Published: 31 July 2018
Accepted: 01 April 2018
Revised: 01 February 2018
Received: 01 July 2016

RESEARCH-ARTICLE

Adaptive Process Migrations in Coupled Applications for Exchanging Data in Local File Cache

JIANWEI LIAO, Southwest University, Chongqing, China

ZHIGANG CAI, Southwest University, Chongqing, China

FRANCOIS TRAHAY, Telecom SudParis, Evry, Ile-de-France, France

JUN ZHOU, Southwest University, Chongqing, China

GUOQIANG XIAO, Southwest University, Chongqing, China

ACM Transactions on Autonomous and Adaptive Systems (TAAS), Volume 13, Issue 2 (June 2018)
https://doi.org/10.1145/3226027
EISSN: 1556-4703

Adaptive Process Migrations in Coupled Applications for Exchanging Data in Local File Cache

JIANWEI LIAO and ZHIGANG CAI, Southwest University, China
FRANCOIS TRAHAY, Telecom SudParis, France
JUN ZHOU and GUOQIANG XIAO, Southwest University, China

Many problems in science and engineering are usually emulated as a set of mutually interacting models, resulting in a coupled or multiphysics application. These component models show challenges originating from their interdisciplinary nature and from their computational and algorithmic complexities. In general, these models are independently developed and maintained, so that they commonly employ the global file system for exchanging their data in the coupled application.

To effectively use the local file cache on the compute node for exchanging the data among the processes of such applications, and consequently boosting I/O performance, this article presents a novel mechanism to migrate a process from one compute node to another node on the basis of block I/O dependency. In this newly proposed mechanism, the block I/O dependency between two involved processes running on the different nodes is profiled as block access similarity by taking advantage of the *Cohen's kappa statistic*. Then, the process is supposed to be dynamically migrated from its source node to the destination node, on which there is another process having heavy block I/O dependency. As a result, both processes can exchange their data by utilizing the local file cache instead of the global file system to reduce I/O time. The experimental results demonstrate that the I/O performance can be significantly improved, and the time required for executing the application can be resultantly decreased, as expected.

Categories and Subject Descriptors: D.4.3 [**File Systems Management**]: Distributed File Systems

General Terms: Design, Algorithms, Performance

Additional Key Words and Phrases: Coupled application, computing process migration, distributed file systems, block I/O dependency, I/O performance

ACM Reference format:
Jianwei Liao, Zhigang Cai, Francois Trahay, Jun Zhou, and Guoqiang Xiao. 2018. Adaptive Process Migrations in Coupled Applications for Exchanging Data in Local File Cache. *ACM Trans. Auton. Adapt. Syst.* 13, 2, Article 9 (July 2018), 25 pages.
https://doi.org/10.1145/3226027

This work was partially supported by Hunan Provincial Natural Science Foundation of China (No. 2018JJ2309), the Fundamental Research Funds for the Central Universities (No. XDJK2017B044 and XDJK2018D013), and the Opening Project of State Key Laboratory for Novel Software Technology (No. KFKT2016B05).

Authors' addresses: J. Liao (corresponding author), College of Computer and Information Science, Southwest University of China, and State Key Lab. for Novel Software Technology, Nanjing University, P.R. China; email: liaotoad@gmail.com; F. Trahay, Department of Computer Science, Telecom SudParis, France; email: francois.trahay@telecom-sudparis.eu; Z. Cai, J. Zhou, and G. Xiao, College of Computer and Information Science, Southwest University of China; emails: {czg, zhouj, gqxiao}@swu.edu.cn.

https://doi.org/10.1145/3226027

1 INTRODUCTION

High-performance distributed computing systems, which generally have compute nodes for running jobs and storage nodes for offering I/O services, are commonly employed to solve certain large-scale scientific applications in both research institutions and industry (Vecchiola 2009). Before running an application, its processes are normally placed onto the compute nodes according to some policies, such as the round-robin fashion and the opportunity cost approach (Amir 2002). To enable dynamic movement of the processes during execution, the approach of process migration is leveraged to move processes from source compute nodes to other nodes in distributed systems for the purposes of load balancing, locality of resource usage, and fault tolerance (Milojicic 2000). Specifically, the consistent state of the process, which contains heap data, stack content, processor registers, address space, and process communication state, is supposed to be migrated, and then the process continues its execution on another node from its break point (Liao 2012b).

The coupled workflows consisting of multiple individually developed component models have been widely used in Big Data processing applications. For example, SCALE-LETKF, which can provide high-resolution, real-time weather forecasting of severe guerrilla rainstorms in Japan, has the SCALE model for data simulation, as well as the LETKF model for data assimilation (Miyoshi 2016). More importantly, both models have data dependency with each other, and they exchange more than 1TB data in total for making a predication by using the global file system. As a consequence, one of the most crucial challenges to be met is about how to speed up data access and transfer requested data by such applications (Dongarra 2011; Zhang 2012). To be specific, a remarkable discovery has arisen from the study area of data I/O in high-performance computing: The I/O transfer time needed for exchanging the data among processes belonging to the same application depends on both the affinity of the processes and their locations of host compute nodes (Jeannot 2014). That is, when two processes are located in the same compute node, both of them can exchange the data by accessing the local file cache instead of the global file system. Therefore, the time required for data I/O exchange can be significantly decreased.

Targeting to accelerate the data exchange in coupled applications, our motivation is to dynamically schedule the processes to compute nodes in high-performance distributed compute systems and then to allow these processes to exchange the data with other processes running on the same nodes. This article intends to propose a mechanism of dynamic process migration according to the block data dependency, which is revealed by the storage nodes, and the migration is transparent to the applications. To put this mechanism to work, we first propose an algorithm to figure out the block I/O dependency between two processes running on the different compute nodes, by utilizing *Cohen's kappa statistic*. Then the storage node may trigger the client file systems to conduct dynamic process migration. As a consequence, not only the I/O workloads can be balanced, but also I/O data throughput can be greatly improved, because of taking advantage of the local file cache to exchange the data. In brief, the article makes the following two contributions:

(1) *Introducing a novel scheme to estimate the block I/O dependency of two involved processes running on different compute nodes.* We first pre-process the block access sequence by dividing it into several epochs after ordering all access events on the basis of their target block identifiers. Then, we employ the *kappa* statistic to indicate the I/O track consistency of two involved block access sequences belonging to different two processes, which is also termed as block I/O dependency in the article.
(2) *Proposing the scheme of process migration, which can dynamically move a process from the source compute node to the destination node, according to the block I/O dependency.* To be specific, the process can be migrated to the node, on which there is another process having tight block data dependency with it. Consequently, the I/O performance can be greatly

boosted by allowing the both processes to exchange their data by making use of the local file cache.

The remainder of the article is organized as follows: Section 2 presents the related work concerning the topics of process migration, block I/O-based optimization strategies, and existing tools for coupling models. The design and implementation details of this newly proposed process migration technique are illustrated in Section 3. Section 4 introduces the evaluation methodology and discusses experimental results. At last, we make concluding remarks in Section 5.

2 RELATED WORK

A large number of related work focusing on process live migration, I/O optimization strategies on the basis of block I/O access on storage nodes, and coupling tools has been consecutively proposed.

Mechanisms of process live migration. There are many approaches of process migration for the different purposes, such as fault tolerance and load balancing (Petri 1995; Barker 2004). To achieve fault tolerance and provide high reliable service, many researchers have introduced various schemes of process migration for different application contexts in HPC (Wang 2008; Ouyang 2010; Ibrahim 2011). These schemes merge health monitoring of nodes with live migration to provide proactive fault tolerance. The migration of process running an Message Passing Interface (MPI) rank is named live, because it occurs in parallel with the execution of the application. Furthermore, Cores (2014) have introduced an application-level checkpoint/restart framework to proactively move the MPI processes when impending failures are notified, instead of restarting the entire application. Besides, for the purpose of balancing workloads on the compute nodes, certain techniques aim to migrate the processes from overloaded nodes to the nodes having less workloads (Vyas 2014). To improve acceleration process migration, X. Ouyang et al. have proposed the RDMA-based migration scheme (Ouyang 2011a) and the user-level file system called CRFS, which is designed with checkpoint/restart I/O traffic to efficiently deal with the concurrent write requests (Ouyang 2011b).

Especially, with the emerging of virtualization technique (Williams 2012; Xu 2014b), several virtual machine replication and migration approaches have been presented to yield fault tolerance and load balancing in high-performance computing (Median 2014; Ahmad 2015; Mashtizadeh 2014; Xu 2014). To be specific, in the ideal copy of a virtual machine (VM), the complete and consistent state from the original virtual machine should be mirrored as an image copy, including its memory, disk, and network connections, and then the copy of image is transferred to another physical compute node. In essence, the VM migration is an extension of the approach of process migration.

Mechanisms of moving computing processes to storage nodes. This kind of mechanism is also called "active storage" (Ridel 1998), which can contribute to I/O-bound applications from two principle ways: (1) parallelism acceleration, where all involved storage nodes can carry out the computation by using their own data at the same time, when they have available computing resources, and (2) bandwidth reduction, where the transferred summary statistics or intermediate results are a very small fraction of the amount size of input data from the storage nodes to the host compute node (Xie 2016; Choudhary 2015). However, because of resource limitations on storage nodes, the migrated processes are likely to use certain restricted kernel functions only offered by the compute nodes in some cases. That means it might be impossible to run this kind of tasks on the storage nodes. Moreover, direct access to the storage nodes in HPC is not commonly allowed in real-world production environments due to the reason of security (Zheng 2013).

Mechanisms of I/O optimization on the basis of block I/O access pattern. Li (2004) introduced a data-mining approach called *C-miner* to disclose block correlations in storage systems on a local machine, and then the file system can take advantage of the discovered block correlations for guiding I/O optimization strategies, such as data prefetching and data movement, to boost system performance. Similarly, S. Jiang et al. (Ding 2007; Jiang 2013) have proposed *DiskSeen*, which employs a frequent sequence-based pattern modeling technique to classify block access patterns, and both temporal and spatial correlations of block access events have been taken into account for advancing the sequentiality of disk accesses and overall prefetching performance.

