Journal of Computations & Modelling, vol.6, no.3, 2016, 25-42
ISSN: 1792-7625 (print), 1792-8850 (online)
Scienpress Ltd, 2016

Mobile Thread Migration for Dynamic Load Balancing in Grid

Masaya Miyashita[1], Andrii Zhygmanovskyi[2], Noriko Matsumoto[3] and Norihiko Yoshida[4]

Abstract

A Grid is mostly heterogeneous system where the computation capability of each node varies. Therefore, load distribution and load balancing are among the most important issues in Grid. Some techniques have been proposed for dynamic and adaptive load distribution, such as relocating jobs from a high-loaded node to a low-loaded node while retaining job execution state based on virtual machine and thread migration. In this paper, we propose a method for dynamic and adaptive load distribution between computation nodes using mobile thread migration which yields lightweight job relocation without unnecessary overhead. Using an example problem called parallel PrefixSpan, whose computation cost is absolutely unpredictable and which is used in analysis of amino-acid sequences, we demonstrate effectiveness of our technique through experiments.

Keywords: grid computing; load balancing; mobile threads; dynamic migration

[1] Saitama University. E-mail: masaya@ss.ics.saitama-u.ac.jp
[2] Saitama University. E-mail: andrew@ss.ics.saitama-u.ac.jp
[3] Saitama University. E-mail: noriko@ss.ics.saitama-u.ac.jp
[4] Saitama University. E-mail: yoshida@ss.ics.saitama-u.ac.jp

Article Info: *Received* : July 27, 2016. *Revised* : August 29, 2016.
Published online : October 1, 2016.

1 Introduction

Grid systems, which are composed of a number of volunteer commodity PCs, are already widely used for various scientific computations. A Grid is mostly heterogeneous system, and the computation capability (as well as network bandwidth) of each node varies. In such system, computation cost of a job assigned to each node often varies as well. Therefore, load distribution and load balancing are among the most important issues in Grid, and several related techniques have been proposed. Some of them are based on static prediction of the computation cost of all the jobs before execution, and can only be applied to problems with predictable computation cost.

Several techniques have been proposed for dynamic and adaptive load distribution by relocating jobs while keeping their execution state from high-loaded node to low-loaded node during execution. Most of them utilize virtual machine migration [1, 2, 3]. In this scheme, the cost of transferring the data presents some overhead. Therefore, more lightweight ways of migration are required.

In this paper, we propose a dynamic and adaptive load distribution technique for relocating jobs between computation nodes using mobile thread migration, which leads to lightweight job relocation without unnecessary overhead. There have already been some proposals on thread migration [4, 5, 6], however they are based on distributed shared memory (DSM) systems. Mobile thread systems do not require such underlying platform, therefore they are more lightweight than DSM-based thread migration. We demonstrate effectiveness of our approach using an example problem called parallel PrefixSpan used in analysis of amino-acid sequences, whose computation cost is absolutely unpredictable. Based on our preliminary study using two PCs [7], we conducted some extended experiments towards practical evaluation. In this regard, this paper is an extended version of our previous conference paper [7].

The organization of this paper is as follows. We explain mobile threads, and our design principle for dynamic migration in Section 2. Section 3 is an introduction of the example problem, and Section 4 describes a Grid system based on our technique. We show the results of experiments in Section 5, and Section 6 presents conclusions of this paper.

2 Migration based on mobile threads

A mobile thread, sometimes also called mobile agent, is a kind of thread, which can move between computers in a network while keeping its own program code and execution state [8]. This idea arose to help development and operation of large-scale network applications. Conventional network applications are designed based on "communication". The idea of mobile threads separates computation and physical platform, and unifies computation and communication instead. Communication is now enclosed within computation, and this encapsulation is also expected to reduce network traffic.

To implement this feature, a mobile thread must be able to transfer not only its information but also its execution context over a network. The source node suspends the mobile thread, transfer information required to continue execution to the destination node, and the destination node resumes the thread. Mobile thread systems already proposed so far include Aglets [9], MOBA [10], AgentSpace [11], and JavaGo [12], all of which are based on the Java language.

