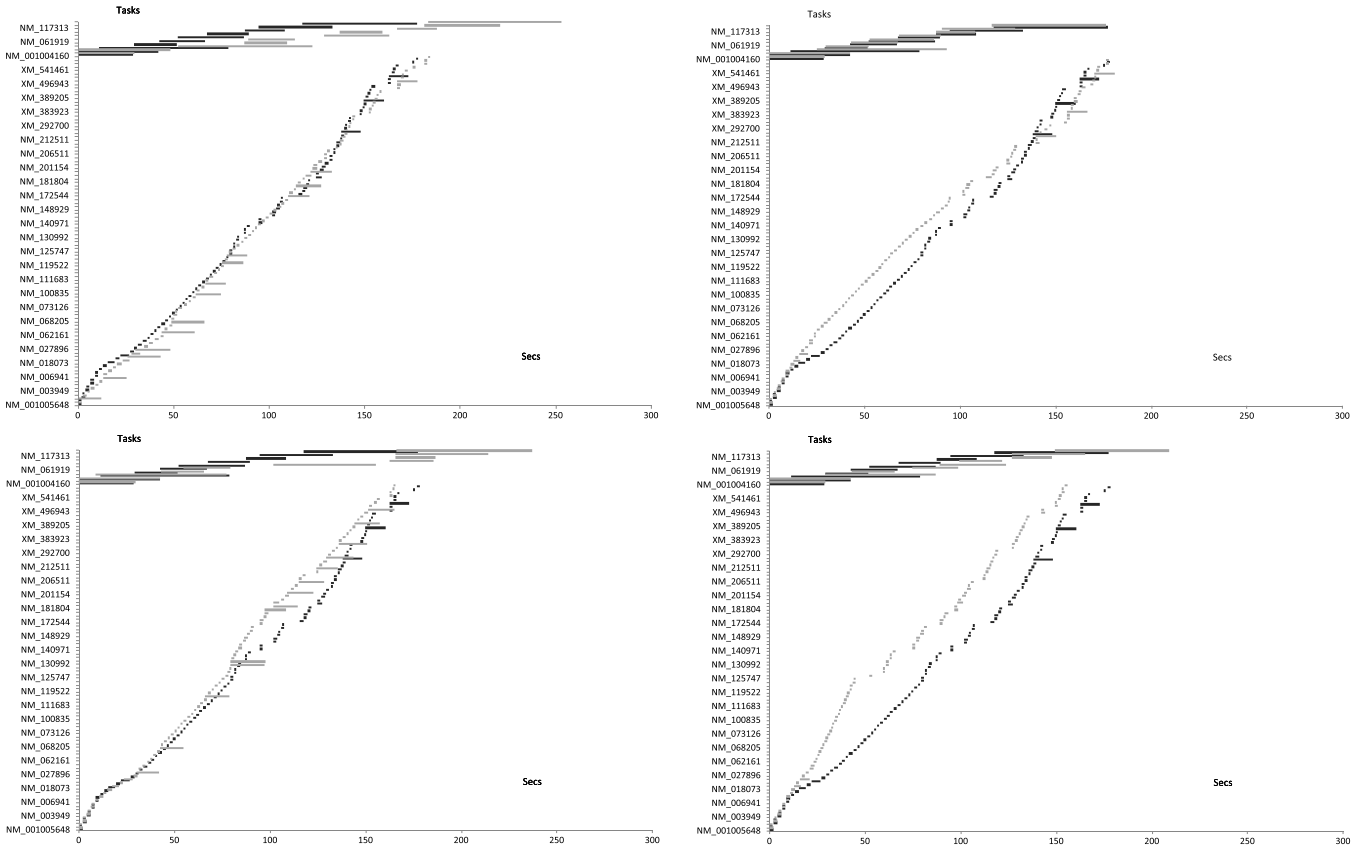

Fig. 9. Overlapped Gantt chart with different parameter settings for DAD and the best cases ($Delay = 2, \beta = 0.8$). The x-axis is the timing axis that records the start times and end times of the tasks which are plotted in the y-axis. The dark lines are the best case when $Delay = 2$ and $\beta = 0.8$, and the light lines are the cases with an altered parameter.