After understanding the fact of the block access events of the process occurred on the storage nodes have certain regularities, we have proposed and implemented a server-side prefetching mechanism in distributed file systems (Liao 2016). This novel I/O optimization scheme takes advantage of the block access patterns to direct prefetching data on the storage nodes, and then the prefetched data are proactively forwarded to the client node. As a consequence, this mechanism can achieve the goals of accelerating the execution of the applications running on the resource-limited compute nodes (i.e., client nodes) and enhancing I/O data throughput.

Coupling tools for separated models or applications. The Model Coupling Toolkit (*MCT*) is a library offering routines and datatypes to build a coupled system and widely used in Community Climate System Model (*CCSM*) (Larson 2005). The *OASIS coupler* is another library that is capable of processing synchronized exchanges of coupling information generated by different models of the climate system and the separate coupler mediates communication among the components (Valcke 2006).

In our previous work (Liao 2017), we have proposed an I/O arbitrator middleware. This middleware employs the MPI communication facility to support direct parallel data transfer among job components that rely on the netCDF interface for performing I/O operations. We have also deployed this middleware in the SCALE-LETKF application and then conducted a case study to verify the effectiveness of the middleware.

However, these coupling tools, including our previously proposed one, require us to modify the source codes of applications more or less to use their specific interfaces. That is, they cannot keep the transparency to application users and fail to provide a general solution to all cases. Therefore, this article aims to propose a technique to accelerate the execution of coupled applications when they rely on file systems for exchanging data, without any modifications to the applications. To this end, it adopts the *Cohen's kappa statistic* to estimate the block I/O dependency of two involved processes running on different compute nodes. After that, it dynamically directs process migrations, according to their block data dependency. Consequently, the paired processes are allocated on the same compute node, so that they can exchange data via accessing the local file cache, to reach the target of shortening the time required by I/O operations.

3 DESIGN AND IMPLEMENTATION

This section presents design and implementation specifications of the proposed dynamic process migration based on data dependency. First, an overview of the proposed mechanism is presented. Then, it depicts the algorithm of estimating block I/O data dependency, which is leveraged to determine whether process migration should be triggered or not. Next, the specifications of dynamic process migration are described. At last, the implementation details are explicitly illustrated.

3.1 Functional Overview

Figure 1 demonstrates the overall architecture of the newly proposed mechanism when the system has eight compute nodes and eight storage nodes. From the figure, we can see that a process running on *Comp. node I* has a tight block I/O dependency relationship with another process running on *Comp. node III*. In other words, the output data of a process running *Comp. node I* are requested by another process running *Comp. node III*. After the former process has been migrated to *Comp. node III*, both processes can employ *File Cache* to exchange their data. So that the time required by

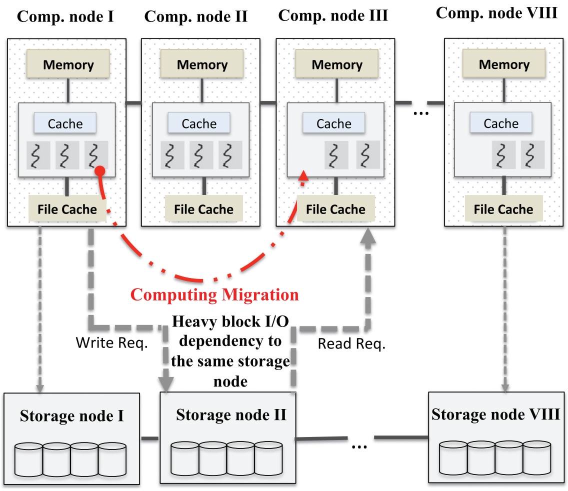


Fig. 1. The illustration of dynamic process migration on the basis of data dependency (note: data dependency can be represented with the block access pattern similarity).

I/O operations can be significantly reduced, in contrast to relying on the storage nodes to exchange the data.

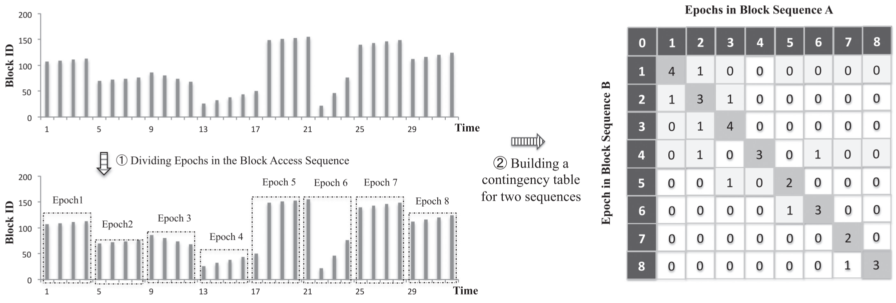
Note that Figure 1 shows the processes having data dependency are executed on the same compute node, after migration. It is also enabled to migrate the target process to a nearby compute node that shares the same distributed cache, so that the paired processes can exchange their data by accessing the shared distributed cache. Note that this newly proposed mechanism does not focus on optimizing file local cache systems; it aims to decrease the time needed for data exchanges among processes via accessing local cache systems.

To effectively manage the cached block data, we employ the write-invalidate algorithm to ensure data consistency in the system. Specifically, the client file system buffers the written contents in the cache and then invalidates other data replicas. When receiving an I/O request, the client file system first checks whether the requested data have been buffered in the local file cache or not. And then it decides to access the cached data or communicate with storage servers for operating the target data.

To keep minimal overhead to conduct process migrations, the proposed mechanism explores the matched process-pairs as much as possible, for migrating multiple processes in parallel, once a matched process-pair (in which two processes have heavy block data dependency) has been found in the routine scan. Therefore, the downtime caused by checkpoint/restart of migration can be greatly cut down.

3.2 Block I/O Dependency

Note that in our mechanism, the block IDs refer to the unique identifications of data chunk at the distributed file system level, and not at the disk level, which are local to each storage server. For rating I/O dependency of two processes running on the different compute nodes, we first employ the mechanism proposed in our previous work (Liao 2016) to piggyback the process identification with I/O requests so that the server side block access history can include the process information. After that, we pre-process the time series of block access sequences, which is related to the processes, for estimating the block I/O dependency of two involved processes.


Block ID
200
150
100
50
0
1
5
9
13
17
21
25
29
Time
① Dividing Epochs in the Block Access Sequence
Epoch1
Epoch2
Epoch 3
Epoch 4
Epoch 5
Epoch 6
Epoch 7
Epoch 8
② Building a contingency table for two sequences
Epochs in Block Sequence A
Epoch in Block Sequence B

0	1	2	3	4	5	6	7	8
1	4	1	0	0	0	0	0	0
2	1	3	1	0	0	0	0	0
3	0	1	4	0	0	0	0	0
4	0	1	0	3	0	1	0	0
5	0	0	1	0	2	0	0	0
6	0	0	0	0	1	3	0	0
7	0	0	0	0	0	0	2	0
8	0	0	0	0	0	0	1	3

(a) Pre-processing the block access sequence belonging to a specific process

(b) Generating an example contingency table

Fig. 2. Pre-processing the block access sequence (a), and building the contingency table for two involved access sequences(b).

3.2.1 Pre-Process Block Sequences of Process-Pair. The target of two block I/O access sequences generated by the different processes are primarily disposed, and then a square contingency table is constructed to express the access consistency relationship between two block access sequences. The procedure is described as follows:

(1) After collecting the sequence of block access events issued by the specific process in a certain period, we split the block access sequence into many epochs, and each epochs has a fixed number of block access events by referring both temporal and spatial correlations of block access events. Figure 2(a) demonstrates the details of this step, and then the block access sequence can be split into multiple epochs in the time order.

(2) We build a contingency table, in which there are $Number_{epoch} \times Number_{epoch}$ elements, for the two block access sequences in the process-pair, when the sequences associated with two separated processes have been pre-processed.

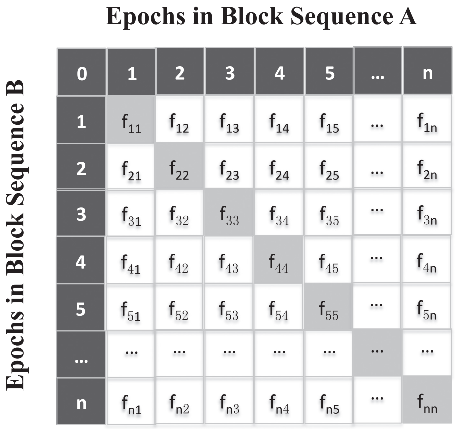
    All elements in the table are initialized as 0, and the value of each element can be set as the value of the coincidence indicator, according to *Definition* 3.1. Once all elements have been set, the construction of the contingency table is completed, and this table will be utilized to rate the similarity of two block access sequences.

Definition 3.1 (Coincidence Indicator). the indicator value of an element in the table is the total count of block access events, which occur in the corresponding epochs of two involved block access sequences.

Figure 2(b) shows an example contingency table of two involved block access sequences, i.e., *Block Access Sequences A* and *B*. To be specific, the value of *Element [2][2]* in the table is 3, which indicates *three* access events appear in *Epoch 2* of both access sequences. And the value of *Element [2][3]* in the table is 1, which means only *one* access event existing in *Epoch 2* of the sequence of *B*, and it has also appeared in *Epoch 3* of the sequence of *A*.

3.2.2 Assessing Block I/O Dependency Using kappa. There are many correlation coefficients that can determine a statistical relationship between two sequences of variables, such as the *Pearson* correlation coefficient (Benesty 2009), the *Spearman* correlation coefficient (Myers 2006), and the *Kendall* correlation coefficient (Abdi 2007). These correlation coefficients are effective to identify that the numbers in two sequences have a same trend, or an exactly opposite trend.

On the other side, the block I/O sequence is a kind of categorical data, as there is no intrinsic ordering to different block numbers. Thus, the commonly used coefficients for leveling the sequence similarity cannot benefit to calculating the similarity of two block access sequences in our context.



