Disk Cache-Aware Task Scheduling For Data-Intensive and Many-Task Workflow

Masahiro Tanaka[†]
Center for Computational Sciences,
University of Tsukuba
Tsukuba, Japan
tanaka@hpcs.cs.tsukuba.ac.jp

Osamu Tatebe[†]
Faculty of Engineering, Information Systems,
University of Tsukuba
Tsukuba, Japan
tatebe@cs.tsukuba.ac.jp

Abstract*—Workflow scheduling to maximize I/O performance is one of the key issues in data-intensive, many-task computing. In our previous work, we proposed locality-aware workflow scheduling method using the Multi-Constraint Graph Partitioning. In this work, we focus on read performance of input files from the disk cache (buffer cache or page cache on main memory). In order to maximize the disk cache hit rate of input files, a LIFO-order scheduling is effective since created intermediate files may be read soon. However, LIFO policy has a disadvantage of so-called "trailing task problem." We propose a hybrid scheduling strategy of LIFO and HRF (Highest Rank First). In our strategy, one of two policies is applied depending on the number of highest-rank tasks in the queue to avoid the problem. In addition, scheduling for the overlap of computation and I/O is proposed. We implement our scheduling strategy for the Pwrake workflow system and the Gfarm distributed file system and evaluate it by executing data-intensive workflows using a computer cluster. Our scheduling strategy improves the performance of copyfile workflow by 30% due to increase in disk cache hit rate, and the performance of Montage workflow by 12% due to increase in core utilization.

Keywords—workflow system; many task computing; task scheduling; distributed file system

I. INTRODUCTION

The increasing amount of science data generated by advanced instruments requires parallel processing on distributed computer resources. Among issues in parallel processing of science data are (1) learning cost in parallel programming such as MPI and (2) needs for legacy programs to process science data in long-used data formats. In view of such situations, one of useful approaches for parallel data processing is process-based parallelism, i.e., the parallel execution of a large number of sequential programs, each of which processes a part of data. However, there are issues in executing a large number of short time jobs such as task throughput. Raicu et al. discussed issues in Many Task Computing (MTC) where the number of tasks is 10^3-10^6 [1].

Pwrake[1] [2] is a system developed for MTC and data-intensive workflows. Pwrake is an extension to Rake, a build tool written in Ruby. Rake is also a powerful workflow language to define many-task workflows. In order to achieve scalable file I/O performance, Pwrake is assumed to use the Gfarm distributed file system [3]. Gfarm file system has a feature to utilize the local storage of compute nodes. The local access to the storage of compute nodes is one of key issues for scalable parallel I/O performance. In our previous work, we proposed the workflow scheduling to minimize data movement using the Multi-Constraint Graph Partitioning (MCGP) [4].

In this paper, we focus on another issue in the I/O-aware task scheduling, i.e., the hit rate of disk cache (buffer cache or page cache on main memory) during workflow execution. This issue relates to the order of task execution. We propose task scheduling using a LIFO queue so that tasks whose prerequisite tasks completed later are executed earlier. The LIFO scheduling improves the cache hit rate of input files. However, The LIFO scheduling has a disadvantage in the CPU core utilization of tasks due to a problem known as the trailing task problem [5] that is mentioned in Section III.C. We propose the modification of the LIFO scheduling to reduce trailing tasks using information on a workflow DAG. In addition, we discuss scheduling for the overlap of computation and I/O. We evaluate our task scheduling by applying to a copyfile workflow and a Montage astronomy workflow using a computer cluster of 12 worker nodes with 96 cores.

The contributions of this paper are the followings:

- New scheduling methods to improve the disk cache hit rate to avoid trailing task problem.
- Implementation and evaluation of our scheduling algorithm by running scientific workflows on a computer cluster.

The remainder of this paper is organized as follows. Section II describes the background of this work. Section III describes the scheduling methods considering disk cache hit rate and CPU utilization. In Section IV, we apply the proposed methods to scientific workflow using a computer cluster. Section V describes related work and Section VI provides conclusion.

[1] http://masa16.github.io/pwrake/

† The Japan Science and Technology Agency (JST) / CREST.

II. Background

A. Pwrake: Parallel Workflow extension for Rake

Pwrake is a parallel and distributed workflow system developed for data-intensive, many-task computing on computer clusters. Pwrake is implemented as an extension for on Rake (Ruby Make). After the first report [2], we continued the development the Pwrake system and improved its performance, functionality, compatibility with Rake. We briefly introduce the overview of Pwrake here.

Pwrake inherits a workflow language from Rake. Rake is a standard tool of Ruby and is widely used by Ruby users, and a powerful workflow language since it has various useful features such as mapper rules and capability to write scripts in the Ruby language. Pwrake inherits also the implementation of Rake::Task class (hereafter Task class) from Rake. The Task class holds information including the self-name of the task, a list of prerequisite tasks that must be completed before the start of itself, and a task action defined as a Ruby code block. The dependency information through prerequisite tasks forms a DAG (Directed Acyclic Graph). Rake::FileTask (hereafter FileTask) is a file generation task class defined as a subclass of the Task class. As for FileTask, the task name is regarded as an output file name, and the prerequisite task names can be regarded as input files. The action of FileTask is executed when the output file does not exist, or its timestamp is older than input files. This mechanism is useful to resume a workflow like UNIX Make.

Fig. 1 is a schematic overview of the Pwrake design. Pwrake organizes compute nodes in the master-worker model. The Pwrake master has the same number of worker threads as the number of total cores of worker nodes. A task action is executed in a worker thread of the Pwrake master process. Every worker thread connects to remote worker nodes via SSH. Shell command lines are sent to a remote node and executed there.

Instead of file staging, Pwrake relies on file sharing through the Gfarm file system. If resource allocation is required, Pwrake is run on resources acquired by batch system. Gfarm has a mechanism to exploit the performance of local I/O: (1) selection of a close replica when reading a file replicated to multiple nodes, and (2) selection of the local storage when creating a new file. However, the locality-aware task scheduling is an issue for the workflow system.

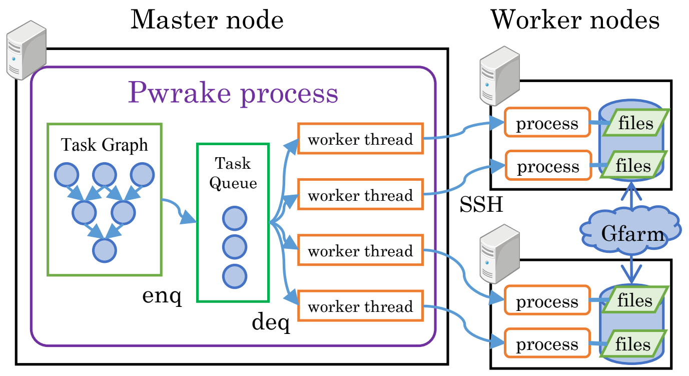

Fig. 1. Schematic overview of Pwrake system (See text for details).

