A New Approach to Configurable Dynamic Scheduling in Clusters based on Single System Image Technologies

Geoffroy Vallée[1], Christine Morin[2],
Jean-Yves Berthou[1], Louis Rilling[3]
IRISA/INRIA, Campus Universitaire de Beaulieu, 35042 Rennes, Cedex, France
EDF R&D, 1 avenue de Général de Gaulle, BP408, 92141 Clamart, France
E-mail:{gvallee,cmorin,lrilling}@irisa.fr, Jy.Berthou@edf.fr

Abstract

Clusters are now considered as an alternative to parallel machines to execute workloads made up of sequential and/or parallel applications. For efficient application execution on clusters, dynamic global process scheduling is of prime importance. Different dynamic scheduling policies that have been studied for distributed systems or parallel machines may be used in clusters. The choice of a particular policy depends on the kind of workload to be executed. In a cluster, it is thus highly desirable to implement a configurable global scheduler to be able to adapt the dynamic scheduling policy to the workload characteristics, to take benefit of all cluster resources and to cope with node shutdown and reboot. In this paper, we present the architecture of the global scheduler and the process management mechanisms of Kerrighed, a single system image operating system designed for high performance computing on clusters. Kerrighed provides a development framework allowing to easily implement dynamic scheduling policies without kernel modification. In Kerrighed, the global scheduling policy can be dynamically changed while applications execute on the cluster. Kerrighed process management mechanisms allow to easily deploy parallel applications in the cluster and to efficiently migrate or checkpoint processes, including processes sharing memory. Kerrighed has been implemented as a set of modules extending Linux kernel. Preliminary performance results are presented.

1 Introduction

Clusters are now more and more widely used to execute scientific workloads made up of tasks that may be sequential or parallel applications. In the domain of high performance computing, the goal of the scheduler is to minimize the time a task remains in the system. It is thus highly desirable to optimize the use of the resources available in the architecture to execute as quickly as possible the tasks. A multiprogrammed environment is needed as generally a task does use all the system resources. Several dynamic scheduling policies have been defined to allocate processors to processes during the execution of tasks. In a cluster, as the processor and memory resources are distributed in different nodes, a global scheduler need to be implemented. Different global dynamic scheduling policies that have been studied for parallel computers and distributed systems may be used in clusters. In fact, there does not exist a unique scheduling policy that is suitable to all kinds of workloads that may be executed on a cluster. Thus, there is a need of a configurable global scheduler in which the dynamic scheduling policy can be selected regarding the characteristics of the workload to be executed.

The work we present in this paper relates to the design and implementation of a configurable global dynamic scheduler for Kerrighed *Single System Image* (SSI) operating system for high performance computing on clusters. Kerrighed extends Linux operating system with a set of distributed services that globally manage the cluster resources to provide the same interface to programmers as the one of Linux running on an SMP machine. Kerrighed provides a development environment to ease the implementation of global dynamic scheduling policies. In Kerrighed, the global scheduling policy can be changed without stopping the execution of tasks running on the cluster. Kerrighed global scheduler relies on global process management mechanisms which allow to easily deploy parallel applications in the cluster and to efficiently migrate or checkpoint processes, including processes sharing memory. Section 2 provides a background on global scheduling in clusters. Section 3 presents the Kerrighed[7] (formerly called

[1]EDF R&D
[2]IRISA/INRIA
[3]ENS Cachan/IRISA

Gobelins) SSI operating system. Section 4 describes the architecture of Kerrighed configurable global dynamic scheduler and the underlying process management mechanisms. Preliminary performance results are presented in Section 5. Section 6 concludes.