Epochs in Block Sequence A

Epochs in Block Sequence B

0	1	2	3	4	5	...	n
1	f_{11}	f_{12}	f_{13}	f_{14}	f_{15}	...	f_{1n}
2	f_{21}	f_{22}	f_{23}	f_{24}	f_{25}	...	f_{2n}
3	f_{31}	f_{32}	f_{33}	f_{34}	f_{35}	...	f_{3n}
4	f_{41}	f_{42}	f_{43}	f_{44}	f_{45}	...	f_{4n}
5	f_{51}	f_{52}	f_{53}	f_{54}	f_{55}	...	f_{5n}
...	...	...	...	...	...	...	...
n	f_{n1}	f_{n2}	f_{n3}	f_{n4}	f_{n5}	...	f_{nn}

Fig. 3. The built contingency table of two involved block access sequences, which is used for calculating the Cohen's *kappa* statistic.

Since *Cohen's kappa statistic* measures inter-rater agreement for qualitative (categorical) items (Joseph 1969; Anthony 2005), it can be naturally leveraged to estimate the similarity of two block access sequences. For instance, when the *kappa* statistic is 1, the involved two variables have a perfect consistency, and the involved two variables do not have any consistency in the case of the *kappa* statistic is 0. Through elaborately modeling for input categorical data of block access sequences, we can figure out whether two sequences are similar or not. As a result, we can conduct process migrations to group relevant processes running on the same compute nodes.

To be specific, the *kappa* statistic is calculated from the observed and expected frequencies on the diagonal of a square contingency table (Joseph 1969). By referring to Figure 3, assume that there are $n \times n$ elements to measure the similarity A and B, and suppose that there are n distinct explicit outcomes (n Epochs in the illustration) for both A and B. Let f_{ij} implies the frequency of the number of subjects with the ith categorical response for sequence B and the jth categorical response for sequence A.

The sum value of the elements in the table can be obtained by the following equation:

$$s = \sum_{i=1}^{n} \sum_{j=1}^{n} f_{ij}. \tag{1}$$

The observed proportional agreement between A and B can be defined as

$$p_0 = \frac{1}{s} \sum_{i=1}^{n} f_{ii}. \tag{2}$$

And the expected agreement by chance is

$$p_e = \frac{1}{s^2} \sum_{i=1}^{n} f_{i+} f_{+i}, \tag{3}$$

where f_{i+} is the total for the ith row and f_{+i} is the total for the ith column.

Consequently, *Cohen's kappa statistic* can be obtained with the following equation:

$$\kappa = \frac{p_0 - p_e}{1 - p_e}. \tag{4}$$

According to Equation (4), the value of *kappa* coefficient in the example case shown in Figure 2(b), i.e., κ is 0.686. Thus, it implies that the two block access sequences have a substantial consistent pattern, according to the interpretation of *kappa* (Anthony and Garrett 2005).

Furthermore, to objectively express the consistency of two block access sequences, we use an approach in the probability theory to analyze the value of κ for the purpose of obtaining a verifiable decision. First, we advance a hypothesis H_0, which assumes that the targeted two access sequences ***do not*** have any consistency, so that the expectation of κ is 0. Then, we can compute the variance of κ with Equation (5) (Wang 2005),

$$Var(\kappa) = \frac{1}{s-1} \frac{p_e + p_e^2 - \sum_{i=1}^{n} \frac{f_{i+} f_{+i}}{s^2} \left(\frac{f_{i+}}{s} + \frac{f_{+i}}{s} \right)}{(1 - p_e^2)}. \tag{5}$$

Next, we can obtain the asymptotic distribution of u, which is the standard normal distribution of $N(0, 1)$, as defined in Equation (6),

$$u = \frac{\kappa}{\sqrt{Var(\kappa)}}. \tag{6}$$

Finally, it is possible to determine the consistency of two block access sequences after computing relevant values of κ and u by using Equations (4) and (6), respectively.

- If $\kappa \leq 0$, then we accept the hypothesis of H_0 and affirm that the two block access sequences are not consistent.
- If $\kappa > 0$, which also indicates $u > 0$, then we then compute the probability of P by using Equation (7). In case P is smaller than a pre-defined threshold (0.05 used in our mechanism (DeGroot 2011)), we reject the hypothesis of H_0. Specifically, a small value of P offers evidence against the hypothesis of H_0, because the data have been observed that would be unlikely if H_0 were correct. As a result, we consider the two block access sequences are consistent, to indicate the related process-pair has heavy block I/O dependency,

$$P = probability(Z > u | H_0), Z \sim N(0, 1), \tag{7}$$

where the symbol of Z represents the normal distribution having a mean of 0 and a standard deviation of 1.

3.2.3 Time Complexity and System Scalability. We have explored the time overhead caused by *kappa*-based consistency analysis from the aspects of building the contingency table and counting the *kappa* coefficient. Assuming we have y access events in the sequence, and the size of epoch is x, we can obtain the worst-case time complexity to build the contingency table that has $(y/x)^2$ elements:

$$T_1 = O(x^2 \times (y/x)^2) = O(y^2). \tag{8}$$

Obviously, the time complexity of constructing the relevant contingency table solely depends on the size of block access sequence rather than the granularity of epoch.

The time complexity to calculate the *kappa* coefficient is related to the number of elements in the table, which can be obtained by the following equation:

$$T_2 = O((y/x)^2). \tag{9}$$

The overall time complexity to assess the similarity of two access sequences is the sum of T_1 and T_2:

$$T = O(y^2 + (y/x)^2). \tag{10}$$

At this point, it is true that less time is required for estimating the sequence similarity, when the size of epoch is becoming larger. However, the granularity of epoch is subject to the capacity of local file cache and the number of processes running on the compute nodes. In other words, the local cache is supposed to buffer a minimum of data blocks in a designated epoch. Otherwise, it is impossible to share the cached data, though two processes (running on the same compute node) have similar block access sequences in the unit of a specific large epoch.

We have also studied the time complexity of the proposed *kappa* statistic-based matching algorithm for a process-pair: It creates a contingency table having $Number_{epoch} \times Number_{epoch}$ elements and then computes the *kappa* value for it, which indicates the time complexity of constructing the table is $O(1)$, as the number of epochs in the sequence is a constant due to the configuration.

Consider that we have a total of n processes of application on the compute nodes; the newly proposed scheme will require $O(n^2)$ matching operations to find the targeted migration process(es) in a round of routine scan on the storage node.

The cost for finding the process to migrate is $O(n^2)$, which indicates this approach fails to scale with process count and the number of compute nodes, while the applications have even more than thousands of processes. To address this issue, we offer a configurable policy to compare only m (a configurable variable of $m << n$) processes, which have heavy I/O communications with the storage servers for exploring the processes to be migrated. That is because migrating the processes having less I/O workloads cannot benefit I/O system performance as expected. As a consequence, the proposed matching algorithm results in $O(nlogn + m^2)$ time complexity, and the first part is needed for sorting the processes according to their total numbers of block access events; the second part is required for discovering a matched process among m selected processes. We summarize that when m is much smaller than n, the expected time complexity of the proposed mechanism is $O(nlogn)$.

Moreover, the evaluation experiments reveal the cost of finding processes to be migrated appears acceptable, and the details will be demonstrated in Section 4.

3.3 Process Migration and Load Balance

3.3.1 Dynamic Process Migration. When there is a pair of processes, in which the two processes have block data dependency, the process migration will be triggered. Our prototype system is designed based on the coordinated checkpoint approach of the *BLCR* module (Duell 2000), which is a system-level library, and transparent to the processes. The course of carrying out a process migration in our implementation is specifically demonstrated in Figure 4, and it has the following steps:

(1) The storage server is responsible for triggering the process of migration after the analysis of block I/O dependency. Consequently, the client file systems on the compute nodes are supposed to complete the task.
(2) All relevant MPI processes have to coordinate to reach a consistent global state. In fact, the facility of *BLCR* supports migrating the MPI processes. We take advantage of the LAM/MPI's job-centric interaction mechanism for the MPI tasks to clear in-flight messages in the MPI communication channels.
(3) To reduce the time needed for transferring the checkpointed state (e.g., memory pages in the address space of the migrated process), we employ the MPI communication facility, e.g., `mpi_send()` and `mpi_recv()`, to convey the data of the checkpointed state from *Comp. node I*, to the destination of *Comp. node III*.
(4) The state of migrated process including the values of registers, signal information, opened files will be restored on the destination node. All other MPI processes are suspended at their point of execution.

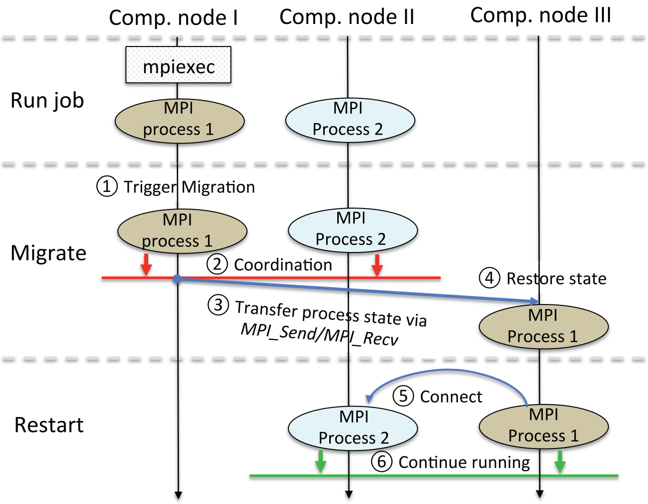


Fig. 4. Process Migrtation: Assuming there is a matched process for *process 1*, running on *Comp. node III*. Then *MPI process 1* is dynamically migrated from *Comp. node I* to *Comp. node III*.

(5) The restarted MPI process sets up a communication channel with all other MPI tasks. Consequently, the drained in-flight messages can be restored, and all the processes can resume execution from the checked point.

(6) All MPI processes are able to continue running, and the restarted process can exchange data with other processes running on the same compute node, by taking advantage of the local file cache.

3.3.2 Load Balance. A difference counter on the client file system is introduced to record the numbers of migrate-in processes and migrate-out processes corresponding to the compute node. All counters are initialized to `0` after processes have been distributed onto compute nodes at the beginning stage. In other words, each counter managed by client client file system corresponds to a compute node, it shows the difference between the number of migrate-in processes and the number of migrate-out processes, after certain dynamic process migrations. For instance, the value of 1 indicates that the compute node has one more process after the initialization distribution.

