

Cross Cluster Migration using Dynamite Remote File Access Support

A thesis
submitted in partial fulfilment
of the requirements for the degree
of
Master of Science
at the
University of Amsterdam
by

Adianto Wibisono

Section Computational Science



Amsterdam, The Netherlands

September 9, 2002

Abstract

Performing computations in dynamic cluster computing environments requires the application to adapt to the changes. Dynamite provides a dynamic load balancing mechanism to enable High Performance Computing in dynamically changing cluster of workstations environment, by performing task migration. A further step was taken, to enable cross cluster migration to obtain more computational power outside the cluster.

This development of cross cluster migration in Dynamite still holds the prerequisites that the application being migrated between clusters does not perform file operations in its initial cluster. In this thesis a solution to this problem is described, supporting remote data access by integrating the existing Dynamite libraries with Global Access to Secondary Storage (GASS) libraries provided by Globus Toolkit.

Acknowledgements

I would like to thank Dr. Dick van Albada for his expert supervision of my thesis work, Kamil Iskra for all his guidance and help in dealing with all the obstacles that I had to face during this thesis work, Dr. Zeger Hendrikse for getting me acquainted with the Globus Toolkit, and Prof P.M.A Sloot for giving me the freedom to choose the direction of my thesis. I also realize that without the company from the Kruislaan guys during the nocturnal works in the lab 215, the local support from my Indonesian Amsterdamers friends, and the remote mental support from those who cared about me, it would be very hard for me to keep myself motivated and to be able to finish this thesis.

Contents

Abstract	iii
Acknowledgements	v
List of Figures	xii
1 Introduction	1
1.1 Motivation	1
1.2 Thesis Outline	3
2 Process Migration Overview	5
2.1 Implementation Level	6
2.1.1 Kernel Level Process Migration	6
2.1.2 User Level Process Migration	7
2.2 File Access Handling	7
2.2.1 Kernel Level File Access	8
2.2.2 User Level File Access	8
3 Checkpoint and Migration in Dynamite	11

3.1	Background	11
3.2	Architecture Overview	12
3.2.1	PVM Parallel Programming Environment	14
3.3	Checkpoint Mechanism	16
3.3.1	Dynamic loader	17
3.3.2	Checkpoint Handler	17
3.4	Migration Mechanism	20
4	Remote File Access using GASS	23
4.1	Basic Assumption	24
4.2	Default Data movement in GASS	24
4.3	GASS Operation	26
5	Design Issues	29
5.1	Minimizing Residual Dependency	29
5.2	Transparency Requirement	30
5.3	Possible Scenarios	31
6	Implementation	35
6.1	Cross-checkpoint data structure	35
6.2	Modification on Dynamic Loader	36
6.3	Modification of Checkpoint Handler	37
6.3.1	Saving GASS File States	37
6.3.2	URL prefixing	38

6.3.3	Restoring GASS File States	38
6.4	File System Calls Wrapping	39
6.4.1	Wrapper for system call <code>open</code>	40
6.4.2	Wrapper for system call <code>close</code>	41
7	Testing and Performance Measurement	43
7.1	Testing Environment	43
7.2	Correctness and Stability Tests	44
7.2.1	Sequential Test	44
7.2.2	Parallel Test	44
7.3	Performance measurement	45
7.3.1	Memory to File	46
7.3.2	File to Memory	47
7.3.3	File to File	48
8	Summary and Future Work	53
8.1	Summary	53
8.2	Future Work	54
	Bibliography	55

List of Figures

2.1	High Level view of Process Migration	6
3.1	Dynamite System.	13
3.2	PVM Architecture	15
3.3	Stages of migration.	20
4.1	Gass Data Movement	25
4.2	Gass Cache Architecture	26
5.1	Write from Initial to Remote.	31
5.2	Read from Initial to Remote.	32
5.3	Write from Remote to Remote.	33
5.4	Read from Remote to Remote.	34
6.1	Cross-checkpoint additional data structure.	36
6.2	File state data structure.	36
6.3	Saving GASS file states.	38
6.4	URL prefixing.	39

6.5	Restoring globus file states.	39
6.6	Wrapper for system call <code>open</code>	40
6.7	Wrapper for system call <code>close</code>	41
7.1	Time for memory to file tests with remote clusters : fs0 at Free University (upper left), fs1 at University of Leiden (upper right), fs2 at Nikhef (down left) and fs3 at TU Delft (down right) to local cluster fs4 at Utrecht.	46
7.2	Transfer rate for memory to file tests with remote clusters : fs0 at Free University (upper left), fs1 at University of Leiden (upper right), fs2 at Nikhef (down left) and fs3 at TU Delft (down right) to local cluster fs4 at Utrecht.	47
7.3	Time for file to memory tests with remote clusters : fs0 at Free University (upper left), fs1 at University of Leiden (upper right), fs2 at Nikhef (down left) and fs3 at TU Delft (down right) to local cluster fs4 at Utrecht.	48
7.4	Transfer rate for file to memory tests with remote clusters : fs0 at Free University (upper left), fs1 at University of Leiden (upper right), fs2 at Nikhef (down left) and fs3 at TU Delft (down right) to local cluster fs4 at Utrecht.	49
7.5	Time for file to file tests with remote clusters : fs0 at Free University (upper left), fs1 at University of Leiden (upper right), fs2 at Nikhef (down left) and fs3 at TU Delft (down right) to local cluster fs4 at Utrecht.	50
7.6	Transfer rate for file to file tests with remote clusters : fs0 at Free University (upper left), fs1 at University of Leiden (upper right), fs2 at Nikhef (down left) and fs3 at TU Delft (down right) to local cluster fs4 at Utrecht.	51

Chapter 1

Introduction

1.1 Motivation

Efforts to perform High Performance Computing (HPC) in clusters of workstations are interesting because we try to obtain more computing power for our applications, not from an expensive massive parallel processor which is specifically designed for HPC, but by harvesting idle cycles from a cluster of workstations.

This cluster of workstations environment was not designed for HPC, it was probably designed only to provide data sharing facilities between its users. The effective computing power available on this environment depends on the behaviour of the users of the workstations. Since the behaviour of the user is dynamic and unpredictable, this environment also behaves as a dynamically changing environment.

In addition to affecting the available computing power, the user's behaviour also influences the cluster's network capabilities. Although the users may not be using the computational nodes that are needed for HPC, they might still increase the network load. High performance applications that rely on message passing are sensitive to this. Moreover, HPC applications may also have dynamic behaviour at run-time. Computational resources needed by each sub-task in this kind of application can also change over time. In fact, this effort to do HPC in cluster of workstation poses many challenges.

The Dynamite, a dynamic execution environment was developed to face these challenges. This environment was developed based on the Parallel Virtual Machine (PVM) parallel

programming libraries. It was designed to provide a load balancing mechanism for parallel applications in a dynamically changing cluster environment. It gives the application ways to adapt to its changing environment by migrating individual tasks from heavily loaded nodes to less loaded nodes within the cluster. This operation is performed in a manner that is robust, efficient and transparent to the user and the application programmer [Iskra et al., 2000b].

Experiments with the Dynamite environment for parallel programming show that it can realize greatly improved utilisation of system resources for long running jobs. It is also currently being used as a research tool, in order to conduct experiments on dynamic task scheduling [Iskra et al., 2000a]. A recent development on this Dynamite environment was to obtain more computing power not only from idle nodes within a cluster but to facilitate access to computing power from outside the cluster, by performing cross cluster migration. This presents a new challenge and also an opportunity.

Migration of tasks within a cluster could be performed utilising the fact that nodes within that cluster share file systems. This condition no longer holds when we perform task migration across clusters. In her thesis, Jinghua Wang addressed this problem by providing a socket migration method [Wang, 2001]. In this case the application being migrated between clusters cannot perform file operations in its initial cluster. In this thesis we try to solve this problem by supporting remote data access, integrating the existing Dynamite libraries with Global Access to Secondary Storage (GASS) libraries provided by the Globus Toolkit.

The steps to gain more computational power from outside sources mean that we are moving into geographically distributed computing which might span multiple organizations. This is relevant with the grid computing paradigm which is becoming increasingly popular in the scientific community. Scientists are now sharing more and more computational resources, storage systems and data services that are distributed and owned by multiple organisations [Baker et al.,].

In 1994, this current wave of widely distributed computing and collaboration had been predicted in [Geist, 1994]. It was mentioned that in order to cope with this wave, several social issues have to be addressed, such as scheduling applications having to consider both internal organization priority and additional tasks from outside the organization that want to use its resources, accounting of the resource usage and support.

The Globus Toolkit itself actually is a widely used middle-ware which is designed to cope with these issues. Now it has become a major infrastructure component of most existing grid computing projects. It provides basic services for enabling grid computing such as security, resource management, communication, information for resource discovery, and executable management [Foster and Kesselman, 1999]. These other services also offer a potential to be used in the Dynamite environment.

1.2 Thesis Outline

The next chapter will start to describe process migration in general, and related research on this topic. Then the Dynamite program environment, how it works and recent developments will be discussed in more detail in Chapter 3. Chapter 4 will discuss the GASS library from the Globus toolkit. Here we will discuss how the library works, what assumptions it is based on and how we are going to utilize the library to solve our problem. Chapter 5 will discuss design considerations in this work; how we are going to integrate the GASS library into the Dynamite libraries. We will consider what will be expected from the result of the integration and how it will behave. In this chapter we also discuss different possible scenarios that need to be handled. The implementation of this design will be discussed in Chapter 6. Chapter 7 will describe the testing of the implementation and discuss some performance measurements. Chapter 8 will summarize and conclude our work and give some future directions which could be followed from this work.

Chapter 2

Process Migration Overview

This chapter describes basic terms related to process migration and gives a brief overview of different process migration approaches. The main interest of this overview is to see how the existing systems solve the environment and resource transfer (e.g handling access to external devices such as open file or other devices) during process migration, which is directly related to this thesis.

Process is a key concept in operating systems [Tanenbaum, 1992]. A process consists of code and data, a stack, register contents, and the state specific to the underlying Operating System (OS), such as parameters related to process, memory, and file management. A process can have one or more threads of control. Threads, also called light-weight processes, have their own stack and register contents, but share the process's address space and some of the operating-system-specific state, such as signals.

Process migration is the act of transferring a process between two machines (the source and the destination node) during its execution [Milojicic et al., 2000]. It involves extracting the state of the process on the source node, transferring it to the destination node where a new instance of the process is created and updating the connections with other processes on communicating nodes (Figure 2.1). The transferred state includes the process address space, execution point (register contents), communication state (e.g., open files and message channels) and other operating system dependent state.

Research on process migration is conducted on both heterogeneous environments and homogeneous environments. Heterogeneous process migration has not been addressed in most

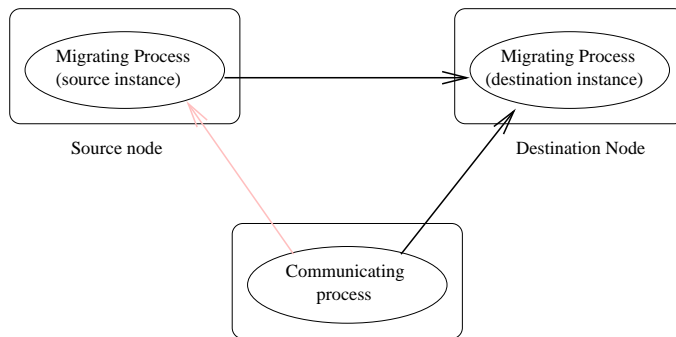


Figure 2.1: High Level view of Process Migration

migration implementations. This is not a significant limitation because most of the work was conducted for workstation clusters typically consisting of compatible machines. However to fully utilize systems that are heterogeneous in either hardware or operating systems, this mechanism is needed. Example of heterogeneous process migration could be found in [Katratos et al., 1998] where the author describes a mechanism to allow process migration of a process running on Puma operating system to a Linux operating system.

Homogeneous process migration involves migrating processes in a homogeneous environment where all systems have the same architecture and operating system but not necessarily the same resources or capabilities. Process migration can be performed either at the kernel-level, user-level or at application level. Dynamite belongs to the user level homogeneous process migration category and we will focus on this class of migration in the next section.

2.1 Implementation Level

2.1.1 Kernel Level Process Migration

Kernel level process migration techniques modify the operating system kernel to make process migration easier and more efficient [Lawrence, 1998]. The advantage of modifying the kernel is that process state is more readily available at this level. This approach also implies that any kind of process running on the kernel can be migrated since the kernel was designed with the process migration in mind. Some implementation of this approach include Locus [Walker and Popek, 1983], Amoeba [Mullender et al., 1990], V [Theimer et al., 1985], Accent [Zayas, 1987], MOSIX [Barak et al., 1995] and Sprite [Ousterhout et al., 1988].

In spite of this straightforward way of capturing process state, this approach has a number of undesirable features in higher context e.g meta-computing or grid computing context. As the number of different architecture and operating system platforms grows, the issue of mechanism portability becomes important in addition to efficiency concerns. Furthermore it is typically unfeasible to mandate replacement of the operating system on all participating nodes. This characteristic is limiting the possibility to deploy this specialized distributed system into wider audience.

