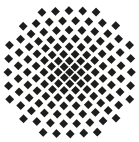

H L R S

Institut für
Höchstleistungsrechnen

FORSCHUNGS- UND ENTWICKLUNGSBERICHTE

PROCESS MIGRATION IN A PARALLEL ENVIRONMENT

Adrian Reber

ISSN 0941 - 4665 May 2016 HLRS-16

Höchstleistungsrechenzentrum
Universität Stuttgart
Prof. Dr.-Ing. Dr. h.c. Dr. h.c. Prof. E.h. M. M. Resch
Nobelstrasse 19 - 70569 Stuttgart
Institut für Höchstleistungsrechnen

PROCESS MIGRATION IN A PARALLEL ENVIRONMENT

von der Fakultät Energie-, Verfahrens- und Biotechnik
der Universität Stuttgart zur Erlangung der Würde eines
Doktor-Ingenieurs (Dr.-Ing.) genehmigte Abhandlung

vorgelegt von

Adrian Reber
aus Esslingen

Hauptberichter:	Prof. Dr.- Ing. Dr. h.c. Dr. h.c. Prof. E.h. Michael M. Resch
Mitberichter:	Prof. rer. nat. Peter Väterlein
	Prof. Dr.-Ing. Stefan Wesner
Tag der mündlichen Prüfung:	03.12.2015
CR-Klassifikation:	I.3.2, I.6.6

D93

ISSN 0941 - 4665 May 2016 HLRS-16

Contents

1 Introduction **21**
1.1 Motivation 21
1.2 Goals 22
1.3 Structure of this work 22

2 State of the Art **25**
2.1 Hypervisor 28
2.2 Para-Virtualization and Container Based 29
2.3 I/O Accesses 29
2.4 Process Migration 30

3 Process Migration **33**
3.1 The Process 33
3.1.1 Process Management 34
3.1.2 Memory Management 35
3.1.3 File Management 36
3.2 Memory Transfer Methods 37
3.2.1 Memory Transfer During Migration 38
3.2.2 Memory Transfer After Migration 39
3.2.3 Memory Transfer Before Migration 40
3.3 Preemptive Migration 41
3.3.1 Single System Image 43
3.4 Checkpoint/Restore Migration 43
3.4.1 Berkeley Lab Checkpoint/Restart 46

3.4.2 Distributed MultiThreaded Checkpointing 47
3.4.3 Kernel-Space-Based 48
3.4.4 User-Space-Based 50
3.5 Post-Copy Migration vs. Pre-Copy Migration 53
3.6 Process Migration 54

4 Parallel Process Migration **57**
4.1 Related Work 57
4.2 Parallel Process Migration 59
4.3 Open MPI 62

5 Results **65**
5.1 Approaches and Implementation 66
5.1.1 Requirements - Constraints - Limitation 66
5.1.2 Kernel-Space-Based Process Migration 69
5.1.3 User-Space-Based Process Migration 70
5.1.4 Process Identifier 72
5.1.5 Environment Variables 73
5.1.6 Security 74
5.1.7 Implementation within Open MPI 75
5.1.8 Re-Parenting 77
5.1.9 Output Redirection 78
5.2 Test and Validation Methods 79
5.3 UDP Ping Pong - udpp 80
5.4 memhog 81
5.4.1 Via Ethernet with a local SSD 82
5.4.2 Via InfiniBand with a local SSD 84
5.4.3 Via Ethernet with a local RAM drive 85
5.4.4 Via InfiniBand with a local RAM drive 87
5.4.5 Test Case Summary with *memhog* 87
5.5 FENFLOSS 89

6 Conclusion and Outlook **97**
6.1 Conclusion 97

6.2 Outlook 102

Bibliography **105**

Glossary

AIX Advanced Interactive eXecutive. 45

BLCR Berkeley Lab Checkpoint/Restart. 46–49, 58

BTL Byte Transport Layer. 63

C/R checkpointing and restoring. 38, 40, 43–52, 55, 57–59, 62, 63, 66, 67, 69–72, 75, 98–103

CLI command-line interface. 73

compute cluster (or cluster) is the combination of all components which are part of a compute cluster. 11, 22

computer simulation is used to simulate a system with the help of programs running on one or multiple computers. 21

CPU Central Processing Unit. 22, 23, 25, 28–30, 32–35, 39, 41, 55, 67, 69, 79, 98, 100, 102, 103

CRIU Checkpoint/Restore in Userspace. 51, 52, 70, 71, 76–79, 98, 99, 101

CRS Checkpoint/Restart Service. 75, 76

DMTCP Distributed MultiThreaded Checkpointing. 47, 48, 72, 100

FENFLOSS Finite Element based Numerical Flow Simulation System. 89, 90, 92, 93, 95, 101

FPU Floating Point Unit. 69

guest is one of (possible) many virtual machines running on a host which is providing a platform for virtualization with the help of a hypervisor. 10

HNP Head Node Process. 78

host (or physical host) is the actual hardware on which multiple virtualized guest systems are running. 10, 25

HPC High Performance Computing. 21–23, 28–30, 32, 37, 38, 43, 45–47, 57, 59, 65, 67, 69, 73–75, 97–103

I/O Input/Output. 30, 58, 76, 103

IP Internet Protocol. 31, 80, 81

ISA Instruction Set Architecture. 31, 66, 67, 69, 100, 101

iSCSI Internet Small Computer Systems Interface. 27

KVM Kernel-based Virtual Machine. 27, 53, 54

LAN Local Area Network. 31

MCA Modular Component Architecture. 75

MPI Message Passing Interface. 22, 23, 32, 57–63, 90, 98, 101, 102

NAS Network-Attached Storage. 27

NFS Network File System. 79

node is a generic term referring to a single computer in compute cluster. 21

ORTE Open Run-Time Environment. 62, 77

OSPF Open Shortest Path First. 31

PID Process Identifier. 35, 48, 72–74, 78, 99, 100

QDR Quad Data Rate. 79

RAM Random-access memory. 35, 68, 69, 71, 79, 82, 85, 90

RPC Remote Procedure Call. 77

SAN Storage Area Network. 27

SMTBF system mean time between failure. 58, 59

SR-IOV Single Root I/O Virtualization and Sharing. 30

SSD Solid-state drive. 68, 69, 71, 79, 82, 84, 85, 89

SSH Secure Shell. 75

SSI single-system image. 43, 57

SUPER-UX Operating system running on NEC SX architecture supercomputers. 45

system see compute cluster. 22

UDP User Datagram Protocol. 65, 80, 81

VLAN Virtual Local Area Network. 31

WAN Wide Area Network. 31

List of Figures

3.1 Process Table 34
3.2 Virtual Memory - "page table" 36
3.3 Memory Transfer During Migration 38
3.4 Memory Transfer After Migration 39
3.5 Memory Transfer Before Migration 40
3.6 Preemptive Migration 42

4.1 MPI migration starting point 59
4.2 MPI migration complete node 60
4.3 MPI migration load balancing 61
4.4 Open MPI layers 62
4.5 Open MPI process tree 63

5.1 Direct vs. Indirect Migration 68
5.2 Open MPI process tree 75
5.3 Open MPI initiate checkpoint 76
5.4 Open MPI initiate restart 77
5.5 Open MPI spawn opal-restart 77
5.6 Open MPI calls CRIU for restore 78
5.7 Open MPI process tree after restore 78
5.8 *udpp* migration 81
5.9 Comparison of migration time via Ethernet using SSDs with and without pre-copy 84

