HAL
archives-ouvertes.fr

Containers : A Sound Basis For a True Single System Image

Renaud Lottiaux, Christine Morin

▶ **To cite this version:**

Renaud Lottiaux, Christine Morin. Containers : A Sound Basis For a True Single System Image. [Research Report] RR-4085, INRIA. 2000. inria-00072548

HAL Id: inria-00072548
https://hal.inria.fr/inria-00072548

Submitted on 24 May 2006

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Containers : A Sound Basis For a True Single System Image

Renaud Lottiaux, Christine Morin

N°4085

Novembre 2000

THÈME 1

ISSN 0249-6399 ISRN INRIA/RR--4085--FR+ENG

Containers : A Sound Basis For a True Single System Image

Renaud Lottiaux*, Christine Morin*

Thème 1 — Réseaux et systèmes
Projet PARIS

Rapport de recherche n°4085 — Novembre 2000 — 19 pages

Abstract: Clusters of SMPs are attractive for executing shared memory parallel applications but reconciling high performance and ease of programming remains an open issue. A possible approach is to provide an efficient Single System Image (SSI) operating system giving the illusion of an SMP machine. In this paper, we introduce the concept of container as a mechanism to unify global resource management at the lowest operating system level. Higher level operating system services such as virtual memory system and file cache can be easily implemented based on containers and transparently take benefit of the whole memory resource available in the cluster.

Key-words: Distributed operating system, cluster, distributed ressource management, single system image

(Résumé : tsvp)

* IRISA / Université de Rennes 1, {rlottiau,cmorin}@irisa.fr

Unité de recherche INRIA Rennes
IRISA, Campus universitaire de Beaulieu, 35042 RENNES Cedex (France)
Téléphone : 02 99 84 71 00 - International : +33 2 99 84 71 00
Télécopie : 02 99 84 71 71 - International : +33 2 99 84 71 71

Conteneurs : un outil pour un véritable système à image unique

Résumé : Les grappes de multiprocesseurs sont des architectures attrayantes pour l'exécution d'applications parallèles utilisant un modèle de programmation par mémoire partagée. Cependant, concilier haute performance et simplicité de programmation reste un problème difficile. Les systèmes à image unique sont une solution pour offrir l'illusion d'une machine unique à mémoire partagée au dessus d'une grappe de calculateurs. Cependant, aucun système n'a encore permis une véritable gestion globale de *toutes* les ressources d'une grappe. Dans cet article, nous présentons le concept de *conteneur* comme un mécanisme de bas niveau permettant d'unifier la gestion des ressources au sein d'un système d'exploitation distribué. Les services systèmes de haut niveau, tels que les mécanismes de mémoire virtuelle, de gestion de fichiers parallèles, ou de cache coopératif peuvent facilement être mis en œuvre au dessus des conteneurs, bénéficiant ainsi d'une utilisation transparente de toute la ressource mémoire de la grappe.

Mots-clé : Système d'exploitation distribué, grappe de multiprocesseurs, gestion de ressources distribuées, système à image unique

1 Introduction

It is now clear that clusters of SMP's have the performance potential to be used as a cheap parallel machine. However, the lack of dedicated software makes clusters difficult to use. The management of resources (memories, disks, processors) which are distributed through the cluster nodes is a burden to the programmer. An efficient management of these resources is difficult and introduces a significant overhead in term of development time and decreases the software reliability.

In order to ease cluster programming, several solutions have been proposed such as shared virtual memory [13, 1], thread migration [14, 3] or parallel and distributed file systems [9, 2]. But the integration of all these mechanisms in a realistic system to offer the view of an unique machine, a Single System Image (SSI), remains an open issue.

The work presented in this paper relates to containers. A container consists of a set of pages that may be located in any node of the cluster and may be shared by different threads whatever their location. Higher level operating system services such as the virtual memory system, a shared virtual memory system, a cooperative caching system, thread migration or parallel file system can be easily implemented relying on containers. Moreover these mechanisms can take benefit of containers efficiency and high availability. Finally, containers can easily be implemented as an extension of an existing kernel by modifying a few low level resource management functions.

The remainder of this paper is organized as follows. In Section 2, we give an overview of previous works for global resource management. In Section 3, we introduce the container concept. In Section 4, we describe the design of containers and their use for the implementation of high level operating system services. Section 5 details the implementation of containers. Section 6 presents preliminary performance evaluation and Section 7 concludes.

2 Single System Image : Background

In a cluster of SMP's, physical resources such as memories, disks and processors are distributed and cannot easily be shared by the different nodes without a good software support. To make all these resources available from any node regardless of their physical attachment, numerous software mechanisms have been studied such as distributed shared memory, distributed and/or parallel file system and global thread scheduling. On the other hand, the multiplication of independent off-the-shelf components increases the risk of a node failure in the cluster. To solve this problem, fault tolerance mechanisms and high availability solutions have been proposed.