2.1.2 User Level Process Migration

A number of systems to date have provided some form of homogeneous process state capture implemented at the user level (i.e. without direct, special kernel support). Sample of implementation includes Condor [Litzkow et al., 1988], Codine [Genias Software GmbH,] which supports sequential jobs migration. Several implementation which supports migration of parallel jobs are MPVM [Casas et al., 1995], CoCheck [Stellner and Pruyne, 1995], tmPVM [Tan et al., 1999], ChaRM [Dan et al., 1999], DAMPVM [Czarnul and Krawczyk, 1999], Hector [Robinson et al., 1996], and our Dynamite [van Albada et al., 1999] system also fall in this class of approach in implementing process migration.

Although this approach is somewhat less efficient in capturing process state than a kernel level implementation, user-space designs are generally more portable. A common argument against user-level state capture schemes is the difficulty involved in capturing and recovering a process's external and kernel-level state. This could pose some restrictions with respect to intra-process communication, and other services that involve external state such as file system.

2.2 File Access Handling

Among the existing approaches of the process migration we want to focus on how they cope with the problem of file access before and after migration. Different approaches can be seen to be depending on the level of process migration implementation.

2.2.1 Kernel Level File Access

LOCUS [Walker and Popek, 1983] was developed to be a distributed, fault tolerant version of UNIX compatible with both System V and BSD. It consist of a number of modifications to the UNIX kernel and an enhanced file system. The system is built around a network wide file system supporting file caching and replication and support for operation in a heterogeneous environment. Locus distributed file system eases the transfer of open files, no need for additional remote file access support in this case.

MOSIX [Barak et al., 1995] supports location and access transparencies and will operate in a heterogeneous environments. It provides a single file system by mounting each node's file system under a single super root. No additional file caching are needed, remote files are accessed via kernel to kernel RPC. Process migration does not affect file access.

In Sprite [Ousterhout et al., 1988], open files are dealt with the source passing information about the process's open files firstly to the target, which then notifies the file server that each open file has been moved. The server queries the source to obtain the most recent state associated with that file. Lastly the new process's Process Control Block is updated from the relevant fields on the current machine. A PCB for the process is also maintained at the home node in order to achieve a very high level of transparency.

From these examples we can see that kernel level process migration was usually implemented as a part of larger distributed system environments. These systems generally also implement a distributed file systems, so the process migration does not need additional file access supports.

2.2.2 User Level File Access

tmPVM [Tan et al., 1999] dynamically rewrites the open and close system routines so that any calls to them with their arguments are noted. During the process migration the states of all opened file descriptors are captured and file descriptors are restored by reopening the corresponding files with appropriate file mode relative to the state at the point of migration. tmPVM implementation is still restricted by the assumption that the same file is available via the same path from any node within the system. This implies that approach on tmPVM

would not work if it has to perform migration across cluster with different file system.

Condor [Litzkow et al., 1988] performs process state capture and recovery in homogenous environments by using a slightly modified core dump of the process to capture and recover memory and processor state. Operating system specific information associated with the process is maintained at the user level by tracking the parameters and return values of all system calls via wrapper routines. As already explained that Condor's approach is at the user level it has no support for a global view of file system.

For Condor all system calls, including file handling, are passed back to a process **shadow** on its home node. Since individual write calls are not traced by checkpointing mechanism, Condor programmers are recommended that all file operations are idempotent – i.e. that multiple reads or writes will not have an adverse affect. Programs that both read to and write from the same file are not guaranteed to work properly.

This approach although it solves the problems of non existence of global file system view in user level implementations, leave a residual dependency on the home node. A home dependency can simplify migration, because it is easier to redirect requests to the home node than to support services on all nodes. However it also adversely affects reliability, because a migrated foreign process will always depend on its home node. In Condor's case the residual dependency was on the shadow process at the home node.

Approach on handling file access by Dynamite system is similar to the approach that has been taken by tmPVM so it faces the same limitation, that migration could be performed only on a system sharing the same file system. This is the problem that this thesis tries to overcome. The mechanism and solution chosen for in this work is described in Chapter 5.

Chapter 3

Checkpoint and Migration in Dynamite

This chapter is about Dynamite, the **DYNAMIC** Task migration **E**nvironment. The first section will discuss the motivation to perform task migration in Dynamite Environment. The next section will give an overview of Dynamite's main components and how they interact. In this section the PVM parallel programming environment which became the base of current implementation of Dynamite will be introduced. The last section will discuss the checkpoint and migration mechanism in more detail. Since we are going to perform modification and support to this system, a clear understanding on how the system work is necessary. Focus in this chapter will be on the checkpoint and restart mechanism, since most of the modification needed for remote data access support will be implemented in this part.

3.1 Background

As described in the first chapter, Dynamite is designed to face the challenge of running HPC in a dynamic cluster computing environment. Dynamite tries to maintain a good performance of parallel applications in a cluster of workstations, where the available computing power of each node may change dynamically.

Performance of a parallel application can be optimized by choosing a good strategy of task allocation. The performance of parallel applications is often determined by the performance of the slowest task, which could become a bottleneck. In a static environment where

the nodes are dedicated, optimal task allocation can be performed at the beginning of the computation, and need not be changed as long as the application run time behavior does not change. This is not the case that Dynamite is dealing with, because optimal allocation of tasks can not be determined at the beginning of application execution.

If in a cluster of workstations nodes used in running a parallel application receive more computational load from other users, performance of the tasks allocated to the nodes affected is degraded. For applications with a dynamic run-time behaviour, similar case can also apply when some of the tasks demand more computation. In both cases the node which receives additional load either from other user or from the application itself, is no longer optimally allocated. Dynamite will try to obtain optimal allocation continuously during the application run time [Iskra et al., 2000a], in order to cope with these changes which might degrade overall performance of the application.

Obtaining optimal task allocation during run time means that Dynamite needs to be able to detect the changes in the environment, and if there is a change that might reduce the performance, respond to this change by performing re-allocation of the tasks. Tasks should be re-allocated in such a way that performance can be maintained by considering optimal load balance and minimum communication overhead. Load monitoring facilities is needed to detect the changes and process migration are needed to do re-allocation of the tasks.

An additional advantage of task migration is that it is possible to free the individual nodes, if they are needed for some higher priority tasks. This reallocation can be performed without losing the results of the computation obtained so far. In grid computing where people are sharing resources, this flexibility will be beneficial. For the owner of the resources, it is more convenient to share their resource for this type of parallel application, since it can be asked to leave their resources, if they are needed for an important application.

3.2 Architecture Overview

The components of the Dynamite system are as follows [Iskra et al., 2000a] :

- resource monitor facilities which monitor computation load on nodes being used,
- scheduling sub system which tries to make optimal task allocation,

- task migration module, which allows task to be re allocated.

The load monitoring components collect information necessary for the scheduler to decide whether or not a certain task needs to be migrated to another node. This information consists of available capacity of each node, network connectivity and capacity, current load on each node and current computing capacity required by the application. The first two items may be considered to be stable so they can be obtained in advance. The last three items are dynamic information that needs to be continuously monitored.

Migration decisions will be performed by the scheduler, based on the information that is available about the cluster. The typical approach taken by most cluster management systems is to measure the load on each available host and of each application process, and decide on re-allocation if necessary.

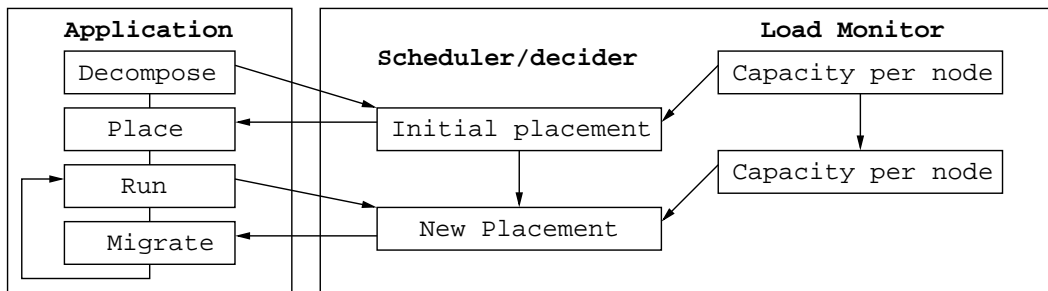


Figure 3.1: Dynamite System.

Figure 3.1 illustrates how an application is run using the Dynamite system. First it is decomposed into several sub tasks. An initial placement is determined by the scheduler, one that needs not to be optimal yet. When the application is running, the load monitoring sub-system checks the capacity per node. Based on this information the scheduler will decide whether or not migration is necessary.

One of possible strategies that can be used by the scheduler is to migrate task on a heavily loaded node (due to external user influence) or the busiest task (due to dynamic run-time behaviour) to the least loaded node until a satisfactory state is achieved. When all machines are evenly loaded with jobs and tasks, Dynamite does nothing. It only performs migration when the system gets out of balance. This strategy has been proven well suited for running independent jobs on networks of workstations, but it performs less well for parallel appli-

cations as it completely neglects communication between interdependent tasks.[van Albada et al., 1999]

The task migration module will perform re-allocation while the application is still running. It requires that the current state of the running application on initial node to be captured. The process of capturing a snapshot of a running application state is called process *checkpointing*. The saved task state information can be used to restart the application in the target node.

3.2.1 PVM Parallel Programming Environment

Current implementation of Dynamite was based on PVM parallel programming libraries. Under PVM, a user defined collection of networked computers appears as one large distributed memory computer. This logically distributed memory computer (which is actually our cluster of workstations environment) is termed *virtual machine*.

PVM supplies functions to automatically start up tasks on the virtual machine and allow the tasks to communicate and synchronize with each other. A *task* is defined as a unit computation in PVM analogous to a Unix process. Applications can be parallelized by using message-passing. By sending and receiving messages, multiple tasks of an application can cooperate to solve a problem in parallel.

The PVM system [Geist et al., 1994] consists of two parts. The first part is a daemon process, called `pvmd`, that resides on all the computers (hosts) making up the virtual machine. When a user wants to run a PVM application, he first creates a virtual machine by starting up PVM. This starts a `pvmd` process on each of the member hosts in the virtual machine (see Figure 3.2). The `pvmd` serves as a message router and controller. It provides a point of contact, authentication, process control and detection. The first `pvmd` (started from console) is designated as the master while the others (started by the master) are called slaves. Only the master can add or delete slaves from the virtual machine.

The second part of PVM system is a library of routines (`libpvm`) that allows a task to interface with `pvmd` and other tasks. `libpvm` contains functions for packing and unpacking messages, and functions to perform PVM *syscalls* by using message functions to send service requests to the `pvmd`.

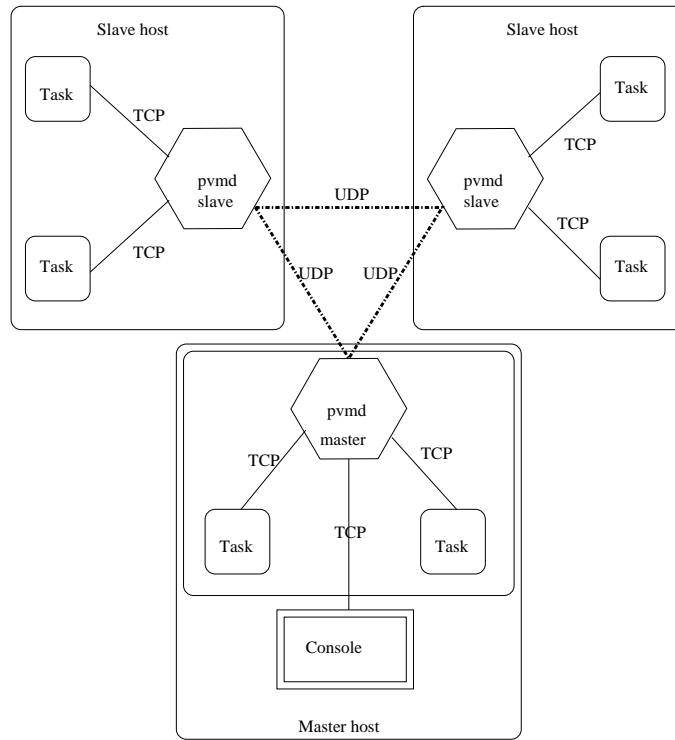


Figure 3.2: PVM Architecture

In order to run an application in the Dynamite environment, the users need to re link their application with the Dynamite versions the of PVM libraries (DPVM) and the Dynamite dynamic loader. This libraries differ from the original PVM libraries in the support for task migration. There is additional PVM console command **move** in DPVM, which enables users to migrate tasks among computational nodes in the virtual machines. Dynamite supports applications written for PVM 3.3.x, running under Solaris/UltraSPARC 2.5.1 and 2.6 and Linux/i386 2.0 and 2.2 (libc 5 and glibc 2.0 binaries are supported).