The Java language and its environment provides multi-platform remote invocation mechanism. Its serialization facility also enables us to send and receive not only simple values (integers and strings) but also complex objects even in heterogeneous network environments. However, Java only provides such mechanisms for mobile objects, but not for mobile threads. The execution context of threads cannot be transferred. The systems mentioned above extend Java language and/or its byte-code interpreter to provide context mobility and basic mechanism for dynamic class loading.

In order to implement job relocation from a high-loaded node to a low-loaded node during execution and achieve dynamic load distribution in Grid, mobile threads have several advantages over virtual machine migration.

- Migration overhead:
  The amount of data to be transferred for mobile threads is much smaller than for virtual machines.

- Variation of execution platform:
  Underlying hardware vary from node to node, and sometimes execution of a virtual machine has some restrictions. Execution of mobile threads does not have any.

To apply the mechanism of mobile threads to dynamic load distribution, the system should migrate a thread executing a job from a high-loaded node to a low-loaded node, or from a low-performance node to a high-performance node. To realize this approach, we introduce the following two parameters:

- Performance of a computation node E_{node}.
- Estimated computation cost of a job E_{cost}.

E_{node} indicates the processing capacity which the node can execute in a unit of time. E_{cost} indicates the workload of a job. We introduce time T required to complete the job on the node as $T = E_{cost}/E_{node}$. We use T as a parameter for job migration. Figure 1 shows some examples of criteria for job migration in the case of two nodes, with the maximum number of jobs being executed concurrently on a single node is set to 2 for simplicity.

We could make a node control the migration in an autonomous manner, however it is actually difficult to find an appropriate destination node for migration in a decentralized manner, therefore we put a central manager which always monitors the load of all nodes and controls every migration. Figure 2 shows an outline of the proposed procedure.

The process flow is shown in Figure 2 in general case of more nodes and larger capacity.

(1) A worker tells its value T to the manager periodically, and asks the manager for an appropriate destination node.

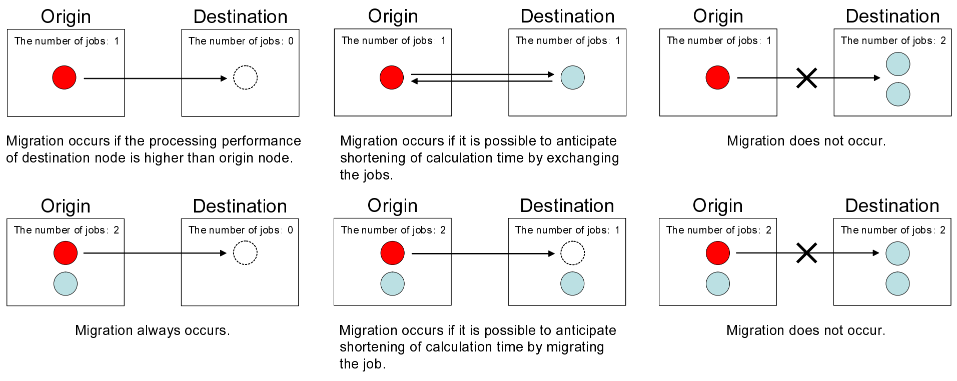


Figure 1: Criteria for job migration.

(2) The manager replies with the information about the lowest-loaded node, which has the smallest value T out of all nodes.

(3) The worker determines whether to migrate its job or not, according to the criteria whose simple version is shown in Figure 1.

3 Example problem: PrefixSpan

As an example problem in which the computation cost of each job changes dynamically and cannot be predicted before computation, we use “PrefixSpan” proposed by Pei et al. [13] [14], which is an algorithm for sequential pattern mining. Sequential pattern mining is extracting frequent substrings out of a given database of sequences, and is often applied to analysis of amino-acid sequences.