5.10 Comparison of migration time via InfiniBand using SSDs with and without pre-copy 85
5.11 Comparison of migration time via Ethernet using a RAM drive with and without pre-copy 87
5.12 Comparison of migration time via InfiniBand using a RAM drive with and without pre-copy 88
5.13 Comparison of migration time using pre-copy 90
5.14 Comparison of migration time without pre-copy 91
5.15 FENFLOSS memory transferred during migration with and without pre-copy 92
5.16 FENFLOSS migration duration with and without pre-copy . . . 94

List of Tables

3.1 Checkpoint/Restart implementations overview 52

5.1 Memory bandwidth measured using the STREAM benchmark . 79
5.2 Comparison of migration time via Ethernet using SSDs with and without pre-copy 83
5.3 Comparison of migration time via InfiniBand using SSDs with and without pre-copy 86
5.4 Comparison of migration time via Ethernet using a RAM drive with and without pre-copy 86
5.5 Comparison of migration time via InfiniBand using a RAM drive with and without pre-copy 89
5.6 FENFLOSS memory transferred during migration with and without pre-copy 93
5.7 FENFLOSS migration duration details with and without pre-copy 95
5.8 FENFLOSS migration duration overview with and without pre-copy 95

Zusammenfassung

Um die immer steigenden Anforderungen an Rechenressourcen im High Performance Computing zu erfüllen werden die eingesetzten Systeme immer größer. Die Werkzeuge, mit denen Wartungsarbeiten durchgeführt werden, passen sich nur langsam an die wachsende Größe dieser neuen Systeme an. Virtualisierung stellt Konzepte zur Verfügung, welche Systemverwaltungsaufgaben durch höhere Flexibilität vereinfachen. Mit Hilfe der Migration virtueller Maschinen können Systemverwaltungsaufgaben zu einem frei wählbaren Zeitpunkt durchgeführt werden und hängen nicht mehr von der Nutzung der physikalischen Systeme ab. Die auf der virtuellen Maschine ausgeführte Applikation kann somit ohne Unterbrechung weiterlaufen.

Trotz der vielen Vorteile wird Virtualisierung in den meisten High Performance Computing Systemen noch nicht eingesetzt, dadurch Rechenzeit verloren geht und höhere Antwortzeiten beim Zugriff auf Hardware auftreten. Obwohl die Effektivität der Virtualisierungsumgebungen steigt, werden Ansätze wie Para-Virtualisierung oder *Container*-basierte Virtualisierung untersucht bei denen noch weniger Rechenzeit verloren geht. Da die CPU eine der zentralen Ressourcen im High Performance Computing ist wird im Rahmen dieser Arbeit der Ansatz verfolgt anstatt virtueller Maschinen nur einzelne Prozesse zu migrieren und dadurch den Verlust an Rechenzeit zu vermeiden.

Prozess Migration kann einerseits als eine Erweiterung des präemptive Multitasking über Systemgrenzen, andererseits auch als eine Sonderform des *Checkpointing* und *Restarting* angesehen werden. Im Rahmen dieser Arbeit wird Prozess Migration auf der Basis von *Checkpointing* und *Restarting* durchgeführt, da es eine bereits etablierte Technologie im Umfeld der Fehlertoleranz ist. Die am besten für Prozess Migration im Rahmen dieser Arbeit geeignete *Checkpointing* und *Restarting* Implementierung wurde ausgewählt. Eines der wichtigsten Kriterien bei der Auswahl der *Checkpointing* und *Restarting* Implementierung ist die Transparenz. Nur mit einer möglichst transparenten Implementierung sind die Anforderungen an die zu migrierenden Prozesse gering und keinerlei

Einschränkungen wie das Neu-Übersetzen oder eine speziell präparierte Laufzeitumgebung sind nötig.

Mit einer auf *Checkpointing* und *Restarting* basierenden Prozess Migration ist der nächste Schritt parallele Prozess Migration für den Einsatz im High Performance Computing. MPI ist einer der gängigen Wege eine Applikation zu parallelisieren und deshalb muss Prozess Migration auch in eine MPI Implementation integriert werden. Die vorhergehend ausgewählte *Checkpointing* und *Restarting* Implementierung wird in einer MPI Implementierung integriert, um auf diese Weise Migration von parallelen Prozessen zu bieten.

Mit Hilfe verschiedener Testfälle wurde die im Rahmen dieser Arbeit entwickelte Prozess Migration analysiert. Schwerpunkte waren dabei die Zeit, die benötigt wird um einen Prozess zu migrieren und wie sich Optimierungen zur Verkürzung der Migrationszeit auswirken.

Abstract

To satisfy the ever increasing demand for computational resources, high performance computing systems are becoming larger and larger. Unfortunately, the tools supporting system management tasks are only slowly adapting to the increase in components in computational clusters. Virtualization provides concepts which make system management tasks easier to implement by providing more flexibility for system administrators. With the help of virtual machine migration, the point in time for certain system management tasks like hardware or software upgrades no longer depends on the usage of the physical hardware. The flexibility to migrate a running virtual machine without significant interruption to the provided service makes it possible to perform system management tasks at the optimal point in time.

In most high performance computing systems, however, virtualization is still not implemented. The reason for avoiding virtualization in high performance computing is that there is still an overhead accessing the CPU and I/O devices. This overhead continually decreases and there are different kind of virtualization techniques like para-virtualization and container-based virtualization which minimize this overhead further. With the CPU being one of the primary resources in high performance computing, this work proposes to migrate processes instead of virtual machines thus avoiding any overhead.

Process migration can either be seen as an extension to pre-emptive multitasking over system boundaries or as a special form of checkpointing and restarting. In the scope of this work process migration is based on checkpointing and restarting as it is already an established technique in the field of fault tolerance. From the existing checkpointing and restarting implementations, the best suited implementation for process migration purposes was selected. One of the important requirements of the checkpointing and restarting implementation is transparency. Providing transparent process migration is important enable the migration of any process without prerequisites like re-compilation or running in a specially prepared environment.

With process migration based on checkpointing and restarting, the next step towards providing process migration in a high performance computing environment is to support the migration of parallel processes. Using MPI is a common method of parallelizing applications and therefore process migration has to be integrated with an MPI implementation. The previously selected checkpointing and restarting implementation was integrated in an MPI implementation, and thus enabling the migration of parallel processes.

With the help of different test cases the implemented process migration was analyzed, especially in regards to the time required to migrated a process and the advantages of optimizations to reduce the process' downtime during migration.

Chapter 1

Introduction

Today's availability of High Performance Computing (HPC) resources and its integration into the product development cycle can lead to a shorter time to market and a more predictable product quality by employing computer simulation at different stages of the product development cycle. The need for computer simulations at multiple stages of the product development cycle as well as the desire to increase complexity and/or granularity, leads to a higher demand for HPC resources. This demand is usually satisfied by increasing the number of nodes which leads to new problems.

1.1 Motivation

One of the problems connected with an increasing number of nodes in an HPC environment is that system management becomes more complex. Existing tools and practices are no longer feasible and driven by the larger number of nodes and other components like power supplies, interconnect and cooling, new system management approaches are needed which include new intelligent management and monitoring tools as well as new underlying technologies which offer much more flexibility.

1.2 Goals

The primary focus of this work is the ability to migrate processes while they are running, without interrupting or even affecting the running processes. This offers new possibilities and flexibilities for system management tasks like updating a system, replacing hardware which has shown defects or distributing the load more evenly. It should be possible to perform all of these tasks independently of the usage of the affected component (node, power supply, interconnect, cooling). It should no longer be necessary to wait for tasks to finish before said system management operations can be performed.

In addition to easing system management tasks, process migration makes it possible to distribute load more evenly. It can be used to migrate processes from a single node which is running out of resources like available memory or Central Processing Unit (CPU) cycles. It can also be used to migrate processes to another part of the compute cluster to free up resources like the interconnect or to distribute the cooling more evenly throughout the whole system. Not only should it be possible to migrate the process inside the cluster, it should also be possible to migrate processes to on-demand spun up instances in the cloud.