As a result, process load balance after dynamic migrations can be guaranteed by resorting to the difference counters kept by client file systems. In the case of paired processes running on different compute nodes have been identified, the storage server triggers a migration event with the movement direction, after checking the difference counters on the relevant compute nodes. The process running on the compute node having a larger value of difference counter is supposed to be migrated to another compute node. Besides, through setting the configuration file, we can refuse not only migrating in a process when the difference value is larger than a pre-defined threshold but also migrating out a process when the difference value is smaller than the threshold to avoid extreme uneven process workloads.

3.4 Implementation

For demonstrating the effectiveness of the mechanism of process migration on the basis of block I/O dependency, we have developed a proof-of-concept implementation of the newly proposed technique in *PARTE*. The PARTE file system is a POSIX-compliant distributed file system that has

Table 1. Node Specification of Clusters I and II

	Cluster I	Cluster II
CPU	4×Intel(R)E5800 3.20G	Intel(R)E5410 2.33G
Memory	1×4GB 1066MHz/DDR3	4GB DDR3-SDRAM
Disk	6×114GB 7200rpm SATA	500GB 7200rpm SATA
Network	Intel 82598EB, 10GbE	Intel 82599ES 10GbE
OS	Ubuntu 13.10	Debian 6.0.4
Nodes	5	12

been implemented from scratch in C by referring the design philosophies of the Google file system. The PARTE file system has three modules and runs in the Linux environment:

- *partmds running on a specified server nodes.* The module of active metadata server, which works to offer metadata services for client file systems and storage servers.
- *parteost running on the storage node.* The module of storage server is responsible for the management of real file data. Moreover, it takes charge of profiling block access patterns, judging block I/O dependency periodically, and issuing the command of process migration if required, in the routine scans.
- *partecfs running on the compute node.* The module of client file system has been designed and implemented based on FUSE (FUSE 2011). The client file system supports caching and managing for the cached data by employing the least recently used (LRU) policy. Furthermore, the client file system is supposed to call the checkpoint/restart library to fulfill dynamic process migration, after receiving the commands from storage servers.

4 EVALUATION EXPERIMENTS

In this section, we first introduce the experimental setup, including the experimental platform and the parameter settings. Then the experimental results of running two real-world applications are presented. After that, a case study of executing the developed benchmark presents varied levels of effectiveness caused by the proposed mechanism, when the application having various kinds of data exchange patterns. Finally, we make a brief summary on findings from experiments.

4.1 Experiment Setup

4.1.1 Experimental Platform. We employed two clusters to conduct the designed experiments. The active metadata server and four storage servers were deployed on the 5 nodes of Cluster I, and all client file systems were assigned on the 12 compute nodes of Cluster II. Both clusters are equipped with Open MPI (*version 1.6.5*), and both clusters are connected with a 10 GigE Ethernet. Table 1 shows the specifications of nodes on both of clusters.

Besides, the netCDF library (*version 4.2.2.1*) is installed on the nodes of Cluster II, for executing the applications that require to process netCDF format data. All four cores on each compute node of Cluster II are dedicated for running the benchmarks.

4.1.2 Application Benchmarks and Comparison Counterparts. To show that the proposed mechanism can yield attractive performance improvements for the targeted applications by enabling dynamic process migration, we have performed some experiments to evaluate the proposed scheme. Apart from running three real-world applications, i.e., *SPEEDY-LETKF*, *GAIML2-CLM3*, and the MapReduce version of *k-means* clustering, we have also carried out a case study to show the benefits resulted by the proposed system. Moreover, the case study also demonstrates the accuracy of *kappa*-based dependency analysis and the effects of using varied sizes of epoch in the proposed mechanism.

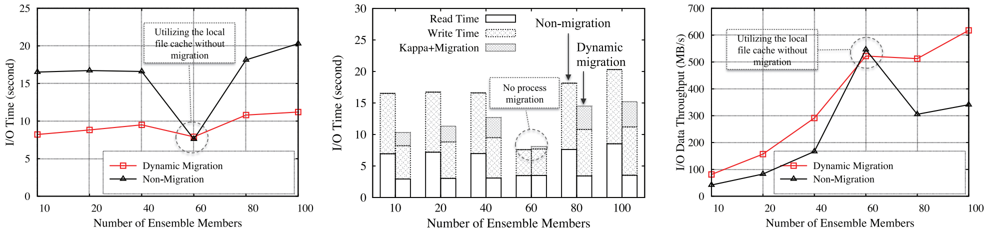
Because there is no related work that migrates on-going processes on the basis of I/O workloads, we used the PARTE distributed file system with the following three configurations to run the benchmarks:

- *Non-Migration*, which is the default process placement policy adopted by most of job schedulers. Specifically, the processes are bound to the compute nodes according to the round-robin fashion, and they are never migrated to other nodes.
- *Dynamic Migration*, which is the distributed file system equipped with the proposed mechanism. It enables dynamic process migration according to block I/O data dependency. At the placement stage, the processes are allocated to the compute nodes by following the fashion of a round-robin. But, the processes can be migrated to other compute nodes during the execution when the data dependency has been detected for the purpose of utilizing the local file cache to exchange the data.
- *OPT Scheme*, which is similar to the *Non-Migration* scheme, as it does not uphold process migration. But it allows initially placing the processes to the compute nodes, on the basis of their data dependency, after analyzing the source codes of benchmarks or post-analyzing the execution logs. In other words, the processes in a process-pair are mapped to the same compute nodes to enable exchanging their data by using the local file cache.

  Note that this scheme is not a general solution, as it must understand the data dependency among the processes, and the layout of processes does not change during the life cycle. That requires us to analyze the source codes of applications or the execution logs. This comparison counterpart is used to illustrate the gain/loss in a global scale caused by our proposed *Dynamic Migration* scheme. Because we have programmed a benchmark to emulate varied data dependency among the processes, and we understand the data dependency among the processes, we use this scheme while executing the developed benchmark.

4.1.3 Default Parameter Settings. In all experiments, the size of the local file cache on the compute node is configured to 128MB (indicating that the client file system can cache $2 \times 1,024$ blocks of data, and each block holds 64KB data). By default, the number of block access events in a block sequence used for characterizing the data access pattern is set as $1,024$, and the number of block access events in each *epoch* is configured as 16. That is, there are 64 *Epochs* in each block access sequence, and the size of contingency table used for calculating the *Cohen's kappa statistic* is 64×64. Two access events are regarded the same when the difference of block IDs of the two events is not greater than 4. Meantime, during the execution of the application, our scheme will profile the block I/O dependency again (i.e., it is called as a routine scan that refreshes the contingency tables and computes the values of κ for process-pairs), if the total number of recent block access events reaches $1,024 \times 12 \times 2$ after the last scan.

Different values of *kappa* indicate diverse levels of agreement of two involved block access sequences, including fair consistency, substantial consistency, and perfect consistency, and so on. In the evaluation experiments, we have employed the level of substantial consistency (the threshold for the value of P in Equation (7) is set as 0.05) for triggering process migration in the experiments, to show the effectiveness of dynamic process migration. Since the total number of processes is not larger, we compare all processes for disclosing the matched process pair in the evaluation experiments, that means the configured value of m is the total number of processes. Besides, we set the permitted range of difference counter managed by the client file system as $[-2, 2]$, for guaranteeing load balance.



(a) I/O Time

(b) Breakdown Analysis of I/O Time

(c) I/O Data Throughput

Fig. 5. Experimental results of running *SPEEDY-LETKF*.

Note that the size of epoch is not ruled to be a constant, and it can become larger to reduce the time caused by *kappa*-based consistency analysis. The larger size of epoch, however, may lack of consideration of time property in the block I/O traces and therefore lead to unnecessary migrations. For the purpose of disclosing both positive and negative effects of using a large size of epoch, we have also performed relevant experiments in our case study in Section 4.3.

4.2 Experimental Results of Real-World Coupled Applications

4.2.1 Application of SPEEDY-LETKF. SPEEDY-LETKF is a typical coupled weather forecasting system constituted of two kinds of separately developed models, i.e., the *SPEEDY* model is for simulation (Molteni 2003), and the *LETKF* model is for data assimilation (Hunt 2007), both of them have the same number of processes. During weather forecasting, the computation input data of each model depends on the computation output of other model, and the *SPEEDY-LETKF* system is supposed to run multiple cycles of simulation and data assimilation for yielding uninterrupted prediction results. The two components run in a cyclic way: After the simulation finishes, the data assimilation starts taking the results from the simulation as its input data, and after the data assimilation finishes, the simulation of the next cycle follows, depending on the results from the data assimilation.

In the experiments, the *SPEEDY* model process executes 1-year integration, and then *LETKF* is responsible for data assimilation. The I/O size is 17.3MB per process, more than 6.7GB for the entire 100 ensemble members, and each ensemble member has four processes.[1] Because *SPEEDY-LETKF* does not change its data dependency during its lifetime, we report the average results of three rounds of execution.

For comparison, we recorded the time required for I/O operations between the *SPEEDY* and *LETKF* processes by using *Non-migration* and the scheme of *Dynamic Migration* based on block I/O dependency. Scaling the number of ensemble instances from 10 to 100, Figure 5(a) demonstrates the time required for carrying out I/O operations between the two kinds of processes. And Figure 5(b) reports the breakdown analysis of the I/O time. As seen, the proposed mechanism can reduce the I/O time, because it can shorten the time required for reading and write the data, though it consumes time for conducting process migration.

But when the number of ensemble member is `60` (i.e., 240 processes for each kind of model), the *Dynamic Migration* scheme does not outperform *Non-migration.* On the one hand, all 480 processes are placed onto 12 compute nodes on the basis of the round-robin fashion, and the *SPEEDY* process is allocated with its corresponding *LETKF* process on the same node. Therefore, these processes can directly use the local file cache for exchanging the data without any migrations. Although the

[1] An ensemble member represents a model state, and then a set of processes is assigned to read an ensemble member in the *SPEEDY-LETKF* program.