The essence of PrefixSpan is as follows (quoted from [13]):

> A sequence is a linear combination of characters such as alphabets. A sequence database is a set of tuples $\langle sid, s\rangle$, where sid is a sequence ID and s is a sequence. A tuples $\langle sid, s\rangle$ is said to contain a sequence α, if α is a subsequence of s, i.e., $\alpha \sqsubseteq s$. The support of a sequence α in a sequence database S is the number of tuples in the database containing α, i.e.,
>
> $$support_S(\alpha) = \mid \{\langle sid, s\rangle \mid (\langle sid, s\rangle \in S) \wedge (\alpha \sqsubseteq s)\} \mid .$$

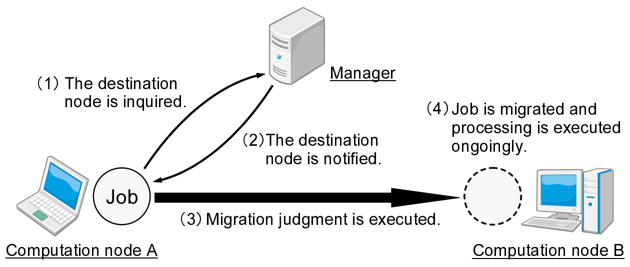

Manager
(1) The destination node is inquired.
(2)The destination node is notified.
(4)Job is migrated and processing is executed ongoingly.
Job
(3) Migration judgment is executed.
Computation node A
Computation node B

Figure 2: Outline of migration mechanism.

Given a positive integer ξ as the support threshold, a sequence α is called a frequent sequential pattern in the sequence database S if the sequence is contained by at least ξ tuples in the database, i.e., $support_S(\alpha) \geq \xi$. Given a sequence database and a minimum support threshold, the problem of sequential pattern mining is to find the complete set of sequential patterns in the database.

Given a sequence $s = \langle a_1, a_2, \ldots, a_m \rangle$, the sequence $\langle a_1, a_2, \ldots, a_j \rangle$ $(1 \leq j \leq m)$ where $a_1 \neq a$, $a_2 \neq a$, $\ldots$, $a_{j-1} \neq a$, $a_j = a$ is called a prefix of s for $a(prefix(s, a))$, and the sequence $\langle a_{j+1}, \ldots, a_m \rangle$ is called a postfix of s for $a(postfix(s, a))$. Let α be a frequent sequential pattern in the sequence database S. A projected database $S|a$ is made by solving $postfix(s, a)$ for each sequence s in S. Searching of frequent sequential patterns is done by making projected databases repeatedly.

Now we present a parallel processing version of PrefixSpan. It is based on the master-worker framework. A frequent sequential pattern with length k is called an k-frequent pattern. A job is to extract $k+1$-frequent patterns and the ones after them from a k-frequent pattern. Initially, the central manager creates some jobs which correspond to k-frequent patterns extracted using the user-specified threshold k. Each job is assigned to a thread extracting all the following frequent sequential patterns with the depth-first search method. A thread completing its job asks the manager for a new job repeatedly until all the jobs in the manager are processed. There has been a proposal similar to this [15], however they are a parallel version of their own "Modified PrefixSpan."

The parallel PrefixSpan has the following characteristics:

(1) A job does not have any mutual dependences, and can be processed independently.

(2) It is not possible to predict the number of frequent sequential patterns extracted out of one job, nor the time required for extraction in a static analysis before computation. This is because both of them depend on the characteristic of the sequence.

(3) The difference between the computation cost of jobs tends to get larger during execution.

4 System design and implementation

Our system prototype implements dynamic load distribution based on mobile threads applied to a problem with inherently unpredictable characteristics, namely the parallel PrefixSpan. Job migration is implemented using a mobile thread system, JavaGo [12], because it does not require any special runtime environment unlike other systems. Figure 3 shows an overview of our system, in which the *migration server* is a part of JavaGo.

According to the process flow shown in Figure 3, each Grid node performs the following steps.

(1) The master process extracts k-frequent patterns from given 1-frequent patterns using a user-specified threshold k with breadth-first search.

(2) All jobs are stored in the *global job pool*.

(3) The master thread picks a job out of the *global job pool*.

(4) The master thread receives a *global job request* from the worker process, and sends a job to it. The worker process extracts s-frequent patterns (where $s > k$) from the k-frequent pattern using breadth-first search.

(5) All jobs are stored in the *local job pool*.