2 Background

There are two different kind of scheduler: static schedulers, dynamic schedulers, and each of these schedulers can be adaptive. The static schedulers create processes according to a policy. Then, the processes cannot be migrated during their execution. So balancing problem during process execution cannot be solved. The dynamic schedulers can migrate processes during their execution. So, if a load balancing problem appears or if a node needs to be removed from the cluster, the scheduler can apply a new placement of processes without stopping their execution. Schedulers can be adaptive to execute differents kinds of workloads: sequential applications, parallel applications communicating by shared memory or by message passing, applications which need a lot of memory or a large CPU usage, etc. These workloads need a different global management of resources, and so need a different scheduling policy. Various schedulers have been proposed for clusters.

A lot of systems offer static scheduling. This is the case in Beowulf clusters[8]. A Beowulf system offers to the programmers a set of programming toolkits and running environments like MPI[10] or PVM[9]. When an application is launched the MPI runtime places its processes on the cluster nodes (without any knowledge of nodes state). This approach has a major drawback: it can be used only by users who can write from scratch their application, and who can develop with the message passing programming paradigm. Unfortunately, this kind of users is a small part of the cluster users. Second, the process placement is statically defined by the runtime without information about nodes, so it cannot place application according to the resource usage or dynamically manage scheduling problems (i.e. if the process placement is not really efficient).

Other systems like batch systems are based on automatic and transparent process placement at creation time, according to cluster resources available. For example, PBS[5] provides an efficient process placement mechanism using system information about all nodes, queues for process management and efficient placement policies. When a user submits an application to the system, it gives some information about the application (computation time, memory use, CPU use). This information allows the system to queue the processes in a suitable waiting queue. Then, with the system information about nodes, the system can determine the adequate nodes and the adequate time to efficiently execute the application.

With dynamic scheduling, the scheduler can manage processes in the cluster in order to use efficiently available resources, in a transparent way for the users. Actual systems which offer a dynamic scheduler are specialized for a given problem. For example, MOSIX[2] and SPRITE[4] provide an efficient dynamic scheduler for computing sequential applications. NOMAD[6] provides a dynamic scheduler for parallel and sequential applications, using a co-scheduling of concurrent applications and queues to create applications. But in these three systems, all the resources of the cluster cannot be globaly managed, and so all the processes cannot be migrated efficiently. For example, in the MOSIX[2] system which does not offer a global management of the network, a communicating process which migrates still depends on its creation node: the process sockets are still linked with the creation node. Finally, the execution of a migrated process is not efficient. Another systems like GENESIS[1], based on the micro-kernel technology (all distributed kernel services are developed from scratch), offers a global management of memory, processor and network. Moreover, GENESIS[1] provides an efficient process group management which allows to deploy parallel application. So, GENESIS[1] provides an efficient mechanism of parallel application deployement (and so process placement) and process migration.

But such systems are specialized for a given workload, and so cannot execute efficiently all kinds of workloads. For example, a system specialized in load balancing for sequential applications maybe unefficient for parallel applications by shared memory which can create false sharing, and so ping-pong of memory pages.

Finally some systems allow to specialize the scheduler, and so provide an configurable scheduler. In PBS, the scheduler is specializable, using the specialized language *Batch Scheduling Language*, Tcl or C. For example, an implementation of the Maui Scheduler[3] has been implemented in PBS.

To offer a complete solution for cluster scheduling which can manage all kinds of workloads, the scheduler needs to be dynamic and adaptative, and based on a cluster SSI.

3 Overview of Kerrighed

Kerrighed is a single system image cluster operating system extending Linux kernel. Kerrighed is implemented by slight modifications of the Linux kernel and a set of modules. Linux system (as any modern system) can be divided in two layers: a virtual layer and a physical layer. The vir-

tual layer implements the application system interface and resource virtualization. The physical layer implements device access (hard disk, memory).

Kerrighed extends traditional system mechanisms thanks to the *container* concept. A container is a software object which allows to store and share memory pages between cluster nodes. Containers are inserted between these two layers (see Figure 1). So, all system events between the virtual and the physical layers can be intercepted by containers, allowing to divert and to extend traditional operating system services, and to access distributed data.