2.1 Distributed Memory Management

Global memory management is a key feature to make the cluster appear as a shared memory multiprocessor. Existing user level software Distributed Shared Memory (DSM) [13, 1] are not sufficient to offer the view of a shared memory multiprocessor. Indeed, they only allow several threads of an application to share memory areas as specified by the programmer. However, they generally support to a limited extent the concurrent execution of multiple applications and inter-application data sharing. Moreover, interactions between software DSM and file systems are not considered.

Shasta [20] and Blizzard-S [21] offer a COMA like fine grain shared memory, enabling transparent execution of unmodified or lightly modified multiprocessor applications. They are implemented inside an operating system kernel. However, they do not offer a full single system image. Creation of shared memory regions passes through explicit memory allocation functions. Moreover, there is no support for (parallel) file access, swapping, thread migration, and fault tolerance. Finally, the system is limited to a single user. To fill these gaps, a deep modification is needed since shared memory is only supported through virtual memory. Sharing data between several processes of several users, caching (parallel) file data and enabling swap would require a deep operating system memory management modification.

A low level memory management system has been proposed in GMS [8] to use memory of remote machines. GMS is a system designed for cluster of workstations to enable a machine to inject pages in the memory of remote machines. This mechanism is implemented at the lowest operating system level, allowing the injection of any kind of page: text, data, and so on. However, neither distributed shared memory nor task migration are offered, which is in contradiction with SSI needs.

2.2 Global Disk Management

To cope with the performance gap between processor speed and disk latency, a lot of work has been done on Parallel File Systems (PFS) [9, 2]. In a PFS, file data is split in sets of blocks stored on several disks, increasing the overall disk bandwidth by enabling read and write operations in parallel. The use of caches in PFS has been extensively studied [12, 6]. A significant performance improvement can be obtained, specially with *cooperative caching* [6, 5]. Thanks to a global management of local caches on client nodes, cooperative caching increases the overall file system performance by increasing the global cache hit rate.

The design of cooperative caching faces the problem of local memory management and data coherence. The fixed size of file system caches does not allow to optimize the overall

memory usage by taking into account dynamic variation in applications memory needs and file accesses. Hence, an application requiring a lot of memory is not able to use the memory allocated to the file system cache to avoid swapping. Conversely, a file system cache located on an idle node can not be extended to use the available memory. Moreover, replication of cache data in different nodes raises a coherence issue, leading to the introduction of extra coherence mechanisms.

2.3 Global Processor Management

To achieve global processor management, a number of mechanisms [14, 3] have been proposed such as process transparent remote execution, thread or process migration and load balancing. Mosix [3] is probably one of the most advanced systems offering global processor management. A Mosix cluster is a set of computers sharing processor resources. When a new process is launched from a node, Mosix chooses a lightly loaded node (in term of processor load and memory usage) to execute it. At execution time, load balancing is performed to maintain a high processor usage level and decrease the overall process execution time. Mosix also takes care of the memory usage. If the memory of the execution node of a process becomes overloaded, Mosix can migrate the process to a less loaded node. However, Mosix has several drawbacks to fully offering an SSI. First, it does not offer shared memory, limiting Mosix applications to sequential one or to threads executing on the same node. Moreover, a process accessing a local file should still be able to access this file after a migration. Hence, file accesses performed by migrated processes are performed through a deputy kept on the home node. This induces an extra software and performance overhead. Finally, Mosix does not offer a native parallel or distributed file system virtualizing distributed disks.

2.4 High Availability

A cluster being made up of a large number of standard machines, the probability of a node failure is not negligible. Several recoverable distributed shared memory systems (RDSM) that implement a backward error recovery strategy have been studied [17]. Many RDSM are based on a global consistent checkpoint approach to guarantee that the set of communicating processes checkpoints forms a consistent system state.

Checkpoints need to be saved in stable storage to ensure that they remain accessible and are not altered by a failure. To overcome the performance issue of disk implemented stable storages, a few RDSM builds a stable storage using the volatile memory of cluster failure-independent nodes [11]. A redundancy mechanism based on parity as in RAID has been proposed to tolerate a node failure in a remote paging system [15]. Concerning the

disk resource, fault tolerance and reconfiguration mechanisms have been implemented in XFS [2]. A lot of work has being carried out on the implementation of checkpointing strategies in distributed systems [7]. However, node failures are not tolerated in Mosix nor in Sprite that implement process migration for global processor management. In those systems that keep residual dependency on the process creation node, both a failure of the creation node and the failure of the current execution node of a migrated process result in the process failure.

2.5 Single System Image : Issues

Mechanisms proposed to globally manage one specific device rely on complex software systems which focus on the management of the given device with no or low consideration to other resources. An SSI offering global management of all resources using these previous works is not reasonable due to the unmaintainable overall software complexity. Moreover, each sub-system designed to globally manage a specific device can take conflicting decisions leading to poor performance.