To complete the usefulness of process migration in an HPC environment it must be possible to migrate one or more processes of a parallel calculation which is running on multiple nodes, to other nodes in the cluster. In the scope of this work this means to support parallel calculations which are parallelized with the help of a Message Passing Interface (MPI) implementation.

1.3 Structure of this work

After establishing the necessity of the existence of process migration in the current chapter, Chapter 2 (page 25) proposes the migration of single processes (or process groups[1]) instead of virtual machines to reduce virtualization induced

[1] a process and its child processes

overheads in CPU and communication. With the help of process migration it is possible to use enhanced system management techniques like migration, without the need to introduce virtualization which is undesirable in an HPC environment due to overheads connected with virtualization.

Chapter 3 (page 33) introduces the general concepts of a process and what needs to be considered, to enable process migration. Different methods to transfer the memory of the process to be migrated are discussed as well as different approaches to migrating the whole process. After ruling out the pre-emptive migration approach, different checkpoint/restart based approaches are discussed, as well as which of the existing checkpoint/restart implementations is the most promising for use as a basis for process migration.

Chapter 4 (page 57) focuses on process migration in a parallel MPI environment. Basing process migration and thus parallel process migration on checkpoint/restart has the additional advantage that parallel process migration can use the results of existing fault tolerance related studies.

Chapter 5 (page 65) presents the actual implementations and the results gained by the implementation.

Chapter 6 (page 97) summarizes this work and provides an outlook identifying which aspects may become the subject of further studies.

Chapter 2

State of the Art

Techniques like virtual memory and preemptive multitasking have made virtualization a core concept of computer sciences for decades. During the last decade the concept of virtualization has gained considerable attention due to the availability of different hypervisor providing operating system level virtualization functionality and virtualization has become one of the primary platforms providing services in a data center. Applications[1] no longer run on physical hardware but are increasingly moved to virtual hardware[2]. With the help of different hypervisors virtual machines are running on top of those hypervisors and this model has many advantages compared to using the physical hardware directly. Each virtual machine can be used to run almost any desired operating system as the virtual machines behave just like a physical machines would. Modern CPUs have special abilities to directly support hypervisors, thus enabling hypervisors to run many operating systems inside many virtual machines on a single host.

Running applications in a virtualized environment has many advantages:

- **Consolidation** - Running in a virtualized environment provides the possibility to consolidate many applications on a single physical hardware.

[1] In this context application is a synonym for any kind of process or service running in the data center

[2] or virtual machines

Instead of trying to buy multiple servers which attempt to offer exactly the right amount of resources, it is possible with the help of virtualization to buy less hardware which is more powerful. Thus decreasing the number of physical systems which in turn decreases the cost of running those systems. But instead of running all applications on the same physical machine and the same operating system, virtualization is used to separate the applications.

- **Separation/Isolation** - Running in a virtualized environment provides separation between the application. With virtualization one can easily control how many resources each virtual machine receives which in turn also controls the resources available to targeted application running inside the virtual machine. But separation is not only helpful for dividing existing resources, it is also a form of security, as one vulnerable application which has been compromised does not automatically endanger all applications running on the same physical hardware[3].

- **Utilization** - Running in a virtualized environment enables better utilization of the physical hardware. The utilization of the resources can be optimized by dynamically deciding how many virtual machines are running on one physical machine.

- **Administration** - On the one hand providing a virtual machine for each application increases the number of virtual machines and on the other hand it makes the administration of those virtual machines easier. Running a dedicated virtual machine for each application makes management tasks like updating the operating system or updating the application running inside the operating system much easier as there are no internal dependencies between the running applications since each application is running in its own virtual machine. Running in a virtualized environment also means that storage resources are shared. Instead of accessing a physical disk directly, it is common in virtualized environments to use storage backends which provide the hard disk from the view of the virtual machine. The

[3]assuming there are no known vulnerabilities in the hypervisor used

hard disk can be a simple file representing the virtual machines hard disk. The virtual machines hard disk can, however, also be provided by Storage Area Network (SAN), Network-Attached Storage (NAS), Internet Small Computer Systems Interface (iSCSI) or object based storage systems.
With the help of snapshotting, fast copies of the virtual disk can be created as backups. These snapshots can then be used to easily restore an application running in a virtual machine after a failure. It can also be used for testing new features more easily, without the need to reinstall the whole system in the case of something going wrong.

- **Deployment** - Using virtual machines, new services can be made available in a very short time. There is no need to buy new hardware and in the optimal case new deployment happens automatically.

- **Availability** - Running in a virtualized environment can also provide higher availability for the virtual machines than for the physical hardware. Most virtual environments make it possible to migrate virtual machines between the existing physical machines running a hypervisor. This enables an automatic or manual reaction to imbalanced use of the existing resources, or the replacement of defect hardware in one of the physical machines. All without interrupting the running applications.

All those advantages are available from most of today's hypervisors and especially from virtual machine migration. This makes it possible to perform system management tasks independent of the applications currently running as those applications can be distributed and balanced over the existing physical hardware without interrupting those applications. Examples for easy-to-use off-the-shelf solutions which support virtual machine migration are the hypervisor implementations from VMware[1] and Kernel-based Virtual Machine (KVM)[2].

2.1 Hypervisor

Hypervisors provide virtual machine migration in different scenarios. Virtual machine migration in its simplest form has all physical hosts in the same network and all physical hosts are using a shared storage system. In the case a virtual machine has to be migrated only the state and the memory of the virtual machine have to be migrated. In scenarios where there is no shared storage system hypervisors also support virtual machine migration in combination with storage migration. This offers the possibility to migrate virtual machines over larger distances as not all physical hosts have to access the same storage system. Virtual machine migration in combination with storage migration requires a larger amount of time for the migration as more data has to be migrated. Another form of virtual machine migration can be used for hot standby scenarios. During the whole runtime of a virtual machine, a second virtual machine on another physical host is continuously synchronized with the first virtual machine. This scenario provides very fast migration in the case migration becomes necessary as most of the data is already transferred to the destination system. It also provides fault tolerance as the second virtual machine can be activated as soon as the first virtual machine has a failure.

Comparing an application which is running in a virtualized environment to an application running on physical hardware leads to the question of whether resources are wasted in a virtualized environment by the hypervisor which is controlling the virtualized environment. In any hypervisor implementation there will be an overhead which requires resources and these resources are not available to the application which is now running in the virtualized environment (see [3], [4] and [5] for attempts to quantify the virtualization overhead). Unfortunately it is not possible to meaningful quantify the overhead as it will vary with the used virtualization technique as well as with the running workload.

In an HPC environment the primary resource in most of the cases is the CPU and therefore it is important to know how many CPU cycles are wasted by the hypervisor. Although the virtual machine performance penalty is minimal, research to optimize the usage of the resources is ongoing and a common approach

in an HPC environment is to use para-virtualization[6] or even container based virtualization[7][8] to reduce the overhead of the virtualization.

Attempts to reduce the virtualization overhead by using simpler virtualization techniques like para-virtualization and container based virtualization are a strong indicator that, no matter how small the overhead is, every CPU cycle is important and if possible should not be wasted.