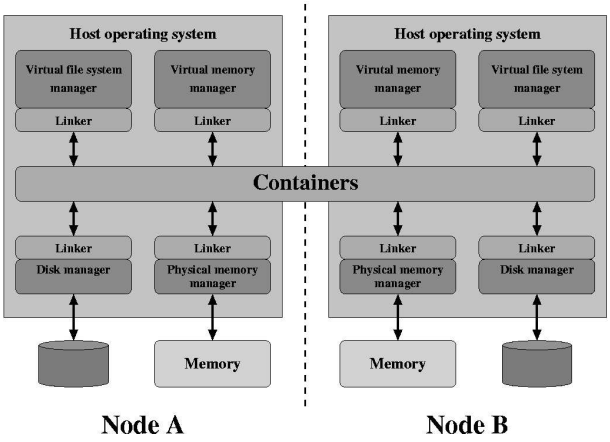


Figure 1. Integration of containers in a standard OS

Each container is associated with two linkers: an *interface linker* which diverts high level kernel functions to containers, and an *i/o linker* which allows the container to access a given device manager. For example, a shared virtual memory (SVM) can be implemented using an interface linker to connect a container to the virtual memory manager, and an I/O linker to connect this container to the physical memory manager. In this case, a container can be seen as an extension of the memory segment concept to the cluster scale. The process address space is divided into memory segments which can be associated to containers, and can be shared between different processes.

Based on the container mechanism, an SVM, a cooperative file cache and a distributed file system are implemented in the system. Kerrighed allows to execute standard Linux applications including pthread applications. As Kerrighed is based on Linux, a thread is implemented by a Linux process, and a Kerrighed process (K-process) is a Linux process whose segments are linked to containers.

4 Global Scheduler

4.1 Requirements

A given workload can be efficiently executed on a cluster if available resources are efficiently used. Thus, the cluster OS needs an efficient scheduler which can efficiently create processes on any cluster node according to the resources available, and which allows to migrate processes during their execution in order to balance the load on the cluster nodes. The scheduler should be configurable to be adapted to the submitted workload. Moreover, a scheduler needs a mechanism which allows to specify and implement new scheduling policies. So, a suitable node has to be chosen when a K-process is created for is execution (**process placement**), or when a K-process is migrated (**process migration**) to benefit of remote resources or to decrease the pressure on local resources. K-processes can also be checkpointed (**process checkpoint-restart**) to decrease the pressure on local resources.

In the standard Linux system, processes and threads can be created with the **fork** and **execv** interface. In SMP machines, the standard interface **pthread** allows to create and manage threads of shared memory applications. We need such mechanisms in a cluster OS in order to be able to execute existing applications developed for SMP machines on a cluster.

We also have an implementation constraint: the scheduler of the Linux kernel must not be modified, all the scheduling mechanisms must be out of the kernel scheduler. Thus, Kerrighed global scheduler just adds or removes processes in the standard Linux scheduler queues of the cluster.

4.2 The Kerrighed Global Scheduler

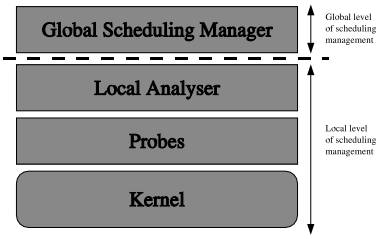
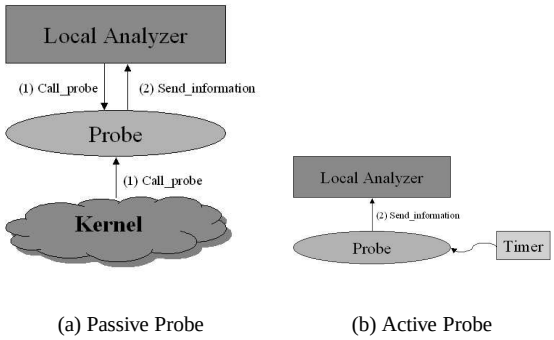


Figure 2. The Kerrighed scheduler layers

We propose a modular global scheduler, composed of three layers (see Figure 2): a probe layer, for the global scheduler to obtain system information giving a view of the cluster state; a local analyzer to detect all the local scheduling problems like high resource pressure or device failures; and a global scheduling manager in order to place the new

processes and to solve global scheduling problems (for example to balance the load) on all cluster nodes.