To design an efficient and fully operational SSI, a new approach to resource management is needed. To avoid mechanism redundancy, conflicting decisions and decrease the software complexity, resource management should be built in a unified and integrated way. To achieve this property, we have proposed the concept of container, which offers a generic mechanism to design and implement high level distributed resource management systems.

3 Containers

In existing operating systems designed for SMP machines, memory management is the core of the system. It ensures two main goals: enabling processors to share data through memory and storing data coming from disk. Physical memory is organized as a set of page frames which can be allocated, deallocated or swapped to disk. This physical memory is used as a page cache for the implementation of thread virtual memory and file cache.

The lack of shared physical memory in a cluster makes it difficult to implement services offered by a traditional operating system. In order to share physical memory between nodes in a cluster, we introduce the concept of *container*, described in the remainder of this section.

3.1 Concept

A container is a virtual entity consisting of a sequence of bytes structured as a set of pages. It is used as a low level mechanism to store and share data between several nodes in a cluster.

A container can be viewed as the extension to a cluster scale of the concept of memory segment. The key point is that a container is not just a kernel level mechanism used to shared data for user level applications but to give the illusion to the kernel that resources are shared. Since the kernel views distributed resources as unique and shared, offering high level distributed services (such as distributed shared memory, cooperative caching, etc) is possible without designing new mechanisms and with a few modification to an existing kernel.

In modern operating systems, memory, disk and even network (socket) management relies on page based mechanisms. All these mechanisms use a few number of basic functions such as page allocation, page mapping, page copy on write, page read from disk, etc. All operating system services are built on top of these functions. We use these functions to make the link between operating system and containers. Containers are then used as an interface between operating system services and distributed physical resources such as memories, disks and block devices (see fig. 1). Each access from the OS to a device passes through kernel low level functions which use containers to virtualize distributed resources. Then, the operating system does not distinguish a standard page frame from a page frame coming from a container. All operations that can be applied to standard page frames can also be applied to page frames belonging to a container.

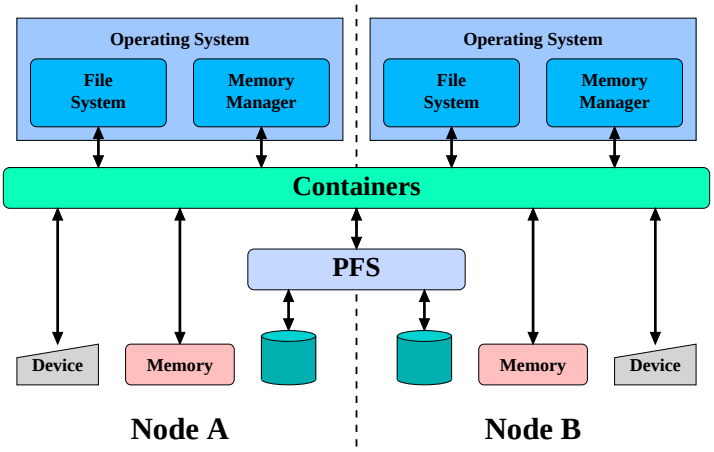

Figure 1: *Containers*

Containers implement a software COMA-like memory management at the operating system level. Similarly to COMA architectures, containers use local memories as caches

but with a memory page management granularity. They ensure the migration or replication of pages between nodes at physical memory level.

We have defined three kinds of containers: *volatile*, *file* or *linked*. A *volatile* container holds temporary data which is not linked to any device. These containers generally have a lifetime equal to the lifetime of applications using them. They exactly behave as a standard software DSM. *File* containers are associated to parallel files. They contain data coming from parallel files, allowing to access them through physical memory, thanks to container handling functions. They allow to map parallel files in DSM, greatly easing their access and use. They can survive to the death of applications using them, then behaving as cooperative parallel file caches. Finally, *linked* containers are associated to devices (disk, CD-Rom reader, etc.) located on a given node. They make it possible to any node to access a device located on a remote node through physical memory. They can also survive to the death of applications using them.

When a page of a container is referenced, it is placed in physical memory inside a page frame, its address being returned to the operating system. Thus, a node physical memory is used like a container page cache. Four functions are provided for accessing this page cache: *FindPage* checks if a page is present in local memory, *GetPage* puts a copy of a page in local memory, *GrabPage* allows write access to a page and *FlushPage* evicts a page from a local memory.

To access a page, the operating system should first check if the page is present in the cache thanks to the *FindPage* function. When this function is called, the physical memory is scanned to check the presence of the requested page. If the page is present in memory, its physical address is returned to the operating system, or else an exception is triggered otherwise. In this last case, the *GetPage* function is used to obtain a copy of the page. When this function is called, a copy of the requested page coming from a node owning the page is placed in local physical memory. If there is no copy in the cluster, the page is created, loaded from disk or from a remote device, depending on the container type.

