Multi-cluster Load Balancing Based on Process Migration*

XiaoYing Wang, ZiYu Zhu, ZhiHui Du, and SanLi Li

Tsinghua National Laboratory for Information Science and Technology
Department of Computer Science & Technology, Tsinghua University, Beijing 100084
{wangxy,zzy,duzh,lsl}@tirc.cs.tsinghua.edu.cn

Abstract. Load balancing is important for distributed computing systems to achieve maximum resource utilization, and process migration is an efficient way to dynamically balance the load among multiple nodes. Due to limited capacity of a single cluster, it's necessary to share the underutilized resources of other sites. This paper addresses the issues in multi-cluster load balancing based on process migration across separate clusters. Key technology and mechanisms are discussed and then the implementation of a prototype system is described in detail. Experimental results depict that by achieving multi-cluster load balance, surplus resources can be efficiently utilized and the makespan is also greatly reduced as a result.

Keywords: Load balancing, Process migration, Multi-cluster.

1 Introduction

Recent years, clusters of inexpensive networked computers are increasingly a popular platform for executing computationally intense and long running applications. The main goal of cluster systems is to share all the resources of the whole system through the interconnections and to effectively share resources via efficient resource management and task scheduling, finally achieving high performance. Thus, a key problem is how to effective utilize the resources. However, in such a loose-coupled computing environment as clusters, load imbalance, which is usually caused by the variable distribution of workload on the computing nodes, leads to performance degradation. Therefore, it is important for a cluster to balance the load among all nodes to improve the system performance.

Methods for load balancing can be classified into two categories. Static approaches are usually achieved by task allocation beforehand, which requires prior knowledge of exact information of tasks (such as execution time) and thus cannot adapt to the run-time load variation. Dynamic approaches are more complex and need the support of process migration, one of the most important techniques

* This work is supported partly by the Natural Science Foundation of China under Grant No.60503090 and No. 10778604, and China's National Fundamental Research 973 Program No.2004CB217903.

to fully utilize the resources of the entire system. Besides, process migration in a cluster system is also of benefit to system maintenance, availability, data locality, mobility and failure recovery [1,2,3]. Process migration can be implemented on different levels, which greatly affects the design choices. Current implementations can be divided into four categories:

- `Unix-like Kernel`. This method requires significant modification to monolithic kernel to achieve full transparency. Nevertheless, kernel-level implementation could sufficiently utilize the functionality of the operating systems(OS), obtain the detailed status about processes and thus provide high efficiency to users. Examples are LOCUS [5] and Sprite [6].
- `Microkernel`. As a separate module, microkernel builds process migration facilities on top of OS. It's relatively easy to implement because of the availability of system transparency support and message passing communication mechanism. Mach [7] is a typical example.
- `User-space migration`. This method doesn't require the modification of the kernel, implying that the entire address spaces are extracted and transferred to rebuild at the destination node. Condor [8] is an example system. Since not all states can be obtained at user level, some processes cannot be migrated. Great overhead has to be paid for breaking the obstacle between kernel space and user space, and thus the efficiency is much lower.
- `Application-level migration`. The function of process migration is integrated into real applications, and optimization methods could be designed based on particular applications. Freedman [9] and Skordos [10] have studied about such approaches. However, transparency and reusability would be significantly sacrificed, since semantics of the application need to be known and the programs have to be rewritten or recompiled. The functions of migrated processes are limited, and each new application has to be specially designed.

OpenMosix [4] is a typical tool which modifies the OS kernel to support process migration and allows multiple uniprocessors and symmetric multiprocessors running the same kernel to work in close cooperation. Dynamic load balancing can be achieved by migrating processes from one node to another, preemptively and transparently. Via openMosix, a cluster is seen as a large virtual machine with multiple CPUs and mass storage, and the cluster-wide performance of both sequential and parallel applications can be improved. However, when the capacity of a single cluster is limited, resources of multiple clusters ought to be shared. Unfortunately, conventional implementations of kernel-level process migration don't support migrating across different clusters. As computational grids emerged and got widely used, resources of multiple clusters became dominant computing nodes of the grid. Cross-cluster process migration can help to balance the load among multiple grid nodes with fine granularity, so it's also valuable for multi-site load balancing in computational grid. In this paper we discuss the key issues in multi-cluster load balancing and considerations in the implementation of a prototype system. Experimental results demonstrate that resource utilization greatly benefits from enabling multi-cluster load balancing, thus leading to significant reduction in task makespan.