2.2 Para-Virtualization and Container Based

Para-virtualization is a virtualization techniques that reduces the hypervisor overhead with simplified interfaces to the guest operating system. This means that the guest operating systems needs to be adapted to run in the para-virtualized environment which at the same time means that the hypervisor is not required to emulate real hardware. The guest operating system is aware that it is running in a para-virtualized environment. Container-based virtualization which is also called operating system-level virtualization is a virtualization techniques with minimal overhead. It does not emulate an operating system but provides mechanisms to separate the processes running in the container instances. Just like hypervisor based virtualization container-based virtualization provides the opportunity to limit the resource usage of each container. Most container-based virtualization implementations, however, do not provide the possibility to migrate the containers.

2.3 I/O Accesses

Although the virtualization penalty for the CPU is nowadays relatively small it is important to also consider other hardware resources besides the CPU. The CPUs which are used in today's HPC environments usually have the necessary hardware extensions to support virtualization with low overheads. These technical advancements are unfortunately not yet widely available in communication

and Input/Output (I/O) hardware components. The fact that communication and I/O hardware has much higher latencies and lower bandwidths than the CPU makes those components the bottleneck even in the non-virtualized case and this bottleneck intensifies even more in the virtualized case[9][10]. Especially in virtualization scenarios where multiple accesses to those components have to be multiplexed without hardware support, the virtualization overhead will increase further and performance prediction will become non-trivial if multiple virtualized environments are running on the same hardware[11].

In contrast to the common approach in virtualization which emulates communication and I/O hardware components in software, there is Single Root I/O Virtualization and Sharing (SR-IOV). With the help of SR-IOV those components provide a virtual interface (function) which provides a dedicated virtual device and for the virtual machine the component appears as a device which is dedicated to one virtual machine. This has the advantage that the functionality does not need to be replicated in the hypervisor's emulation and latencies will be much lower than in the emulated case. Although this technology has mainly been implemented by communication hardware like Ethernet adapters it can now also be found in InfiniBand hardware[12] which makes it more attractive in an HPC environment. Unfortunately it does not yet provide the full performance for all use cases[12][13].

Sadly using SR-IOV has the disadvantage that physical components cannot be paused and the state cannot be transferred during migration, which still requires operating system support for a successful virtual machine migration.

2.4 Process Migration

To avoid the disadvantages of virtualization like hypervisor overhead which wastes CPU cycles, high latency communication due to emulation of communication hardware components, or state loss during migration with SR-IOV,

this work proposes to migrate single processes or process groups[4]. To migrate just a process continues the trend of minimizing the virtualization overhead with the help of simpler virtualization techniques like para-virtualization or container based virtualization. Operating systems have continued to develop and now also provide features for separation and isolation which used to require virtualization.

Another advantage of migrating processes and not complete virtual machines is that it requires a smaller amount of memory to be transferred during migration as only the process affected has to be transferred and not the complete operating system. Only migrating a single process makes migration over data center boundaries easier as less memory has to be transferred and the destination is independent of the underlying technology used. Process migration is independent of running on a virtual machine or a physical machine, independent of the underlying hypervisor and independent of the storage backend (no shared storage or storage migration required). As long as the source and destination of the migration share the same Instruction Set Architecture (ISA) and operating system (see 5.1.1 (page 66)), a process can be migrated. This independence of the underlying technology also provides the opportunity to migrate non-parallel jobs of an overloaded compute cluster to on demand provisioned systems in a compute cloud.

Although the migration of virtual machines within the same Internet Protocol (IP) subnet is easy and supported by most hypervisors, migration of virtual machines over the boundaries of a Virtual Local Area Network (VLAN) or even over a Wide Area Network (WAN) and larger distances can also be solved by directly routing to the virtual machines (for example with Open Shortest Path First (OSPF)). But this usually is a more complicated setup and requires additional work compared to the solution provided by existing hypervisors out of the box. Process migration however is independent of the migration destination and can be used for migration inside a Local Area Network (LAN) as well as over a WAN.

In addition to the previous points there is another scenario in which process

[4] a process with all its child processes

migration is not just better than virtual machine migration but also the only possible solution. If there are multiple processes running on a single machine in an HPC environment and if these processes do not all place the same requirements on existing resources like memory and CPU, thus creating an imbalance in the usage of those resources, process migration can help to restore a more balanced usage of the existing resources by migrating the processes to other systems. This way a starvation of the processes due to the lack of a resource can be avoided and the processes can be balanced on the existing resources. With virtual machine migration, every process needs to run in its own virtual machine if it ever needs to be migrated. With process migration it is not necessary to decide in advance what should be possible to migrate. With process migration any process can be migrated at any time.

To efficiently use process migration in an HPC environment it has to support some kind of parallelization. With MPI being one of the primary approaches for parallelizing a computational task over multiple nodes and cores, any kind of process migration has to support MPI parallelized application. If the MPI environment can handle process migration it becomes easier to migrate processes as the knowledge of the underlying communication technology is no longer necessary to the instance triggering the migration.

This chapter proposes to migrate single processes (or process groups) instead of virtual machines to reduce virtualization induced overheads in CPU and communication. Process migration also requires less data to be transferred during the migration and reduces the requirements on the source and destination system of the migration. With the help of process migration, it is possible to use enhanced system management techniques like migration without the need to introduce virtualization which is undesirable in an HPC environment due to overheads connected with virtualization.

Chapter 3

Process Migration

To migrate a process it is important to understand what a process is and which approaches and optimization can be used to effectively migrate a process. This chapter provides an overview of the fundamental concepts of processes and process migration.

3.1 The Process

A process is a container or instance of an application or program which is currently being executed. A UNIX based operating system provides an environment in which multiple processes are running in a time-sharing configuration. In a time-sharing configuration the operating system process scheduler schedules the processes to and from the CPU (context switch) to give each process its share of the CPU(s).

According to [14, 89] a process consists of an entry in the process table "with one entry per process". Each entry in the process table (see Figure 3.1 (page 34)) includes all the information about the process and the resources which have been allocated to it.

To be able to provide a time-sharing configuration, modern operating systems

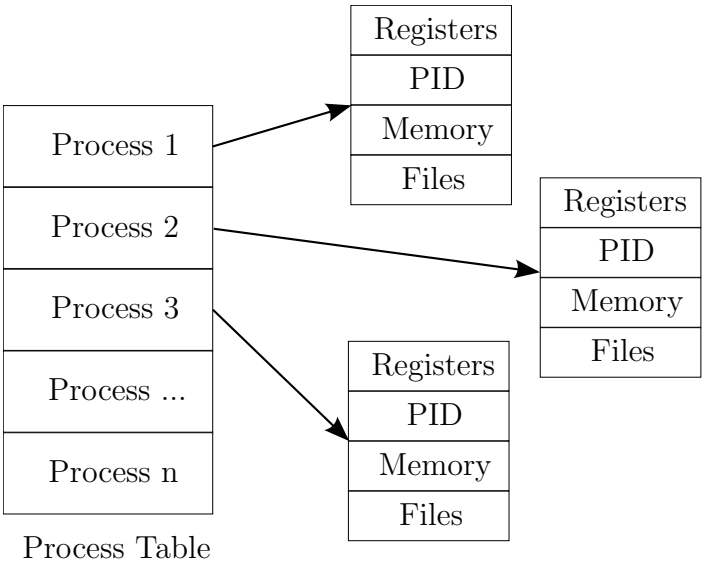

Figure 3.1: Process Table

which are of interest in the scope of this work already provide abstraction layers between the process and the hardware. These abstraction layers can already be seen as some kind of virtualization. The scheduling which is performed by the operating system can, to some extent, already be seen as a method of migrating the processes to and from the CPU(s).

In the context of migrating a process from one system to another the following parts of a process have to be taken in account which will be discussed in the following sections.

- Process management
- Memory management
- File management

3.1.1 Process Management

Using the process table, it is possible to access vital process data which contains information on where the operating system has stored the data containing the

Process Identifier (PID), stack pointer, program counter and content of the registers. This part also contains the information on the current scheduling state and pending signals.