The containers semantic imposes to use the *GrabPage* function before writing to a page. This function ensures that the copy present in local memory is the only one existing in the cluster, allowing writes without any coherence problem. When this function is called, every existing copy in the cluster is invalidated.

When there is no more free page frames in a node memory, the operating system replacement algorithm chooses a page to evict and calls the *FlushPage* function. If there exists other copies of the selected page in the cluster, the local copy is simply discarded. If this copy is the only one in the cluster, it is injected into the memory of a remote node. If there is no memory space left in the cluster and the chosen page belongs to a *volatile* con-

tainer, the page is discarded after being written to disk. Finally, if the chosen page belongs to a *file* or *linked* container, it is discarded after updating the device copy if needed.

3.2 Advantages

The use of containers to globally manage memory in a cluster offers many advantages. By virtualizing resources at a very low level, all services offered by a modern operating system can be offered cluster wide without adding any new mechanisms and with just a few kernel modifications.

Containers offer a generic mechanism for the implementation of shared memory mechanisms, file mapping and cooperative file caching. By unifying all these mechanisms within a single system, software complexity is decreased, increasing the system robustness.

The use of containers to manage the file cache makes it possible to obtain an efficient cooperative cache without doing a dedicated implementation. The management of file cache data is ensured by the container management mechanism. This mechanism ensuring the management of every data located in memory, it is possible to offer the best possible balance between the amount of cache data and volatile data. The system automatically adjusts the balance between several types of data according to applications needs.

Thread migration raises the problem of remote accesses to files located on a particular node disk. If a process which has opened a file located on a disk local to its execution node is to be migrated, the system must ensure remote access to this file. This implies the implementation of complex mechanisms linking a migrated process and the file opened on a remote disk. The use of containers to manage the file cache solves this problem without implementing extra mechanisms. Thanks to containers, the file cache is accessible from any node in the cluster. Thus, a migrated process can transparently access a remote disk by simply accessing the global file cache.

Finally, the introduction of quality of service in containers, such as fault tolerance [16] or high availability benefits to all the operating system services using containers. The robustness of containers ensures the robustness of the whole system. By focusing the design and development effort on containers, we simplify the overall system architecture and easily offer all mechanisms needed by an SSI.

4 Design

Containers have been implemented in Gobelins operating system. We describe in this section the design of containers in Gobelins and how we use them to build high level services.

4.1 The Gobelins Operating System

Gobelins is an SSI operating system designed for cluster of PCs. We use a global and integrated management of disks, memories and processors. Gobelins offers a shared memory programming model. A parallel application is executed by a *g-process*, which contains a number of threads called *g-threads*. The execution of a *g-process* is distributed among the cluster nodes, its *g-threads* being automatically migrated to idle nodes to increase parallelism. This programming model is the one provided by an operating system designed for SMP machines. With Gobelins, we extend this model to clusters of SMPs.

To offer a shared memory multiprocessor view, Gobelins extends traditional operating system memory management to a cluster scale. Distributed resources are managed through a Distributed Shared Memory (DSM), a Parallel File System and a thread migration mechanism. These subsystems are integrated to avoid redundancy of common mechanisms. This integration also avoids to take conflicting decisions in the system. For example, taking into account both memory and processor load in a thread migration mechanism can avoid a negative impact of a thread migration on the memory system. Finally, information known by a subsystem can be used to optimize another subsystem.

The address space of each *g-process* is divided in several segments called Shared Memory Areas (SMA). As segments rely on physical memory in traditional operating systems, SMAs rely on *containers*. SMAs are shared between *g-threads* of the same *g-process*. Thus, *g-threads* can share data through SMAs. Moreover, several *g-processes* can share memory through a special SMA mapped in the virtual address space of these *g-processes* extending mechanisms such as *IPC system V* segments to a cluster scale.

Gobelins is composed of a standard UNIX file system and a Parallel File System. These systems are implemented on top of a cooperative cache relying on containers. The implementation of a PFS and a DSM in the operating system has led us to design a single level storage system with file mapping as a basic file system interface. When a file is mapped in memory, an access to this memory area behaves as if it has been done to the corresponding file area. By suppressing the traditional *read/write* file system interface, the programmer task can be significantly eased specially for the design of out-of-core applications (i.e. applications working on a data set greater than the available memory).

4.2 Virtual Memory Design

In the Gobelins operating system, memory sharing between nodes is provided through SMAs. Each SMA is a container mapped in the virtual address space of a *g-process* (see figure 2). Each page of the *g-process* address space is associated to a page of the corresponding container. The link between a page frame holding container data and a virtual

page is carried out by the operating system virtual memory manager. However, when a page referenced by a *g-thread* is not present in the physical memory of its execution node, the local node contacts the corresponding container manager to obtain a copy of the page.