2 Process Migration Techniques

In this section, we discuss the key techniques and mechanisms of process migration based on openMosix [11,12]. Each process has only one Unique Home Node(UHN) where it was created, usually the node which the user logged in. Every process seems to be running on its UHN, but in fact it may have been transferred to another node.

2.1 Deputy/Remote Mechanism

A migrated process consists of two parts - user context (called *remote*) and system context (called *deputy*). *Remote* contains the program code, stack, data, memory-maps and registers of the process, encapsulating the process when it's running at user level. Meanwhile, *deputy* encapsulates the process when it's running at kernel level, containing the description of resources that the process is attached to and a kernel-stack for the execution of system code on behalf of the process. Thus, *remote* can be migrated several times among different nodes, while *deputy* is site-dependent and can only stay on UHN. The interfaces between user context and system context are well defined. It's possible to intercept every interaction between the two contexts, and forward it across the network.

In Linux operating system, a process can only enter the kernel level via system calls. The interception and forwarding of every interaction between two contexts are done by the Link Layer. When a process has been migrated, its *deputy* and *remote* still keep connected. *Remote* deals with UHN-dependent operations through *deputy*, like getting environment variables or doing I/O. High transparency is achieved by their interaction. When *remote* meets system calls or resource requests when running the process, it sends them back to *deputy*. *Deputy* is always looping itself in kernel level, waiting for the requests from *remote*. Then, it deals with them and sends the results back to *remote*.

2.2 Migration Procedure

After the OS with migration support boots up, three daemons are started on each node running as kernel-level threads, including: *mig_daemon*, which listens at a certain port and deals with incoming requests; *info_daemon*, which collects and distributes the load information; and *mem_daemon*, which monitors memory utilization. Necessary steps of the process migration procedure include: (1)*Target selection*. A target node can be either specified by user manually or automatically decided by the scheduling system according to history and current load information of the cluster. When a process *P* is selected to migrate, it is marked and its priority is increased in order to get CPU timeslices more easily. Then, a request is sent to the *mig_daemon* on the target node. (2)*Request and negotiation*. After receiving the migration request of process *P*, the target node forks a new process *P'* to deal with it. *P'* first asks the load-balancing module whether to accept the incoming request and the module judges it according to predefined algorithms. *P'* is marked as remote, and then a TCP connection is established between *remote* and *deputy*, exchanging messages between two nodes. (3)*Process state transfer*. If the target node accepts P to migrate there, it sends back an acknowledgement to the source node. After receiving the acknowledgment message, the source node starts to extract the states of process P and forwards them to the remote process. The *remote* modifies its own process states according to the received data. (4)*Execution resuming*. Once the sending phase is finished, P becomes *deputy* and enters a waiting loop in kernel level, until the process is terminated or migrated back to its UHN. *P'* is modified to $READY$ state and added into the running queue, waiting for scheduling. When it is scheduled and enters the use space, it resumes executing from the instruction before migrating, which indicates the completion of the entire process migration procedure.

3 Multi-cluster Load Balancing Implementation

In reality, cluster systems often belong to different organizations or locate in isolated districts. Extending and merging the capability of existing cluster systems which are geographically distributed is helpful to achieve unified management and also decrease the cost of system expansion. In this section we present the main issues to implement a system supporting multi-cluster load balancing.

3.1 Multi-cluster Architecture

In a cluster system, computing nodes are usually configured inside a Local Area Network. A gateway node is connected to all the nodes via one Network Interface Card(NIC), and to the public network via another NIC, assuring that external users can log in and operate. For security consideration, users from public network cannot access the internal nodes of a cluster directly. The gateway node acts as a router, and also guarantees the security of the cluster system via firewall configurations. The topology of multi-cluster architecture is shown in Fig. 1.

As seen, the computing nodes inside different clusters are unable to communicate with each other. Traditional process migration approaches require direct connection of UHN and the remote node. As a result, when a cluster is under heavy load while other clusters are light-loaded, the imbalance problem emerges from a global point of view. To support multi-cluster load-balancing, lower-level operation system needs to be modified and the gateway can be used to forward migration data and necessary messages. Here, we add a property *groupID* into the data structure representing a node, indicating the cluster it belongs to. The gateway node is not involved in computing and has two entries in the configuration file - one for internal IP address, which contains a *groupID* of the cluster it belongs to; another for public IP address, which has a *groupID* of 0. An example file of two clusters is shown in Fig. 2.