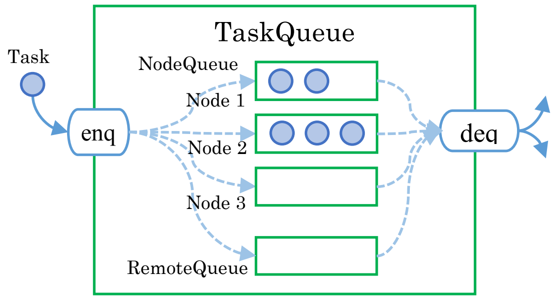

Fig. 2. Schematic design of locality-aware TaskQueue. Candidate nodes are determined based on locality, tasks are queued into corresponding NodeQueue.

B. Pwrake Task Scheduling

The task scheduling problem is to find the best assignment of tasks to computer resources. The objective of scheduling is to minimize the makespan. The makespan is defined as estimated time from the start of the first task until the completion of the last task. The task scheduling problem is NP-hard. Therefore, various heuristic scheduling algorithms have been proposed for heterogeneous environments such as HEFT (Heterogeneous Earliest Finish Time) [6]. Those algorithms normally assume that the execution time of tasks (task cost) is known in advance, and the schedule is decided before workflow execution.

In the case of MTC workflows, however, it is difficult to predict task costs for thousands or millions of tasks. As for the task scheduling of Pwrake, we assume that knowledge on the cost of each task and the performance of nodes is not given in advance. Instead, we take an approach of TaskQueue-based dynamic scheduling. This is a pull-based approach to enable efficient balancing and fast dispatch of tasks. In this approach, the key issue of the task scheduling is the selection of a task retrieved by an idle worker from TaskQueue.

The locality-aware design of the TaskQueue class in the Pwrake system is shown in Fig. 2. TaskQueue has “enq” (enqueue) and “deq” (dequeue) methods, and contains NodeQueue assigned to every worker node. The locality-aware scheduling of Pwrake is as follows.

In the “enq” phase, the scheduler assigns candidate worker nodes where a task is to be executed. We implemented two algorithms for the selection of candidate nodes. The first algorithm is the “close to input files” scheme. The second algorithm is “Multi-Constraint Graph Partitioning (MCGP)” developed in our previous work [4]. The former scheme is to select worker nodes which store input files of a task. The latter algorithm can improve locality for workflows that deal with geometrically-partitioned data. After the determination of candidate nodes, the task is queued into the corresponding NodeQueue.

The “deq” method is invoked by an idle worker to retrieve a task from TaskQueue. In this phase, the scheduler tries to retrieve a task from NodeQueue assigned to the worker nodes. If the NodeQueue contains no task, worker node retrieves tasks in the following order; (a) find a task whose input files are stored at other than compute node. If no task is found, then (b) “steal” a task assigned to other nodes. This mechanism enables load balancing among worker nodes.

III. Cache-Aware Task Scheduling

A. *Performance of Reading a Local and Cached File*

TABLE I. shows the read performance of a Gfarm file at the Tohoku cluster (See Section IV.A). The read performance of *local* and *cache* access is roughly 10 times as high as that of other cases. This table shows that local and cache access are important for I/O performance. In the previous section, we visited the Pwrake scheduling to improve the local access ratio. The main subject of this paper is cache-aware task scheduling.

TABLE I. I/O PERFORMANCE OF GFARM FILES (MEASURED USING CLUSTER SHOWN IN TABLE II.)

			Bandwidth (MiB/s)
Read	Local	disk	70
		cache	592
	Remote	disk	39
		cache	71
Write	Local	disk	59

B. *LIFO as a Cache-Aware Scheduling*

In this paper, we do not step into the mechanism of disk cache. Instead, we assume that a later-saved file has a higher probability that an input file is cached in main memory. This assumption is applicable in situations where a file I/O-based workflow is executed on standard Linux computers. In this assumption, access time of input files is a good criterion. This is achieved by a priority queue based on file access time. However, the priority queue requires sorting costs. As a more simple approach, the Last-In-First-Out (LIFO) queue produces a similar result since task queuing time is an alternate indicator of creation time of input files.

In the design of Pwrake TaskQueue, NodeQueue shown in Fig. 2 is implemented as a queue which defines the order of tasks such as LIFO.

We here compare the LIFO and FIFO scheduling using an example workflow whose DAG is drawn in Fig. 3. In this example, a vertex represents a task and an edge represents a dependency through input/output files. This workflow consists of three steps of tasks; A*i*, B*i*, C where *i* = 1..*n*. Here the input file of task B*i* is the output file of task A*i*, and task C reads all the output files of B*i*. For simplicity, we assume that all the tasks occupy single core and task costs are equal.

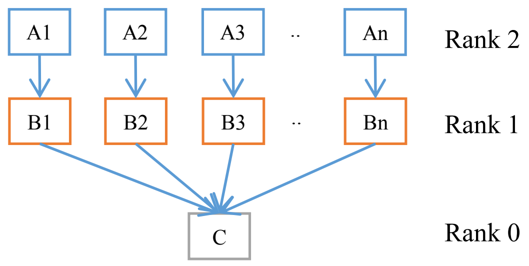

Fig. 3. DAG of the example workflow. Vertex and edge represent task and dependency, respectively.

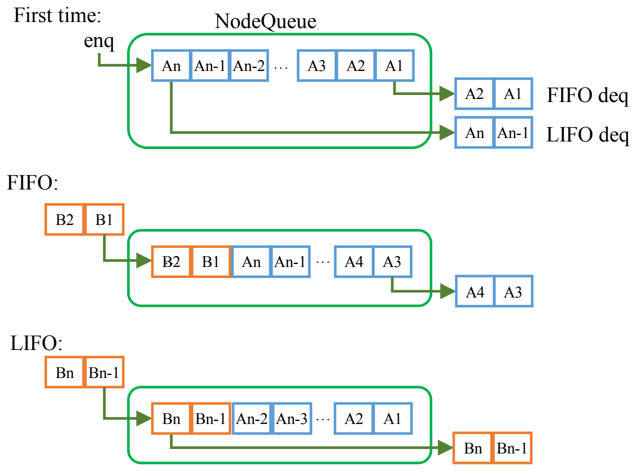

Fig. 4. Schematic illustration of FIFO and LIFO queues for the Fig. 3 DAG on a two-core machine. Top: at the beginning, all the tasks A are queued, and A1 and A2 are retrieved (FIFO), or An and An-1 are retrieved (LIFO). In the next step, Middle (FIFO): B1 and B2 are queued, and A4 and A3 are retrieved. Bottom (LIFO): Bn and Bn-1 are queued and retrieved.