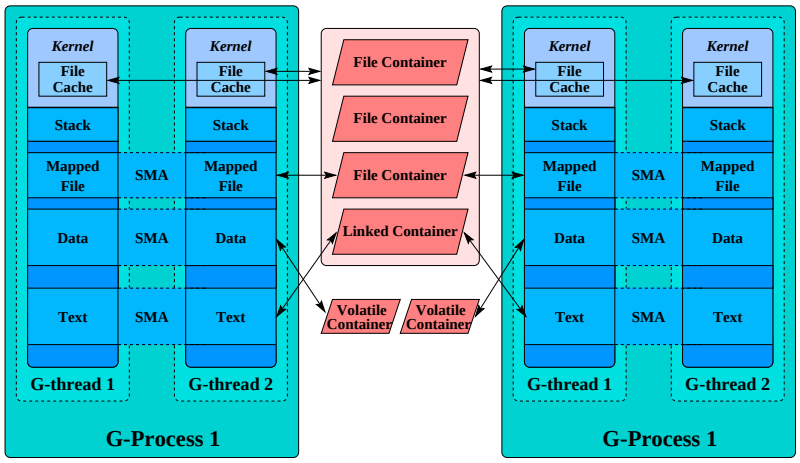

Figure 2: *Global view of the Gobelins memory management (here, two instances of the same application)*

The *G-process* data SMA is associated to a *volatile* container. This SMA behaves as a traditional software DSM. Memory sharing between several applications is possible, since a container can be mapped in the address space of several *g-processes*. In this case, each *g-process* sharing a memory area holds in its virtual memory an SMA linked to a shared container.

4.3 File System Cache

Containers are also used by the operating system to share kernel data between several nodes, such as file cache data. In modern operating systems, this file cache is structured as a set of pages. The basic unit of file management is the memory page. Each file access passes through the file cache, which caches pages from both mapped files and files accessed through standard *read/write* calls. Thus, it is possible to use containers to implement this cache. As a node can use data cached by another node through containers, the cache hit rate is increased. Moreover, cache data selected by a local replacement algorithm can be kept in global memory if it can be injected into the memory of a remote node. This file cache is also used to cache data from PFS files. Each data coming from disk, from standard UNIX files

or parallel files, is stored in the file cache. The key point is that no cooperative file cache is implemented in the PFS. Caching is entirely managed by the underlying containers.

4.4 Parallel File Mapping

Parallel files and UNIX files can be mapped in the virtual memory of *g-processes*. For each mapped file, there is a dedicated SMA in the virtual address space of the *g-processes* mapping the file. For instance, the text SMA is associated to a container *linked* to the file holding the application code. A mapping SMA is associated to a *file* container associated to a parallel file. Containers used to map files are also part of the operating system file cache.

5 Implementation Issues

Implementation of Gobelins has been carried out using Linux. Our experimentation platform is a 28 dual-processor PCs cluster interconnected with 100 Mb Ethernet and Gigabit Ethernet. Gobelins is implemented at kernel level without modifying the Linux core kernel. Gobelins functionalities are added using *kernel modules* [19].

5.1 Overview of the Implementation

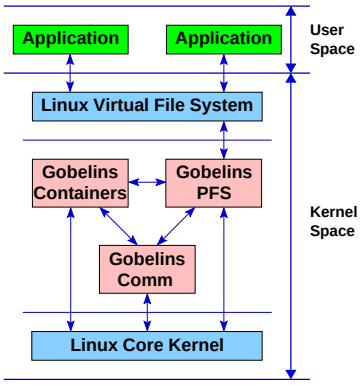

Figure 3: *Integration into Linux*

The actual implementation consists of 3 modules. A communication layer has been implemented on top of Ethernet network adapters. It offers high level communication primitives like *send*, *receive* and *active messages* at the kernel level. This library has been

developed on top of a modified version of Gamma [4], a low level Ethernet communication library offering very low latency and high bandwidth.

A module implements *containers*. The container system consists of 3 components: a *container manager*, a *page manager*, and a *page server*. The *container manager* implements the interface used by the operating system services. It receives requests from the kernel and forwards them to the *page manager*. The *page manager* manages a table containing the owner of each page, i.e. the node which was the last one to write the page. The *page server* handles requests from remote nodes. It sends copy of local pages or invalidates local page copies.

Finally, the parallel file system is a new Linux file system, usable like any other file system. On each node, the PFS is composed of a *PFS manager*, a *PFS server* and a *disk manager*. The *PFS manager* deals with a list of open files and their corresponding mapping address. It makes the correspondence between a page number in a container and its disk location. The *PFS server* handles requests from remote nodes. The *disk manager* accesses data of the local disk.