As the operating system schedules each process to and from the CPU, depending on its scheduling state, information like the content of the registers is already stored in a format that is copied to and from the CPU as needed.

To migrate a process, all those memory structures need be exported from the operating system the process is currently running on and imported into the operating system the process should be migrated to. As each process already uses the abstraction provided by the operating system, it should be possible to extract this data in order to migrate a process to another system just as the operating system schedules different processes on a single system.

3.1.2 Memory Management

With the process' entry in the process table the memory management information can be retrieved. This includes the location of the executable code, the stack and the heap. As only modern operating systems are of interest in the context of this work, it can be assumed that the memory management uses virtual memory which also provides another abstraction for the memory accesses just as the operating system does with the time-sharing configuration of the CPU.

Virtual memory provides its own address space for each application and virtualizes the address space so that the process does not need to know which kind of physical memory (Random-access memory (RAM) or secondary storage (e.g., disk)) backs each memory address and wether the memory is contiguous (see Figure 3.2 (page 36)). The operating system can decide wether the virtual memory address is backed by actual physical memory or if the virtual memory address has not been used, it can be paged out. Through the usage of virtual memory the process has no direct knowledge of the physical memory addresses actually used, which means that the memory access is virtualized (like already

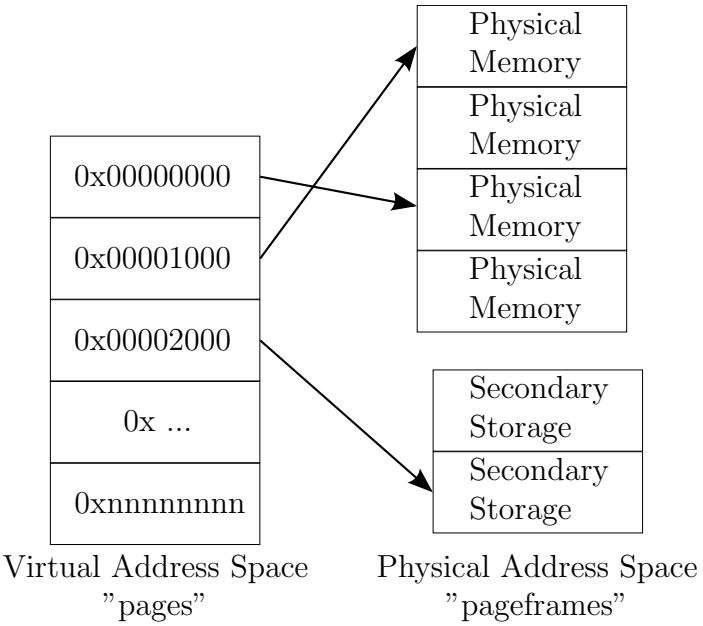

Figure 3.2: Virtual Memory - "page table"

implied by the name "virtual memory"). Virtual memory is also used to protect/restrict the access to each process' memory as each process can only access memory mapped through the page table. So just like in the case of process management, the process is running in a virtualized environment which provides an abstraction layer for all accesses to memory addresses. As the address space is virtualized and the operating system already knows how to write the used memory pages to disk (paging), it should be also possible to page all of the memory out to another system to which the process should be migrated.

3.1.3 File Management

In the process table there is also an entry concerning the file management for each process (the file descriptor table). It provides information on the working directory, root directory and all file descriptors. The file descriptor contains the details about the files which are in use and for UNIX based operating systems, which are of interest in the context of this work, a file descriptor can refer to any file type like a regular file, directory, socket, named pipe or character and

block device file. The process gets an identifier with which the file descriptor can be accessed in the file descriptor table. To migrate a process these memory structures (file descriptor table and file descriptors) need to be transferred to the destination system and can then be used to open the files with the same identifier and at the same position they used to be on the source system. It is important to remember this only includes the location of the file and its position and not the actual content of the file. Either the file needs to be transferred additionally to the destination system or the systems involved in the migration need to use shared storage system. There are multiple shared file-systems which can be used to fulfill this constraint and especially in an HPC environment it is common to use a shared file-system which all systems included in the process migration can access.

3.2 Memory Transfer Methods

After looking at what needs to be transferred to migrate a process, it is important to know how the data can be transferred. The largest portion is the actual memory used by the process. The data structures from the process table containing the information defining the process require, compared to processes' memory, only a minimal amount of memory. Therefore it is important to choose the right method to transfer the memory efficiently.

In the scope of this work three different methods of transferring the memory to the destination system have been studied. The methods differ in the point in time at which the memory is transferred. In all cases the process needs to be suspended for a certain amount of time during which it is migrated. The memory can now either be transferred before, during or after the process has been suspended. The process' information from the process table is transferred during the suspension in each of those scenarios as it is, compared to whole amount of memory used, negligibly small. This way it can also be ensured that the information from the process table does not change during the transfer.

3.2.1 Memory Transfer During Migration

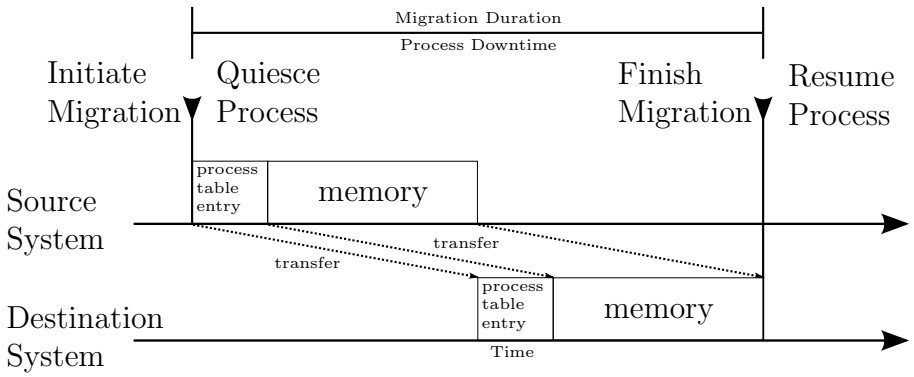

Figure 3.3: Memory Transfer During Migration

Leaving any optimization aside to reduce the time in which the process is suspended provides the most simple memory transfer method. To migrate a process, the process is quiesced and then all necessary parts of the process are transferred to the destination system including the process' memory and the entries from the process table. On the destination system the information transferred is included in the operating system and the process is then resumed (see Figure 3.3 (page 38)). This method is straight forward and requires no additional effort as there is no optimization. It has, however, the longest downtime of the migrated process which can be, depending on the used memory and the interconnect used to transfer the data, of significant duration (over 600 seconds for 50GB of memory (see Figure 5.9 (page 84) and 5.10 (page 85))). This is especially important in an HPC environment where this downtime has to multiplied by the number of processes involved. This memory transfer method is very similar to the concepts used by checkpointing and restoring (C/R). The information of the process is extracted from the operating system and can be stored on a disk for classic C/R or it can be transferred to another system in order to migrate the process, as it will be suggested below.