Next, we consider the execution of this workflow on machine with two cores. Fig. 4 schematically shows the behavior of FIFO and LIFO queues. In both cases, all tasks A*i* (*i* = 1..*n*) are queued at the beginning. When a task A*i* is retrieved from the queue, it is executed. After the task A*i* completes, a task B*i* is queued. FIFO and LIFO have the following difference. In the FIFO order, all the tasks A are retrieved earlier than the tasks B. In the LIFO order, lately-queued tasks B are executed earlier.

Fig. 5 shows the result of scheduling on machine with two cores, under four scheduling policies including FIFO and LIFO. In this figure, time advances from top to bottom and the only last task C is not shown. The LIFO scheduling (Fig. 5 (1)) maximize the disk cache hit rate since B*i* task is invoked immediately after the task A*i*. On the other hand, in the case of the FIFO scheduling (Fig. 5 (2)), the average interval of A*i* and B*i* is (execution time of one task) × (*n*/2). If the output file of A*i* expires from the disk cache during this interval, the read performance becomes worse. (There is no problem in the arrows toward the other column since tasks are in the same node.) On the other hand, the LIFO scheduling has a disadvantage in core usage because of the "trailing task problem" [5] described in the next subsection.

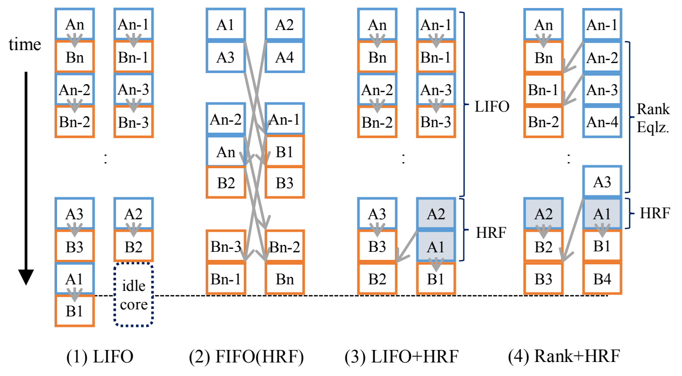

Fig. 5. Schematic diagram of the scheduling result of the Fig. 3 workflow on a two-core machine under four scheduling policies. Time advances from top to bottom. The task C is omitted. The gray arrows indicate task dependencies.

C. Trailing task problem and HRF

The trailing task problem [5] is a problem in MTC. In the course of workflow execution, remaining tasks becomes fewer than workers, and some worker becomes idle. If many trailing tasks are left, the CPU utilization becomes worse. In the LIFO case (Fig. 5 (1)), the task A*n* and B*n* are trailing tasks and occupy only one core. Since our scheduling strategy does not assume prior information on task cost, it is impossible to predict CPU idle time caused by trailing tasks. Instead, we consider the probability of the number of trailing tasks. In the LIFO scheduling of Fig. 5 (1), the maximum time span of trailing task is the total execution time of A*n* + B*n*, since A*n* and B*n* cannot be executed in parallel. On the other hand, FIFO scheduling (Fig. 5 (2)), the maximum time span of trailing task is only the execution time of B*n*. This is because all the tasks A are invoked before the start of the tasks B. The difference between task A and B is the distance from the target task C.

We here define the "rank" of a task as a distance from the last target task in a workflow DAG as follows. The target task is defined as rank 0. The prerequisite task of a rank *i* task is numbered as rank *i*+1. If a prerequisite task is followed by tasks having different ranks, the rank of the prerequisite task is obtained by adding one to the maximum rank of the subsequent tasks. In the example of Fig. 3, the tasks C is rank 0, the tasks B are rank 1, and the tasks A are rank 2.

An effective way to mitigate the trailing task problem is earlier execution of higher rank tasks. We call this policy Highest Rank First (HRF). Similar policy can be seen in the static scheduling algorithms such as HEFT [6]. In the HEFT algorithm, it calculates a priority called "upward rank" for each task in the first phase, and tasks are assign tasks to workers starting with the highest priority in the second phase.

Since LIFO and FIFO (same order as HRF) conflict with each other, only one of these schemes can be selected at the same time. If *n* is large, LIFO tends to be better since the interval of task A*i* and B*i* becomes large. If the number of worker cores is large, FIFO may be better due to the trailing task problem. In the next subsection, we propose methods with advantages of both LIFO and FIFO in most of the situations.

D. Proposed Methods

To solve the problem of disk cache hit rate and CPU utilization simultaneously, we propose two algorithms to determine the order of task retrieval from NodeQueue.

1) LIFO + HRF

We call the first method LIFO+HRF, where the LIFO or HRF scheduling is applied according to the number of tasks in NodeQueue. We define N_{HR} as the number of tasks with the highest rank in NodeQueue and N_{core} as the number of cores of the corresponding node. The LIFO+HRF scheduling algorithm is:

- If $N_{\mathrm{HR}} > N_{\mathrm{core}}$, select the task in the order of LIFO
- If $N_{\mathrm{HR}} \leq N_{\mathrm{core}}$, select the task in the order of HRF

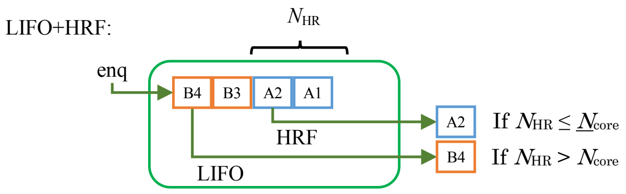
LIFO+HRF:
N_{HR}
enq
B4
B3
A2
A1
HRF
LIFO
A2
B4
If $N_{\mathrm{HR}} \leq N_{\mathrm{core}}$
If $N_{\mathrm{HR}} > N_{\mathrm{core}}$

Fig. 6. Schematic illustration of the LIFO+HRF algorithm.