5.2 Basic Data Structures

To manage containers, the table *containerList* is replicated on each node. Each entry of this table contains information on a given container: container identifier, size, type, identifier of the linked parallel file or device, location of this device and list of *g-threads* sharing the container on the local machine.

For each container, we also use two tables to store state and location of pages in the cluster. The first one is a statically distributed table called *pageInfo* used to locate page copies. This table contains for each page the location of its master copy (i.e. the last modified copy). A second table called *ctnrPageTable* exists on each node. It contains for each page present locally, its coherence state (*invalid*, *read* or *read/write*) to implement a write invalidate coherence protocol and the list of existing copies in the machine (if the page is the master copy).

5.3 Implementation of Container Manipulation Functions

When the *findPage* function is called, the *container manager* looks up in local tables if the requested page is present in local memory. If the page is present, its physical address is returned. Otherwise, a null pointer is returned.

When the *GetPage* function is called, the *container manager* sends a page copy request to the concerned *page manager*. This one determines if there is a copy of the requested page in the memory of a node. If there exits a copy in a remote node, the request is forwarded to

the *page server* of this node, which sends back a copy to the requesting node. If there is no existing copy in the cluster, several actions are possible according to the container type. If the page belongs to a *volatile* container, a request is sent back to the requesting node, which creates a new empty page. If the page belongs to a *file* container, a request is sent to the PFS manager which loads the page in memory and sends it to the requesting node. Lastly, if the page belongs to a *linked* container, a request is sent to the *page server* of the node on which the linked device is located. Data is retrieved from the device and sent to the requesting node.

When the *GrabPage* function is called, the *container manager* sends an invalidation request to the *page manager* which forwards it to the owner of the page. This one invalidates its copy in memory and forwards the invalidation request to each node holding a copy. On these nodes, the page is discarded from physical memory and removed from the virtual address space of each *g-threads* attached to the corresponding container. If the page belongs to a *file* or *linked* container, the corresponding entry in file system cache data structure is invalidated.

5.4 Implementation of SMA

The virtual address space of a Linux process is split in several segments called *Virtual Memory Area* (VMA) and managed by a set of functions which can be replaced by programmer specific functions. SMAs are implemented by modifying these functions and adding a link to a container. Functions diverted by *Gobelins* are *no_page*, called when a page is touched for the first time and *wp_page*, called to perform a copy on write. Our functions are very simple. They mainly consist of calls to appropriate container functions and manipulations of the *g-process* page table.

When a page fault occurs in an SMA, specific *no_page* or *wp_page* functions are called through VMA manipulation functions. If the fault is a read fault, these functions check the presence of the requested page in local memory thanks to *findPage*. If the page is not present, the *getPage* function brings a copy back in local memory. This page is then inserted in the virtual address space of the faulting process by the Linux memory manager. If the fault is a write fault, the *grabPage* function is used to invalidate remote copies and the page access right is changed to write.

6 Performance Evaluation

We present in this section a preliminary performance evaluation of containers. This evaluation focuses on the use of containers as a mechanism to share memory between nodes. We

have used a parallel application designed for an SMP machine and compared performance results obtained on a 4 ways SMP machine to those obtained with a 4 nodes cluster running Gobelins. On the SMP machine, data is shared between thread through physical main memory, while on the cluster data is shared between threads through memory thanks to the underlying use of containers.

The cluster consists of 4 nodes based on Intel Pentium Pro (200 MHz, 256 KB L2 cache on chip) processors interconnected with Fast Ethernet technology. Each node has a 128 Mbyte local memory. The SMP machine is a DELL station based on 4 Intel Pentium III (550 MHz, 512 KB L2 cache on chip) processors with 1 Gbyte physical memory.

We used the Modified Gram-Schmidt (MGS) algorithm as a test parallel application. The MGS algorithm produces from a set of vectors an orthonormal basis of the space generated by these vectors. The algorithm consists of an external loop running through columns producing a normalized vector and an inner loop performing for each normalized vector a scalar product with all remaining ones.

The MGS test program has been written in C for the SMP machine. We have just modified 1 line in the source code and recompile it to be able to run it on Gobelins. In the final version of Gobelins, an SMP code will run without any modification.

We made two sets of experiments : a first one with a 512x512 double float matrix and a second one with 1024x1024 matrix. For each set we have executed the MGS algorithm with 2, 3 and 4 processors.