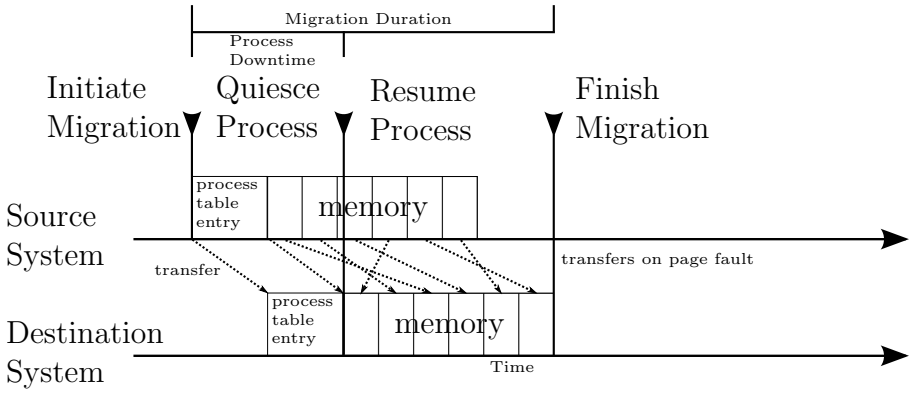

Figure 3.4: Memory Transfer After Migration

3.2.2 Memory Transfer After Migration

A possible optimization of the previous method involves transferring the memory at the moment it is actually accessed. Instead of requiring a downtime of the process during which all related data are migrated to the destination system, the downtime is much shorter and only the process table entries are transferred to the destination system. After the migration, the process is resumed and the process' memory is transferred on-demand whenever it is actually accessed. This method is very similar to the approach of an operating system which schedules a process on the CPU. If the newly scheduled process accesses memory which has been paged out, this generates a page fault and the operating system transfers the missing pages into main memory. The same method can be used for process migration between systems. If the migrated processes tries to access non-migrated memory, this generates a page fault and the memory is then transferred at this very moment (see Figure 3.4 (page 39)). This significantly reduces the process' downtime during the migration but introduces high latencies every time a non-migrated page is accessed. As this method for process migration (post-copy migration) applies existing practices, which are found in many operating systems, it seems like a good candidate for avoiding long process downtime during migration.

3.2.3 Memory Transfer Before Migration

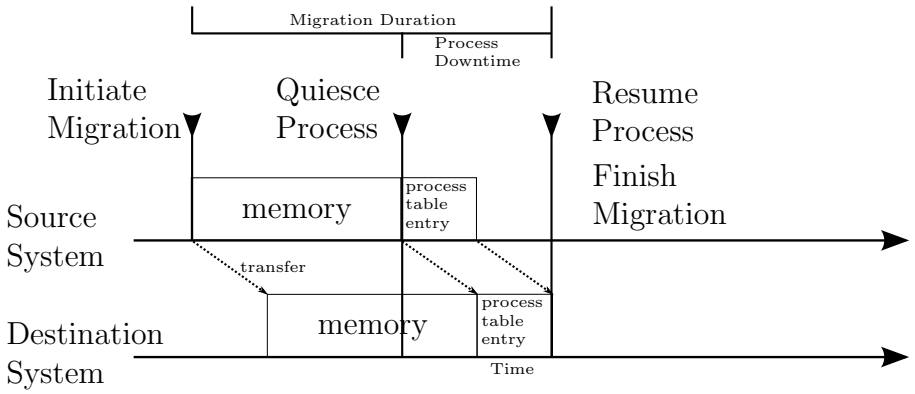

Figure 3.5: Memory Transfer Before Migration

Instead of transferring the memory on-demand on a page fault, it is also possible to transfer the memory before the process is quiesced and the process table entry is migrated (see Figure 3.5 (page 40)). This method has the advantage that it does not introduce the possibility of high latencies on every page fault as the memory has to be transferred from the system the process has been migrated from. This method has similar short process downtimes to the previous method because the process can keep running during the memory transfer, but once the process table entry has been transferred, all the memory is already on the destination system and the process can continue running with no additional delays due to missing memory pages. This makes this memory transfer method much more predictable in its behavior. The disadvantage is that the memory used by the process still running will change meaning some pages need to be transferred again. At this point additional effort is required to re-transfer only those pages which have changed during the previous memory transfer. This method (pre-copy migration) is again closer to C/R than the previous method (post-copy migration) which was very similar to the process scheduler of an operating system.

3.3 Preemptive Migration

Process migration can be seen as a special case of regular scheduling as it is performed by every preemptive multitasking operating system with the difference being that the process can be scheduled to a different physical (or virtual) system instead of scheduling the process on a local CPU. However process migration can also be seen as a specialized form of checkpointing and restarting (see 3.4 (page 43)) where the checkpointing is not used to write a process image on disk but instead is directly transferred to the memory of the destination system.

Basing process migration upon the preemptive multitasking of the operating system is one possible approach to supporting process migration. The process scheduler of the operating system could be extended to schedule, and thus migrate, the processes to another node instead of scheduling processes only on the local operating system.

Figure 3.6 (page 42) shows a diagram with the possible steps required during the migration.

Once the decision to migrate a process has been made, the process to be migrated has to be quiesced and the migration starts with the transfer of the process table entry. The process scheduler running in the kernel space requests the transfer of the process table entry by calling a program in user space. This program then carries out the actual transfer of the process table entry to the destination system's user space. From the user space the process is then transferred to the kernel space where the process scheduler integrates it in the process table of the destination system. Once the process is continued on the destination system and it attempts to access its memory, a page fault occurs which results in a request for that memory page on the source system. The request has to be passed to the user space which then transfers the request over the network to the source system. From the source system's user space it is forwarded to the kernel space. Now the requested memory page is transferred to the destination system in the same way the process table entry previously.

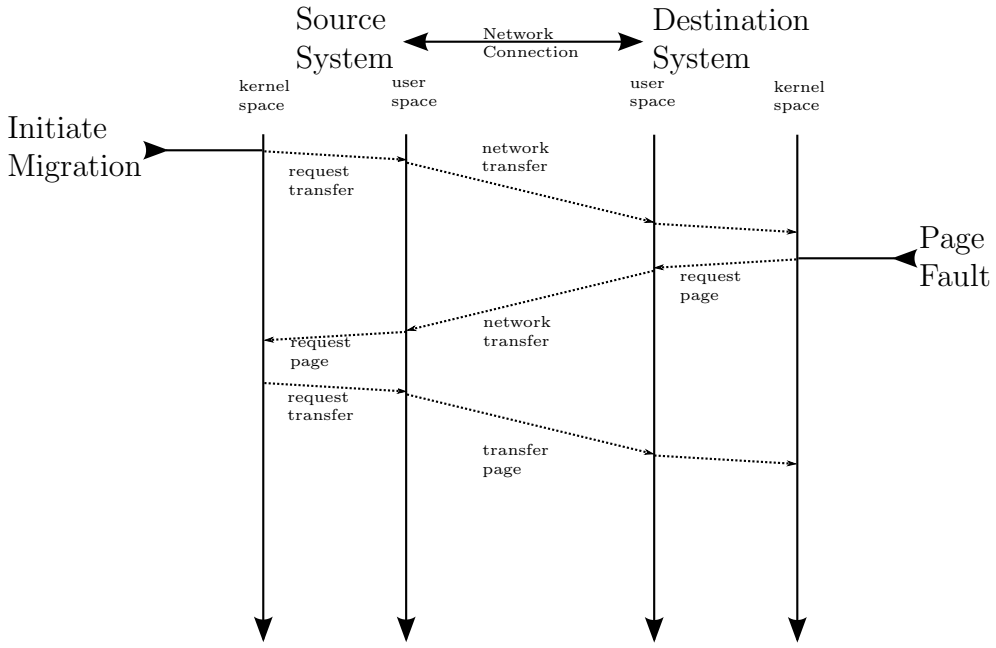

Figure 3.6: Preemptive Migration

This design has many kernel space - user space transition, which make it very complex and error-prone. Another approach would be to omit the many kernel space - user space transitions. This other approach unfortunately has further drawbacks. All methods and programs which are available in user space and which provide means of data transportation, data security and data integrity would have to be re-implemented in kernel space. Re-implementation of existing functionality would require a lot of time and introduce many errors which have already been solved in the user space implementations. Running the data transfer in kernel space introduces many possible functional and security related errors. If such errors are exploited while running in kernel space, such an error can compromise not only the program running the transfer, but the whole operating system.