Fig. 6 schematically shows the behavior of the LIFO+HRF algorithm. The result of LIFO+HRF scheduling for the DAG of Fig. 3 is shown in Fig. 5 (3). In the example, the LIFO+HRF algorithm selects tasks in the following policies:

- Task A*i*, B*i* (i = 3..*n*) : LIFO
- Task A2, A1, B2, B1 : HRF

The tasks A*i* and B*i* (i = 3..*n*) is retrieved in the cache-aware LIFO policy. On the other hand, when only A2 and A1 remain in the queue, they are executed earlier under the HRF policy.

2) Rank Equalization + HRF

The second proposed method relates to the overlap of I/O and computation. In workflow patterns in Fig. 3, there may be a situation where the tasks A are computationally intensive, and the tasks B are I/O intensive. In such a situation, there is a chance to reduce execution time by overlapping the tasks A and B. Fig. 5 (4) (labeled as Rank+HRF) shows an example of overlap scheduling. Although the interval between the task A*i* and B*i* in Fig. 5 (4) is longer by one task than those of LIFO; it is much shorter than the interval of FIFO. This situation can be achieved by the rank equalization of the number of tasks running at any time. We call this policy *Rank Equalization*. The algorithm of Rank Equalization is as follows.

- Get *R*, a set of ranks of queued tasks at scheduling time.
- Calculate weights *w*[*r*] for each rank *r* in *R*. (see below)
- Select a rank *r* in *R* randomly using weights *w*[*r*].
- Retrieve a task from tasks with the rank *r* under the LIFO policy.

The weights of rank selection are not equal in the following reason. As is shown in Fig. 7, the number of task invocations is inversely proportional to task execution time. Therefore, the weight of rank selection is defined as the inverse of average task execution time, which is measured at run time. However, at the beginning of the workflow, task execution time is not given. Therefore, equal weights are provided initially. After the first task completes, the average of the measured task execution time is used.

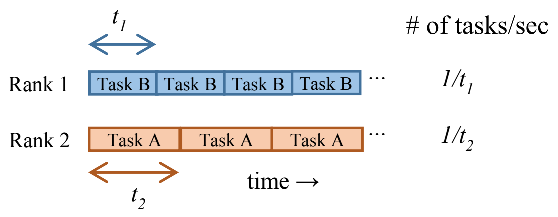

Fig. 7. The relationship between task execution time and the number of task invocations in the same timespan.

The retrieval from the selected rank is the LIFO order. Since the trailing task problem still exists, the algorithm is changed to HRF when the number of tasks is less than the number of cores.

IV. Performance Evaluation

A. Evaluation environment

The environment used for the evaluation is summarized in TABLE II. The computer cluster is the Tohoku site of InTrigger platform [7]. We used up to 12 worker nodes. Gfarm File System Node (FSN) is located on each worker node. Another node is used for the Gfarm metadata server and the Pwrake master. In order for the Gfarm system to write a file to the local storage every time, the option schedule_idle_load_thresh in gfarm2.conf is set to be 100.

TABLE II. Evaluation environment

Cluster	InTrigger Tohoku site
CPU	Intel Xeon E5410 2.33GHz
Main Memory	32 GiB
# of cores / node	8
Max # of compute node	12
Network	1Gb Ethernet
OS	Debian 5.0.4
Gfarm	ver. 2.5.8.6
Ruby	ver. 2.1.1
Pwrake	ver. 0.9.9.1

B. Copoyfile Workflow

In this section, we investigate the effect of the LIFO scheduling using an I/O-only workflow. As a workflow task, we made a program named copyfile[2] written in the C language. The copyfile program loads an input file into main memory, after that it writes the data to an output file. (This behavior is similar to scientific data processing without calculation.) The DAG of this workflow is same as the DAG in Fig. 3. Both the tasks A and B are copyfile task, i.e., an input file is copied twice. Before the workflow execution, 100 input files with 3 GiB are created in the storage of ten worker nodes. Since each input file is copied twice, the total read and write size is 600 GiB, respectively.

In this measurement, we used ten worker nodes of the InTrigger Tohoku cluster. Since the copyfile task is I/O-intensive, only one core per node is used. Thus, ten processes run in parallel during workflow execution. Immediately after the creation of 3 GiB file, it is surely cached since the main memory size of compute nodes is 32 GiB. On the other hand, after reading 100 input files, the first file is surely evicted from the disk cache. We evaluate only the LIFO and FIFO scheduling. HRF is not evaluated since it does not change the behavior in the situation here of one core per node. The elapsed time is measured three times. The averaged result is shown on left blue bars in Fig. 8. It shows that the LIFO scheduling improved performance by 30% from FIFO scheduling.

We compare the experiment with the estimation of the workflow elapsed time from the I/O performance. The elapsed time of the copyfile workflow is estimated as $t = I / R + O / W$, where I and O are the sizes of input and output files in bytes and

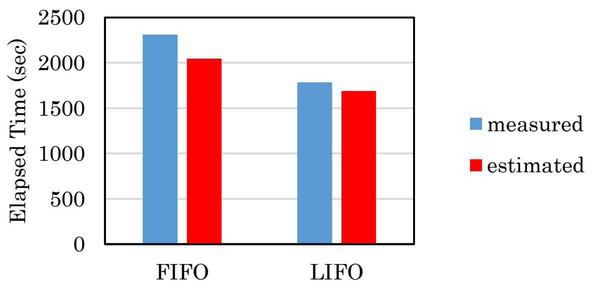

Fig. 8. Elapsed time of copyfile workflow.

R and W is the read and write bandwidth in bytes/sec, respectively. The values of R and W are the I/O performance of Gfarm shown in TABLE I. We assume the read bandwidth as follows: For FIFO, the tasks A and B read from local *disk*. For LIFO, the tasks A read from local *disk* and the tasks B read from the local *cache*. Thus, execution time is estimated and shown as red bars in Fig. 8. The estimation is consistent with the experiment since the elapsed time of the workflow includes other time than I/O. The result shows that the performance improvement by the LIFO scheduling is attributed to cache read.

C. Montage workflow

In this section, we evaluate the cache-aware scheduling using astronomical image processing software Montage [8] as a data-intensive scientific workflow. Montage is a collection of programs for combining multiple shots of astronomical images and creating an image with a large sky area. We implemented a Montage workflow as Rakefile[3]. Fig. 9 shows an example DAG of Montage workflow. In the actual processing, the number of tasks is larger depending on the number of input files.