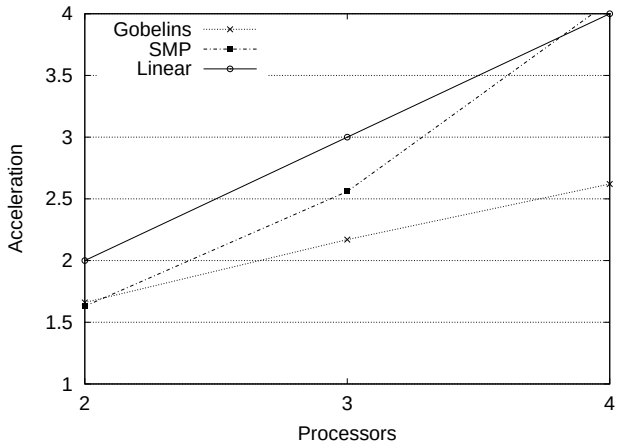

Figure 4: *Acceleration of MGS with a 512x512 Matrix*

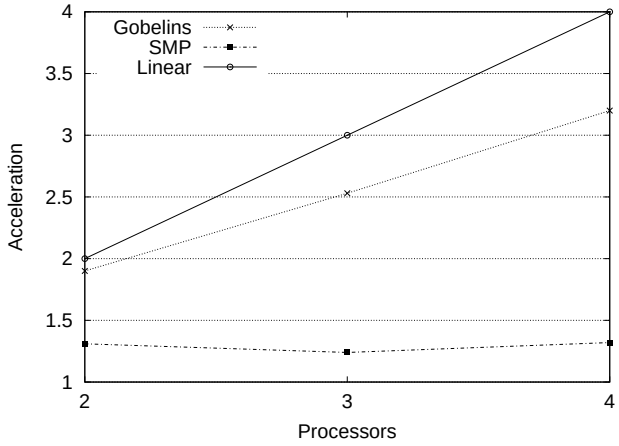

Figure 5: *Acceleration of MGS with a 1024x1024 Matrix*

Figure 4 shows the speed-up obtained for a 512x512 matrix on a SMP machine and a cluster running Gobelins. The speed-up of MGS running on top of Gobelins grows linearly up to 2.62 with 4 processors. On the SMP machine, the speed-up grows up to 4.1 with 4 processors. We can see a dramatic performance improvement on 4 processors with the SMP machine, since the speed-up is super-linear. This result can be explained by observing that in this case, the whole matrix fits in the processors L2 cache which is not the case in the sequential case. The sequential algorithm can be optimized to increase the cache hit ratio even with a large matrix, but the modification is far from trivial. In the Gobelins case, we do not observe this performance peak because of the smaller cache size of the cluster processors. This performance peak could be observed with more nodes which is not possible with the current Gobelins cluster.

Figure 5 shows the speed-up obtained for a 1024x1024 matrix. The speed-up of MGS with Gobelins still grows linearly up to 3.2 with 4 processors. On the SMP machine, the speed-up does not grow with the number of processors and stay very low around 1.3. This surprising result can be explained by the insufficient main memory bandwidth. With this problem size, data does not fit in cache but trashes it for each step of the algorithm. Thus, each memory access induces a cache miss. On the SMP machine during the whole application life time, several processors are competing to read data from the main memory to fill L2 caches. This competition overloads the memory system which is not able to feed caches on time, decreasing the whole machine performance. On the Gobelins cluster, this problem

does not arise since each processor on each machine can access his local memory without any competition.

In summary, results obtained with containers as a low level mechanism to share memory between nodes of a cluster are encouraging. For small problem size, performance is a bit lower than a hardware shared memory machine and is dramatically higher for large problems. Moreover, we could expect even higher performance by increasing the number of nodes. With an SMP machine, increasing the number of processors faces the problem of main memory throughput bottleneck and associated hardware cost overhead.

7 Related Work and Conclusion

Few other work has been done to globally manage *all* resources in a cluster. The Nomad [18] project tends to manage all resources of a cluster, but if global memory management is studied, no support for shared memory if offered. Millipede [10] is a user level implementation of an SSI on top of Windows NT. Shared memory is offered but support for thread migration is not allowed when a thread uses local resources. Some commercial products such as SCO UnixWare or Solaris MC claim to offer an SSI, but a little is known about their real capabilities.

In this paper we have presented the concept of container and described how it can be used for global distributed memory management in Gobelins operating system. Containers unify mechanisms needed to implement complex operating system services like distributed shared memory, thread migration, parallel file mapping in shared memory and management of cooperative caches. We have shown that good performance can be obtained. This make it reasonable to use containers as a low level memory mechanism to make a cluster looks like an SMP.

An implementation of Gobelins in the Linux kernel is in progress. This implementation makes minor modifications to core kernel. Gobelins functionalities are added in the kernel using modules. The current implementation does not include thread migration and fault tolerance. Fault tolerance mechanisms are currently studied [16] on a user level prototype. Thread migration on top of containers is actually studied and will be implemented in the next few months.

Our future work includes deeper experimental evaluation of containers as memory sharing mechanism. We also plan to evaluate the performance of containers as a mechanism to implement cooperative caching, file mapping in shared memory and file system support for thread migration. At last, fault tolerance mechanisms to tolerate node and disk failures will be introduced in the system.