4.2.1 System Probes



(a) Passive Probe (b) Active Probe

Figure 3. Kerrighed Probes

The probe layer implements system probes measuring for example the CPU or the memory use. This layer is the more complex to implement (*i.e.* operating system programming). There are two different kinds of probes: the passive probes and the active probes, and each probe can be linked with a set of local analyzers (a probe sends information to local analyzers). Active probes are regularly woken up by a system timer, whereas passive probes are woken up by a system event (see Figure 3). There are two different system events: Linux kernel events, and global scheduler events (in order to get local information). When a passive probe is woken up by a system event, the probe sends information about the entity to probe to the local analyzer.

For example, we can use an active probe to probe the CPU usage (the CPU is probed at regular time), whereas we can use a passive probe to probe the ping-pong of memory pages between two threads of a shared memory application (when a page arrives on a node, the probe is woken up to detect page ping-pong.).

To simplify the implementation of the global scheduler, a set of system probes is provided within Kerrighed OS: a memory probe, a CPU probe, and a probe to detect ping-pong of memory pages. Additional probes can be implemented by operating system programmers.

4.2.2 Local Analyzer

The second layer is the Local Analyzer (LA). This layer gets probe information, analyzes it and detects abnormal local system state. This layer is also in charge to send probe information to the global scheduler. A set of local analyzers run on each nodes (see Figure 4).

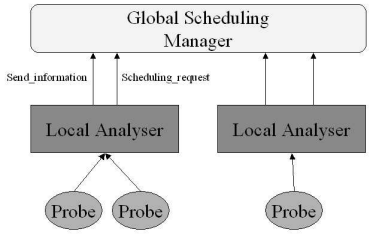


Figure 4. Local Analyzer

Each local analyzer can be linked with a set of probes. For example, if we have a probe for CPU usage and another for CPU temperature, a local analyzer linked with these two probes can detect local CPU high pressure, and local thermal problems. If a CPU problem is detected, the local analyzer sends a scheduling request to the global scheduling manager (a local analyzer has not a global vision of the cluster state and so cannot apply directly a process schedule).

4.2.3 Global Scheduling Manager

The third layer is the global scheduler. Global Scheduling Manager (GSM) is running on each node, and is linked with a set of local analyzers. Global scheduling manager executing on different node communicate together to exchange information on the node state. This layer is the only layer which has a global view of the cluster. This global view is made with the probe information and allows to detect global load unbalance (see Figure 5), and so each global scheduling manager implements a global scheduling policy. When a scheduling problem is detected, the global scheduling manager can decide to migrate some processes or to checkpoint an application, according to the scheduling policy, in order to have an efficient use of cluster resources.

The global scheduler can execute any dynamic scheduler policy, performing batch or preemptive scheduling.

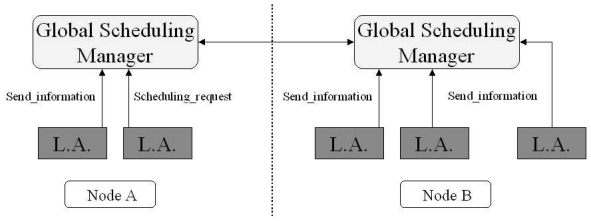


Figure 5. Global Scheduling Manager

4.2.4 Configuration and Development

All the layers of the global scheduler are implemented as Linux kernel modules and are managed by XML configuration files, which allow dynamic management. So, it is possible to dynamically load and unload probes, local analyzers and global scheduling managers. Moreover, each layer provides a development framework to simplify the programing of a new component. So, the Kerrighed development framework allows the OS programmer to simply create new global scheduling policies.