TABLE III. Sizes of experiment Montage workflow

Input file	SDSS DR7
# of input files	421
Size of one input file	2.52 MB
Size of total input files	1061 MB
# of intermediary files	4720
Size of intermediary files	63.5 GB
# of tasks	2707

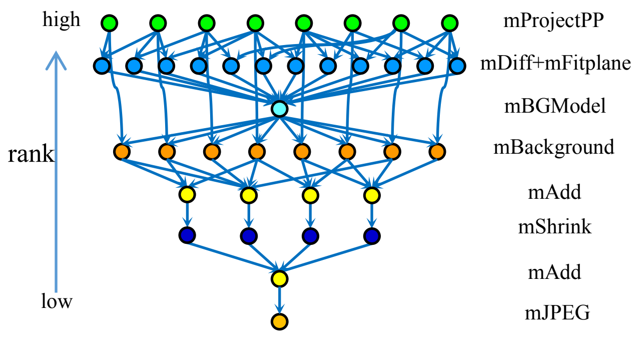

Fig. 9. DAG of Montage workflow. Vertex and edge represent task and dependency, respectively.

[2] https://gist.github.com/masa16/5956881

[3] https://github.com/masa16/pwrake-demo

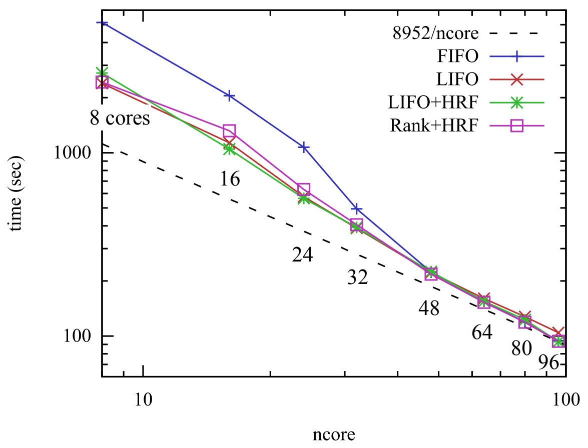

Fig. 10. Elapsed time of Montage workflow for scalability in a double logarithmic graph. The broken line indicates inverse proportion as a guide to scalability.

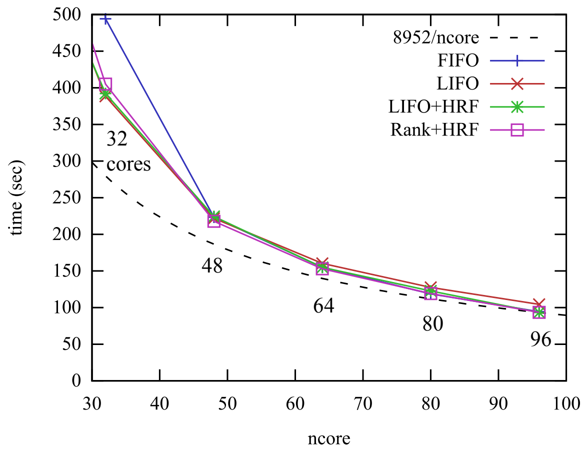

Fig. 11. Elapsed time of Montage workflow scalability (Same as Fig. 10 but shows only >32 cores, in linear axes). The broken line indicates inverse proportion.

As input files, we use the image of SDSS DR7 [9]. The file size and task size of the evaluated workflow are summarized in TABLE III.

1) Scalability and Performance Enhancement with Cache

In order to investigate the scalability, we measured elapsed time of Montage workflow using 1-12 nodes with data size fixed (i.e., strong scaling). The number of used cores is 8-96 (eight cores per node). We measured four scheduling methods; FIFO, LIFO, LIFO+HRF, and Rank+HRF. In all cases, we apply the locality scheduling with a node assignment algorithm using MCGP. We measured once for 1-3 nodes, three times for 4-8 nodes, and five times for 10-12 nodes, selected better half experiments, and averaged them since there may be unexpected performance degradation due to OS noise, etc. The elapsed time of sequential tasks (mBGModel, last mAdd, mJPEG) are excluded from the results. Fig. 10 is the plot in all the range of 8-96 cores in logarithmic scale, and Fig. 11 is the plot in the range of 32-96 cores in linear scale. In these plots, the broken line indicates inverse proportion between the number of cores and elapsed time as a guide to scalability.

In Fig. 10, the slopes of the measurement data are steeper than the broken line. This means that the performance is improved beyond the scalability. We consider that the reason is attributed to disk cache hit rate since the data size processed by a single node decreases as the number of nodes increases. In the range of 8-32 cores, the elapsed time of FIFO is significantly longer than that of the other scheduling. The speedup from FIFO to LIFO+HRF in the 8, 16, 24-core experiments is about 1.9 times. The result shows that the disk cache hit rate is a significant factor for performance when data size per node is large.

In the range of 48-96 cores, the elapsed time of all the scheduling methods is close to each other. In the experiment using 12 nodes with 96 cores, the file size per node is about 1/12 of the total size. For example, the total file size of mProjectPP output is about 20 GB, but the files size processed per node is about 1.7 GB. This size is small enough to load in the main memory of 32 GiB. Therefore, cache access is dominant even for the FIFO scheduling. Among the four scheduling methods, only the elapsed time of the LIFO scheduling is longer than other scheduling methods. On the other hand, the LIFO+HRF scheduling produces the best performance over the entire range of the number of cores.

2) Improvement of Core Utilization with HRF

In this subsection, we investigate the relationship between scheduling and core utilization. The elapsed time of the workflow using 12 nodes with 96 cores is plotted in Fig. 12. The result shows 12 % improvement by HRF. The result of FIFO is close to the LIFO+HRF and Rank+HRF since cache access is dominant in this experiment.

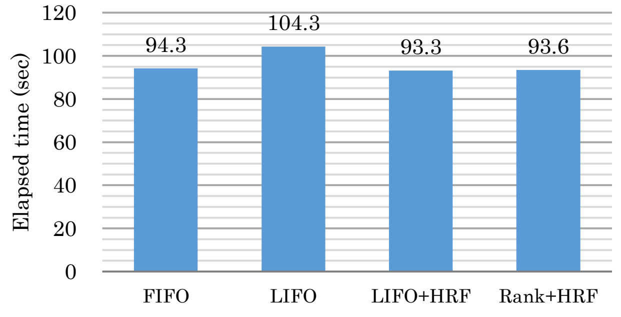

Fig. 12. Elapsed time of Montage workflow using 96 cores.