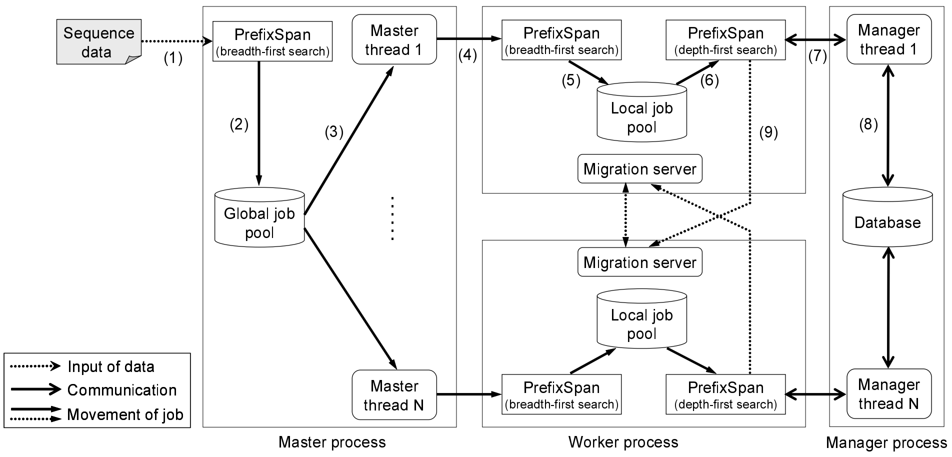


Figure 3: System overview.

(6) The worker process picks a job out of the *local job pool* unless the *local job pool* is empty, and does extraction using depth-first search.

(7) The worker process periodically communicates with the manager process during extraction, reporting its searching cost, and asking for appropriate *destination migration server*.

(8) The manager thread updates and maintains the database based on the reports and requests from the worker process.

(9) If necessary, a thread is migrated to the *migration server* in another process on another node.

(10) When the *local job pool* is empty, the flow goes to (11). Otherwise, it goes back to (6).

(11) When the *global job pool* is empty, all the processes terminate. Otherwise, the flow goes back to (3).

We need two parameters to realize the flow, which are mentioned in Section 2: the performance of a computation node and the estimated computation cost of a job.

The performance is approximated by the number of branches in a search tree which the computation node can search in one second. The value can be obtained from a preliminary benchmarking of the system.

The estimated computation cost of a job must be estimated using both (1) the current progress of searching N_{done}, and (2) the number of branches already traversed T_{past}. From the computation cost for the past search T_{past}, the computation cost for the remaining search T_{future} is calculated as

$$T_{future} = T_{past} \times (1 - N_{done})/N_{done}$$

We estimate N_{done} assuming that the search tree is balanced, i.e. the size of each subtree is the same. Figure 4 shows the detail of estimation using three subtrees. In this case, after completing the search in one subtree, we estimate that $N_{done} = 1/3$.

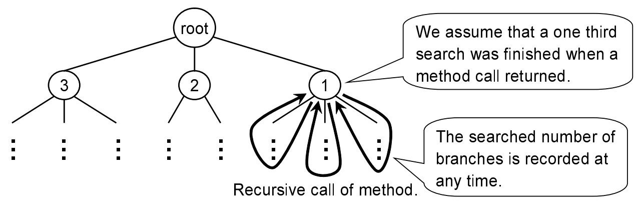

root
3
2
1
We assume that a one third search was finished when a method call returned.
The searched number of branches is recorded at any time.
Recursive call of method.

Figure 4: Example of cost prediction.

5 Experiments and evaluation

We conducted some preliminary experiments, using two and four PCs.

5.1 Experiments using two PCs

We have implemented a prototype for experimental evaluation of our system design. It is composed of two PCs of different performance as shown in Table 1. Table 2 shows migration overhead between PCs measured during the experiments. JavaGo does not actually achieve the optimal performance compared to other mobile thread systems, however it is much faster than virtual machine migration [16].

Table 1: Experiment setup.

	High-perf. PC	Low-perf. PC
CPU	Intel 3.00GHz	AMD 1.80GHz
Memory	2GB	1GB
OS	Linux 2.6	
Network	1000Base-T	
Mobile threads	JavaGo 3.1.1	
Performance (branches/sec)	180,000	17,000

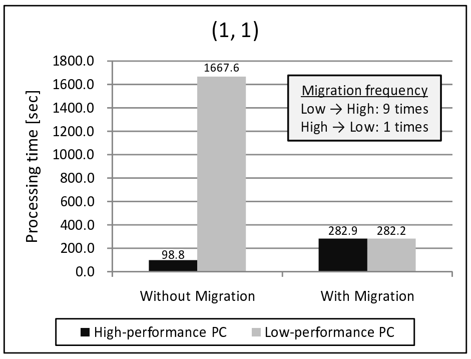
Table 2: Migration overhead.