4.2.5 Example of a Global Scheduler in Kerrighed

To illustrate our how approach to global scheduling in Kerrighed, this section presents a simple example of a dynamic scheduler which balances only the CPU load. In this example, when the CPU usage is over 80%, some processes are migrated.

To implement this example of scheduler, the three following layers are needed: a probe which probes the CPU usage periodically, implemented as an active probe, a local analyzer sends a scheduling request to the global scheduling manager when high pressure is detected, a global scheduling manager which triggers a process migration when it receives a scheduling request.

Listing 1. XML configuration files

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<monitors>
 <timer_monitor>
  <file>cpu_probe</file>
  <name>probe1</name>
  <time>5</time>
 </timer_monitor>
</monitors>

<?xml version="1.0" encoding="ISO-8859-1"?>
<primitives>
 <primitive>
  <file>cpu_local_analyzer</file>
  <name>cpu_analyzer</name>
  <monitors>
   <monitor>probe1</monitor>
  </monitors>
 </primitive>
</primitives>

<?xml version="1.0" encoding="ISO-8859-1"?>
<schedulers>
 <scheduler>
  <nodes>all</nodes>
  <name>scheduler1</name>
  <file>cpu_scheduling_manager</file>
  <primitives>
   <primitive>
    <name>cpu_analyzer</name>
    <global_infos>cpu_table</global_infos>
    </primitive>
   </primitives>
 </scheduler>
</schedulers>
```

The XML files presented in Listing 1 defines the three components of the considered scheduler. These three files are stored on each cluster node. The cpu probe named **probe1**, which is woken up each five seconds is first declared. Then, an associated local analyzer named **cpu_analyzer** is declared. Finally, the global scheduling manager associated with the **cpu_analyzer**, which is running on all the cluster nodes is declared.

The CPU Probe In this example, we use the Kerrighed CPU probe, so we do not describe the probe implementation here. This probe is an active probe (the probe is woken up periodically).

The Local Analyzer The local analyzer needs to receive the probe information and if the CPU usage is higher than 80%, the local analyzer sends a migration request to the global scheduler. By default, each probe information sent to a local analyzer are automatically send to the associated global scheduling manager.

```
int cpu_local_analyzer (void *arg)
{
/* informations send by the probe */
weight_t weight;
if (PROBE_WEIGHT > 80){
  /* migration request sends */
  send_scheduling_request (
    ANALYZER_ID, process_to_migrate);
  }
return 0;
}
```

We can see in the local analyzer code than some system information is available. The macro ANALYZER_ID gives the unique identifier of the local analyzer, and the probe information is available using the macro PROBE_WEIGHT.

Global Scheduling Manager When a global scheduling manager receives probe information from a local analyzer, the value is automatically saved in the associated table (**cpu_table** in the example), and sent to other node (the nodes list is defined in the XML file).