So both approaches (completely in kernel space, kernel space - user space transitions) have their drawbacks and would add a lot of complexity to a central part of the operating system like the process scheduler. Changes to such a central part of the operating system would drastically decrease the acceptance in an HPC production environment as the risk of unintentional side effects would be very high.

3.3.1 Single System Image

Preemptive multitasking over system boundaries is similar to the functionality provided by single-system image (SSI). SSI provides an abstraction with which multiple systems and their distributed resources can be accessed as a single system. The SSI implementation migrates/distributes the processes between the existing hosts and provides a single interface to access the resources. There are different SSI implementations like OpenMosix, OpenSSI and Kerrighed[15]. Unfortunately the SSI approach is not very useful in an HPC environment because the programs used in an HPC environment are usually aware that they will be running on many nodes and SSI was therefore not studied further.

3.4 Checkpoint/Restore Migration

Checkpoint/Restore, which is also known as Checkpoint/Restart, is known primarily as an approach for providing fault tolerance. All the necessary information defining a process or a group of processes is collected and stored (periodically) in one or multiple files (checkpointing). In the case of fault tolerance this checkpointed information is used to restore/restart the process after the cause of the fault has been remedied. By employing C/R only the results since the last checkpoint are lost and not since the beginning of the entire calculation. C/R can be used on different levels. It ranges from application level C/R to fully transparent operating system level C/R. Periodically saving the results can be seen as the simplest form of application level checkpointing. The application

writes its results since the last checkpoint to a form of of permanent storage and it also knows how to restore this data in case of a restart. This application level checkpointing is easy to implement because it is specially tailored for its application. The disadvantage is that it has to be re-designed and re-implemented for every application and thus it can be seen as the opposite of fully transparent.

Trying to be more transparent leads to a variant of application level checkpointing which is provided by an external library. This is designed to support checkpointing of as many different applications as possible but it still requires massive changes to the actual program to be checkpointed. It does not require as much work to implement as application level checkpointing provided by the application, but it still requires a significant number of code changes. The increase in transparency also leads to higher complexity in the C/R library used. To be useful in many different applications, it needs to provide more functionality than the self implemented application level C/R.

The next step in providing a more transparent checkpoint solution is to remove the requirement to modify the existing code. A user-space based C/R solution could require certain libraries to be pre-loaded to intercept operating system calls, in order to be able to checkpoint and restore the targeted application. Again, this increases the complexity of the C/R implementation while at the same time the checkpointing becomes more transparent. At this level no more changes are required to actual application which opens C/R for programs without access to the source code as re-compilation is no longer required. The environment still needs to be correctly set up so that, for example, certain libraries which are intercepting calls from the application to the operating system are pre-loaded.

Every C/R technique presented came closer to being fully transparent and the last step is to provide this kind of transparency by implementing C/R on the operating system level. Thus the application which needs to be checkpointed has neither to be modified on the source code level, recompiled nor started in a specially prepared environment.

The C/R implementation on the operating system level has the highest complexity but at the same time provides highest transparency and flexibility.

Although having the highest level of complexity, the fully transparent operating system level C/R implementation is the one with the greatest chance of actually being used[16]. Every other C/R implementation mentioned has the drawback that it requires additional work for the application developer or HPC system administrator and is therefore less likely to actually being employed.

A fully transparent operating system level C/R implementation can also be the basis of a process migration implementation. Instead of periodically storing the data of the process on a storage system, the data are transferred directly from the main memory of the source node to the main memory of the destination node, thus migrating the process by employing C/R techniques.

As there are multiple existing C/R implementations the most promising candidates have been studied in more detail to be able to decide which C/R implementation is most suitable as the basis for migrating processes. There are not just multiple existing C/R implementations but also multiple operating systems like IBM's Advanced Interactive eXecutive (AIX) or NEC's SUPER-UX that support C/R[17]. In the scope of this work only Linux based C/R implementations have taken into account. According to the TOP500 list of supercomputer sites, Linux is used on over 90% of the worlds fastest systems[18]. In addition to its wide adoption in supercomputing the open nature of Linux makes it a perfect basis for this work.

To successfully support process migration, the Linux based C/R implementation needs to be as transparent as possible to support as many different programs as possible. Transparent C/R is important to avoid re-compilation or running the program in a special environment (e.g., library pre-loading). The requirement to re-compile a program and to a lesser extent the requirement to pre-load a library to re-route system calls, hinders the usage of C/R, especially if the source code of the program to be C/R is not available. Although the pre-loading of a library is a good solution to prove a concept, it is not desirable for a production environment as it adds an additional layer which will decrease the performance

even if the penalty is only minimal. It also requires additional maintenance as the system call library might be changed for security reasons or to fix bugs, which then requires an update of the wrapper library which, depending on the mode of operation, will take much longer than fixing the system call library. Thus the system will be unusable for an unknown time until the wrapper library has been fixed.

In the following, four operating system level C/R implementations, providing transparent C/R, will be evaluated to identify the most promising as basis for process migration.

3.4.1 Berkeley Lab Checkpoint/Restart

One of the more prominent C/R implementations is Berkeley Lab Checkpoint/ Restart (BLCR)[19] which has now been in existence for about ten years. It was originally developed as a project which was not part of the official Linux kernel tree and has been adopted in many HPC environments. Being developed outside of the official Linux kernel tree has the advantage that its design does not have to be accepted by the Linux community. On the other hand this development model has the disadvantage that its development lags behind the official Linux tree versions and upgrading to a new Linux version always depends on the availability of a new BLCR release. Another drawback of BLCR's development model is that not all Linux distributions include externally (outside of the official Linux kernel) developed code as its unavailability might block an important security update. But not being included in Linux distributions used in HPC requires additional work for the cluster maintenance and it is also not part of any test suites involving a release of a Linux distribution being covered by its vendor.

BLCR's kernel based functionality is located in Linux kernel modules[20]. This approach makes it easier to maintain the code outside of the official Linux with its fast changing interfaces. This design also makes it easier to install BLCR on a system, as the operating system kernel does not require changes and re-

compilation. If the BLCR modules can be compiled against the kernel to which the system has just been upgraded, this makes maintenance easier. The approach of locating all kernel required functionality in one place has, however, the drawback that it reduces BLCR's transparency and the successful use of BLCR requires the application to be checkpointed have to either be re-compiled or certain libraries pre-loaded.

The lack of full transparency and the additional steps during cluster software upgrade were the reasons BLCR was not selected as the basis for C/R based process migration. Not being part of the official Linux kernel requires additional work during cluster maintenance and it also increases the risk of not being able to upgrade due to uncertainty as to whether BLCR will work with the newly installed kernel.

This leads to an additional requirement of the C/R implementation. It has either to be included in the official Linux kernel, or it has to be implemented only in user-space, making it independent of the Linux kernel version and its changing internal interfaces.

3.4.2 Distributed MultiThreaded Checkpointing

Distributed MultiThreaded Checkpointing (DMTCP) "is a transparent user-level checkpointing package for distributed applications"[21] which is implemented completely in user-space. It targets Linux and requires no changes to the Linux kernel and therefore it fulfilled the requirements of transparency and running in user-space.

To successfully checkpoint a process, the targeted process needs to be started in a special environment which preloads certain libraries providing wrappers for different system calls. By completely running in user-space most of the problems connected with BLCR concerning its kernel modules do not exist. There is, however, the disadvantage that every system call needs to go through the pre-loaded wrappers which probably only means a minimal performance penalty for HPC programs as the compute intensive parts do not usually use many system calls.