Maximum	1,356ms	Minimum	598ms	Average	820ms

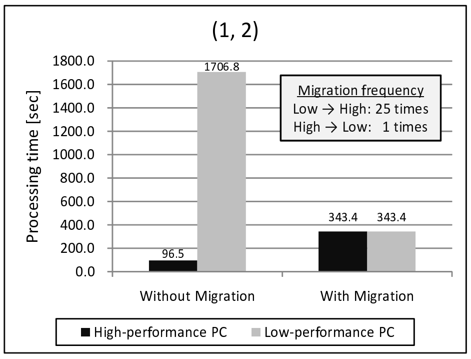
Character strings of the alphabet used in the experiments are taken from public sequence data at PROSITE [17]. They contain 6 sequences of 50 characters with 300 characters in total. The minimum support is set to 6. The threshold of the workers is between 1 and 3, and the threshold of the master is also between 1 and 3. The number of jobs that a node can execute simultaneously is set to 2. Therefore, the policy of the job migration is the same as one shown in Figure 1.

We compared the processing times of execution without migration and execution with migration. The results are shown in Figures 5 (a), (b), (c) and Figures 6 (a), (b). The numbers in parentheses indicate the threshold of the master and the threshold of the workers. Each worker communicates with the master every 100,000 cycles of branch traversals.

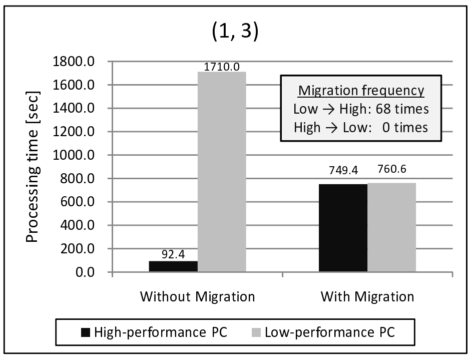
Figures 5 (a), (b), and (c) show that the processing times of the PCs



(a) The case of (1,1).



(b) The case of (1,2).



(c) The case of (1,3).

Figure 5: Simulation results (1).

differ much without migration, while they are almost equal with migration. This is because a job with high computation cost is migrated from the low-performance PC to the high-performance one.

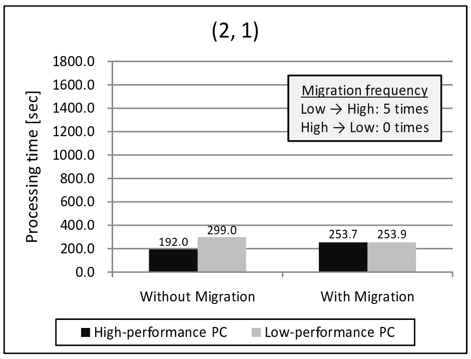
Figures 6 (a) and (b) show that without migration, the difference of the processing times between two PCs becomes small when the threshold of master is increased. This is because when the threshold is high, the master makes many jobs, each of which is small enough, so that the load of workers is well balanced. However, in this case, most searches are done not on the workers but on the master. Such case does not make sense in practical Grid systems. Figures 6 (a) and (b) also show that the processing time is shorter without migration. This is because of the migration overhead.

Figure 7 shows the result when the communication intervals between the workers and the master are changed from 100,000 to 300,000 cycles of branch traversals. The figure shows that the communication interval does not affect the outcome much.

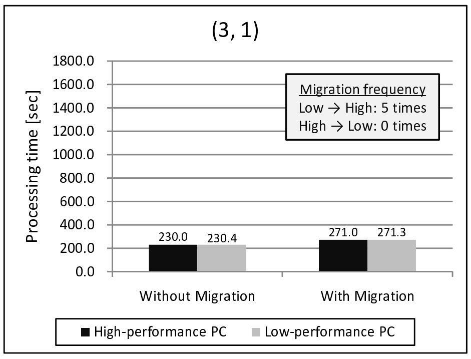
Next, we investigated the effect of the PC performance. We used two high-performance PCs, one of which had variable CPU utilization from 10% to 100% and acted as a low-performance PC. Then we measured the performance increase rate (or improvement), which is defined as

$$(1 - T_{mig}/T_{nomig}) \times 100\ [\%]$$

where T_{mig} is the processing time with migration, and T_{nomig} is the one



(a) The case of (2,1).