```
int cpu_scheduling_manager (void *arg)
{
pid_t pid_of_process_to_migrate ;
node_id_t node_id;
 pid_of_process_to_migrate =
   find_a_kerrighed_process ();
node_id = find_a_node_for_migration ();
do_process_migration (tsk , node_id);
return 0;
}
```

The function *find_a_kerrighed_process* just finds the first Kerrighed process running locally whereas

find_a_node_for_migration finds the node with the lower CPU load, according to the table *cpu_table*.

4.3 Kerrighed Process Management

Kerrighed schedulers are based on three mechanisms: process placement, process migration and process checkpoint/restart. For process placement, two mechanisms are provided in Kerrighed. First, processes can be created at the creation time using a distributed fork interface. Second, processes can be created during the application execution and so, the system needs to extract an image of the process and to transfer it on a remote node in order to create a running clone. Process migration needs to extract a process image and to transfer it on a remote node in order to create a running clone (like in the process placement), but then the initial process is stopped. Process checkpoint needs to extract a process image and then to store it on disk or in remote memory. So, these mechanisms use the same underlying mechanism: the process extraction (see Figure 6).

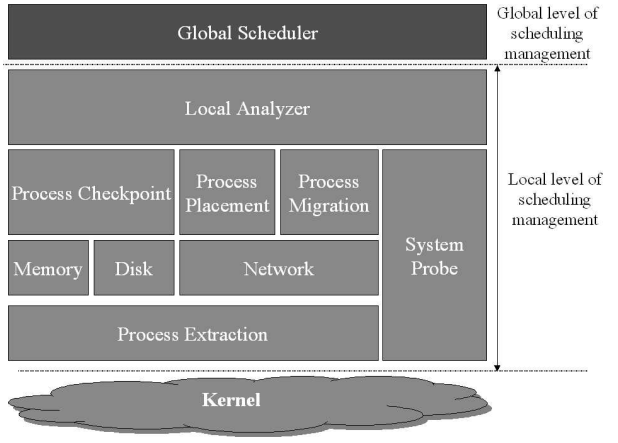


Figure 6. Global Scheduler Architecture

The process extraction consists in creating a **ghost process**. Some kernel information about the process is extracted. First, we have to extract the process address space, second we have to manage the process open files, third we have to manage the Process IDentifier (**PID**), then the value of the processor registers associated to the process, and finally the signals associated to the process.

4.3.1 Management of the Process Address Space

In a standard Linux kernel, all the process memory information needs to be extracted in order to create a coherent ghost process. In the Kerrighed OS, a K-process is linked to containers. Containers allow physical memory pages of a process to be accessed anywhere in the cluster. So, for process migration, memory pages do not need to be extracted. New links with containers are just created after the process transfer, and then memory pages are migrated on demand by the container mechanism during the process execution. So, for process extraction, the container mechanism eases the creation of the ghost process: only container information needs to be extracted instead the whole process memory space.

4.3.2 Management of Process Open Files

In a standard Linux kernel, like for the extraction of the memory segments, all the process file information needs to be extracted in order to create a coherent ghost process. The information to extract is very complex to manage. First, kernel structures need to be extracted: *inode*, *dentry* and *files* structures. Second, two linked list need to be managed: the list of opened files in the system and the list of opened files by the process. In the Kerrighed OS, a process can be linked to containers. In this case, containers allow to access remote files. So, to create the ghost process, the process extraction is simpler: new links with containers are just created and file lists are updated after the process transfer.

So, for process extraction, like for the extraction of the memory segments, the container mechanism simplifies the creation of the ghost process: containers provide transparent access to remote file, allowing to transparently migrate a process which access files.

4.3.3 Management of the process PID

In a standard Linux kernel, threads are implemented by processes and by the **pthread** library. So, processes are identified by a kernel unique identifier: the Process IDentifier (**PID**). Threads are identified by internal identifiers in the **pthread** library.

The Kerrighed OS implements a **pthread** interface which extends the kernel process informations. To identify running threads, the system needs a unique identifier: the Kerrighed Process IDentifier (**KPID**) The **KPID** of a K-process is composed of the Linux process PID created for executing the K-process, of the current node identifier, and of a thread identifier (which is by default zero). The Kerrighed **pthread** interface is implemented using a master thread: the first created thread is the master thread and centralizes the information about other threads belonging to the same application. The thread identifier of the master thread is zero, and for each new thread, this value is incremented.