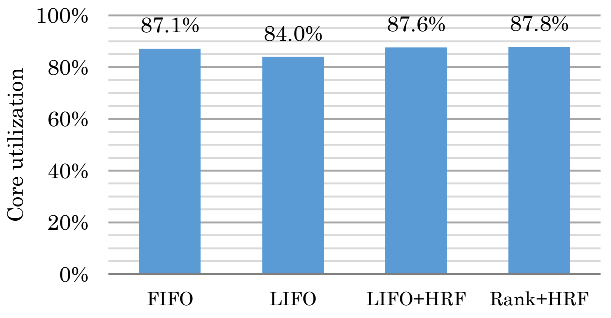

Fig. 13. Core utilization during Montage workflow.

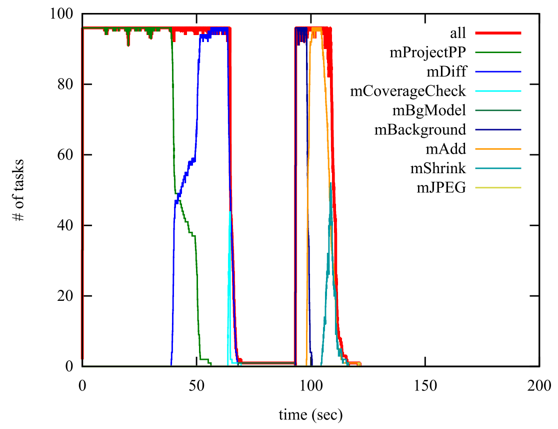

Fig. 14. The time transition of the number of tasks during Montage workflow. Scheduling: FIFO for task order, MCGP for locality.

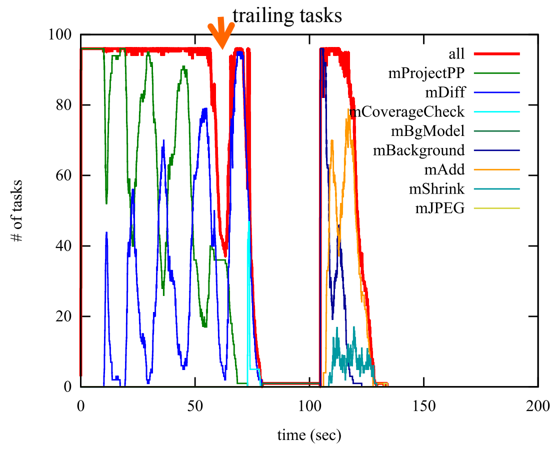

Fig. 15. The time transition of the number of tasks. Scheduling: LIFO, MCGP. The arrow indicates decrease in core utilization due to the trailing task problem.

We define the core utilization as $\frac{t_{\mathrm{cum}}}{t_{\mathrm{elap}} n_{\mathrm{cores}}}$, where t_{cum} is the cumulative execution time of tasks, t_{elap} is the elapsed time of the workflow, and n_{cores} is the number of cores. The cumulative execution time t_{cum} is derived as the summated elapsed time of all the tasks. The obtained core utilization is plotted in Fig. 13. The result shows that the core utilization of LIFO is about 84%, while that those of other methods are about 87-88%. The result shows that the major cause of lower performance of LIFO is core utilization.

In order to see in detail how the execution order affects the core utilization, we plot the time variation of the number of tasks during the workflow execution in Fig. 14 - Fig. 17. In these plots, red thick line shows the total number of tasks, and the other lines show the number of each kind of tasks.

Fig. 14 is the result of the FIFO scheduling. The number of running tasks is close to 96 (same as the number of cores) almost all the time except the period of singly-executed tasks (mBgModel, last mAdd and mJPEG). Fig. 14 shows that tasks are invoked in the descending order of rank. For example, all the mProjectPP tasks are invoked at first, and then all the mDiff tasks are invoked.

Fig. 15 is the result of the LIFO scheduling. It shows that the mProjectPP and mDiff tasks are invoked alternately. This execution order is intended to raise disk cache hit rate (In this experiment, however, it is not effective for disk cache hit rate since the data size is small.) On the other hand, the total number of tasks decreases around 60 sec at the position indicated by an arrow. This is because the number of the mProjectPP tasks that remain in the queue becomes less than the number of worker cores. This is the trailing task problem.

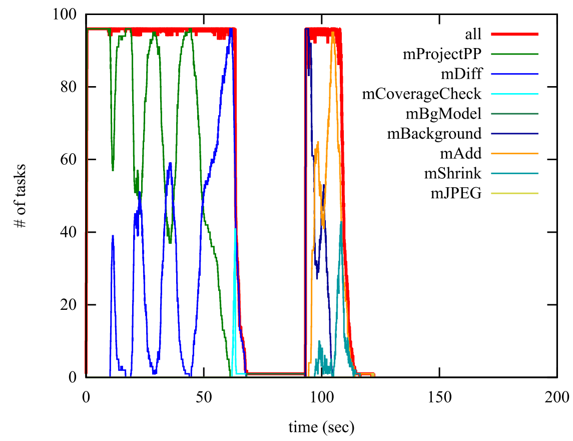

Fig. 16. The time transition of the number of tasks. Scheduling: LIFO+HRF, MCGP.

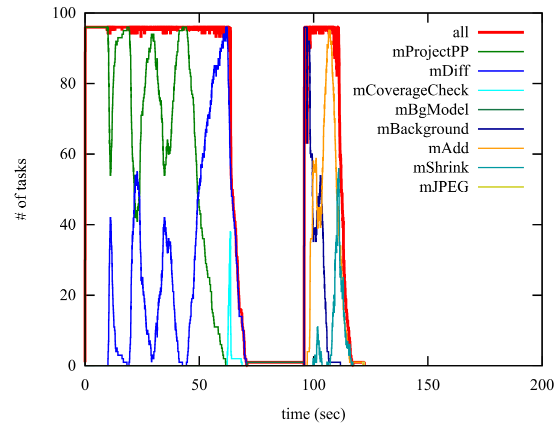

Fig. 17. The time transition of the number of tasks. Scheduling: Rank+HRF, close-to-file.

Fig. 16 is the result of the LIFO+HRF scheduling. It shows alternate task invocation similar to LIFO. However, a decrease in the total number of tasks due to trailing tasks is *not* seen in the LIFO+HRF experiment. This result demonstrates that the proposed LIFO+HRF scheduling resolves the trailing task problem for the Montage workflow which is more complex than the example workflow shown in Fig. 3.