(b) The case of (3,1).

Figure 6: Simulation results (2).

without migration. Figure 8 shows that the larger is the difference between the PC performance, the more is the effect of migration. If one PC has only 10% power of the other, the migration improves 87.7% of the processing time, while if two PCs have the same performance, migration actually degrades the processing time.

We also measured load transition in two PCs. Figure 9 shows the transition when one PC's CPU utilization is limited to 50%. The solid line corresponds to the high-performance PC, while the dashed line shows the low-performance PC. Migration took place around the 370th time slot, then the load of the low-performance PC decreased and the load of high-performance PC increased. The same was observed around the 20th, 490th, 540th, and 660th time slots as well. The load of the low-performance PC was rising from the 220th time slot to the 370th, however the migration took place when the high-performance

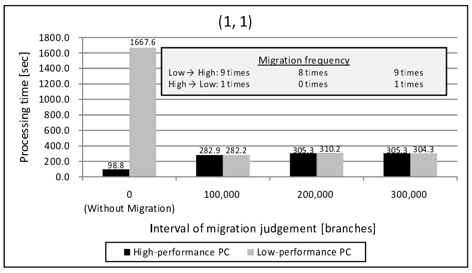


Figure 7: Result of (1,1) with different intervals.

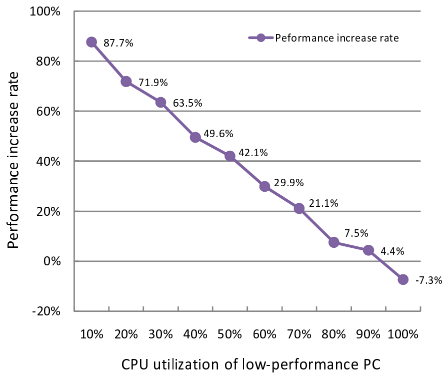


Figure 8: Performance improvement by variable PC performance.

PC became ready to accept a new job after completing its own jobs.

5.2 Experiments using four PCs

The system configuration composed of four PCs is shown in Table 3. Table 4 shows migration overheads between PCs measured during the below experiments in this configuration. The thresholds of the master and workers are set to 1. We compared the processing times of following three executions:

(1) Execution without migration.

(2) Execution with random migration.

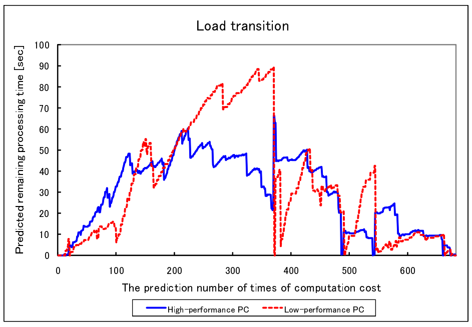


Figure 9: Load transition in two PCs.

Table 3: Experiment setup.

	PC1 & PC2	PC3	PC4
CPU	Intel	Intel	Intel
CPU	Core 2 Duo	Pentium 4	Core Duo
	3.00GHz	3.06GHz	1.20GHz
Memory	2GB	1GB	1GB
OS	Linux 2.6		
Network	1000Base-T		
Mobile threads	JavaGo 3.1.1		
Performance ratio (PC1 as 100)	100	36	30

(3) Execution with migration based on the appropriate cost prediction.

In random migration, a job is migrated to any random destination node with a fixed probability. The probability is determined so that the total migration number of times becomes almost the same as (3). In the experiment 1, CPU utilization is not limited in each PC. In the experiment 2, CPU utilization of PC2 is limited to 60%. By repeating each experiment five times, we obtained average values. The results are shown in Figure 10 (a) and (b).

Figure 10 (a) and (b) reveal that the processing times of the PCs differ much without migration, while they are almost the same with random migration and with our migration technique. Furthermore, the processing time with our migration technique is faster than the time with random migration. This is because a job with much computation cost is migrated from the low-performance PC to the high-performance one appropriately.