A thread manager is running on each node in Kerrighed OS. This thread manager allows to communicate with all the threads running in the cluster. For example, when the system has to execute a function on each thread, a request is broadcast to all the thread managers of the cluster which

receive the request, find the local threads associated with the **KPID**, and then apply the function. This mechanism allows to manipulate threads (thread kill for example), to globally manage the threads states.

4.3.4 Extraction of the value of the processor registers

The values of the processor registers associated to each process are not available at any time in the system. In the Linux kernel, these values are available during an exception return, an interruption return, or after a system call return. So, we have created in the Linux kernel a new state to start Kerrighed mechanisms like process migration. This kernel modification is very light (just a few lines of code) and very similar to the kernel state for the signal treatment. If a process is marked as needing a Kerrighed mechanism, such as migration (like marked for a signal treatment), the kernel is hijack to the Kerrighed process management module.

4.3.5 Signal Management

Another important feature for process execution is signals. How a signal can be sent to a migrated process, and how a migrated process receives a signal? So, a distributed mechanism to manage signals is needed.

Our approach is to create a distributed kernel service for signal management. All nodes execute a kernel thread, the signal manager. Signal managers have a local table in order to localize migrated processes which were previously running on the local node, and to identify migrated processes which are locally running. So, the signal managers have two goals: send signals concerning a migrated process to the destination node if the destination process is not on the current node, and receive signals sent by remote processes and send these signal to the local destination processes.

5 Evaluation

A preliminary evaluation of the process management mechanisms used by the scheduler has been performed.

We have used a two node platform, each node being a Pentium II, 233 MHz with 128 MB of memory. The two nodes are interconnected by a 100Mb/s Ethernet network. Nodes communicate using the TCP protocol. For our experiments, we have considered the migration of a single mono-thread process executing a sequential version of the MGS algorithm. Different sizes of the MGS matrix have been used. Only one migration operation is activated (using the migration system call) during the process execution. Three different times have been chosen for triggering the migration: (i) one second after the beginning of the application, (ii) exactly in the middle of the application execution, (iii) three seconds before the end of the application execution. This allows us to quantify the impact of the application size on the migration performance.

First, we have evaluated the time needed to transfer the process state from the source node to the destination node. This time is measured as the time between the instant when the process is suspended on its source node and the time when it is ready to restart its execution on the destination node. For this set of experiments, the memory segments of the Kerrighed process are linked to containers when the process is created. In Kerrighed, the migration time is constant and equals 13.5 ms in average. Indeed, pages of the process address space are migrated on demand, once the process has resumed its execution on the destination node.

Secondly, we have measured the overhead due to migration on the process execution time. This overhead comprises the process transfer time and the overhead on the execution time of the migrated process on the destination node. Indeed, once a process is migrated, memory pages in containers need to be migrated to the destination node on demand. Thus, the execution time of the process on its destination node is different from what it would have been on its source node if it had not been migrated. The overhead is calculated as the difference between the total application execution time obtained without migration and the total execution time obtained when the process migrates once during its execution. For this set of experiments, the Kerrighed process is created as a standard Linux process. The process address space is linked to containers when the process migrates for the first time. Thus, migration times provided in the following of this section for Kerrighed include the time needed to create containers for the application memory segments. Table 1 presents the migrating process execution time in Kerrighed and Figure 7 shows the migration overhead on the process execution time. The process execution time increases with the matrix size: the bigger the matrix is, the more pages need to be migrated. Even if the time needed to transfer pages increases with the matrix size, the migration overhead does not increase too. Figure 7 shows that the higher the computation time of the application is, the lower the migration overhead is[1]. Moreover, it shows that the migration overhead in Kerrighed is between 0.5 and 7 percents of the execution time. The less the pages on the remote node are accessed, the less the overhead is.