1) Upper left ($Delay = 2$, and $\beta = 0.8$ versus 0.0): DAD degrades to a normal FIFO scheduler and thus has a similar execution time as the original Torque scheduler.
2) Upper right ($Delay = 2$, and $\beta = 0.8$ versus 1.0): Because the CPU load does not experience a bottleneck for dominant tasks, the slight change in β does not lead to an obvious difference in execution time.
3) Bottom-left ($\beta = 0.8$, and $Delay = 2$ versus 0): DAD does not try to delay a job, causing many more remote jobs (and longer lines).
4) Bottom-right ($\beta = 0.8$, and $Delay = 2$ versus 4): In the best case, some longer tasks are executed earlier, resulting in a shorter total execution time.

load average, and therefore, the performance is identical to that of the original Torque scheduler. On the other hand, because the BLAST benchmark has a mixed workload, the improvement with DAD is not drastic but is still significant. The makespan is reduced from 266.9 to 178.0 s.

VI. Conclusion and Future Work

For the nonuniform storage access file system, it is quite important to take advantage of effective local access. DAD offers a method to schedule tasks while striking a balance between CPU load average and file locality. In an evaluation using thput-gfpio, the average throughput of a task dispatched by the original Torque scheduler is 780.11 MB/s, whereas that dispatched by DAD is 2803.61 MB/s, which is about a 3.59 times increase in throughput. Moreover, an evaluation using readgf shows a 90% decrease in makespan compared with the original Torque scheduler. Finally, in an evaluation using the BLAST benchmark, the best case ($\beta = 0.8$ and $Delay = 2$) has a total execution time of 178.0 s, which is a 33% time reduction from the 266.9 s of the original Torque scheduler.

Moreover, because different tasks might refer to different files and the tasks may read/write with a different size even when referring to the same file, it would be inappropriate to set a fixed β value for different kinds of tasks. Therefore, as a future study, we considered adding a function for a task to specify the β value when being submitted.

References

[1] T. Sugimoto *et al.*, "Large-bandwidth data acquisition network for XFEL facility, SACLA," in *Proc. ICALEPCS*, Grenoble, France, 2011, pp. 626–629.
[2] SLAC, Menlo Park, CA, USA. [Online]. Available: https://portal.slac.stanford.edu/sites/lcls_public/Pages/Default.aspx
[3] F. Schmuck and R. Haskin, "GPFS: A shared-disk file system for large computing clusters," in *Proc. Conf. FAST*, 2002, pp. 231–244.
[4] P. Schwan, "Lustre: Building a file system for 1,000-node clusters," in *Proc. Linux Symp.*, 2003, pp. 380–386.
[5] B. Pawlowski, D. Noveck, D. Robinson, and R. Thurlow, "The NFS version 4 protocol," in *Proc. 2nd Int. SANE*, 2000, pp. 1–20.
[6] R. B. Ross, R. Thakur, P. Carns, and W. Ligon, "PVFS: A parallel file system for Linux clusters," in *Proc. 4th Annu. Linux Showcase Conf.*, 2000, pp. 391–430.
[7] O. Tatebe, K. Hiraga, and N. Soda, "Gfarm grid file system," *New Gener. Comput.*, vol. 28, no. 3, pp. 257–275, Jul. 2010.