Table 2. Number of Migrations and Time Overhead while Running *SPEEDY-LETKF*

# of Ensemble	# of Migration	Kappa Analysis Time (ms)	Migration Time (ms)
10	40	388	1,834
20	80	412	2,243
40	160	447	2,619
60	**0**	**476**	**0**
80	320	502	3,192
100	400	522	3,402

scheme of *Dynamic Migration* does not trigger any process migrations in this case, it consumes less than 500ms to perform *kappa*-based similarity comparisons.

After verifying the proposed mechanism can indeed reduce the time needed for exchanging the data between *SPEEDY* and *LETKF* in our tests, we also measured the I/O data throughput while performing experiments with various settings, and Figure 5(c) shows the relevant results. The scheme of *Dynamic Migration* is able to increase the read I/O throughput by up to 81.3%, when the size of ensemble members is 100 in the experiments.

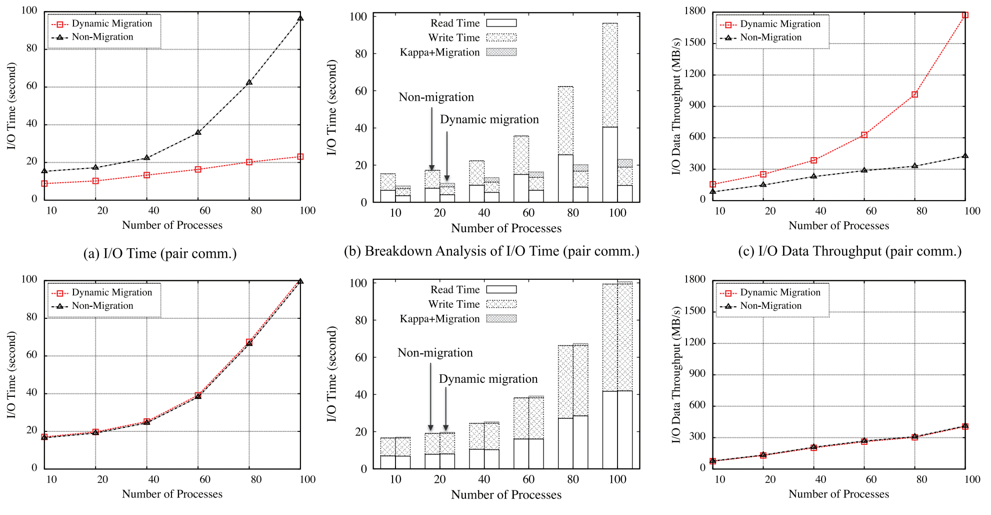
The statistics relating to the number of migrations, and the time needed for conducting migrations are collected in Table 2. In the table, the number of ensemble members indicates the quantity of one kind of processes (i.e., $\# \; of \;\; Ensemble \times 4$), and the number of migrations represents the quantity of all migrated processes. The metric of *Migration Time* is about the checkpointing overhead. More exactly, *Migration Time* consists of the time of saving the state of the checkpointed process, the time for transferring the checkpointed state from the source node to the destination node, and the time of resuming the stopped process.

Clearly, less than half of total processes have been migrated in all cases. We emphasize that migrating the processes takes some time, but it can save more time needed to carry out I/O operations for exchanging data between paired processes. Moreover, the time used for *kappa*-based dependency analysis is presented in Table 2, as well. The analysis time does not increase greatly, while the quantity of *kappa*-based similarity comparisons is getting larger. This means the *kappa*-based analysis is able to scale preferably in the context of a large amount of processes.

4.2.2 Application of GAMIL2-CLM3. We also derived another application benchmark from a real coupled model version of *GAMIL2-CLM3* (Zhang 2015). To be specific, *GAMIL2* is an atmosphere model, and *CLM3* is a land surface model; the processes of *CLM3* need to read the result data generated by the processes of *GAMIL2*. Generally, both models have the same number of processes and support both point-to-point and all-to-all communication patterns for exchanging data.

In the experiments, we set *GAMIL2-CLM3* having 128 variables, and each variable is supposed to be output 4 times. Then each process operates the same horizontal grid of 7680 (128×60) grid points, so that each process requires to deal with 40MB data. Two models use the point-to-point data communication pattern for exchanging data, when both models have the same parallel decompositions. And the all-to-all communication pattern is leveraged in the case of two models share the data with different parallel decompositions. Thus, we separately run the benchmark utilizing two data communication patterns, and the average results of three rounds of execution are recorded.

The I/O time required for exchanging data between *GAMIL2* and *CLM3* is measured, and Figure 6 illustrates the results. Specifically, Figure 6(a), (b), and (c) report I/O time, the breakdown analysis of the I/O time, and I/O data rate respectively, when the benchmark adopts the



(a) I/O Time (pair comm.)

(b) Breakdown Analysis of I/O Time (pair comm.)

(c) I/O Data Throughput (pair comm.)

(d) I/O Time (all-to-all comm.)

(e) Breakdown Analysis of I/O Time (all-to-all comm.)

(f) I/O Data Throughput (all-to-all comm.)

Fig. 6. Experimental results of running *GAMIL2-CLM3*.

point-to-point data communication pattern. Obviously, *Dynamic Migration* can yield over 4 times speedup on I/O operations, when there are 100 processes in each model. This is because it can cut down the time required for reading and writing the data, though it needs some time for conducting *kappa*-based dependency analysis, and completing process migrations. In brief, the less I/O time implies that more time can be devoted to perform atmosphere data processing and land surface data analysis, and that the total execution time can be consequently decreased.

Figures 6(d) to (f) present the results on relevant metrics, when the application adopts the all-to-all data communication pattern. As seen, the newly proposed mechanism does a little worse than *Non-migration* by 2.7% to 4.3%. In these cases, the proposed scheme does not trigger any process migrations, when both kinds of processes in *GAMIL2-CLM3* employ the all-to-all pattern to exchange data. Oppositely, the proposed scheme results in slight performance degradation, compared with *Non-migration*, as previously illustrated in Figures 6(d) to (f). However, it is worth to mention that the all-to-all communications are the worst cases for our proposal, since all the processes communicate with all the processes. But, there is no optimal solution for properly allocating such processes on compute nodes for decreasing the time needed by I/O data exchanges among them.

Furthermore, the statistics data about the number of migrations, and the time needed for completing migrations when running *GAMIL2-CLM3* with the point-to-point data communication pattern, are presented in Table 3. Similarly to the case of *SPEEDY-LETKF*, the proposed mechanism can effectively find corresponding *CLM3* processes, for the *GAMIL2* processes if the application employs the pair communication pattern to exchange data.

Through running two real-world coupled applications, we have verified that the proposed scheme of dynamic process migration can greatly reduce I/O time by dynamically allocating paired processes on same compute nodes, in the case of coupled applications take advantage of the point-to-point communication pattern to exchange data. Since existing data transfer tools, such as *OASIS* (Valcke 2006), *MCT* (Larson 2005), and our previous work (Liao 2017), primarily support the pattern of pair communication for data transfer between component models. We believe the pair data

Table 3. Number of Migrations and Time Overhead while Running *GAIML2-CLM3* (`point-to-point comm.`)

# of Processes	# of Migration	Kappa Analysis Time (ms)	Migration Time (ms)
16	16	386	1,546
32	32	422	2,011
64	64	462	2,326
128	128	489	2,812
256	256	526	3,231
512	512	556	4,062

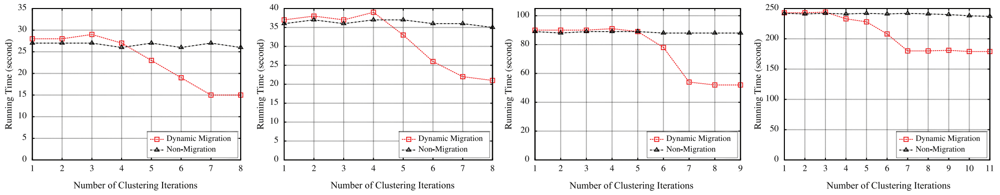
communication pattern is commonly used in many real coupled applications. As a result, the newly proposed mechanism is beneficial to such targeted application contexts.

4.2.3 Application of k-means Clustering with MapReduce. The algorithm of k-means clustering is able to classify semi-structured or unstructured data sets into k clusters (Cui 2014). In its MapReduce version, the map task reads the k centroids into memory from a sequence file or randomly selects them from the input dataset in the first round of clustering. Then the map task iterates over each centroid for each input key/value pair in the dataset. As a result, it writes the centroid with its vector to the file system after measuring the distances and calculating the nearest centroid. The reduce task iterates over each value vector to figure out the average vector and then saves the new centroids into the sequence file. At last, the reduce task checks whether the k centroids have been updated or not from the previous round of execution to decide whether to carry out another iteration of clustering.

As seen, the output of the map task is written to the disk, and the reduce task reads the output from the relevant map task(s) in each clustering iteration, so that it can be categorized as a coupled application. In fact, the data dependency between map tasks and reduce tasks of this application is unknown before execution. Moreover, the data access similarity between two kinds of tasks may change in different iterations. Thus, we selected this benchmark to evaluate our proposal, when the data dependency relationship among processes dynamically changes.

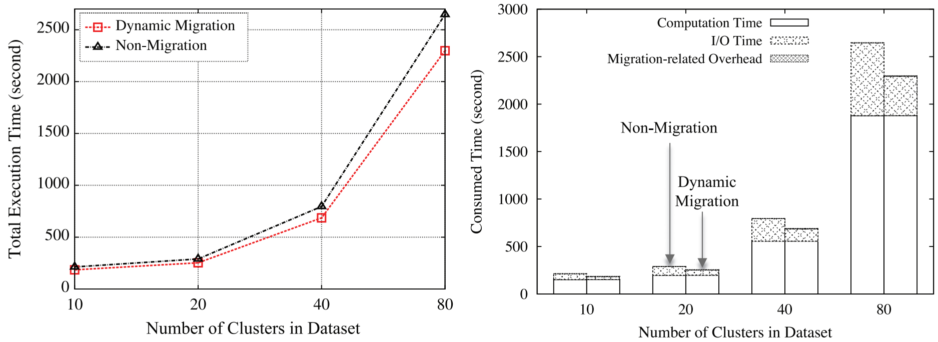
To run this MapReduce application on the top of the PARTE file system, we have built the experiment platform by employing *C-MapReduce* (Wong 2015) to emulate the MapReduce framework. In the experiments, one dedicated compute node was selected as the master node, and other `10` nodes of Cluster II worked as slaves. Each slave node runs 4 map or reduce tasks at a time. We then used *Mixsim* (Melnykov 2012) to generate 4 datasets of `10M` objects (each object has 4 attributes) with a Gaussian distribution, and the variable number of clusters ranging from `10` up to `80`. The total size of the input file is `6.2GB`. Besides, we configured the application adopting the first k objects as the default initial centroids to start the task of object clustering. We emphasize that there is no optimal policy for placing the processes before running this benchmark, since its data dependency relationship cannot be disclosed by using a static analysis method.