6 Conclusion

Kerrighed offers a configurable global dynamic scheduler which allows to adapt the global scheduling policy to the workload to be executed on a given cluster. Moreover, Kerrighed provides a development framework which eases the implementation of new global schedulers. These

[1] A cache effect explains the result obtained for a 500x500 matrix and a migration 3 seconds before the end of the application.

	150x150	300x300	500x500	750x750	1000x1000
Without migration	0,3149	3,329	16,3621	55,8434	133,17
Migration after 1 second of execution	0,3254	3,377	16,4636	56,0096	133,625
Migration exactly in the middle of the execution	0,334	3,4404	16,7976	57,0886	135,415
Migration 3 seconds before the end of execution	0,3341	3,4532	17,3988	58,0887	136,301

Table 1. Execution time of a process migrating once in Kerrighed

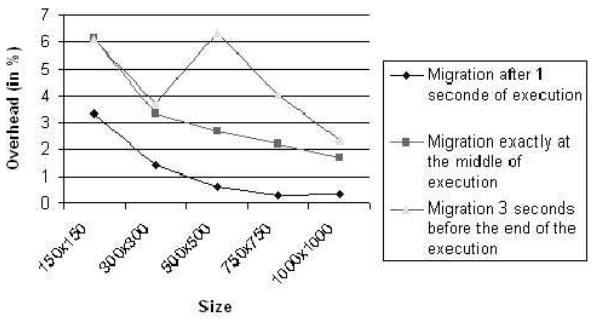


Figure 7. Overhead of a process migration on the application execution time in Kerrighed

schedulers may rely on new components, developed without any kernel modification or based on components existing in Kerrighed. Various kinds of dynamic scheduling policy can be implemented in Kerrighed. All the components of a global scheduler in Kerrighed can be hot-plugged or hot-stopped.

The global scheduler relies on efficient process management mechanisms allowing to deploy application processes on the cluster nodes and to migrate or checkpoint them. Based on the Kerrighed containers concept the scheduling policies can act not only on individual processes but also on processes sharing memory.

We plan to evaluate the Kerrighed configurable scheduler with various dynamic scheduling policies and realistic workloads. Our future work includes global management of data streams (pipe, sockets, ...etc.) to support parallel applications using message passing. It will also be interesting to study scheduling policies for such applications.

References

[1] M. H. A.M. Goscinski and J. Silock. Genesis : The operating system managing parallelism and providing single system image on cluster. Technical Report TR C00/03, School of Computing and Mathematics, Deakin University, February 2000.

[2] O. t. Amnon BARAK and A. SHILOH. Scalable cluster computing with MOSIX for LINUX. In *Proc. Linux Expo '99*, pages 95–100, May 1999.

[3] B. Bode, D. M. Halstead, R. Kendall, and Z. Lei. The portable batch scheduler and the Maui scheduler on Linux clusters. In *4th Annual Linux Showcase and Conference*, October 2000.

[4] F. Douglis. *Transparent Process Migration in the Sprite Operating System*. PhD thesis, Computer Science Division, Department of Electrical Engineering and Computer Sciences, University of California, Berkeley, California 94720, September 1990.

[5] R. L. Henderson. Job scheduling under the portable batch system. In D. G. Feitelson and L. Rudolph, editors, *Job Scheduling Strategies for Parallel Processing*, pages 279–294. Springer-Verlag, 1995. Lecture Notes in Computer Science vol. 949.

[6] E. Pinheiro and R. Bianchini. Nomad: A scalable operating system for clusters of uni and multiprocessors. In *Proceedings of the 1st IEEE International Workshop on Cluster Computing*, page 5570, December 1999.

[7] R.Lottiaux and C.Morin. Containers : A sound basis for a true single system image. In *Proceeding of IEEE International Symposium on Cluster Computing and the Grid*, pages 66–73, May 2001.

[8] T. Sterling, D. Savarese, D. J. Becker, J. E. Dorband, U. A. Ranawake, and C. V. Packer. BEOWULF: A parallel workstation for scientifi c computation. In *Proceedings of the 24th International Conference on Parallel Processing*, pages I:11–14, Oconomowoc, WI, 1995.

[9] V. S. Sunderam. PVM: A framework for parallel distributed computing concurrency. In *Practice and Experience*, volume 2, pages 315–339, Dec 1990.

[10] T. Takahash, F. O'Carroll, H. Tezuka, A. Hori, S. Sumimoto, H. Harada, Y. Ishikawa, and P. Beckman. Implementation and evaluation of MPI on an SMP cluster. In *Paralel and Distributed Processing. IPPS/SPDP'99 Workshops*, volume 1586 of *Lecture Notes in Computer Science. Springer-Verlag*, April 1999.