We also measured working ratio in four PCs. Working ratio is defined as

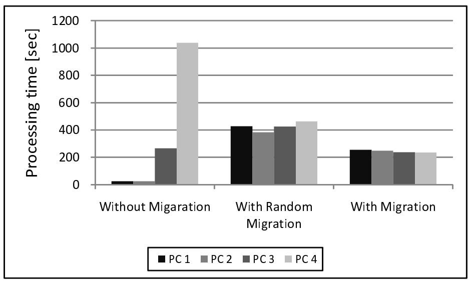
$$T_{run}/T_{max} \times 100\ [\%]$$

where T_{run} is the running time of the PC, and T_{max} is the latest processing time in four PCs. The results are shown in Figure 11 (a) and (b). All values are average of five trials.

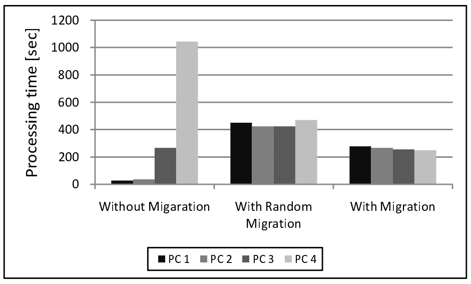
Figure 11 (a) and (b) reveal that the working ratios of the other than PC4 are low without migration and with random migration. As a result, the entire

Table 4: Migration overhead.

Maximum	1,216ms	Minimum	583ms	Average	836ms



(a) Experiment 1.

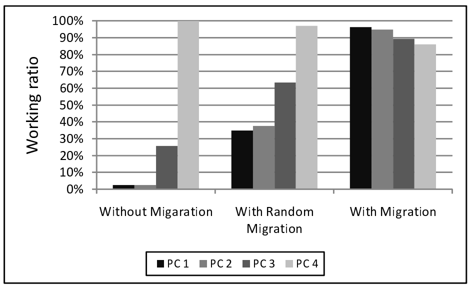


(b) Experiment 2.

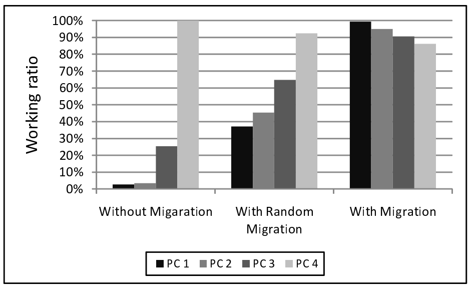
Figure 10: Processing time.

processing time gets longer. With our migration technique, the working ratios of more than 80% are kept for PCs, and any PC of the computation resource is used effectively. The results of Experiment 1 and 2 are similar regarding processing time and working ratio.

Finally, we show an example of load transition. Figure 12 shows the load transition of four PCs with random migration and Figure 13 shows the one with our migration technique. In Figure 12, the loads of PC3 and PC4 are high, while the loads of PC1 and PC2 are low, and the entire processing time becomes long. Meanwhile, Figure 13 shows that the loads of PC3 and PC4 are less than half of those with random migration, and as a result, the entire processing time becomes short.



(a) Experiment 1.



(b) Experiment 2.

Figure 11: Working ratio.

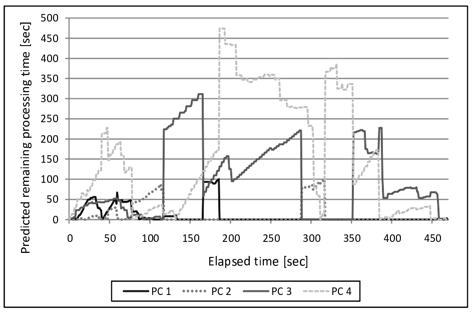


Figure 12: Load transition with random migration.

6 Conclusion

In this paper, we proposed dynamic live migration for Grid using the mobile thread mechanism and proved its efficiency compared to Grid without migration and with random migration, especially in the case where the disproportion of the jobs' computation cost is large. Our migration technique is especially effective for applications of dynamic nature such as PrefixSpan. However, it is necessary to design an appropriate system for each specific application, for example, to introduce appropriate parameters.