3) *Rank Equalization*

In Montage workflow, mProjectPP is compute-intensive and mDiff is I/O-intensive. Therefore, the overlap of mProjectPP and mDiff can improve performance. Fig. 17 show the number of tasks in the experiment of Rank Equalization with HRF (Rank+HRF). The result shows almost same behavior and performance as the Fig. 16 (LIFO+HRF). In this configuration of workflow, the rank equalization is failed to increase performance since the core utilization of mDiff tasks is almost always less than 50%.

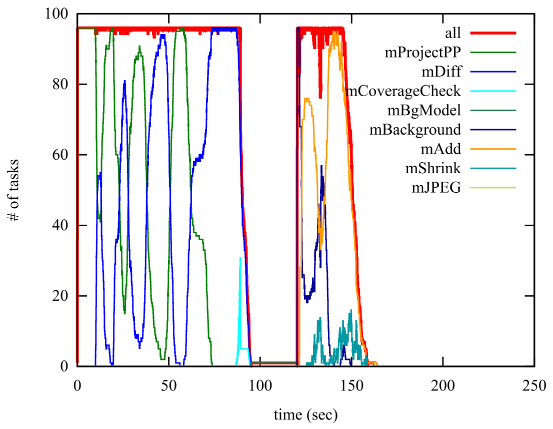

Fig. 18. The time transition of the number of tasks. Scheduling: LIFO+HRF, close-to-file for locality.

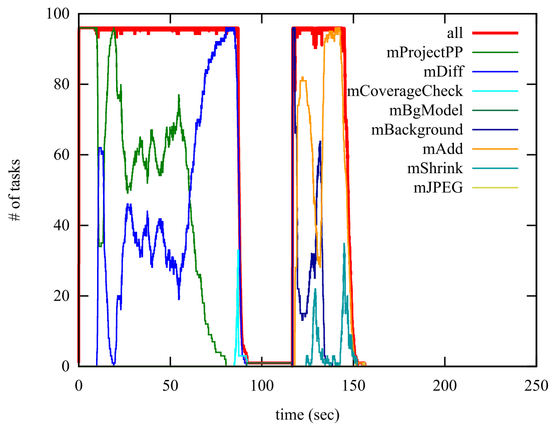

Fig. 19. The time transition of the number of tasks. Scheduling: Rank+HRF, close-to-file.

We here demonstrate another experiment where the Rank Equalization is effective. This is achieved by changing the locality option from MCGP to the location of input files. This change degrades local access rate and makes mDiff execution time longer. The result of this experiment is shown in Fig. 18 and Fig. 19. The plot of LIFO+HRF (Fig. 18) shows that the number of mDiff tasks oscillates and exceeds 48 (50% utilization). On the other hand, the plot of Rank+HRF (Fig. 19) shows that the number of mDiff tasks is flat and less than 48 in the period of 20-60 sec. The elapsed time (excluding sequential tasks) is 134.4 sec for LIFO+HRF and 128.7 sec for Rank+HRF. Thus, Rank Equalization improves performance by 4.4%. This improvement is due to the overlap of mProjectPP (computation-intensive task) and mDiff (I/O-intensive task). In conclusion, the scheduling to overlap computation and I/O can improve the performance of particular workflows.

V. Related Work

A. *Workflow System*

Pegasus [10] is a workflow system for grid computing. Kepler [11] is a workflow system for connecting Web services. These systems are not targeted at MTC since task throughput (tasks/sec) is not pursued. MapReduce [12] is a programming model for a large number of data-intensive parallel tasks. It is only applicable to specific patterns of workflows.

Swift [13] + Falkon [14] + Data diffusion [15] is a combination of frameworks targeted at MTC on Grid systems. Swift is a workflow system which employs Swiftscript dedicated for scientific workflows. On the other hand, Pwrake is based on Rake, a powerful workflow language with scripting capability. Falkon is a framework for fast task dispatch as which reports 1500 tasks/sec at the peak performance. It requires piggy-backing and task bundling. Data diffusion is a mechanism for task scheduling and management of staged files to the execution node, in consideration of the locality. In the Pwrake system, explicit file staging is not required since we assume files are shared by the Gfarm file system.

GXP make [16] is a system using the GNU make as a workflow definition language, built on the GXP system for distributed parallel execution. The mechanism of GXP make is that the "mksh" process traps command lines issued by GNU make and dispatches to remote nodes. Therefore, the order of execution depends on GNU make implementation. In Pwrake, we implemented the scheduling to control the order of execution.

B. *Workflow Scheduling*

Yu et al. [17] surveyed various workflow scheduling algorithms for Grid systems. These algorithms are heuristic or meta-heuristic algorithms based on the prior information of tasks. On the other hand, our work is aimed at MTC workflows comprised of thousands or millions of tasks on computer clusters, not provided by prior information on task execution time.

Locality-aware scheduling for data-intensive workflows has been studied by [18][19][20][21]. Shankar and DeWitt [22] studied DAG-based data-aware workflow scheduling for the Condor system. They focused on cached data on a local disk in order to avoid data movement. On the other hand, our study is aimed at the scheduling for cached data on memory.

Armstrong et al. [5] discussed the trailing task problem that occurs in MTC and proposed the tail-chopping approach, where remaining tasks are run on a smaller allocation of compute resources. We study another approach HRF, which determines the order of task execution using the information of DAG.

VI. Conclusion

Pwrake is a workflow system developed for data-intensive and many-task computing and utilizes the Gfarm file system for file sharing among worker nodes instead of file staging. We discussed that the LIFO scheduling is effective to maximize the probability that input files are cached. We proposed two scheduling methods for improving the performance of every range of workflows. One is the hybrid scheduling of LIFO and HRF (Highest Rank First) in order to mitigate the "trailing task problem" which is a disadvantage of the LIFO scheduling. The other is a hybrid of Rank Equalization and HRF, in quest of the overlap of computation and I/O. We evaluated the proposed scheduling using a computer cluster by executing data-intensive workflows. In the experiment of copyfile workflow, the LIFO scheduling improves the workflow performance by 30% from the FIFO scheduling. In the experiment of the Montage astronomy workflow with 96 cores, the HRF scheduling eliminates trailing tasks and improves by 12%. On the other hand, the Rank Equalization improves performance if computation and I/O are effectively overlapped.

ACKNOWLEDGMENT

This work is supported by JST CREST "System Software for Post Petascale Data Intensive Science" and JST CREST "Extreme Big Data (EBD) Next Generation Big Data Infrastructure Technologies Towards Yottabyte/Year"

REFERENCES

[1] I. Raicu, I. T. Foster, and Y. Zhao, "Many-task computing for grids and supercomputers," in *Workshop on Many-Task Computing on Grids and Supercomputers, 2008 (MTAGS 2008)*, 2008, pp. 1–11.