Figure 7 shows the number of iterations, and the consumed time in each iteration on the selected four datasets. Obviously, *Dynamic Migration* does not perform better than *Non-Migration* in the first four or five iterations. This is because the coupled map and reduce tasks have not been primely allocated on the same compute node at these points but carrying out data dependency analysis and process migrations results in time overhead. The relevant map and reduce processes are likely to be placed on the same compute node after a number of dynamic process migrations. Therefore, data exchanges between two kinds of processes can be partially served by accessing the local file cache, and the execution time is remarkably reduced in the late rounds of iterations.



(a) Number of clusters:10 (b) Cluster number: 20 (c) Number of clusters:40 (d) Cluster number: 80

Fig. 7. Number of iterations and time required for clustering objects.



(a) Overall Execution Time (b) Breakdown Analysis of Execution Time

Fig. 8. Number of iterations and time required for clustering objects.

Table 4. Number of Migrations and Time Overhead while Running *k-means* (`40 Map/Reduce processes`)

# of Clusters	# of Migration	Kappa Analysis Time (ms)	Migration Time (ms)
10	40	435	3,136
20	42	428	3,463
40	42	432	3,489
80	44	438	3,922

As seen in Figure 8(a), *Dynamic Migration* can cut down the overall execution time required for clustering the datasets by `12.7%–13.8%`. For specifically disclosing the I/O reduction caused by the proposed mechanism, we have also recorded the breakdown analysis on the running time, which includes the time required for computations, the time required for I/O operations, and the time required for conducting process migrations in the case of adopting the newly proposed mechanism. As illustrated in Figure 8(b), the time needed for computing the Euclidean distance between two objects in k-means clustering of using both mechanisms is same, but other parts of the execution time are rather different. Compared with *Non-migration*, the newly proposed *Dynamic Migration* can reduce the I/O time by `39.1%–45.8%`, even though it causes extra overhead for conducting dependency analysis and process migrations.

Table 4 presents the statistics data regarding dynamic migration in the scheme of *Dynamic Migration*, when running the k-means clustering algorithm on the selected datasets. Similarly to the results of running *SPEEDY-LETKF* and *GAMIL-CLM3*, there is no much overhead caused by data dependency analysis and process migrations. The most important clue shown in this table

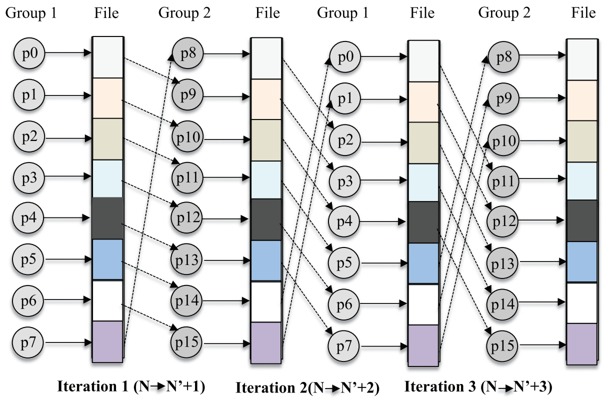


Fig. 9. A simplified example of data input and output flow among 16 MPI processes having the shifting permutation data exchange pattern in the developed benchmark. Note that the data dependency between two groups of processes is set to change in the different rounds of iteration.

is about there are more than `40` migrations in all cases except for processing the dataset of `10` clusters. By referring the *kappa*-based analysis, we have observed this application may change its I/O relationship between the map process and the reduce process from one iteration to another. As a result, migrating a specific process more than one time becomes a case.

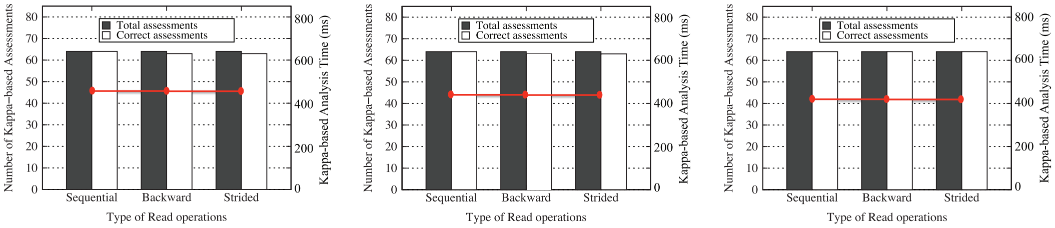
In brief, we argue that our proposal is serviceable for such iterative applications running on the top of MapReduce, even though their data exchange patterns between map tasks and reduce tasks cannot be unveiled without execution, or they may change their data dependency in the different iterations.

4.3 An In-House Benchmark Case Study

For the purpose of exploring the accuracy of the *kappa*-based dependency analysis, and the overall effectiveness of the proposed mechanism of dynamic process migration, a case study has been conducted. First, we have developed an in-house benchmark to emulate the applications having changed I/O data dependency. Specifically, all the processes are divided into two groups, the first group sorts the elements in order, but the second group sorts the elements in reverse order, and both of groups employ the bubble sort algorithm. Moreover, the output results of the processes are supposed to be read by their corresponding processes, according to the shifting permutation, for another round of computation.

Figure 9 demonstrates the data I/O exchanges among the processes of the benchmark in the different rounds of execution. As seen, the processes write their datasets into a shared file, and then the dataset in the file is read by the corresponding process in another group, so that each process has another data-dependency process. Similarly to the *IOZone* benchmark (IOZONE 2015), we programmed the processes to read the file data by following different reads, including `sequential`, `backward`, and `strided` operations. To put it from another angle, a process sequentially writes a piece of dataset to the file, and another paired process will read the same piece of dataset by using diverse read operations in varied cases.

In the experiments, there are three rounds of execution in the program. In each loop of iteration, the program produces 8GB of integer data in the file, and it has multiple processes to carry out integer computations to sort the elements in the corresponding datasets, i.e., different parts of the file. In other words, the process reads a segment of data in the file (*size of segment* = 2 × *size of*



(a) Small Size of Epoch (4 blocks)

(b) Default Size of Epoch (16 blocks)

(c) Large Size of Epoch (4096 blocks)

Fig. 10. Accuracy and Time overhead of *kappa*-based dependency analysis with varied sizes of epoch (with 128 processes).

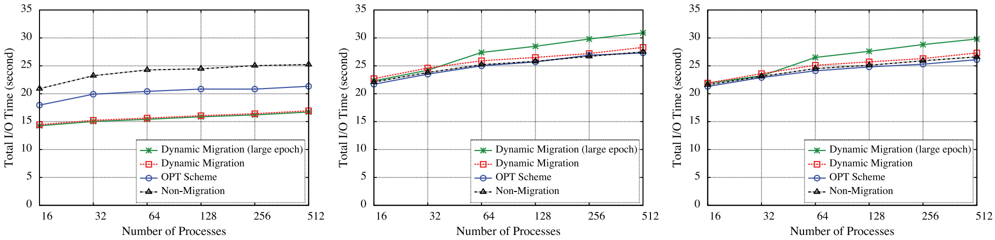
file/number of processes) and then writes the data to the same offset in the file after sorting the numbers. There are no inter-communications during the computing phases in the same group, and each process only sorts its relevant dataset in the file.[2]

4.3.1 Accuracy of kappa-Based Dependency Analysis. This section intends to explore the accuracy of *kappa*-based dependency analysis when having three different sizes of epoch. To be specific, we employ the default size of epoch (i.e., 16 blocks), and a small size of epoch (i.e., 4 blocks), when 1,024 access events in the sequence are used for similarity analysis. Besides, we have introduced a large size of epoch in this case study to show it can yield a better accuracy of *kappa*-based analysis and less time for completing the analysis. In this situation, each epoch has all block accesses in a round of execution, i.e., there are *(size of segment/64KB)* blocks, and only 2 epochs are employed to calculate the *kappa* statistic. In this series of experiments, the target process in one group is compared with all process in another group, and then the assessment results are recorded.

We have run the benchmark with varied number of processes, and obtained very similar results. Thus, we only report the results when running the benchmark with 128 processes to locate a paired process for the target one. Figure 10 shows the statistic data about the number of *kappa*-based assessments and the number of correct assessments. To be specific, there should be total 64 comparisons, including 63 unmatched pairs and 1 matched pair for the target process. As shown, the accuracy of process pair identification (including matched and unmatched assessments) varies from 98.4% to 100%. Specially, the *kappa*-based scheme can correctly identify the process pair, when the process read the file data sequentially, regardless of the size of epoch. But, it might fail to explore the matched peer for a target process, while the process read data by following `backward` and `strided` modes, as illustrated in Figure 10(a) and (b).

Figure 10(c) demonstrates that the large size of epoch can assess data dependency with 100% percent accuracy. It compares all block accesses in a round of execution, it can thus explore the paired processes that operate on the same set of data blocks, by ignoring the access order. Strictly speaking, we refer such process pairs as desired process pairs, as the two processes operate on the same piece of data, although they access the data with different time orders. Nevertheless, it cannot ensure better system performance, by allocating such paired processes onto the same nodes. This is because the local file cache can only buffer a limited number of data blocks, rather than all block data in a round of execution, refer to Section 4.3.2 for more information.

[2] The data in the file is not well sorted after running the benchmark, since there is no intercommunication to merge the sorted datasets.



(a) Sequential read (b) Backward read (c) Strided read

Fig. 11. I/O time of running the developed benchmark with different read operations.