We are still at the starting point of this research, therefore there are still many issues that must be dealt with, such as examining our design in more rigorous manner on a practical Grid platform.

References

[1] Franco, T., et al., Seamless Live Migration of Virtual Machines over the MAN/WAN, *Future Generation Computer Systems*, **22**(8), (2006), 901–907.

[2] Clark, C., et al., Live Migration of Virtual Machines, *Proc. 2nd ACM/USENIX Symp. on Networked Systems Design and Implementation*, (2005), 273–286.

[3] Tatezono, M., et al., Making Wide-Area, Multi-Site MPI Feasible Using Xen VM, *Frontiers of High Performance Computing and Networking, Lecture Notes in Computer Science*, **4331**, Springer, (2006), 387–396.

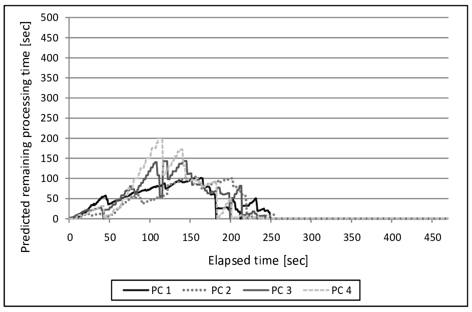


Figure 13: Load transition with migration.

[4] Thitikamol, K. and Keleher, P., Thread Migration and Load balancing in Non-Dedicated Environments, *Proc. 14th IEEE International Parallel and Distributed Processing Symposium*, (2000), 583–588.

[5] Hai, J. and Chaudhary, V., MigThread: Thread Migration in DSM Systems, *Proc. IEEE Workshop on Compile/Runtime Techniques for Parallel Computing*, (2002), 583–588.

[6] Cheng, P.C., et al., A Multi-Layer Resource Reconfiguration Framework for Grid Computing, *Proc. 4th ACM International Workshop on Middleware for Grid Computing*, (2006), 13 pages.

[7] Miyashita, M., et al. Dynamic Load Distribution in Grid Using Mobile Threads, *Proc. 3rd IEEE International Workshop on Internet and Distributed Computing Systems*, (2010), 629–634.

[8] Chess, D., et al., Mobile Agents: Are They a Good Idea?, *Mobile Object Systems towards the Programmable Internet, Lecture Notes in Computer Science*, **1222**, Springer, 1997, 25–45.

[9] Lange, D.B. and Oshima, M., *Programming and Deploying Java Mobile Agents with Aglets*, Addison-Wesley, 1998.

[10] Shudo, K. and Muraoka, Y., Asynchronous Migration of Execution Context in Java Virtual Machines, *Future Generation Computer Systems*, **18**(2), (2001), 225–233.

[11] Satoh, I., A Mobile Agent-Based Framework for Active Networks, *Proc. IEEE Systems, Man, and Cybernetics Conference*, (1999), 71–76.

[12] Sekiguchi, T., et al., A Simple Extension of Java Language for Controllable Transparent Migration and Its Portable Implementation, *Coordination Languages and Models, Lecture Notes in Computer Science*, **1594**, Springer, (1999), 211–226.

[13] Pei, J., et al., PrefixSpan: Mining Sequential Patterns Efficiently by Prefix-Projected Pattern Growth, *Proc. 17th IEEE International Conf. on Data Engineering*, (2001), 215–224.

[14] Yamamoto, K., et al., Learning Sequence-to-Sequence Correspondences from Parallel Corpora via Sequential Pattern Mining, *Proc. 2003 Workshop on Building and Using Parallel Texts: Data Driven Machine Translation and Beyond*, **3**, (2003), 73–80.

[15] Sutou, T., et al., Design and Implementation of Parallel Modified PrefixSpan Method, *High Performance Computing, Lecture Notes in Computer Science*, **2858**, Springer, (2003), 412–422.

[16] Voorsluys, W., et al., Cost of Virtual Machine Live Migration in Clouds: A Performance Evaluation, *Cloud Computing, Lecture Notes in Computer Science*, **5931**, Springer, (2009), 254–265.

[17] Swiss Institute of Bioinformatics, PROSITE - Database of Protein Domains, Families and functional sites, http://prosite.expasy.org/ *(Last accessed on 27 July 2016)*, (2014).