Communication in PVM

PVM allows tasks to communicate indirectly via the PVM daemons, or directly by using TCP connection. For the indirect communication scheme, the PVM daemons have a routing table of the location of tasks that wish to communicate with each other.

Each task in PVM has a unique task identifier. Part of the task identifier denotes the node on which the task run, and another part denotes the task number within that node. In this way if the tasks communicate with each other, the location of the task can be obtained directly

from the task identifier.

In DPVM this identifier will be the same, but it will no longer contain the correct information about which node a task reside on. An additional routing is designed to handle this problem. Communication states of a task that is going to be migrated need to be preserved, as it was before the migration occurs. Later in the Section 3.4 this mechanism will be discussed in more detail.

3.3 Checkpoint Mechanism

One of the important part of migration is the process checkpointing. By capturing the state of the running process on the source node, the process can be restarted again to resume the computation on the target node. State includes processor registers, memory states and communication state, such as open message channels or open files and signal masks. Memory state includes text (program code), data, and dynamically allocated data of the program, also the shared libraries and stack used.

Dynamite must provide saving and restoring interfaces that allow the process migration mechanism to extract a process s state from the source node and restore this state on the destination node. Signals are used in Dynamite to trigger the checkpointing mechanism. Using signals allows checkpointing to be triggered asynchronously from outside of the running process.

Dynamite implements the checkpointing mechanism in the user level, specifically it is implemented directly in the dynamic loader, a low-level user space component of a running Unix process. The main purpose of the dynamic loader is to load the shared libraries needed by a program, prepare the program to run, and then run it.

In addition to dynamic loader (for checkpoint initialization) and checkpoint handler (to handle the asynchronous signals), the checkpoint mechanism in Dynamite also involves system call wrapping. During run time, applications use system calls that might change process state. When the system calls deal with external devices e.g. files or memory mapped disks, the states can only be obtained by wrapping and tracking the changes made during run time.

3.3.1 Dynamic loader

Implementation of Dynamite checkpointer is based on the the Linux ELF dynamic loader version 1.9.9. Necessary modification has been made in order to enable capturing process state for checkpointing and to wrap some system calls that might influence the process state. Low level data needed by dynamic loader when restoring is added as a private data structure in the data section. This data structure will be used to keep locations of all the memory segments, information on the state of open files, and other cross-checkpoint data which need to be preserved.

On preparing to run the application, the dynamic loader records the locations of all the memory segments: text, data and stack in the data structure. The requested shared libraries are loaded into memory, and dynamic linking is performed. The handler for specific user defined signal for checkpointing (SIGUSR1) is installed. This signal is handled by a check-point handler function `ckpt_handler`, which will perform the actual process state saving.

The wrapping of the system calls are prepared also in the dynamic loader. The pointers to the system calls that are going to be wrapped, are redirected to the modified version of the system calls. We still keep the real pointer of the system call, which might be used in the modified version of the system call.

3.3.2 Checkpoint Handler

The checkpoint handler will perform checkpointing for a running application, and also is responsible for restoring an application from a checkpoint file or socket connection.

Checkpointing

After the signal handler has been installed and cross checkpoint data has been initialized in the dynamic loader, the application is checkpointable. If user send signals SIGUSR1 the running application will be suspended and control is given to the checkpoint handler. The steps of checkpointing by the checkpoint handler are as follows :

- *Saving process registers.* Before the `ckpt_handler` signal handler starts executing, the operating system preserves the contents of all CPU registers on the stack. By calling `setjmp` function the state of the process registers will be saved in a buffer, and it can be used later on when restoring from checkpoint.
- *DPVM specific clean up.* If checkpointing a DPVM application, all the socket connections need to be flushed and disconnected. This is performed by `dpvm_user_save` function. This function will be discussed further with the DPVM migration mechanism.
- *Saving file states.* The state of open files is saved subsequently. For every open file a position of the file pointer is obtained with the `lseek` call.
- *Saving signal states.* The state of signal handler is saved using `sigaction` system call.
- *Saving dynamic loader state.* Not all state of the dynamic loader is preserved but to enable dynamic loading after restarting from checkpoint certain data structures such as list of loaded modules needs to be preserved.
- *Creating checkpoint file or socket connection.* The checkpoint file is created if the migration is using the file method, otherwise the checkpointed process will create a socket connection to the target of migration. The `ckpt_create` procedure writes the checkpoint file to the disk.
- *Storing the checkpoint data.* All the data segments must be stored (to file or sent via socket to the destination), since the process has very likely modified them. This applies both to initialized and originally uninitialized segments, which are merged when checkpointing. The dynamically allocated heap segment, the top of which is obtained using the `sbrk` call is also stored. Both the application data segments and those of the shared libraries are stored, as is the CPU stack segment. The text segments of the process itself and those of the dynamically linked shared libraries are also stored. Essentially all the process address space is written into file.

Restoring

Both file and socket checkpointing method are restored from files. For the socket method it will be a temporary file created on the target of migration. Restoring of a checkpoint file begins in the dynamic loader, It will notice a checkpoint specific parts from program headers (marked with type `PT_NOTE` and flags `PT_CKPT`), and will give the control to the checkpoint handler via `ckpt_restore` function. Before giving the control to the checkpoint handler, data and text segment are already loaded by default by the operating system, and the dynamic loader has already prepared the cross checkpoint data structure to be used further in restoring.

The steps in the checkpoint handler itself are as follows :

- *Restoring heap and shared objects.* The cross-checkpoint data structure contain information about the heap and shared objects used in the application. Restoring heap is performed using `brk` function to allocate heap memory and its contents are initialized with `read` from checkpoint file. Shared library segments are restored by memory mapping (using `mmap`) from checkpoint file.
- *Restoring stack.* Restoring stack is complicated since there is a considerable danger of overwriting the frame and return address of the currently executing function. To prevent this from happening, the restoring routine is called recursively until its stack frame is safely beyond the frame and return address of the executing function.
- *Restoring file states.* Files are re opened again using `open` and `lseek` system call . If the new descriptor after restoring is different from the old file descriptor, it will be duplicated using `dup` system call.
- *Restoring signal states.* Signal states are restored using the `sigaction` system call.
- *DPVM reconfiguration.* On restoring a DPVM application it needs to initialize the `libpvm` and configure the process as a task. Socket connection to talk to the local `pvm` need to be prepared, so that the `pvm` task can be resumed properly.
- *Re-installing Signal Handler.* The signal handler for `SIGUSR1` need to be installed to handle the subsequent checkpoint signals.

3.4 Migration Mechanism

The previous section describes the checkpointing mechanism for general applications. For DPVM application the communication state between DPVM tasks need to be preserved. Direct connections between tasks need to be flushed and closed. The connection with the PVM daemon has to be terminated before checkpointing and after restoring the task needs to be reconnected to the local PVM daemon.

Migration in DPVM is triggered by a PVM task that calls the migration function `pvm_move` (possibly called from PVM console). The protocol used for performing migration with regards to preserving the communication states consists of 4 main stages as shown in Figure 3.3, where *task 1* is migrating from *node 1* to *node 3*. The nodes that are active at particular stage are shaded.

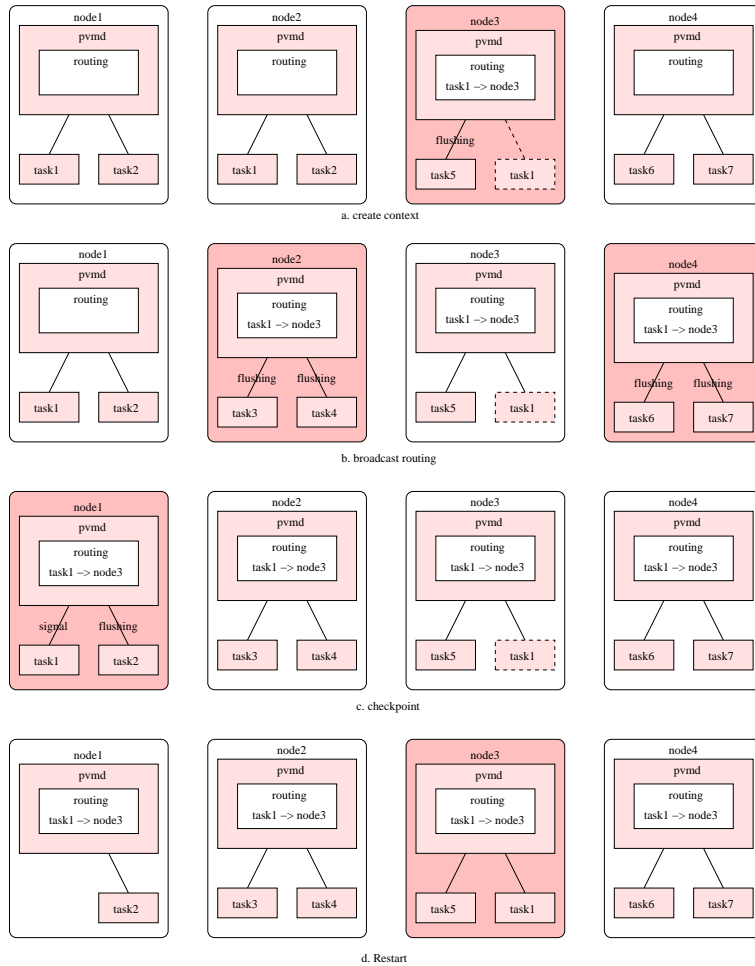


Figure 3.3: Stages of migration.

In the first stage (*create context*) a new task context for the migrating task is created on the destination node (*node3*). The routing table on that node is updated to indicate that the task to be migrated is running on this node. Only control structures in the PVM daemon are allocated, so the daemon can accept and temporarily store any messages it receives that are addressed to the migrating task, the task itself is not started yet. Notification is sent to other tasks in the node using TC_MOVED messages. Both the source and destination claim that they have the migrating task running on them and they still accept messages for it. Messages from all but destination node are still routed to the source node.

In the second stage (*broadcast route*), the new routing information is broadcasted throughout the PVM virtual machine, all the nodes but the source and destination update their routing table (*node 2* and *node 4*). The next stage (*checkpointing*) consist of two different operations executed on the source node (*node 1*). First the routing table is updated to the destination node, and then the checkpoint signal is sent to the task. The task terminates its PVM connections, creates the checkpoint file and terminates.

In the last stage (*restart*) the checkpointed task is restored on the destination node, as described in Section 3.3.2. The checkpoint handler invokes the `dpvm_userrestore` routine which in turn calls `pvmbeataask`, a standard PVM startup function. This function re-connects the task to the destination PVM daemon using a somewhat modified connection protocol (there is no need to allocate a new task identifier, so parts of the initialization can be skipped). Control is passed back to the application code and the PVM daemon on the destination node can finally deliver all the messages addressed to the migrating task which it had to store during the migration.

Chapter 4

Remote File Access using GASS

In wide area computing, programs frequently execute at sites that are distant from their data. Data access mechanisms are required that place limited functionality demands on an application or host system yet permit high-performance implementations. The Global Access to Secondary Storage (GASS) library was developed to address these aforementioned requirements. [Bester et al., 1999]

This service defines a global name space via Uniform Resource Locators and allows applications to access remote files via standard I/O interfaces. High performance is achieved by incorporating default data movement strategies that are specialized for I/O patterns common in wide area applications and by providing support for programmer management of data movement.

GASS forms part of the Globus toolkit, a set of services for high-performance distributed computing. GASS itself uses the Globus services for security and communication, and other Globus components use GASS services for executable staging and real-time remote monitoring. Application experiences demonstrate that the library has practical utility.

We consider that GASS library could be used to support the cross cluster migration in Dynamite project. One of the reasons is that it is designed to achieve a high performance for the basic file access pattern of an application. Since the support that we need for Dynamite for cross cluster migration was to handle input and output files we could use this limited functionality that is provided by GASS library. We are not aiming to provide a full support of distributed file system for Dynamite cross cluster migration.

4.1 Basic Assumption

The GASS library was designed to provide a support for default data movement strategies that are common in wide area computing environment. It was not designed to provide a general library for distributed file system. The common pattern of data movement assumed in design of GASS library was as follows :

- *Read only access to an entire file which is assumed to contain "constant" data.* In the common case of an application that requires processing of static input files, such data may be accessed by multiple readers (perhaps by every process in parallel program). With read only accesses and data that is assumed to be constant no coherency control is needed (Figure 4.1 (a)).
- *Shared write access to an individual file is not required, meaning we can adopt the policy that if multiple writers exist, the last thing written will be the final value.* For example a parallel application in which all processes generate the same answer, or in which any answer is valid. Multiple writers are supported but coherency is simple to implement because it is enforced only at the file level (Figure 4.1 (b)).
- *Append only access to a file with output required in "real time" at a remote location.* For example, a log of program activity, used perhaps to monitor execution. In case of multiple writers, output may be interleaved, either arbitrarily or on a record-by-record basis (but with source ordering preserved). Again, the simple access pattern means that coherency control is simple. (Figure 4.1 (c)).
- *Unrestricted read/write access to an entire file with no other concurrent accesses:* for example, output data produced by a scientific simulation. Because only one process access the file at the time, no coherency control is required.