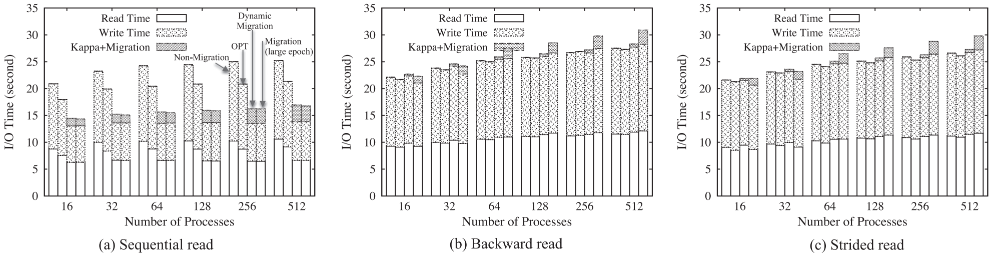
Besides, the time required for completing *kappa*-based similarity analysis (indicated with red lines), when using varied sizes of epoch, is reported in Figure 10, as well. The scheme adopting the large size of epoch consumes the least time to perform *kappa*-based comparisons, compared with others, as expected. More importantly, the proposed mechanism having varied sizes of epoch does not bring about noticeable differences in the time required for *kappa*-based dependency comparisons. This is because constructing the contingency table occupies a major part of time in a *kappa*-based comparison, and it does not change while using different sizes of epoch.

In fact, the time required to conduct *kappa*-based analysis depends on not only the size of epoch, but also the number of involved processes. In other words, more processes generally bring about more time to finish a *kappa*-based consistency check, and Section 4.3.2 will verify this fact with more details.

4.3.2 Performance Results. This section shows the performance results of running the benchmark to evaluate the proposed approach. Figure 11 shows the experimental results of running the developed benchmark regarding the required I/O time, when the processes read the data by following different modes.

As demonstrated in Figure 11(a), *OPT Scheme* outperforms the *Non-Migration* scheme, which initially place the processes in a round-robin fashion. Because *OPT Scheme* binds the processes according to their data dependency in the first round, the processes can exchange their data through accessing the local file cache, though it cannot accelerate data exchanges in the remaining two rounds of execution. Obviously, the proposed scheme of *Dynamic Migration* performs attractively if the processes read the data sequentially, even if the pattern of data dependency changes in different rounds of execution. As discussed, *Dynamic Migration* can dynamically migrate the processes for allocating paired processes onto the same compute nodes, on the basis of recently occurred block accesses. As a result, in contrast to *Non-Migration* and *OPT Scheme*, the newly proposed scheme can reduce the I/O time by 30.8–49.3%.

In the cases of reading data backwardly or stridedly, as illustrated in Figure 11(b) and (c), *Dynamic Migration* demands more time for completing I/O operations. More interestingly, the scheme of *Dynamic Migration* having a large epoch requires more time by up to 31.6%, but the same scheme having the default size of epoch does not bring about a noticeable increase in I/O time. This is because the paired processes can be found to trigger process migrations regardless of the read modes if we use a large epoch in these cases. However, the local file cache is not able to hold all blocks flushed by a write process, so that the paired read process may not obtain the requested data by accessing the cache, though both processes are dispatched on the same node. The situation changes if the scheme equipping with the default size of epoch cannot disclose the paired processes, so there is no time overhead for carrying out unprofitable process migrations, though a little bit more time is needed for *kappa*-based analysis.

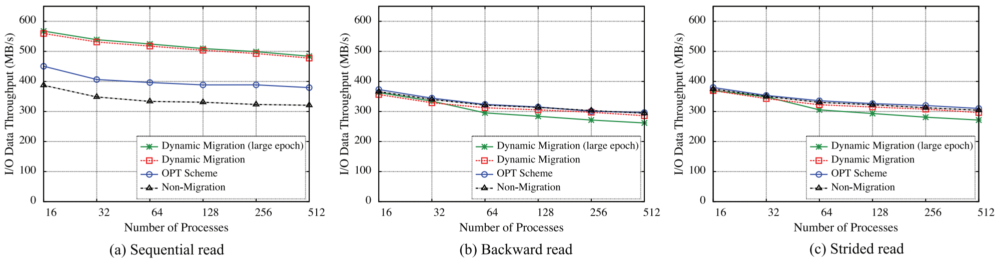


(a) Sequential read

(b) Backward read

(c) Strided read

Fig. 12. I/O time distribution of running the developed benchmark with different read operations.



(a) Sequential read

(b) Backward read

(c) Strided read

Fig. 13. I/O data throughput of running the developed benchmark with different read operations.

Similarly, the breakdown analysis of I/O time is presented in Figure 12, and the results about I/O data throughput are shown in Figure 13. Specially, from the illustration of Figure 12(a), we can understand that the scheme of dynamic process migration on the basis of block I/O dependency can accelerate the execution of the application. This is due to the read and write time can be greatly decreased, even though it results in the time overhead for carrying out process migrations and the *kappa*-based dependency analysis. Figure 13(a) shows the attractive I/O data throughput improvements can be yielded when employing the proposed scheme, as it can cut down the I/O time by using the local file cache to exchange data among processes.

Another notable clue revealed in Figures 11 to 13 is that the newly proposed mechanism cannot benefit the contexts of processes read the data (output by its paired process) by complying with the *backward* and *strided* modes. In fact, a very limited number of real world applications employ the *backward* mode to access the file data. The *strided* mode is widely used for reading data, it can be treated as a variant of the all-to-all data communication pattern in coupled applications. The newly proposed mechanism (with default configurations) does not identify matched processes, for directing process migrations in such situations. Consequently, it does only introduce acceptable time overhead to conduct *kappa*-based dependency analysis.

Furthermore, Table 5 collects the number of migrations and the time overhead on performing *kappa*-based analysis and process migrations, when processes read data sequentially, with default configurations. Since all processes (the number increases by a factor of 2) are initially placed onto 12 compute node in the round-robin fashion, there is no process located with its corresponding process on the same compute node. As shown in the table, half of the processes have been migrated to the proper compute nodes in each round of execution, for the purpose of locating with the corresponding processes in the matched process-pairs. Thus, they can exchange their data by making use of the local file cache for boosting the I/O performance.

Table 5. Number of Migrations and Time Overhead while Running the Benchmark (`sequential read`)

# of Processes	# of Migration	Kappa Analysis Time (ms)	Migration Time (ms)
16	8	312	1,368
32	16	388	1,633
64	32	416	2,102
128	64	442	2,469
256	128	481	2,882
512	256	512	3,017

4.4 Summary

With respect to comparing the *Dynamic Migration* scheme and the *Non-migration* scheme in the evaluation experiments, we emphasize the following two key observations. First, moving the processes to another compute node to take advantage of the local file cache for exchanging the data among processes can indeed reduce the time needed for I/O operations. Second, the proposed mechanism may benefit the coupled systems, in which the component requires to exchange data with other components. This is true for the applications having multiple iterations, and there is I/O dependency among the processes, which leverage the point-point communication pattern to exchange data. In brief, we conclude that this newly proposed mechanism is able to significantly reduce the time required by exchanging data among the processes when they have certain data I/O dependency.

Through executing the developed application in our case study, we can understand that the *kappa*-based dependency analysis is able to yield accurate dependency assessments. Moreover, the time overhead for completing dependency assessments does not increase greatly, even if the number of involved processes grows by a factor of 2.

5 CONCLUDING REMARKS

This article has proposed, implemented and evaluated a dynamic process migration scheme on the basis of block I/O dependency, which is able to boost I/O data rate, and reduce the time required for executing the applications, such as the coupled systems. This scheme has the feature of adaptivity, to support dynamic process migrations, according to the data dependency. Specifically, it can identify the changes in data dependency among processes, and then adaptively cluster relevant processes onto the same or nearby compute node, for exchanging data via accessing the shared file cache.

To put this mechanism to work, First, we have constructed per-process block I/O access sequences on the storage server by resorting to the piggybacking technique. Then, we have proposed an algorithm to estimate the block I/O dependency of two processes running on the different compute nodes, by using *Cohen's kappa statistic*. Next, the process migration will be triggered when a process has a heavy block I/O dependency relationship with another process executing on another node, so that both of them can use the local file cache to exchange data after migration. Finally, not only the execution time of application can be greatly decreased, but also the I/O data throughput can be remarkable improved.

The implementation based on the *PARTE* file system is to illustrate the feasibility and applicability of the main ideas presented in this article. In fact, these ideas can be applied to other conventional parallel file systems such as *Lustre*, the Google file system, *PVFS* and *HDFS*, or their extensions, as well. We emphasize that the newly proposed mechanism does not work well for the cases, in which the processes do not have any I/O dependency relationship, or the processes may read the output data backwardly or stridedly.

The current implementation of the proposed scheme may cause “migration thrashing”, when the target application has different block access patterns during the execution. Therefore, we are planning to analyze more block access events to determine whether the process migration is supposed to be triggered or not in the near future. Moreover, the current implementation aims to migrate processes on the basis of I/O dependency, thus, taking some other critical metrics, such as the energy consumption, computing workloads, and intra-communications, into account for triggering process migration, is another direction of future work.

ACKNOWLEDGMENTS

We want to thank Ming Tang for letting us using the high performance clusters in Southwest University of China, with full control. Especially, we would also like to thank Wei-keng Liao@Northwestern University, for suggesting using a MapReduce application in evaluation.

REFERENCES

H. Abdi. 2007. The Kendall rank correlation coefficient. In *Encyclopedia of Measurement and Statistics*. Sage, Thousand Oaks, CA. 508–510.

R. Ahmad, A. Gani, and S. Hamid. 2015. Virtual machine migration in cloud data centers: a review, taxonomy, and open research issues. *J Supercomput.* 71, 7 (2015), 2473–2515.

Y. Amir, B. Awerbuch, and A. Barak et al. 2000. An opportunity cost approach for job assignment in a scalable computing cluster. IEEE Trans. *Parallel Distrib.* Syst. 11, 7 (2000), 760–768.

V. Anthony and J. Garrett. 2005. Understanding interobserver agreement: the kappa statistic. *Fam. Med.* 37, 5 (2005), 360–363.