3.2 Information Collection

There is no central control or master/slave relationship between nodes. For automatic load-balancing, every node has to know the current status of other nodes.

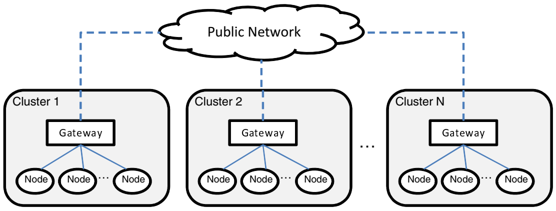


Fig. 1. Architecture of Multiple Clusters

```
# ID  IP  number-of-nodes groupID
# ================================
   1 10.0.0.1      1       1
   2 10.0.0.5      1       1
   3 10.0.0.6      gw      1
   3 192.168.4.15  gw      0
   4 192.168.4.11  gw      0
   4 10.0.0.111    gw      2
   5 10.0.0.9      1       2
   6 10.0.0.12     1       2
```

Fig. 2. Config File Example

This is achieved by each node sending its own info to others periodically. Hence, it's necessary to enable the support for information collection in multi-cluster scenario. Load information is forwarded using UDP protocol, as the information messages needn't acknowledgement and have no critical requirements for reliable transfer. Each node starts an *info_daemon*, waiting for info messages from other nodes. A node sends back an info message about itself immediately after receiving another's.

In order to let the gateway know where to send these messages, two additional segments are added, each carrying a *sockaddr* structure of the network address. The converting and sending procedure of the encapsulated data is shown in Fig. 3, which involves three types of nodes working: 1)*Sender(Src)*. The source node first queries the *groupID* of the destination node and compares to its own *groupID*. If unequal, then they are not in a same cluster and cross-cluster migration is needed. Then, it queries the configuration table again to get the external address of the gateway node of the destination cluster together with the internal address of the gateway node of the local cluster. Data are encapsulated as a three-segment format, with the first segment containing the destination gateway address(*Gw2 addr*), the second segment containing the destination node address (*Dest addr*), and the last segment containing the original message content. The encapsulated data are sent to the local gateway. 2)*Gateway node(Gw)*. The gateway is listening and waiting for the encapsulated data from *Src*. It doesn't care the content, only processing and converting messages. It reads the first segment and extracts the network address as the next destination, then copies the second segment to the first segment, and fills the second segment with the address of the original sender. 3)*Receiver(Dest)*. After converted by two gateway nodes sequentially, the first segment of the encapsulated data becomes the address of *Src* and the second segment is the gateway of the source cluster(*Gw1*). The receiver can query the configuration table and find the *ID* of *Src*, and then obtain load info about *Src* from the last segment of received data.

3.3 Cross-Cluster Process Migration

According to the source and destination of the process to be migrated, we describe following three different scenarios.

Local to remote (migrating to a remote node). Before migrating, the local node first connects to its *mig_daemon* and sends requests containing the reason to migrate. After acknowledged by the remote node, necessary data are transferred

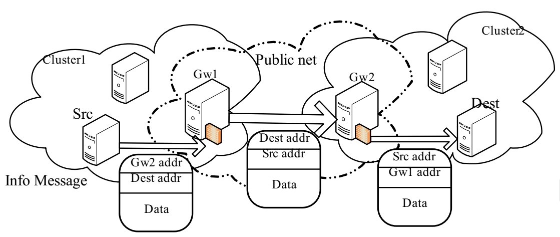


Fig. 3. Info Messages Forwarding

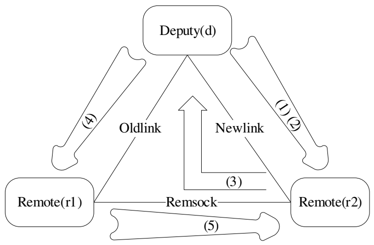


Fig. 4. remote to remote

continuously, including process memory states, virtual areas, physical pages and so forth. Once finishing, the local process becomes deputy and enters a loop waiting for system-calls from remote. At the remote node, *mig_daemon* accepts incoming migration requests and sends back acknowledgement if it agrees to receive the process. Then, a user-level process is created and after receiving all necessary data, the process is changed to *READY* state and waits for scheduling.