4.2 Default Data movement in GASS

Optimal execution of grid applications requires careful management of the limited network bandwidth available. Typically, distributed file systems hide data movement operations from the application programmer. When control is provided, it is focused towards latency

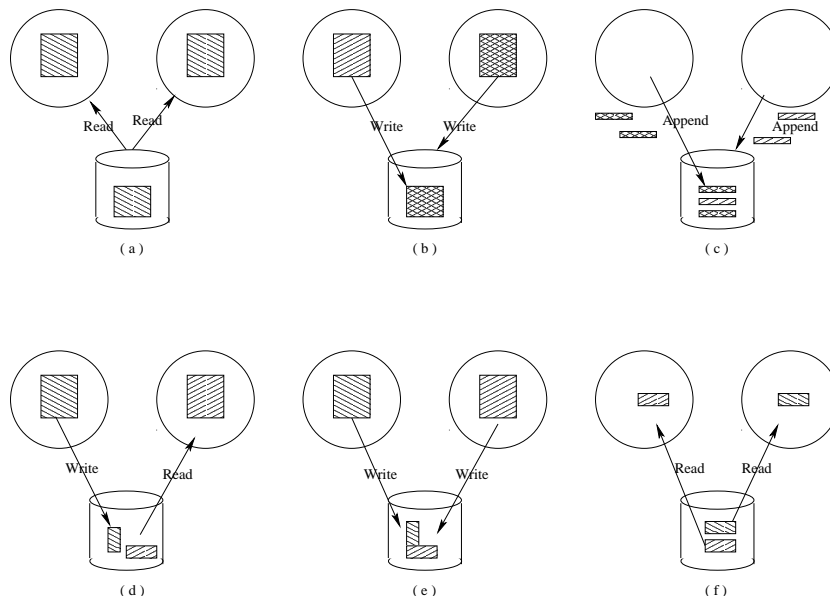


Figure 4.1: Gass Data Movement

management rather than bandwidth management. GASS addresses bandwidth management issues by providing a file cache: a local secondary storage area in which copies of remote files can be stored (Figure 4.2). By default, data is moved into and out of this cache when files are opened and closed, except for files opened in append mode, according to standard strategies.

The first strategy is to fetch and cache on first open for reading. When a remote file is opened for reading, the local cache is checked and the entire remote file is fetched only if it is not already present. The local copy is then opened, a file descriptor is returned, and a reference count associated with the cached copy is incremented. The file can then be accessed within the application using conventional I/O calls on the cached file. This first strategy optimizes data movement for a common situation in parallel computing, namely an input file that is accessed by many processes. Because input files are typically read in their entirety, no unnecessary data transfers are performed and a potentially large number of round-trip communications are avoided. However, the strategy may be inappropriate if a file is large: computation may be delayed too long while the file is transferred, or the local cache may be too small to hold the entire file. Alternative strategies such as prestaging and specialized GASS servers can be used in such situations; these are discussed below.

The second strategy is to flush cache and transfer on last close on writing. When a remote

file that has been created or opened for writing is closed, the reference count associated with that file is checked. If this count is one, then the file is copied to the associated remote location and then deleted from the cache; otherwise, the reference count is simply decremented. This caching strategy reduces bandwidth requirements when multiple processes at the same location write to the same output file. Conflicts are resolved locally, not remotely, and the file is transferred only once. As a special case, a remote file that is opened in appending mode is not placed in the cache; rather, a communication stream (on Unix systems, a TCP socket) is created to the remote location, and write operations to the file are translated into communication operations on that stream. This strategy allows streaming output for applications that require it.

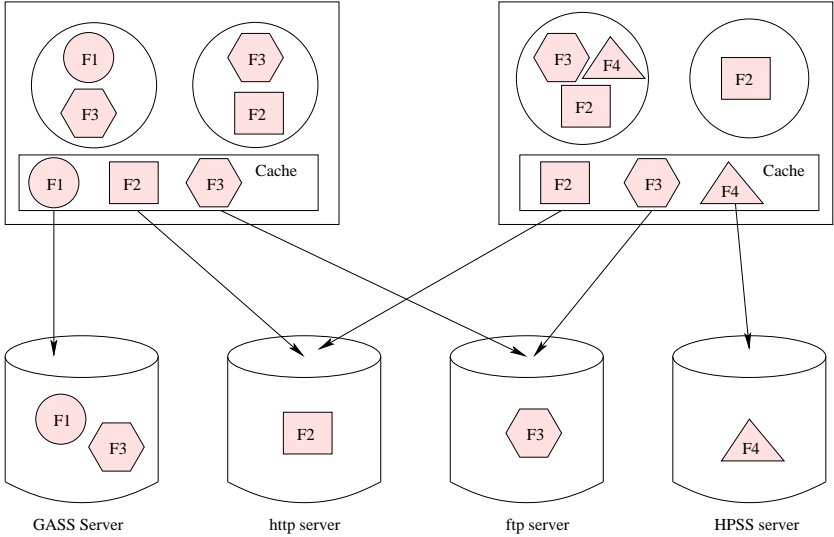


Figure 4.2: Gass Cache Architecture

4.3 GASS Operation

Grid applications access remote files using GASS by opening and closing the files with specialized open and close calls (i.e. using `globus_gass_open`, `globus_gass_fopen`, `globus_gass_close`, `globus_gass_fclose`). These trigger the GASS cache management operations described above to optimize performance based on the default data movement strategies.

From an application viewpoint, the GASS open and close calls act like their standard Unix I/O counterparts, except that a URL rather than a file name is used to specify the location

of the file data. File descriptors or streams returned from these calls can be used with normal read and write operations: only open and close calls need to be modified, all other I/O calls can be used unchanged. The use of specialized open and close calls means that some program modifications are required before an application can use GASS. However, the difficulty of inserting these calls is minimized by ensuring that the GASS calls have the same arguments, are semantically equivalent, and are backward compatible to the Unix functions that they replace.

Re-linking (as in Condor) or kernel support (DFS, WebFS) could be used to avoid the need for application modification, at the cost of some increase in implementation complexity and decrease in portability. A URL used in a GASS open call specifies a remote file name, the physical location of the data resource on which the file is located, and the protocol required to access the resource. An advantage of thus making the location of the file server explicit (unlike for example DFS, which hides file server locations in global file names) is that an application can use domain-specific heuristics to select from among alternative copies. For example, in Figure 4.2, file F3 is replicated on two different servers; different processes may choose to access the nearer copy. The figure also emphasizes the point that GASS can be used to access files stored in a variety of storage devices: specialized GASS servers or FTP servers (already supported) or HTTP, HPSS, DPSS, or other servers (work in progress). The GASS system also exposes additional lower-level APIs which can be used to implement specialized data movement and access strategies.

Chapter 5

Design Issues

The objective of this work is to provide support for remote file access when an application is being migrated to a different cluster. Since we are not working on a specialized distributed operating system, migration across multiple clusters implies that we no longer have a shared file system.

For the application, it can no longer access its input or output files which reside on the initial clusters. By utilizing the GASS libraries we will allow this application to have access to its files in the initial cluster. In regards to the migration process itself, this means that we cannot use the File Migration method. Instead we have to use the socket migration method.

We already mentioned in the introduction that we will use the GASS libraries from the Globus Toolkit to solve this problems. The behaviour of the GASS library has already been explained in the previous chapter. In this design section we will discuss what would be expected from the result of the integration, and how it should perform its task in different possible remote data access scenarios.

5.1 Minimizing Residual Dependency

The solution for this remote file access problem is different from the other approaches that were discussed in the Chapter 2. The distributed file system approach as used by Sprite [Ousterhout et al., 1988], LOCUS or MOSIX is not feasible, since we are not working on a distributed operating system.

Another approach used by Condor [Litzkow et al., 1988] to stream the system calls back to the shadow process in the home node was not feasible either. The implementation of Dynamite leaves no part of the migrated process on the initial node (e.g. no residual dependency). We want to maintain this minimal residual dependency characteristic, as introducing a shadow process on the initial node to our solution might increase the run-time cost of the migration. In our solution we will link our application with the GASS libraries and leave the remote file access operations to the GASS library.

A GASS server needs to be started on each file system from which we wish to perform migration. This GASS server can handle remote file system calls from every process migrated out of this file system. It is an entirely different approach than Condor's approach since this server is not dedicated to a single process migration, as the shadow process.

Additional run-time cost would still occur since the operation of GASS function calls (see Chapter 4) perform data transfer at the beginning of a read operation or at the end of write operation when file is closed.

5.2 Transparency Requirement

The Dynamite libraries were designed to be transparent. Applications need not to be changed in order to run in the Dynamite environment. They only need to be linked with the Dynamite libraries. Additional support for remote data access should preserve this transparency requirement.

For the file system calls, it is required that the application can be written using normal libc file system calls such as (`open`, `close`, `read` and `write`). On run-time, they should be replaced by GASS file calls. This transparency may be achieved by utilizing the existing system calls wrapping. Since Dynamite already perform this, we only need to modify this file system calls wrapping.

The resulting program should be able to decide when to use remote data access, and when to use normal file operations. Whenever the application is performing file operations on the local cluster, it should use the normal file operations. Although the GASS libraries could be used to open local files, it would still cause an overhead on the module initialization.

Although the Globus GASS libraries uses URL for accessing remote data files, the user also need not be aware of this fact. This means that sometimes our libraries need to automatically prepend a prefix to the file name that is opened by the user, according to the initial cluster where the application is started.

The states of the input or output files which are being accessed by the application should be maintained after the migration, as it was before the migration. This means that we need to keep track of the states of the files during migration.

5.3 Possible Scenarios

In this section we will consider possible scenarios for process migration with regard to the file operations. This will be used to elaborate how our remote file support should handle each possible scenario. The scenario was defined from the point of view of the application being migrated.

- Initial Cluster to Initial Cluster. This type of migration is not cross cluster migration. The support for the GASS libraries is not needed.
- Initial Cluster to Remote Cluster.

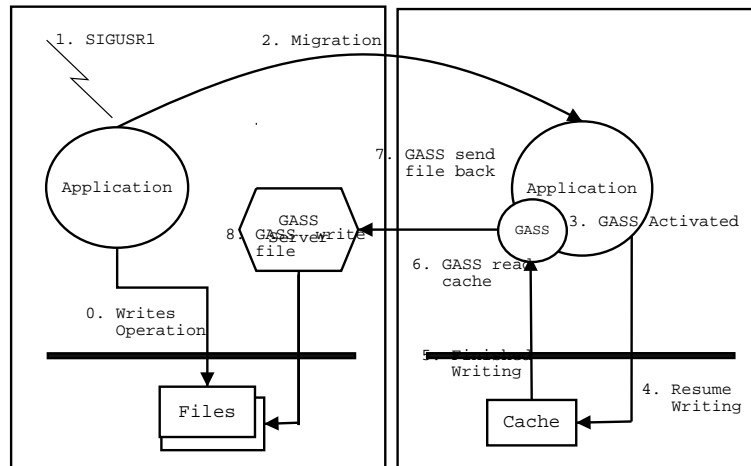


Figure 5.1: Write from Initial to Remote.

At the initial cluster GASS library was not active. Checkpointing would be performed as if there is no support for GASS. File state saving is performed in a normal way,

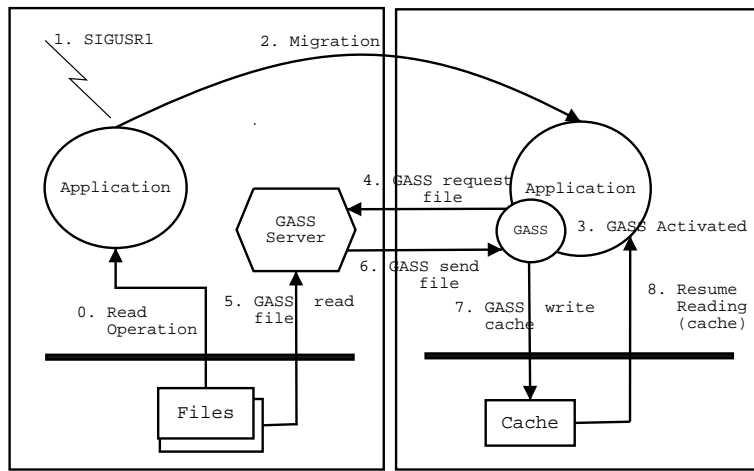


Figure 5.2: Read from Initial to Remote.

i.e. it is treated as a normal file. On restoring the process, we need to activate the GASS Library. The implementation should be able to detect that it was moved from the initial cluster in order to do this. It needs to know the GASS Server of the initial cluster, in order to be able to restore access to the initial files.