[2] M. Tanaka and O. Tatebe, "Pwrake: A parallel and distributed flexible workflow management tool for wide-area data intensive computing," in *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing (HPDC '10)*, 2010, pp. 356–359.

[3] O. Tatebe, K. Hiraga, and N. Soda, "Gfarm Grid File System," *New Gener. Comput.*, vol. 28, no. 3, pp. 257–275, Aug. 2010.

[4] M. Tanaka and O. Tatebe, "Workflow Scheduling to Minimize Data Movement Using Multi-constraint Graph Partitioning," in *2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (ccgrid 2012)*, 2012, pp. 65–72.

[5] T. G. Armstrong, Z. Zhang, D. S. Katz, M. Wilde, and I. T. Foster, "Scheduling many-task workloads on supercomputers: Dealing with trailing tasks," in *2010 3rd Workshop on Many-Task Computing on Grids and Supercomputers*, 2010, pp. 1–10.

[6] H. Topcuoglu, S. Hariri, and M.-Y. Wu, "Performance-effective and low-complexity task scheduling for heterogeneous computing," *IEEE Trans. Parallel Distrib. Syst.*, vol. 13, no. 3, pp. 260–274, Mar. 2002.

[7] H. Saito, Y. Kamoshida, S. Sawai, K. Hironaka, K. Takahashi, T. Sekiya, N. Dun, T. Shibata, D. Yokoyama, and K. Taura, "InTrigger: A Multi-Site Distributed Computing Environment Supporting Flexible Configuration Changes," *IPSJ SIG Tech. Rep. 2007-HPC-111*, vol. 2007, no. 80, pp. 237–242, 2007.

[8] J. C. Jacob, D. S. Katz, G. B. Berriman, J. C. Good, A. C. Laity, E. Deelman, C. Kesselman, G. Singh, M.-H. Su, T. A. Prince, and R. Williams, "Montage: a grid portal and software toolkit for science-grade astronomical image mosaicking," *Int. J. Comput. Sci. Eng.*, vol. 4, no. 2, pp. 73–87, Jul. 2009.

[9] K. N. Abazajian, J. K. Adelman-McCarthy, M. A. Agüeros, S. S. Allam, C. A. Prieto, D. An, K. S. J. Anderson, S. F. Anderson, J. Annis, N. A. Bahcall et al., "The Seventh Data Release of the Sloan Digital Sky Survey," *Astrophys. J. Suppl. Ser.*, vol. 182, no. 2, pp. 543–558, Jun. 2009.

[10] E. Deelman, G. Singh, M.-H. Su, J. Blythe, Y. Gil, C. Kesselman, G. Mehta, K. Vahi, G. B. Berriman, J. Good, A. Laity, J. C. Jacob, and D. S. Katz, "Pegasus: A framework for mapping complex scientific workflows onto distributed systems," *Sci. Program.*, vol. 13, no. 3, pp. 219–237, 2005.

[11] B. Ludäscher, I. Altintas, C. Berkley, D. Higgins, E. Jaeger, M. Jones, E. A. Lee, J. Tao, and Y. Zhao, "Scientific workflow management and the Kepler system," *Concurr. Comput. Pract. Exp.*, vol. 18, no. 10, pp. 1039–1065, Aug. 2006.

[12] J. Dean and S. Ghemawat, "MapReduce: simplified data processing on large clusters," *Commun. ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[13] Y. Zhao, I. Raicu, I. Foster, M. Hategan, V. Nefedova, and M. Wilde, "Realizing Fast, Scalable and Reliable Scientific Computations in Grid Environments," in *Grid Computing Research Progress*, Nova Publisher, 2008.

[14] I. Raicu, Y. Zhao, C. Dumitrescu, I. Foster, and M. Wilde, "Falkon: a Fast and Light-weight tasK executiON framework," in *Proceedings of the 2007 ACM/IEEE conference on Supercomputing*, 2007, pp. 43:1–43:12.

[15] I. Raicu, Y. Zhao, I. T. Foster, and A. Szalay, "Accelerating large-scale data exploration through data diffusion," in *Proceedings of the 2008 international workshop on Data-aware distributed computing - DADC '08*, 2008, pp. 9–18.

[16] K. Taura, T. Matsuzaki, M. Miwa, Y. Kamoshida, D. Yokoyama, N. Dun, T. Shibata, C. S. Jun, and J. Tsujii, "Design and implementation of GXP make — A workflow system based on make," *Futur. Gener. Comput. Syst.*, vol. 29, no. 2, pp. 662–672, Feb. 2013.

[17] J. Yu, R. Buyya, and K. Ramamohanarao, "Workflow scheduling algorithms for grid computing," in *Metaheuristics for Scheduling in Distributed Computing Environments*, vol. 146, Fatos Xhafa and A. Abraham, Eds. Springer Berlin Heidelberg, 2008, pp. 173–214.

[18] W. Xiaohui, W. W. Li, O. Tatebe, X. Gaochao, H. Liang, and J. Jiubin, "Implementing data aware scheduling in Gfarm using LSF scheduler plugin mechanism," *Int. Conf. GRID Comput. Appl. (GCA'05*, pp. 3 – 10, 2005.

[19] A. Ramakrishnan, G. Singh, H. Zhao, E. Deelman, R. Sakellariou, K. Vahi, K. Blackburn, D. Meyers, and M. Samidi, "Scheduling Data-Intensive Workflows onto Storage-Constrained Distributed Resources," in *Seventh IEEE International Symposium on Cluster Computing and the Grid (CCGrid '07)*, 2007, pp. 401–409.

[20] Z. Ding, X. Wei, Y. Zhu, Y. Yuan, W. W. Li, and O. Tatebe, "Implementation of the Grid Workflow Scheduling for Data Intensive Applications as Scheduling Plug-ins," in *2008 Second International Conference on Future Generation Communication and Networking Symposia*, 2008, vol. 5, pp. 14–20.

[21] I. Raicu, I. T. Foster, Y. Zhao, P. Little, C. M. Moretti, A. Chaudhary, and D. Thain, "The quest for scalable support of data-intensive workloads in distributed systems," in *Proceedings of the 18th ACM international symposium on High performance distributed computing - HPDC '09*, 2009, p. 207.

[22] S. Shankar and D. J. DeWitt, "Data driven workflow planning in cluster management systems," in *Proceedings of the 16th international symposium on High performance distributed computing - HPDC '07*, 2007, p. 127.