References

[1] C. Amza, A. L. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, W. Yu, and W. Zwaenepoel. TreadMarks: Shared Memory Computing on Networks of Workstations. *IEEE Computer*, pages 18–28, February 1996.

[2] T.E. Anderson, M.D. Dahlin, J.M. Neefe, D.A. Patterson, D.S. Roselli, and R.Y. Wang. Serverless network file systems. In *Proceedings of the 15th Symposium on Operating Systems Principles*, pages 285–302. ACM Press, December 1995.

[3] A. Barak, S. Guday, and R. G. Wheeler. The MOSIX distributed operating system. 672, 1993.

[4] G. Ciaccio. Optimal communication performance on Fast Ethernet with GAMMA. *Lecture Notes in Computer Science*, 1388, 1998.

[5] T. Cortes, S. Girona, and J. Labarta. PACA: A cooperative file system cache for parallel machines. In *Proc. of the 2nd International Euro-Par Conference*, pages 477–486, August 1996.

[6] M.D. Dahlin, R.Y. Wang, T.E. Anderson, and D.A. Patterson. Cooperative caching: Using remote client memory to improve file system performance. In *Proc. of the First Symposium on Operating System Design and Implementation*, November 1994.

[7] E.N. Elnozahy, D.B. Johnson, and Y.M. Wang. A survey of rollback-recovery protocols in message passing systems. Technical Report CMU-CS-96-181, Department of Computer Science, Carnegie Mellon University, September 1996.

[8] M. J. Feeley, W. E. Morgan, F. H. Pighin, A. R. Karlin, and H. M. Levy. Implementing global memory management in a workstation cluster. In *Proc. of the 15th ACM Symp. on Operating Systems Principles*, pages 201–212, December 1995.

[9] J.V. Huber, C.L. Elford, D.A. Reed, and A.A. Chien. PPFS: A high performance portable parallel file system. In *Conference proceedings of the 1995 International Conference on Supercomputing*, pages 385–394. ACM Press, July 1995.

[10] Ayal Itzkovitz, Assaf Schuster, and Lea Shalev. Thread migration and its applications in distributed shared memory systems. *The Journal of Systems and Software*, 42(1):71–87, July 1998.

[11] A.-M. Kermarrec, C. Morin, and M. Banâtre. Design, implementation and evaluation of icare. *Software Practice and Experience*, 28(9):981–1010, July 1998.

[12] D. Kotz and C. Schlatter Ellis. Caching and writeback policies in parallel file systems. In *1991 IEEE Symposium on Parallel and Distributed Processing*, pages 60–67, December 1991.

[13] K. Li and P. Hudak. Memory coherence in shared virtual memory systems. *IEEE Transactions on Computers*, 7(4):321–359, November 1989.

[14] Michael J. Litzkow, Miron Livny, and Matt W. Mutka. Condor - A hunter of idle workstations. In *8th International Conference on Distributed Computing Systems*, pages 104–111. IEEE Computer Society, June 1988.

[15] E. Marcatos and G. Dramitinos. Adaptative and reliable paging to remote main memory. *Journal of Parallel and Distributed Computing*, 1999.

[16] C. Morin, R. Lottiaux, and A.-M. Kermarrec. High-availability of the memory hierarchy in a cluster. In *19th IEEE Symposium on reliable Distributed Systems*, pages 134–143, October 2000.

[17] C. Morin and I. Puaut. A survey of recoverable distributed shared memory systems. *IEEE Transactions on parallel and Distributed Systems*, 8(9), September 1997.

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

[19] Alessandro Rubini. *Linux Device Drivers*. O'Reilly, 1998.

[20] D. J. Scales and K. Gharachorloo. Towards transparent and efficient software distributed shared memory. In *Proc. of the 16th ACM Symp. on Operating Systems Principles (SOSP-16)*, October 1997.

[21] I. Schoinas, B. Falsafi, A. R. Lebeck, S. K. Reinhardt, J. R. Larus, and D. A. Wood. Fine-grain access control for distributed shared memory. In *Proc. of the 6th Symp. on Architectural Support for Programming Languages and Operating Systems (ASPLOSVI)*, pages 297–307, October 1994.

Unité de recherche INRIA Lorraine, Technopôle de Nancy-Brabois, Campus scientifi que,
615 rue du Jardin Botanique, BP 101, 54600 VILLERS LÈS NANCY
Unité de recherche INRIA Rennes, Irisa, Campus universitaire de Beaulieu, 35042 RENNES Cedex
Unité de recherche INRIA Rhône-Alpes, 655, avenue de l'Europe, 38330 MONTBONNOT ST MARTIN
Unité de recherche INRIA Rocquencourt, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex
Unité de recherche INRIA Sophia-Antipolis, 2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS Cedex

Éditeur
INRIA, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
http://www.inria.fr
ISSN 0249-6399