Remote to local(migrating back home). When a process is coming back from the remote node, the UHN sends a *DEP_COME_BACK* instruction, and then receives all of the process states. Remote starts "going home" after being acknowledged by UHN. After finishing sending the necessary data, remote kills the migrated process and exits normally.

Remote to remote(migrating more than once). It's a little more complex when a process needs to migrate from one remote node to another. Figure 4 illustrates the steps of migration from remote *r1* to *r2*. (1)*oldlink* is the link between *d* and *r1*, and *newlink* connecting *d* and *r2* is created. (2) *d* sends a probing request to *r2*, and notifies it a migration event is going to occur. (3) *r2* sends information of itself to *d*. (4) *d* copies the information about *r2* and sends it together with a request to *r1* via *oldlink* and tells it "Please Migrate". (5) *r1* opens a new link to *r2*, and then sends a migration request to *r2*. After sending necessary data to *r2*, it releases the local process. Till now, the process is migrated to *r2* and only related to *d* and *r2*.

In order to transfer process states reliable TCP connections should be established among nodes and gateways. Unlike info messages, migration messages need bidirectional transfer. The source node sends an "open-connection" command first, the structure of which is similar to the packed info message, with the third segment containing a command string. Since the *mig_gateway* acts almost in the same way as the *info_gateway*, detailed descriptions are omitted here. Note that if the migration type is "remote to remote", the listening port will be randomly selected. Once the process terminates, the source node notifies the gateway to close all open connections.

3.4 Load Balancing Strategy

The main load balancing algorithms consists of CPU load-balancing and memory ushering. Due to space constraints, here we only focus on the CPU load-balancing strategy. The dynamic CPU load-balancing algorithm continuously attempts to reduce the load differences between pairs of nodes, by migrating processes from high loaded to less loaded nodes. This scheme is decentralized with all the nodes regarding each other as equal-position peers and executing the same algorithm. The whole balancing strategy is comprised of following steps.

1)Calculate the load values. Since the nodes in the system may be heterogeneous, the load calculation is related to the number of processors(denoted as N_p) and their speed. First, the number of running processes is added to an accumulator at each clock interrupt. Then, the accumulated load L_{acc} is normalized to the CPU speed of this processor using the maximum CPU speed S_{MAX} in the system versus the local node's calculated CPU speed S_{cal}, namely,

$$L_{acc} = L_{acc} \cdot \frac{S_{MAX}}{S_{cal} \cdot N_p}. \tag{1}$$

Under the consideration of preventing migration thrashing, the actual load value sent to other nodes, called "the export load", is calculated to be slightly higher than the load calculated for internal use. Denote the internal load as L_{int} and the export load as L_{exp}. To calculate L_{exp}, a value representing the highest load over any period has to be recorded, denoted as L_{up}. Denote L_{income} as the summarized load brought by processes"recently" migrated to this node. Then, the internal load and the export load can be computed in the following way:

```
If ($L_{acc}>L_{int}$) //slowly up
    $L_{int}=L_{int}$*decay+$L_{acc}$
Else //quickly down
    $L_{int}=L_{acc}$
End If
If ($L_{acc}>=L_{up}$) //quickly up
    $L_{up}=L_{acc}$
Else //slowly down
    $L_{up}=(L_{up}$*7+$L_{acc})$/8
End If
$L_{exp}=L_{up}+L_{income}$ //prevent receiving too many new processes too quickly
```

where *decay* is predefined constant value between 0∼1. Each node maintains a local load vector storing both the internal load value of its own and export load values received from other nodes.

2)Choose a process to migrate. Processes cannot be migrated if they are constrained, such as being locked, in creation phase, being transferred, or using a shared memory region. Moreover, if a process has not accumulated enough CPU usage, it is not considered for migration. For each process, a migration priority value is first calculated based on the CPU use since it is last considered for migration, including the CPU use of all its children processes. Then, this priority value is combined with a value which attempts to measure the process's contribution to the load currently on the machine. Consequently, a process which forks frequently is more attractive for migration, because once migrated it will continue to fork children thus spreading the load as it bounces from one node to another. Once a process is chosen to be migrated, a flag will be set, indicating that it's the candidate process selected for load-balancing consideration.