The tasks that perform write operations at the time of migration will continue the operation on cache files at the current cluster (i.e. the remote). After writing is finished the cache would be flushed back to the original files as shown in figure 5.1. The GASS library would upload the file needed for task that is interrupted while it performs reading operation, before it could continue reading from the local cache as shown in Figure 5.2.

- Remote Cluster to Remote Cluster.

On checkpointing the process, GASS library has already been activated, We need to deactivate the module after saving the state of the files. The files need to be saved as Globus GASS files. On restoring the process, we also need to activate the GASS Library. It needs to know the GASS Server of the initial cluster, in order to be able to restore its files. For a task that performs writing at the point of migration, what has been written in the cache needs to be flushed back to the initial cluster (see Figure 5.3). The rest of the operation will be performed similar with the writing task that is migrated from initial cluster to remote cluster. Migration of a reading task, from remote cluster to another remote cluster is similar to the case of a reading task migrated from initial cluster to remote cluster (compare Figure 5.4 and Figure 5.2).

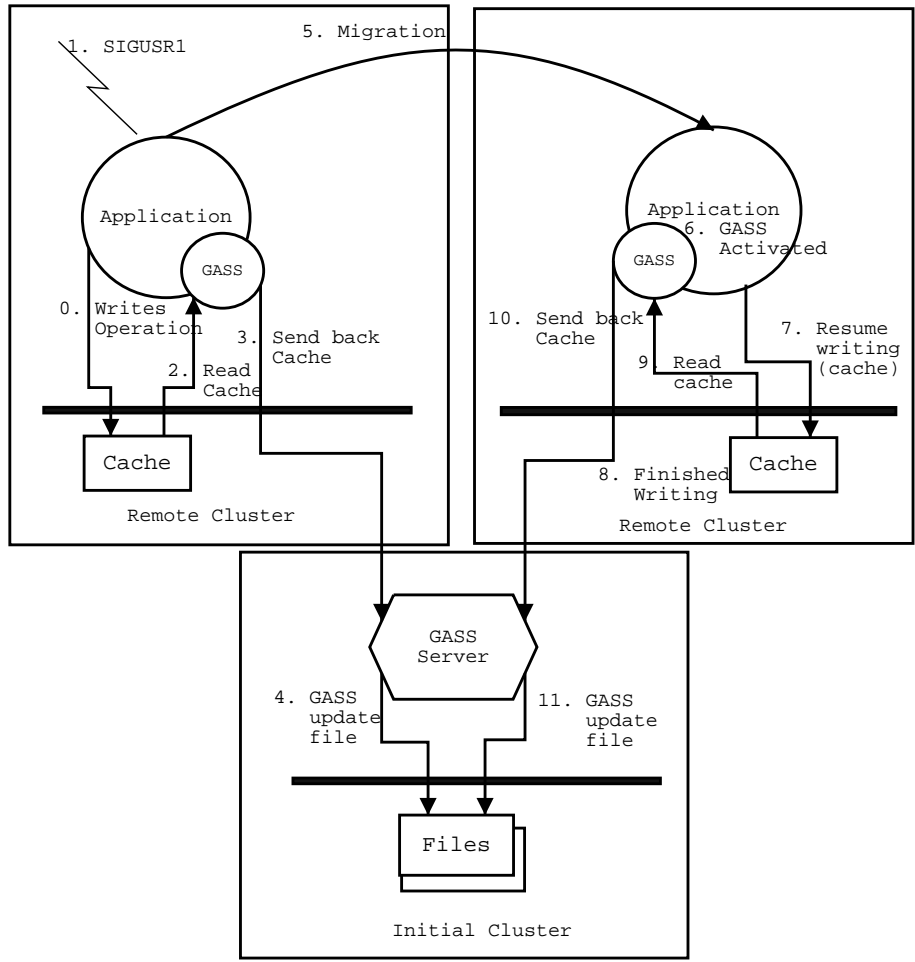


Figure 5.3: Write from Remote to Remote.

They need to perform uploading of the file being read before resuming the operation.

- Remote Cluster to Initial Cluster.

On checkpointing the process we need to deactivate the module after saving the state of the files. File states need to be saved, noticing the fact they are not ordinary files, but GASS files. On restoring the process, we must notice that the application no longer needs to use the GASS libraries and URLs, so we have to remove the URLs. The subsequent file operations will be performed in normal way.

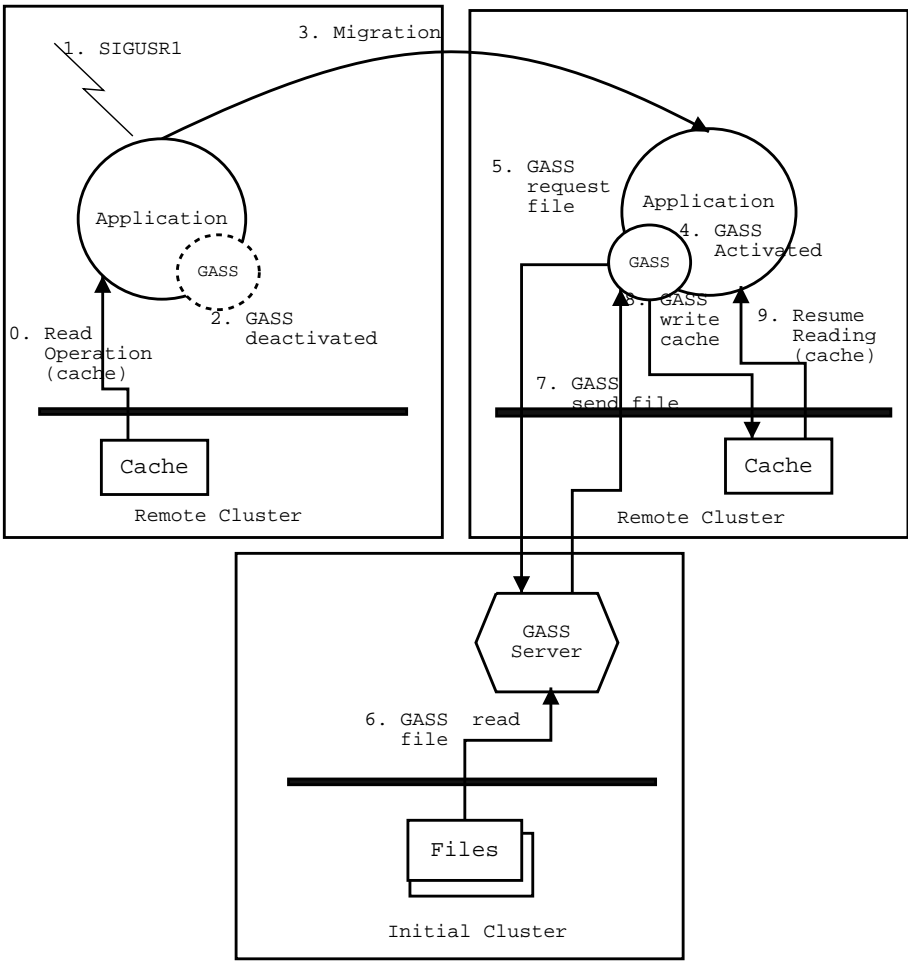


Figure 5.4: Read from Remote to Remote.

Chapter 6

Implementation

To provide the remote file access support, the checkpointing library of Dynamite was modified. The main parts that were modified are the cross-checkpoint data structure, the dynamic loader, the checkpoint handler, and the system call wrapping. In addition to these modifications we need also to start a Globus GASS server on each cluster that we want to include in our system.

6.1 Cross-checkpoint data structure

The cross-checkpoint data structure is modified to have a reference to the GASS functions and to have additional data structures for the GASS files. We need to add certain functions such as GASS module activation/deactivation and GASS counterparts of C file system calls. Using the GASS libraries we have to deal with two different kinds of files. They are the normal files opened directly using original file system calls (normal files) and the files opened using GASS libraries (GASS files).

The GASS library uses cache files that are normal files on its local cluster. As we already described in the previous chapter, these cache files will not be preserved during migration, so there we do not need to save the cache files states. In order to avoid overlapping between normal files opened by the application and cache files opened by GASS libraries we need a separate temporary file state data structure during migration (the `struct filestate gass_fstate` in Figure 6.1). The URL of the GASS server used and the initial cluster

name also need to be preserved during migration (`gass_url` and `initial_cluster` in Figure 6.1).

```
globus_module_descriptor_t * globus_i_gass_file_module;
struct filestate gass_fstate[OPEN_MAX];
char gass_url[MAXPATHLEN];
char initial_cluster[MAXPATHLEN];
```

Figure 6.1: Cross-checkpoint additional data structure.

The data structure for the temporary GASS file is the same type of data structure for a normal file. The states of the files are stored in a structure of file state shown in Figure 6.2. Each available file descriptor is associated with this data structure. We have `OPEN_MAX` available file descriptors, and to keep the states of all files we have an array with `OPEN_MAX` elements. Its index refers to a file descriptor. The `f_privflags` member of the data structure is an indicator whether or not a certain file descriptor is used; this flag will be set whenever application opens a file. The `f_off` represents the offset of the file descriptor; the value will be obtained with `lseek` system call. The `f_oflags` contains the flags that are used when opening the file. The `f_fflags` contains all the flags of the file descriptor, which are obtained using `fcntl` system call.

```
struct filestate {
    int f_privflags;
    off_t f_off;
    int f_oflags;
    int f_fflags;
    char f_path[PATH_MAX];
};
```

Figure 6.2: File state data structure.

6.2 Modification on Dynamic Loader

In Section 3.3.1 we have said that the dynamic loader will load the libraries needed to run a process. It needs to be modified so that the GASS functions are loaded. Another modification required to the dynamic loader is the addition of code to obtain the initial location of a task (see Figure 6.1). This location is stored in the `initial_cluster` variable so that it can be used later on to determine whether or not we need remote file support. The location is determined by using `gethostname` function.

GASS Server Setup

Accessing remote files using the GASS library requires a GASS server to be started on the remote machine. The Globus toolkit provides a tool to start this server externally using the `globus-gass-server` command. It will return the URL of this server with the number of the port used, which will be stored in environment variable `CKPT_GASS_URL`; this environment variable will be loaded in the dynamic loader. We need to start this GASS server on each file server node of the cluster that we wish to perform migration.

6.3 Modification of Checkpoint Handler

In the checkpoint handler additional saving and restoring functions for GASS opened files are needed. In checkpointing mechanism (see Section 3.3.2) saving process registers and DPVM specific clean up part will not be affected. Then additional saving of the GASS file states is performed before saving the states of normal files. The rest of the checkpointing operations which consist of saving signal states, saving dynamic loader state, creating socket connection, and storing checkpoint file will not be affected.

In restoring mechanism, restoring heap and stack will not be affected. After these operations, the checkpoint handler should check whether or not it is in initial cluster. If it is, then no additional operation is needed i.e. the normal files can be restored. Otherwise URL prefix addition needs to be performed. Then restoring of GASS files will be performed and after that the signal states and DPVM user states can be restored.

6.3.1 Saving GASS File States

The GASS library uses cache mechanism that basically uses a normal local file. While the GASS support module is active, these cache files exist. If all GASS files are closed or the module is deactivated, necessary cached file operations (sending back the cache) will be performed and cache will be removed.

Saving the GASS file states is performed separately from the normal files. On checkpointing, GASS files need to be closed and deactivated to flush all the cached file operations.

This operation has to be performed before saving the state of normal files. In this way we avoid saving unnecessary cache files, which will continue to exist but will be no longer useful after migration.

Saving of the GASS files will be performed in separate data structures i.e. `ckpt_gass_fstate`. The reason for this is that while performing the deactivation of the module, the GASS libraries will open and close several cache files using the wrapped system calls. If the data is not separated during this deactivation process, the wrapped system calls might overwrite the GASS files that we want to save.

```
for all files opened {
  if it is a GASS file {
    store the file states [offset, open flags, priv flags]
      in temporary gass file states array
    close file using globus_gass_close
  }
}
if module gass is activated
  deactivate gass support with globus_module_deactivate
```

Figure 6.3: Saving GASS file states.

6.3.2 URL prefixing

Whenever a task is migrated from the initial cluster to a remote cluster, the URL of its initial cluster (`CKPT_GASS_URL` environment variable) needs to be prepended to the files opened by this task. The URL prefixing function performs this addition. It will not add an URL if the migration is from one remote cluster to another remote cluster. Files which are opened locally in initial cluster were saved in the normal files data structure. Therefore after URL prefixing, they need to be copied into the temporary GASS data structure during migration (Figure 6.4). This is needed in order to make sure that all files will be restored with the GASS function calls.