K. Barker, A. Chernikov, N. Chrisochoides et al. 2004. A load balancing framework for adaptive and asynchronous applications. *IEEE Trans. Parallel Distrib. Syst.* 15, 2 (2004), 183–192.

J. Benesty, J. Chen, and Y. Huang et al. 2009. Pearson correlation coefficient. In *Noise Reduction in Speech Processing*. Springer, Berlin, 1–4.

BTIO Benchmark. 2011. Retrieved from http://www.nas.nasa.gov/.

A. Choudhary. 2015. Active Storage with Analytics Capabilities and I/O Runtime System for Petascale Systems. Report No. DOE-NWU-25848. Northwestern University, Evanston, IL.

I. Cores, G. Rodriguez, and P. Gonzalez et al. 2014. Failure avoidance in MPI applications using an application-level approach. *Comput. J.* 57, 1 (2014), 100–114.

X. Cui, P. Zhu, and X. Yang et al. 2014. Optimized big data K-means clustering using MapReduce. *J. Supercomput.* 70, 3 (2014), 1249–1259.

M. DeGroot and M. Schervish. 2011. Probability and Statistics, 4th ed. Pearson Education Limited, London.

X. Ding, S. Jiang, F. Chen, and X. Zhang et al. 2007. DiskSeen: Exploiting Disk Layout and Access History to Enhance I/O Prefetch. In *Proceedings of the USENIX Annual Technical Conference (ATC'07)*. USENIX Association.

J. Dongarra and P. Beckman et al. 2011. The International Exascale Software Roadmap. *Int. J. High Perf. Comput.* Appl. 25, 1 (2011), 3–60.

J. Duell. 2000. The design and implementation of berkeley labs linux checkpoint/restart. Technique Report, Lawrence Berkeley National Laboratory.

FUSE: Filesystem in Userspace. Retrieved from http://fuse.sourceforge.net/.

B. Hunt, E. Kostelich, and I. Szunyogh. 2007. Efficient data assimilation for spatiotemporal chaos: A local ensemble transform Kalman filter. *Physica D* 230, 1 (2007), 112–126.

K. Ibrahim, S. Hofmeyr, C. Iancu, and E. Roman. 2011. Optimized pre-copy live migration for memory intensive applications. In *Proceedings of the International Conference on High Performance Computing, Network, and Storage Analysis (SC'2011).*

Iozone Filesystem Benchmark. Retrieved from http://www.iozone.org/.

S. Jiang, X. Ding, Y. Xu, and K. Davis. 2013. A prefetching scheme exploiting both data layout and access history on disk. *ACM Trans. Stor.* 9, 3 (2013), Article 10.

E. Jeannot, G. Mercier, and F. Tessier. 2014. Process placement in multicore clusters: Algorithmic issues and practical techniques. *IEEE Trans. Parallel Distrib. Syst.* 25, 4 (2014), 993–1002.

F. Joseph, J. Cohen, and B. Everitt. 1969. Large sample standard errors of kappa and weighted kappa. *Psychol. Bull.* 72, 5 (1969), 323–327.

KDD Cup 1999 Data. Retrieved from http://kdd.ics.uci.edu/databases/kddcup99/kddcup99.html.

J. Larson, R. Jacob, and E. Ong. 2005. The model coupling toolkit: A new Fortran90 toolkit for building multiphysics parallel coupled models. *Int. J. High Perform.* C 19, 3 (2005), 277–292.

Z. Li, Z. Chen, S. Srinivasan, and Y. Zhou. 2004. C-Miner: Mining Block Correlations in Storage Systems. In *Proceedings of the 3rd USENIX Conference on File and Storage Technologies (ATC'04)*. USENIX.

J. Liao, Y. Ishikawa. 2012a. Partial replication of metadata to achieve high metadata availability in parallel file systems. In *Proceedings of 41st International Conference on Parallel Processing (ICPP'12)*. 168–177.

J. Liao. 2012b. A new concurrent checkpoint mechanism for embeded multi-core systems. *Comput. Inform.* 31, 3 (2012), 693–709.

J. Liao, F. Trahay, B. Gerofi, Y. Ishikawa. 2016. Prefetching on storage servers through mining access patterns on blocks. *IEEE Trans. Parallel Distrib. Syst.* 27, 9 (Sep. 2016), 2698–2710.

J. Liao, B. Gerofi, G. Lien , S. Nishizawa, T. Miyoshi, H. Tomita, W. Liao, A. Choudhary, and Y. Ishikawa. 2017. A flexible I/O arbitration framework for netCDF based big data processing workflows on high-end supercomputers. *Concurr. Comput. Pract. Exper.* 29, 15 (Aug. 2017), 12 pages.

A. Mashtizadeh, M. Cai, and G. Tarasuk-Levin et al. 2014. XvMotion: Unified virtual machine migration over long distance. In *Proceedings of the 2014 USENIX Annual Technical Conference* (USENIX ATC'14).

V. Medina and J. Garcia. 2014. A survey of migration mechanisms of virtual machines. *ACM Comput. Surv.* 46, 3 (2014), Article 30.

V. Melnykov, W. Chen, and R. Maitra. 2012. Mixsim: An R package for simulating data to study performance of clustering algorithms. *J. Stat. Softw.* 51, 12 (2012), 1–25.

F. Milojicic, F. Douglis, Y. Paindaveine, and S. Zhou et al. 2000. Process migration. *ACM Comput. Surv.* 32, 3 (2000), 241–299.

T. Miyoshi, G. Lien, S. Satoh, and Y. Ishikawa et al. 2016. "Big data assimilation" toward post-peta-scale severe weather prediction: An overview and progress. *Proc. IEEE* 104, 11 (Nov. 2016), 2155–2179.

F. Molteni. 2003. Atmospheric simulations using a GCM with simplified physical parametrizations. I: Model climatology and variability in multi-decadal experiments. In *Climate Dynamics*, Vol. 20, 175–191.

L. Myers and J. Sirois. 2006. Spearman correlation coefficients, differences between. Wiley StatsRef: Statistics Reference Online. https://onlinelibrary.wiley.com/doi/pdf/10.1002/0471667196.ess5050.pub2.

X. Ouyang, S. Marcarelli, R. Rajachandrasekar, and D. Panda. 2010. RDMA-based job migration framework for MPI over infiniband. In *Proceedings of the IEEE International Conference on Cluster Computing (Cluster'10)*. 116–125.

X. Ouyang, R. Rajachandrasekar, X. Besseron, and D. Panda. 2011a. High performance pipelined process migration with RDMA. In *Proceedings of the 11th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID'11)*. IEEE Computer Society, 314–323.

X. Ouyang, R. Rajachandrasekar, X. Besseron, and D. Panda et al. 2011b. CRFS: A lightweight user-level filesystem for generic checkpoint/restart. In *Proceedings of 2011 International Conference on Parallel Processing (ICPP'11)*. 375–384.

S. Petri and H. Langendorfer. 1995. Load balancing and fault tolerance in workstation clusters migrating groups of communicating processes. *ACM SIGOPS Operat. Syst. Rev.* 29, 4 (1995), 25–36.

E. Riedel, G. Gibson, and C. Faloutsos. 1998. Active storage for large-scale data mining and multimedia applications. In *Proceedings of 24th Conference on Very Large Databases (VLDB'98)*. 62–73.

S. Valcke, R. Budich, and M. Carter et al. 2006. The PRISM software framework and the OASIS coupler. In *Proceedings of the 18th Annual BMRC Modelling Workshop*.

C. Vecchiola, S. Pandey, and R. Buyya. 2009. High-performance cloud computing: A view of scientific applications. In *Proceedings of the 10th International Symposium on Pervasive Systems, Algorithms, and Networks (ISPAN'09)*. 4–16.

R. Vyas, H. Maheta, and V. Dabhi et al. 2014. Load balancing using process migration for linux based distributed system. In *Proceedings of International Conference on Issues and Challenges in Intelligent Computing Techniques (ICICT'14)*. 248–252.

C. Wang, F. Mueller, C. Engelmann, and S. Scott. 2008. Proactive process-level live migration in HPC environments. In *Proceedings of the International Conference on High Performance Computing, Networks, and Storage Analysis (SC'08)*. 1–12.

J. Wang and X. Liang. 2005. *Qualitative Data Analysis*. East China Normal University Press, 92–93. [in Chinese]

D. Williams, H. Jamjoom, and H. Weatherspoon. 2012. The Xen-Blanket: Virtualize once, run everywhere. In *Proceedings of the 7th ACM European Conference on Computer Systems (EuroSys'12)*. ACM, New York, NY, 113–126.

J. Wong. 2018. C-MapReduce. Retrieved January 2018 from https://github.com/jeffrey-garcia/C-MapReduce.

Y. Xie, D. Feng, Y. Li, and D. Long. 2016. Oasis: An active storage framework for object storage platform. *Future Generation Computer Systems* 56 (2016), 746–758.

F. Xu, F. Liu, L. Liu, and H. Jin et al. 2014. iaware: Making live migration of virtual machines interference-aware in the cloud. *IEEE Trans. Comput.* 63, 12 (2014), 3012–3025.

F. Xu, F. Liu, L. Liu, and H. Jin et al. 2014b. Managing performance overhead of virtual machines in cloud computing: a survey, state of the art, and future directions. *Proc. IEEE* 102, 1 (2014), 11–31.

X. Zhang, K. Davis, and S. Jiang. 2011. Qos support for end users of i/o-intensive applications using shared storage systems. In *Proceedings of the 2011 ACM/IEEE Conference on Supercomputing (SC'11)*. ACM, New York, NY.

F. Zhang, C. Docan, M. Parashar et al. 2012. Enabling in-situ execution of coupled scientific workflow on multi-core platform. In *Proceedings of IEEE 26th International Parallel & Distributed Processing Symposium (IPDPS'12)*. 1352–1363.

F. Zheng, H. Zou, and G. Eisenhauer et al. 2013. Flexio: I/O middleware for location-flexible scientific data analytics. In *Proceedings of IEEE 27th International Symposium on Parallel and Distributed Processing (IPDPS'13)*. 320–331.

Received July 2016; revised February 2018; accepted April 2018