3)Choose a destination node. It is a complicated problem to determine the optimal location for the job to be migrated, since available resources are usually heterogeneous and even not measured in the same units. Here we try to reconcile these differences by standardizing the resource measurements and adopt a method based on economic principles and competitive analysis. The target node for migration is determined by computing the opportunity cost for each of the nodes in the local load vector, which is a concept from the field of economies research. The key idea is to convert the total usage of several heterogeneous resources, such as memory and CPU, into a single homogeneous "cost". Jobs are then assigned to the machine where they have the lowest cost, just like in a market oriented economy. A simple way is to compute the marginal cost of a candidate node, namely, the amount of the sum of relative CPU usage and memory usage would increase if the process was migrated to that node. The goal is to find a node with minimal marginal cost and select it as the destination.

As a whole, the above load-balancing algorithms respond to variation in the runtime characteristics of the processes, as long as there is no extreme shortage of other resources such as free memory or empty process slots.

4 Performance Evaluation

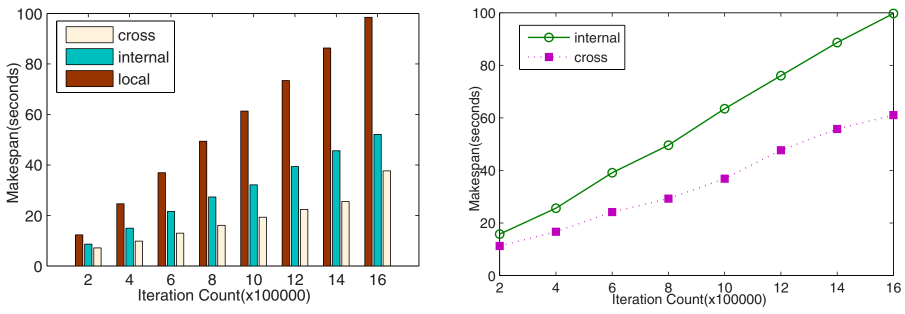
This section presents the results of performance evaluation experiments conducted on the prototype system we implemented. Our testbed is comprised of two clusters: *Cluster 1* has nodes with dual PIII 500MHz CPUs and 128M physical memory; *Cluster 2* has nodes with dual PIII 733MHz and 256M physical memory. There are two independent networks of each cluster, and they own a gateway node respectively, connecting to the public net. Gateways can communicate to each other directly. Both nodes inside the two clusters and the gateways are connected by 100Mb/s Ethernet. After the startup of our prototype system on each node, all the load information used by the scheduling module can be monitored by upper-level tools.

We use a simple CPU-intensive program for exemplification of the experiment, which requires two parameters - the iteration count and the number of child processes. As iteration count becomes larger, the CPU load will be accordingly heavier. We tested and compared three scenarios respectively: without migration support (*local*), with intra-cluster migration support (*internal*) and with cross-cluster migration support (*cross*). Figure 5 shows the task completion time (i.e. makespan) achieved in the above three scenarios, in which Fig. 5(a) has 8 child processes forked and Fig. 5(b) has 16 ones.

During the period when the program is running, we can monitor the load situation on the nodes by user-level tools. While the operating system treats all processes as a whole task, through the userspace monitoring tool implemented in our prototype system, we can distinguish every child process clearly. Take Fig. 5(b) for example, when the program has been submitted to a node and starts to run, the load of the local cluster quickly increases, while the other cluster is idle without any heavy-load tasks; after enabling multi-cluster load balancing, 4 processes are distributed on each node, and the CPU utilization of all nodes reaches nearly 100%.

From Fig. 5, it can be observed that without migration support, it results in low efficiency and remarkably takes more time to complete the task. If the intra-cluster migration support is enabled, a half time can be saved (because there are two nodes in the cluster to share the load), but nodes inside the cluster are still under heavy burden. With multi-cluster load balancing, load can be shared across different clusters and thus the task makespan is greatly reduced. The speedup is computed and shown in Table 1, which depicts the significant improvement by enabling multi-cluster load balancing. Moreover, from the table we can also observe that as the iteration count becomes larger, the speedup of *cross* increases proportionally. That's because in this program the iteration count represents the computation amount of the tasks, and long-running tasks are more tolerant of "non-computation" overhead because of their urgent requirement for additional resource. As the results demonstrate, multi-cluster load balancing can make the resource of multiple clusters more effectively utilized and workload more balanced in a total view.