6.3.3 Restoring GASS File States

Restoring GASS files will be performed after restoring normal files. As explained above, activation of GASS module while restoring will generate cache files. In this case we want

```

ckpt_add_url_prefix() {
  for all files opened{
    add the CKPT_GASS_URL to the filename
    store the file state in gass temporary file state
    occupy the file descriptor
  }
}

```

Figure 6.4: URL prefixing.

to avoid that the normal files overwrite the cache files. If we restore normal files before the GASS files, the descriptor needed by the normal files might have been occupied by some of the GASS cache files.

```

if (gass file counter > 0)
  activate gass support with globus_module_activate
else
  return
for all files in temporary gass file states array {
  if at initial cluster and it is URL prefixed files {
    remove the URL prefix
    open as a normal file
    decrease the gass file counter
  }
  else
    open the file using the globus gass open
    duplicate the file descriptor as the old one
    store back the temporary gass files to the file states
}
if there are no more gass files
  deactivate gass support with globus_module_deactivate
}

```

Figure 6.5: Restoring globus file states.

6.4 File System Calls Wrapping

The system call wrappers has to determine whether to perform remote file access or just local file operations. The decision to use GASS file support is based on the file name (whether or not it is a URL filename) and on the current location of the task. If the task is running on a remote cluster, an URL of the initial cluster (CKPT_GASS_URL environment variable) needs to be added.

The GASS file libraries eventually will perform the original file system calls. Wrapping all calls with the GASS file functions will lead to an infinite recursive calling of itself. To avoid this, we restrict the wrappings only for the system calls that are called directly from the application, not from the GASS file functions. This can be performed by using a flag, every time we want to call GASS file functions. This flag indicates that we don't have to use the GASS file functions, but to use the normal file operation instead. The GASS module activation and deactivation functions also use these flags, since those functions also perform file system operations.

6.4.1 Wrapper for system call open

There are two cases that open system calls need to be redirected to GASS file operations. The first case is when it is already an URL file (the user knows about GASS support, and wanted to access URL files from the application). The second case is when the task that has already been migrated to a remote cluster performs a file open. Unaware of the fact that it has been migrated, the task needs remote file support to perform the file open, since it wants to access the files on its initial cluster.

```
if not a call from globus and not call from pvm and
  (is url name or is not at initial cluster) {
  if ( gass module is not activated )
    activate the gass module support
  increase the globus gass file counter
  if (it is already an URL)
    open with globus_gass_open --> fd
  else
    if (not at initial cluster)
      add URL prefix
      open with globus_gass_open --> fd
  } else {
    open with normal open
  }
}

save the file information in the file states array

return fd
```

Figure 6.6: Wrapper for system call open.

In this wrapper, the file will be opened in the normal way if it is a system call from the GASS

libraries, PVM libraries, or a system call in the initial cluster. Files which are already an URL or files which are opened in the remote cluster will be opened with GASS functions. Activation of the GASS module must be performed at the first time this wrapper function is called. The file states have to be saved at the end of the wrapper. This is the main purpose of the system call wrapping for additional remote file support (Figure 6.6).

6.4.2 Wrapper for system call `close`

The wrapper of system `close` also needs to avoid an infinite recursion by avoiding to wrap calls from GASS library functions. Only files which are URL and not called from GASS library are closed using `globus_gass_close`. The GASS module needs to be deactivated if there are no more GASS file opened (Figure 6.7).

```
if [ not from globus and is URL file ] {
    close with globus_gass_close
    decrease the gass file counter
    if the gass file counter == 0
        deactivate the gass file support
}
else
    close with normal close
```

Figure 6.7: Wrapper for system call `close`.

Chapter 7

Testing and Performance

Measurement

7.1 Testing Environment

Testing the remote file system access is performed within the DAS-2 (Distributed ASCI Supercomputer) cluster. DAS-2 is a wide-area distributed cluster designed by the Advanced School for Computing and Imaging (ASCI). The DAS-2 machine is used for research on parallel and distributed computing by five Dutch universities:

- Free University (VU),
- Leiden University,
- University of Amsterdam (UvA)
- Delft University of Technology,
- University of Utrecht

DAS-2 consists of five clusters, located at the five universities. The cluster at the Free Universiteit contains 72 nodes, the other four clusters have 32 nodes (200 nodes with 400 CPUs in total). The system was built by IBM and the operating system of the DAS-2 cluster is RedHat Linux.

Each node in the cluster contains:

- Two 1-Ghz Pentium-IIIs with at least 512 MB RAM (1 GB for the nodes in Leiden and UvA, and 2 GB for two "large" nodes at the VU)
- A 20 GByte local IDE disk (80 GB for Leiden and UvA)
- A Myrinet interface card
- A Fast Ethernet interface (on-board)

The nodes within a local cluster are connected by a Myrinet-2000 network, which is used as high-speed interconnect, mapped into user-space. In addition, Fast Ethernet is used as OS network (file transport). The five local clusters are connected by the Dutch university Internet backbone (SurfNet).

7.2 Correctness and Stability Tests

7.2.1 Sequential Test

To test the correctness of the remote file access support implementation on Dynamite's checkpoint library, we first use a sequential application which performs subsequent file operations on several files. Checkpointing is performed when the application has opened several files for reading, or has opened several files for writing. In this test we make sure that the file operations can be resumed normally after the checkpoint. For this test we use the file method for checkpointing.

The checkpoint file is migrated manually by copying the file to remote locations, and then it is executed directly from the shell. The test is repeated several times between file servers in the DAS-2 clusters. From all the sequential tests that have been performed, the remote file support is working properly. All the file operations on the initial cluster are completed successfully after the migration.

7.2.2 Parallel Test

Parallel tests are performed to check whether the remote file access support works properly in PVM applications which need remote file operation. A master/slave application where

multiple slaves were spawned was tested. The slaves perform a simple computation and need to perform communication with the master. In addition to these operations, each of them writes to output file, or reads from the shared input file.

Migration is performed according to the various scenarios considered in Chapter 5. The socket migration method is necessary to perform this test. The focus of this testing is to make sure that checkpoint and migration of the process will preserve the file operations without hindering the existing cross cluster process migration mechanism in the Dynamite. This parallel tests show that the implementation of remote file access support is not completely stable. There is a limitation that after several initial migration, the migration tends to fail. We suspect that the reason for this failure is because after several migrations, the temporary checkpoint files generated by socket migration sometimes have an unusually large sizes or unusually small sizes compared to the size when the migration is succeed.

7.3 Performance measurement

Performance of remote file access support is measured using sequential programs. The first sequential test writes from the memory to a file on a remote cluster. In this test, the time needed to perform a write in the cache of the remote cluster and the time to transfer the file back into the initial cluster will be measured. The second test performs a read from a file on a remote cluster to memory. For this test the time to load the file to the cache and the time to read from the cache are measured. The last test is to perform a copy operation of a file in the remote cluster. The operations involved in this test will be to read the file in the cache of the local cluster, read and write to the local cache and send the cache back to the remote cluster. The time needed for these operations will be measured.

The tests are performed between the file servers in the DAS2 clusters, which are connected with the internet backbone. Tests are repeated using different file sizes ranging from 1 kilobyte to 32 Mbyte. File size is increased by doubling it in each experiment, and for each file size the measurement was repeated 10 times. Log scale is used in graphing the results of the measurements in order to evenly separate the measurement points. Error bars are used and the average of the ten measurements for each file size is shown.

7.3.1 Memory to File

In the memory to file test (Figure 7.1), for small files, the time measured for sending the cache back to the local cluster (after the files are closed) does not grow linearly with the size of the file. In this experiment, it increases linearly with the size of the file only for files that are larger than 128 KBytes. This behaviour is due to the latency on starting the file transfer. The transfer rate of this memory to file test is increasing as the file size grows, and eventually reaches a certain value as shown in Figure 7.2. As the file size reaches 1 Mbyte the transfer rate becomes stable. We will summarize the latencies and the transfer rates of these tests in Tables 7.1 and 7.2.

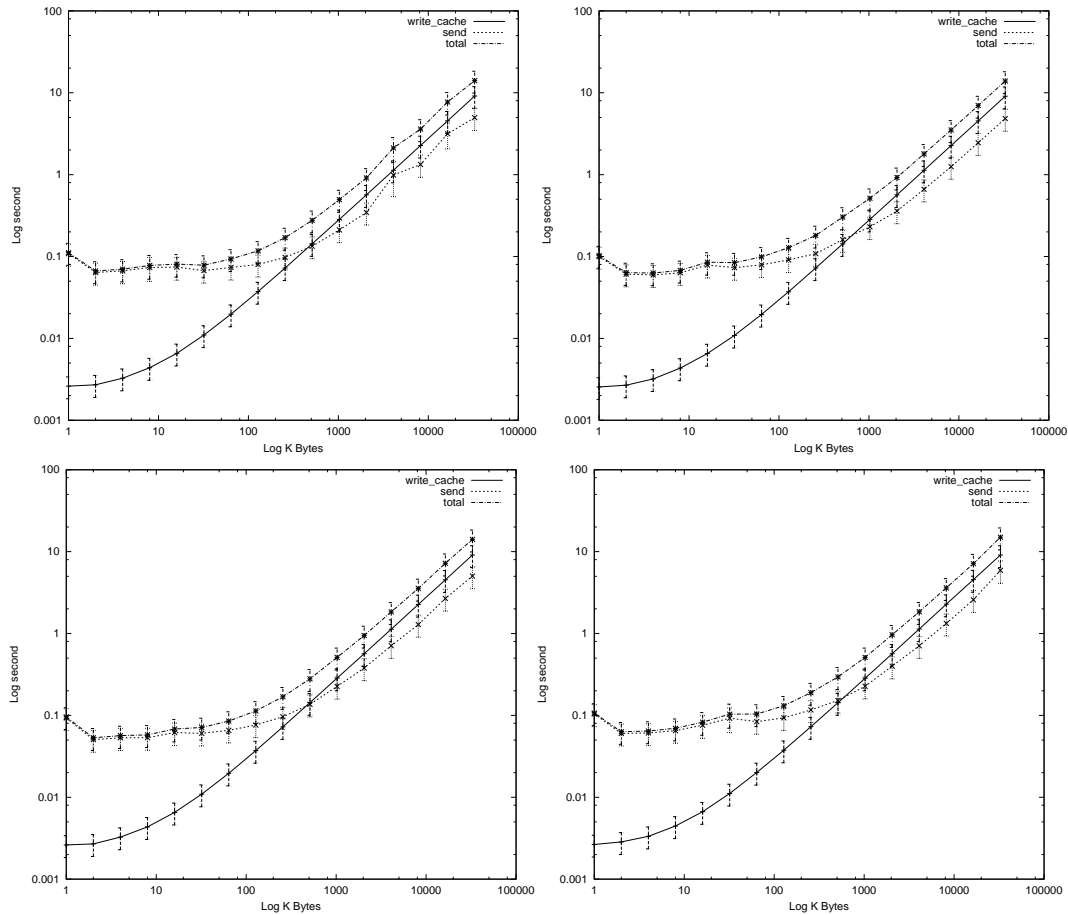


Figure 7.1: Time for memory to file tests with remote clusters : fs0 at Free University (upper left), fs1 at University of Leiden (upper right), fs2 at Nikhef (down left) and fs3 at TU Delft (down right) to local cluster fs4 at Utrecht.

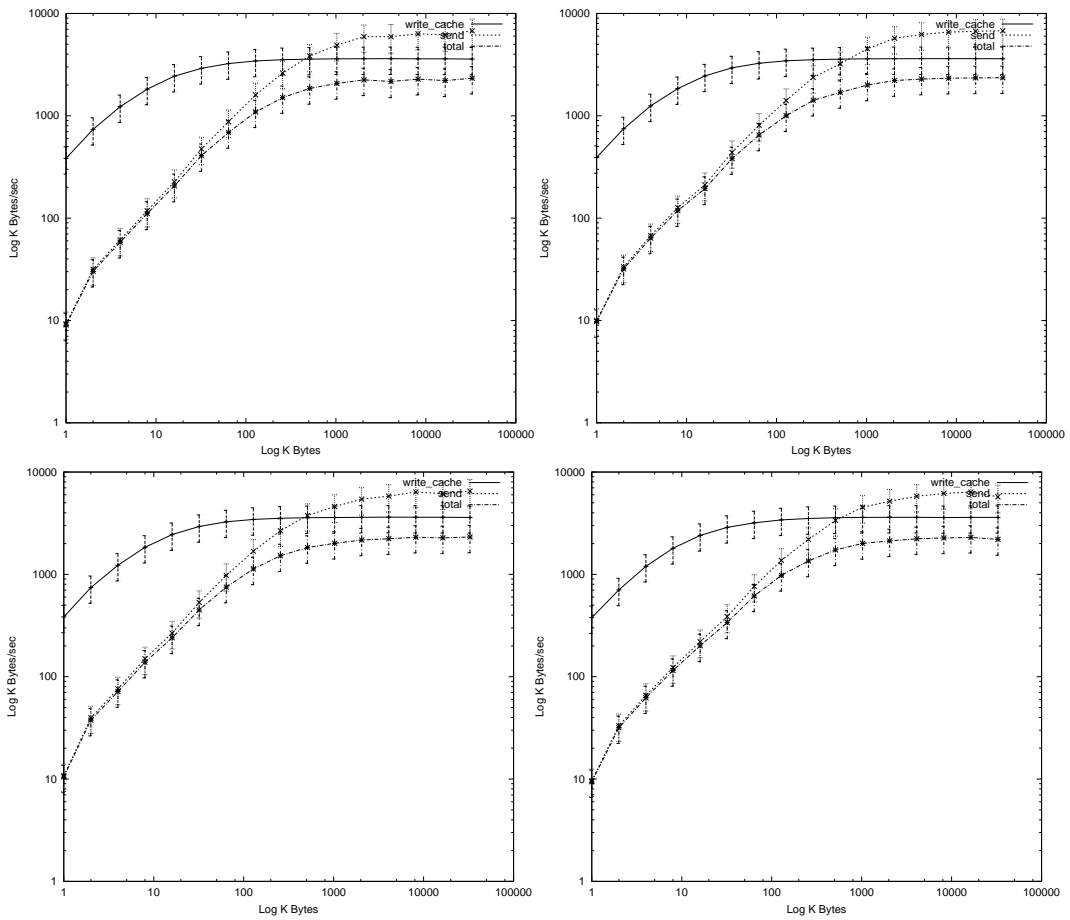


Figure 7.2: Transfer rate for memory to file tests with remote clusters : fs0 at Free University (upper left), fs1 at University of Leiden (upper right), fs2 at Nikhef (down left) and fs3 at TU Delft (down right) to local cluster fs4 at Utrecht.