[8] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The Google file system," *ACM SIGOPS Operating Syst. Rev.*, vol. 37, no. 5, pp. 29–43, 2003.
[9] "Adaptive Computing," Torque Resource Manager, Provo, UT, USA. [Online]. Available: http://www.adaptivecomputing.com/products/open-source/torque/
[10] T. Tannenbaum, D. Wright, K. Miller, and M. Livny, "Condor: A distributed job scheduler," in *Beowulf Cluster Computing With Linux*. Cambridge, MA, USA: MIT Press, 2001, pp. 307–350.
[11] LSF.IBM. [Online]. Available: http://www-03.ibm.com/systems/platformcomputing/products/lsf/index.html
[12] S. Zhou, X. Zheng, J. Wang, and P. Delisle, "Utopia: A load sharing facility for large, heterogeneous distributed computer systems," *Softw., Practice Experience*, vol. 23, no. 12, pp. 1305–1336, 1993.
[13] Open Grid Scheduler. SUN. [Online]. Available: http://gridscheduler.sourceforge.net
[14] J. Dean and S. Ghemawat, "MapReduce: Simplified data processing on large clusters," *Commun. ACM*, vol. 51, no. 1, pp. 107–113, Jan. 2008.
[15] M. Zaharia *et al.*, "Delay scheduling: A simple technique for achieving locality and fairness in cluster scheduling," in *Proc. 5th ACM Eur. Conf. Comput. Syst.*, 2010, pp. 265–278.
[16] X. Wang and Y. Wang, "An energy and data locality aware bi-level multiobjective task scheduling model based on MapReduce for cloud computing," in *Proc. IEEE/WIC/ACM Int. Conf. WI-IAT*, 2012, vol. 1, pp. 648–655.
[17] M. Hammoud and M. F. Sakr, "Locality-aware reduce task scheduling for MapReduce," in *Proc. IEEE 3rd Int. Conf. CloudCom*, 2011, pp. 570–576.
[18] S. Ibrahim *et al.*, "LEEN: Locality/fairness-aware key partitioning for MapReduce in the cloud," in *Proc. IEEE 2nd Int. Conf. CloudCom*, 2010, pp. 17–24.
[19] B. Palanisamy, A. Singh, L. Liu, and B. Jain, "Purlieus: Locality-aware resource allocation for MapReduce in a cloud," in *Proc. ACM Int. Conf. High Perform. Comput., Netw., Storage Anal.*, 2011, pp. 1–11.
[20] X. Zhang, Z. Zhong, S. Feng, B. Tu, and J. Fan, "Improving data locality of MapReduce by scheduling in homogeneous computing environments," in *Proc. IEEE 9th ISPA*, 2011, pp. 120–126.
[21] T. Kosar and M. Livny, "Stork: Making data placement a first class citizen in the grid," in *Proc. IEEE 24th Int. Conf. Distrib. Comput. Syst.*, 2004, pp. 342–349.
[22] T. Kosar and M. Balman, "A new paradigm: Data-aware scheduling in grid computing," *Future Gener. Comput. Syst.*, vol. 25, no. 4, pp. 406–413, 2009.
[23] J. Jin, J. Luo, A. Song, F. Dong, and R. Xiong, "BAR: An efficient data locality driven task scheduling algorithm for cloud computing," in *Proc. IEEE/ACM 11th Int. Symp. Cluster, Cloud Grid Comput.*, 2011, pp. 295–304.
[24] S. L. Olivier, A. K. Porterfield, K. B. Wheeler, M. Spiegel, and J. F. Prins, "OpenMP task scheduling strategies for multicore NUMA systems," *Int. J. High Perform. Comput. Appl.*, vol. 26, no. 2, pp. 110–124, May 2012.
[25] A. Muddukrishna, P. A. Jonsson, V. Vlassov, and M. Brorsson, "Locality-aware task scheduling and data distribution on NUMA systems," in *OpenMP in the Era of Low Power Devices and Accelerators*. Berlin, Germany: Springer-Verlag, 2013, pp. 156–170.
[26] M. Durand, F. Broquedis, T. Gautier, and B. Raffin, "An efficient OpenMP loop scheduler for irregular applications on large-scale NUMA machines," in *OpenMP in the Era of Low Power Devices and Accelerators*. Berlin, Germany: Springer-Verlag, 2013, pp. 141–155.
[27] H. Li, S. Tandri, M. Stumm, and K. C. Sevcik, "Locality and loop scheduling on NUMA multiprocessors," in *Proc. IEEE ICPP*, 1993, vol. 93, pp. 140–147.
[28] B. Khan *et al.*, "Architectural support for task scheduling: Hardware scheduling for dataflow on NUMA systems," *J. Supercomput.*, vol. 71, no. 6, pp. 2309–2338, 2015.
[29] X. Wei *et al.*, "Implementing data aware scheduling in Gfarm(R) using LSF(TM) scheduler plugin mechanism," in *Proc. Int. Conf. Grid Comput. Appl.*, 2005, pp. 3–5.
[30] J. Jiang, G. Xu, and X. Wei, "An enhanced data-aware scheduling algorithm for batch-mode data intensive jobs on data grid," in *Proc. IEEE ICHIT*, 2006, vol. 1, pp. 257–262.
[31] FUSE. [Online]. Available: https://portal.slac.stanford.edu/sites/lcls_public/Pages/Default.aspx
[32] G. Coulouris, "BLAST Benchmarks," Nat. Inst. Health, Bethesda, MD, USA. [Online]. Available: http://fiehnlab.ucdavis.edu/staff/kind/Collector/Benchmark/Blast_Benchmark

Xieming Li received the M.Eng. degree from the Graduate School of System and Information Engineering, University of Tsukuba, Tsukuba, Japan, in 2014, where he is currently working toward the Ph.D. degree.

His main research interests are grid computing and distributed file systems.

Mr. Li is a member of the Information Processing Society of Japan.

Osamu Tatebe received the Ph.D. degree in computer science from the University of Tokyo, Tokyo, Japan, in 1997.

He was with the Electrotechnical Laboratory and National Institute of Advanced Industrial Science and Technology (AIST) until 2006. He is currently a Professor with the Department of Computer Science, University of Tsukuba, Tsukuba, Japan. His research areas are high-performance computing, data-intensive computing, and parallel and distributed system software.