Nevertheless, the performance overhead cannot be ignored, especially when the migration experiences two hops at the gateway nodes. Now we investigate the multi-cluster load balancing overhead, which involves information analysis and making scheduling decisions, process states transfer, communication over network, and results finalization. We figure out the average summarized overhead per computing node and compared *cross* with *internal* in order to make clear how much additional overhead is caused by the processing of gateway nodes, as shown in the bottom rows of Table 1. Again we can see that larger amount of computation makes the overhead relatively smaller, as explained previously. Meanwhile, as the computation amount increases, the additional overhead incurred by *cross*(relative to *internal*) becomes more significant too, due to massively more data passing through the intermediate gateway nodes. However, to sum up, *cross* only incur less than 3% more overhead compared to the inter-



(a) 8 processes (b) 16 processes

Fig. 5. Performance Evaluation Results

Table 1. Comparison of Speedup and Overhead

Iteration($\times 10^5$)		2	4	6	8	10	12	14
SpeedUp	Internal	1.43	1.65	1.71	1.81	1.91	1.86	1.89
	Cross	1.72	2.51	2.84	3.07	3.17	3.28	3.38
Average Overhead Per Node	Internal	14%	8.8%	7.1%	4.8%	2.3%	3.4%	2.7%
	Cross	14%	9.3%	7.2%	5.8%	5.2%	4.5%	3.9%

nal load balancing, which is acceptable especially when dealing with long-term tasks. In practice, the upper-level scheduler should be able to decide whether to migrate according to the overhead and possible efficiency gain.

5 Conclusions and Future Work

In this paper, we conduct researches on multi-cluster load-balancing based on the study and analysis of process migration mechanism. The main problem is that the internal node of a cluster cannot be directly accessed by other machines outside the cluster. Hence, we have designed a system which supports multi-cluster load balancing by employing gateway nodes to forward and transfer necessary data. The results of performance evaluation experiments based on the prototype system have demonstrated the availability and efficiency of multi-cluster load balancing in reducing the task makespan. Our work can be regarded as a preliminary step into the research on dynamical resource sharing and load balancing of multiple large nodes in computational grid environment (especially for CPU-intensive applications). This prototype system has been put into practice and achieved unified management and resource sharing among multiple clusters. As a possible direction for future work, we plan to investigate some potential problems, such as the bottleneck of gateway nodes when dealing with frequent system-calls. Moreover, in the case of decentralized grid environment, the latency between gateway nodes may be considerable, which requires the decision of whether and when there exists the need to migrate. As a further step, we are also considering to employ modeling and simulation to test the performance of multiple large cluster systems in the grid under different workloads.

References

1. Milojicic, D.S., Douglis, F., Paindaveine, Y., et al.: Process migration. ACM Comput. Surv. 32(3), 241–299 (2000)
2. Powell, M.L., Miller, B.P.: Process migration in DEMOS /MP. In: Proc. Ninth Symposium on Operating System Principles, pp. 110–119. ACM, New York (1983)
3. Smith, P., Hutchinson, N.C.: Heterogeneous process migration: The tui system. Technical Report TR-96-04, University of British Columbia. Computer Science (1996)
4. The openMosix Project, `http://openmosix.sourceforge.net`
5. Popek, G.J., Walker, B.J., Johanna, M., et al.: LOCUS - A Network Transparent, High Reliability Distributed System. Proceedings of the 8th Symposium on Operating System Principles, 169–177 (1981)

6. Douglis, F., Ousterhout, J.K.: Transparent Process Migration: Design Alternatives and the Sprite Implementation. Softw., Pract. Exper. 21(8), 757–785 (1991)
7. Accetta, M., Baron, R., Bolosky, W., et al.: Mach: A New Kernel Foundation for UNIX Development. In: Proceedings of the Summer USENIX Conference, pp. 93-112 (1986)
8. Litzkow, M., Solomon, M.: Supporting Checkpointing and Process Migration outside the UNIX Kernel. Proceedings of the USENIX Winter Conference, pp. 283-290 (1992)
9. Freedman, D.: Experience Building a Process Migration Subsystem for UNIX. In: Proceedings of the WinterUSENIX Conference, pp. 349-355 (1991)
10. Skordos, P.: Parallel Simulation of Subsonic Fluid Dynamics on a Cluster of Workstations. In: Proceedings of the Fourth IEEE International Symposium on High Performance Distributed Computing (1995)
11. Argentini, G.: Use of openMosix for parallel I/O balancing on storage in Linux cluster. CoRR cs.DC/0212006 (2002)
12. Katsubo, D.: Using openMosix Clustering System for Building a Distributed Computing Environment, `http://www.openmosix.org.ru/docs/omosix.html`