7.3.2 File to Memory

In the file to memory test (Figure 7.3), the time needed for loading the files to the cache (when the file is opened) also does not grow linearly with the file size when the files are small. In this experiment, it is only when the size of the file is larger than 128 KBytes that the loading time increases linearly with the size of the file. Meanwhile the time needed for reading from the cache grows linearly with the size of the file. The transfer rate of this file to memory test is also increasing and it reaches a certain stable value after the size of the file is 1 Mbyte, as shown in Figure 7.4.

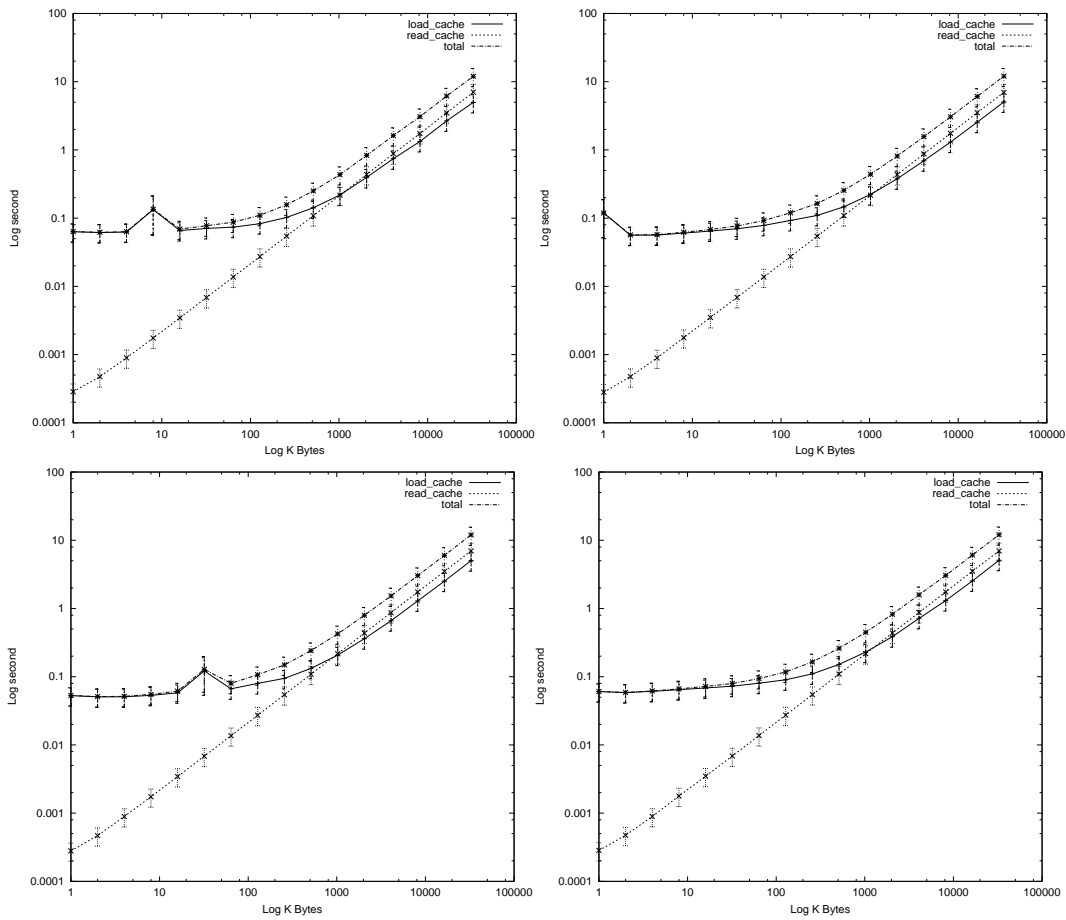


Figure 7.3: Time for file to memory tests with remote clusters : fs0 at Free University (upper left), fs1 at University of Leiden (upper right), fs2 at Nikhef (down left) and fs3 at TU Delft (down right) to local cluster fs4 at Utrecht.

7.3.3 File to File

For the file to file test (Figure 7.5) the dominant cost needed is for loading the data to the cache and sending it back to the remote cluster. For both of these cases, we also observe a latency. The reading and writing from the local cache takes a relatively small amount of time compared to the loading and sending back of the cache. The transfer rate of this file to file test is also increasing and it reaches a certain stable value after the size of the file is 1 Mbyte, as shown in Figure 7.6.

From these tests we try to observe the latencies that occur in all cases and summarize the results in Table 7.1. The results in this table are the averages of time needed to send files with sizes from 1 Kbyte up to 64 Kbyte, where we observe that the time needed to send or to load the cache is not increasing. This result shows that for small files the latency that occurs

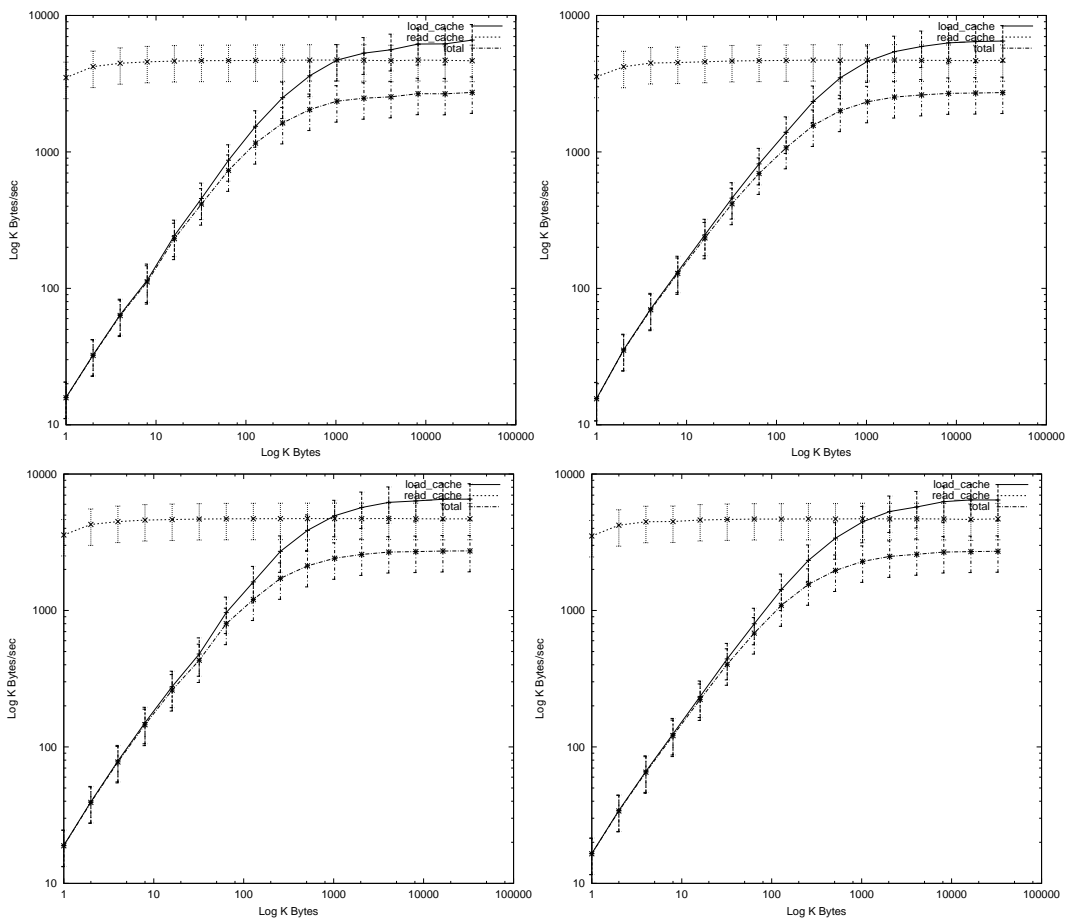


Figure 7.4: Transfer rate for file to memory tests with remote clusters : fs0 at Free University (upper left), fs1 at University of Leiden (upper right), fs2 at Nikhef (down left) and fs3 at TU Delft (down right) to local cluster fs4 at Utrecht.

is quite significant. For a 1 KByte file, for example, with latency around 0.075, we could only achieve transfer rate at ± 13 KBps. Meanwhile, for the large files, when this latency is no longer significant, we observe that the transfer rate could achieve ± 2 MBps as shown in Table 7.2. The results in this table are obtained by averaging the transfer rate of files with size larger than 4 Mbyte.

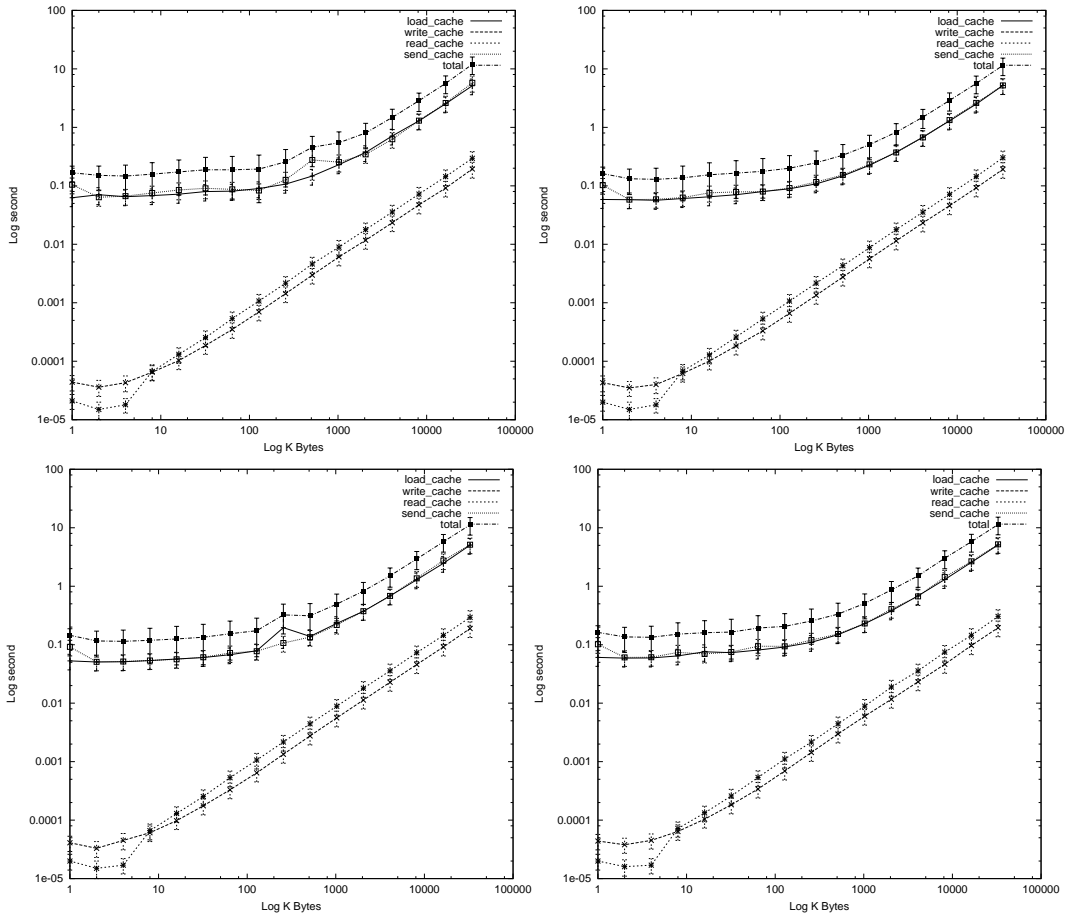


Figure 7.5: Time for file to file tests with remote clusters : fs0 at Free University (upper left), fs1 at University of Leiden (upper right), fs2 at Nikhef (down left) and fs3 at TU Delft (down right) to local cluster fs4 at Utrecht.

	Memory to File	File to Memory	File to File	
	Send Back	Load	Send back	Load
fs0 VU	0.075 ± 0.025	0.075 ± 0.015	0.071 ± 0.007	0.081 ± 0.001
fs1 Leiden	0.072 ± 0.022	0.073 ± 0.014	0.064 ± 0.005	0.073 ± 0.001
fs2 NIKHEF	0.064 ± 0.028	0.062 ± 0.014	0.056 ± 0.003	0.062 ± 0.001
fs3 DELFT	0.066 ± 0.007	0.077 ± 0.016	0.067 ± 0.007	0.076 ± 0.001

Table 7.1: Latency time (in seconds)

	Memory to File	File to Memory	File to File
fs0 VU	2.65 ± 0.03	2.25 ± 0.06	1.67 ± 0.08
fs1 Leiden	2.63 ± 0.01	2.29 ± 0.01	1.67 ± 0.03
fs2 NIKHEF	2.71 ± 0.01	2.28 ± 0.02	1.58 ± 0.06
fs3 DELFT	2.66 ± 0.01	2.24 ± 0.05	1.55 ± 0.09

Table 7.2: Transfer rate (in MByte/sec)

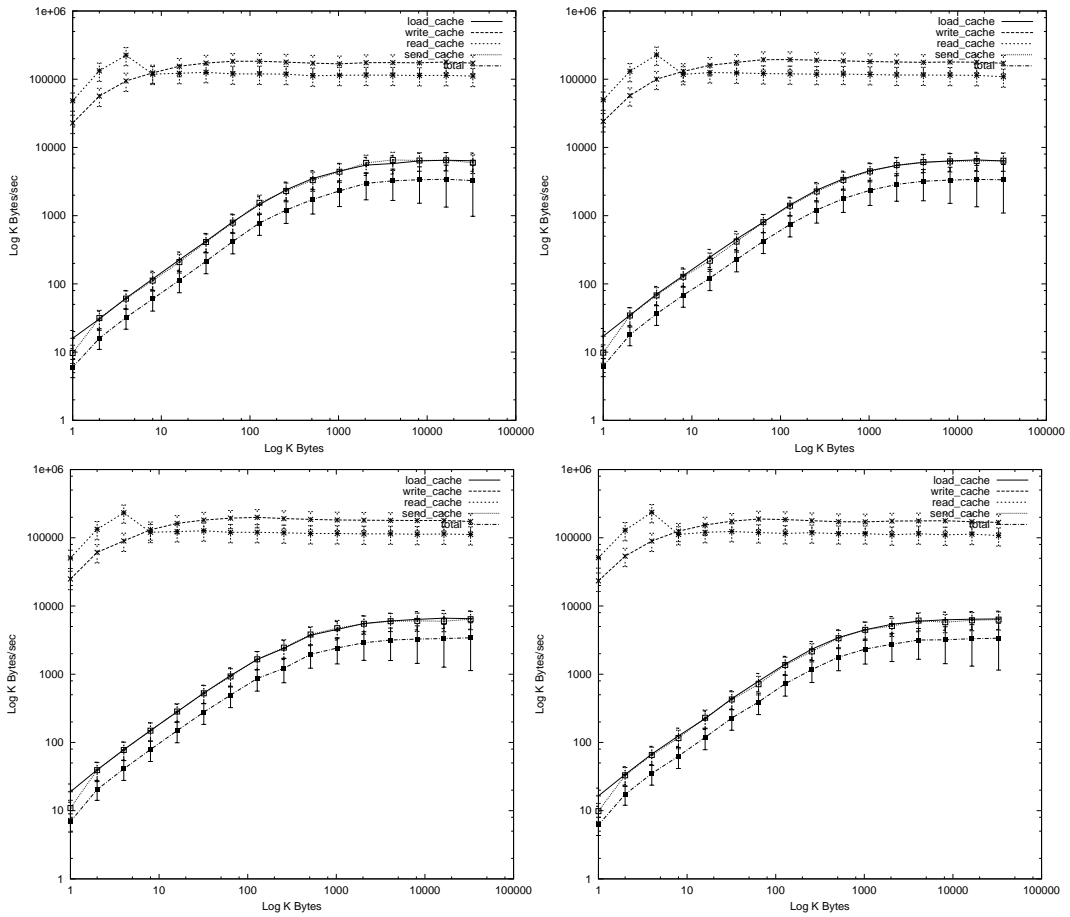


Figure 7.6: Transfer rate for file to file tests with remote clusters : fs0 at Free University (upper left), fs1 at University of Leiden (upper right), fs2 at Nikhef (down left) and fs3 at TU Delft (down right) to local cluster fs4 at Utrecht.

Chapter 8

Summary and Future Work

8.1 Summary

In this thesis we have described how we designed and implemented the remote file access support for cross cluster migration in the Dynamite system. This thesis was developed in an environment which is not a specialized distributed operating system, so that migration across multiple clusters implies that we no longer have a shared file system.

Continuing the work of Jinghua Wang, which provides a socket migration method for cross cluster migration, this thesis work eliminates the prerequisite of a shared file system. We provide this remote data access by integrating the Global Access to Secondary Storage (GASS) libraries provided by the Globus Toolkit into the existing Dynamite libraries. One of the reasons for using GASS is that it is designed to achieve a high performance for the basic file access pattern of an application. The GASS library is designed to provide a support for default data movement strategies that are common in wide area computing environments.

The implementation of this file support is made to be transparent, no additional modification of the user's application is needed. This support was implemented in the checkpoint library of the Dynamite system. With this support, a parallel application running in the Dynamite system can preserve its file operations while being migrated across clusters of workstations that do not share file systems. The checkpoint library could also be used for a non-Dynamite application. This means that this implementation can also be useful for sequential applications that need a checkpointing mechanism.

Correctness and stability tests has been performed for sequential and parallel applications. This is to make sure that checkpoint and migration of the process will preserve the states of the application's file operations. The tests are also performed to guarantee that the additional remote file access support will not cause any conflict with the existing cluster process migration mechanism in Dynamite. Sequential tests show a good stability, meanwhile there are still some limitation on the parallel stability tests. Simple performance tests have shown that the GASS mechanism will induce some additional overhead on loading and sending back the cache for file operations.

8.2 Future Work

This work only uses the remote data access parts of the Globus toolkit libraries. There are still many other possible features of Globus that can be exploited and may have some benefit to be incorporated into Dynamite. For the resource discovery, for example, there is meta directory service that can provide information about available resources in a grid environment.

Currently the ubiquitous Globus toolkit which is accepted as standard middleware for performing grid computing is not supporting PVM as a type of job to be submitted to a grid resource. There is only support for MPI applications. In order to be able to participate in the wave of Grid computing the Dynamite library needs to be extended to support not only PVM applications, but also MPI applications.

The current implementation of Dynamite uses the PVM 3.3.x as the basis of the development. Some PVM users are now already acquainted with PVM 3.4, which provides more functionality and flexibility. In addition to this, the current implementation of the checkpoint library used by Dynamite uses glibc 2.0. Since nowadays applications are developed using a newer version of library (both PVM and glibc), this fact is limiting the usage of the Dynamite system. A future version of Dynamite implementation which uses the latest library and supports not only PVM but also MPI would be very desirable.

Bibliography

- van Albada, G. D., Clinckemaillie, J., Emmen, A. H. L., Gehring, J., Heinz, O., van der Linden, F., Overeinder, B. J., Reinefeld, A. and Sloot, P. M. A. [1999]. Dynamite - blasting obstacles to parallel cluster computing. In Boasson, M., Kaandorp, J. A., Tonino, J. F. M. and Vosselman, M. G. (Eds.), *Proceedings of the fifth annual conference of the Advanced School for Computing and Imaging ASCI, June 15–17, 1999* (pp. 31–37). Delft: ASCI. URL <http://citeseer.nj.nec.com/247187.html>.
- Baker, M., Buyya, R. and Laforenza, D. Grids and grid technologies for wide-area distributed computing. URL <http://citeseer.nj.nec.com/458291.html>.
- Barak, A., Laden, O. and Braverman, A. [1995]. The NOW MOSIX and its Preemptive Process Migration Scheme. Bulletin of the IEEE Technical Committee on Operating Systems and Application Environments.
- Bester, J., Foster, I., Kesselman, C., Tedesco, J. and Tuecke, S. [1999]. GASS: A Data Movement and Access Service for Wide Area Computing Systems. In *Sixth Workshop on I/O in Parallel and Distributed Systems*. URL <http://www.globus.org/research/papers.html#gass>.
- Casas, J., Clark, D., Konuru, R., Otto, S. W., Prouty, R. M. and Walpole, J. [1995]. MPVM: A Migration Transparent Version of PVM. In *Usenix Computing System*, Volume 8 (pp. 171–216).
- Czarnul, P. and Krawczyk, H. [1999]. Dynamic Assignment with Process Migration in Distributed Environments. In *Proceedings of the 6th European PVM/MPI User's Group Meeting* (pp. 509–516). Springer Verlag.

- Dan, P., Dongshen, W., Youhou, Z. and Meiming, S. [1999]. Quasi-asynchronous Migration: A Novel Migration Protocol for PVM tasks. In *Operating Systems Review*, Volume 33 (pp. 5–14).
- Foster, I. and Kesselman, C. [1999]. *The Grid: Blue print for a new computing infrastructure*. Morgan Kaufmann Publisher.
- Geist, A., Beguelin, A., Dongarra, J., Jiang, W., Manchek, R. and Sunderam, V. [1994]. *PVM: Parallel Virtual Machine - A Users Guide and Tutorial for Network Parallel Computing*. MIT Press.
- Geist, G. A. [1994]. Cluster computing: The wave of the future? In *PARA* (pp. 236–246). URL <http://citeseer.nj.nec.com/geist94cluster.html>.
- Genias Software GmbH. CODINE: Computing in Distributed Networked Environments. URL <http://www.genias.de/genias/english/codine>.
- Iskra, K. A., Hendrikse, Z. W., van Albada, G. D., Overeinder, B. J., Sloot, P. M. A. and Gehring, J. [2000a]. Experiments with migration of message-passing tasks. In *Research and Development for the Information Society* (pp. 295–304).
- Iskra, K. A., van der Linden, F., Hendrikse, Z. W., Overeinder, B. J., van Albada, G. D. and Sloot, P. M. A. [2000b]. The implementation of Dynamite - an environment for migrating PVM tasks. In *Operating System Review*, Volume 34 (pp. 40–55).
- Katramatos, D., Chapin, S. J., Hillman, P., Fisk, L. A. and van Dresser, D. [1998]. Cross-operating system process migration on a massively parallel processor. Technical Report CS-98-28. URL <http://citeseer.nj.nec.com/361958.html>.
- Lawrence, R. [1998]. A Survey of Process Migration Mechanism. Technical report. URL <http://citeseer.nj.nec.com/361958.html>.
- Litzkow, M., M.Linvy and Mutka, M. W. [1988]. Condor – a hunter of idle workstations. In *Proceedings 8th International Conference on Distributed Computing System*. San Jose, California.
- Milojicic, D., Douglis, F., Paindaveine, Y., Wheeler, R. and Zhou, S. [2000]. In *ACM Computing Surveys*.

- Mullender, S. J., van Rossum, G., van Renesse, R. and van Staveren, H. [1990]. Amoeba – a distributed operating system for the 1990s. In *IEEE Computer*, number 23(5) (pp. 44–53).
- Ousterhout, J., Cherenon, A., Douglass, F., Nelson, M. and Welch, B. [1988]. The Sprite Network Operating System. In *IEEE Computer 21* (pp. 23–36).
- Robinson, J., Russ, S., Flachs, B. and Heckel, B. [1996]. A task migration implementation of the message passing interface. In *Proceedings of the 5th IEEE International Symposium on High Performance Distributed Computing* (pp. 61–68).
- Stellner, G. and Pruyn, J. [1995]. Resource Management and Checkpointing for PVM. In *Proceedings of the 2nd European User's Group Meeting* (pp. 131–136).
- Tan, C., Wong, W. and Yuen, C. [1999]. tmpvm - task migratable pvm. URL cite-seer.nj.nec.com/442497.html.
- Tanenbaum, A. [1992]. *Modern Operating Systems*. Prentice Hall, Englewood Cliffs, New Jersey.
- Theimer, M. M., Lantz, K. A. and Cheriton, D. R. [1985]. Preemptable remote execution facilities for the v-system. In *Proceedings 10th ACM Symposium on Operating System Principles*.
- Walker, B. and Popek, G. [1983]. The locus distributed operating system. In *Proceedings 9th ACM Symposium on Operating System Principles* (pp. 49–70). Bretton Woods, New Hampshire.
- Wang, J. [2001]. Cross-cluster migration of parallel tasks. Master's thesis, University of Amsterdam, The Netherlands.
- Zayas, E. [1987]. Attacking the process migration bottleneck. In *In Proceedings 11th ACM Symposium on Operating System Principles* (pp. 13–24).