Unfortunately this still introduces an overhead and as previously mentioned, the goal is to avoid overheads wherever possible. Use of the wrapper DMTCP also tries to solve the problem of PID collisions (see 5.1.4) by intercepting the related system calls and providing a virtual PID. On the one hand this provides a solution to problems connected with PID collisions but on the other hand it introduces an incompatibility with existing interfaces. A process trying to read information about its state or its files by accessing the `/proc` file-system will fail due to the virtual PID.

With the requirement to pre-load a wrapper library DMTCP is not as transparent as possible and will always depend on the wrapper library and which system calls it proxies. This implementation has advantages (independent of the Linux version) over BLCR but still requires a special setup to pre-load its wrapper library.

3.4.3 Kernel-Space-Based

Both C/R approaches (BLCR and DMTCP) studied so far still have drawbacks. By design they are both not completely transparent and require re-compilation and/or libraries to be pre-loaded. For a completely transparent C/R solution another design is required. A user-space implementation like DMTCP always needs to pre-load libraries to intercept system calls. To provide a transparent C/R solution, a kernel-based approach is needed. BLCR's decision to locate the required functionality in kernel modules makes it easy to maintain the code outside of the official Linux kernel although it limits its functionality. To develop a transparent kernel-based C/R solution it has to be much more integrated into the kernel. Such a tightly integrated C/R solution will be difficult to develop outside of the official Linux kernel. This leads to a new requirement for the C/R implementation to be used. In addition to the previously mentioned requirement, that the C/R implementation has to be as transparent as possible, it also needs upstream inclusion. For a transparent C/R implementation the code has to be integrated at different places of the Linux kernel and the development of such functionality can only work if it is part of the official Linux kernel and

accepted by the Linux community. Trying to develop C/R functionality outside of the official Linux kernel would increase the development effort dramatically due to the fast development model of Linux and its often changing internal interfaces. Another advantage of upstream inclusion is that C/R will more likely be picked up by Linux distributions which will increase the adoption of C/R in many different areas.

Because of precisely of these reasons another C/R approach was developed by Ladaan and Hallyn[22]. As this approach tries to implement C/R as part of the Linux kernel, it will be called kernel-based-C/R. This kernel based approach was started in 2008[23]. To avoid the same problems as BLCR and other attempts to add support for C/R into the kernel, the kernel-based approach tried to work with the Linux community from the start. One goal was that the changes for C/R had to go upstream and be part of the official Linux kernel tree. To achieve this the authors published their work as soon as possible and always worked with the Linux community and their feedback. The development stalled somewhere around the beginning of 2011 with the release of Linux 2.6.37.

This was also the time this work started and as the kernel-based approach was developed in collaboration with the Linux community and was targeted for upstream adoption it seemed to be a good starting point for process migration. As the project appeared to have been abandoned by the original developers the code was ported, as a part of this work, to the latest (at that time (January 2012)) Linux kernel release version 3.2. As there have been four releases of Linux between 2.6.37 and 3.2 (2.6.38, 2.6.39, 3.0, 3.1) and as the Linux kernel changes fast, it took some time to port the over one hundred changes from Linux version 2.6.37 to 3.2.

Once all those patches had been adapted to the then latest Linux version 3.2 it was possible to use the kernel-based approach with the then latest kernel for C/R. On top of those patches, process migration was successfully implemented and it was possible to move a running process from one system to another without any requirements on the running program (see 5.1.2 (page 69)).

Although the kernel-based approach was developed with upstream inclusion in mind it had, unfortunately, no chance of being included. The number of patches became too large and they were touching too many Linux kernel subsystems. Although the kernel-based C/R approach started with only nine patches it grew during its initial development to over 100 patches. For such a big and invasive change to be accepted by the Linux community, a well-known person, group or company is required to prove that he, or it, will continue to maintain the newly introduced changes. As the code was abandoned by the original developers who moved on to work on other projects it seems that the Linux kernel community made the right decision.

Although the kernel-based approach, which was the third approach studied in greater detail, provided transparent C/R without the need to re-compile programs or pre-load libraries it was not selected as the basis for process migration in this work. The main reason was, that although is was developed with upstream inclusion in mind, it was not accepted by the Linux kernel community and that would mean that no stable C/R would be available. In particular, the future of this C/R approach was unclear as no further active development was taking place.

This led to a new requirement for the C/R approach to be used. The goals transparency and upstream inclusion are not enough. The new additional requirement is that the C/R approach cannot be too invasive as is the case with the kernel-based approach. This in particular when looking at the integration in the Linux kernel. A successful C/R implementation should use existing interfaces as far as possible and only add new interfaces to the Linux kernel if the problem cannot be solved in another way. This new requirement to use existing interfaces led to the next C/R approach.

3.4.4 User-Space-Based

Seeing all the shortcomings and failures of the previously studied C/R implementations it became clear that a new C/R approach was needed. At the

Linux Plumbers Conference 2011 a new approach was presented[24] by Pavel Emelyanov and Kir Kolyshkin which tries to avoid the failures of the other attempts to get a working C/R implementation:

- **transparent**: it should be possible to checkpoint and restart as many applications as possible without re-compilation or library pre-loading.
- **not too invasive**: the changes for a working C/R to the Linux kernel have to be as minimal as possible. Reusing existing interfaces instead of creating new ones is one way to achieve this.
- **upstream inclusion**: a C/R implementation should be included in the official Linux kernel to achieve transparency and wide adoption.

With these problems (transparent, not too invasive, upstream inclusion) in mind the new C/R approach was presented. The goals were to use existing kernel interfaces as far as possible and to do as much as possible in user-space instead of kernel-space. The project was named Checkpoint/Restore in Userspace (CRIU)[25].

With most of its functionality and logic in user-space, it was possible to enhance the Linux kernel interfaces in such a way as to reveal all the necessary information for a successful C/R. With this approach, only minimal changes to existing interfaces, and no functional changes, it was possible to get the changes accepted by the Linux kernel community and provide a C/R solution which can work out of the box on any Linux system with just the installation of the necessary user-space tools and a kernel with the added interfaces. The C/R functionality offered by CRIU is therefore included in official Linux kernel (upstream for the different Linux distributions (downstream)) as only minimal changes are required, it is not too invasive to the Linux kernel, again as only minimal changes are required and has been designed to be as transparent as possible for the programs which have to be C/R'ed.

CRIU fulfills all the requirements for a C/R implementation to be accepted by the Linux kernel community. In August 2012 the first release of the user-space tools (`crtools` (has later been renamed to `criu`) version 0.1) was made with

the necessary changes to the Linux kernel (version 3.5). With this combination of this version of the Linux kernel (and later) and user-space tools, it was possible to transparently C/R programs without applying additional patches or installing additional kernel modules. To prove that downstream integration of this new C/R approach is possible, CRIU was integrated in the Linux distribution Fedora[26]. Starting with Fedora version 19 it is possible to use C/R by only using elements provided by that Linux distribution[27]. No external software, patches or kernel modules are required and C/R is possible out of the box.

Providing transparency, not requiring invasive code changes, and thus being accepted by the Linux kernel community led to the decision, to use CRIU as the C/R implementation on which process migration should be based (also see Table 3.1 (page 52) for an overview). This also means that process migration in the scope of this work will not be based on the preemptive migration approach discussed in section 3.3 (page 41) but on checkpoint/restore. With the additional inclusion of CRIU in the Linux distribution Fedora, it was shown that upstream inclusion is important for downstream acceptance of a new functionality. With the availability of C/R in Linux distributions it is much easier to use process migration for system management tasks as there is no additional overhead to employ C/R on a system providing it as an integral part of that Linux distribution.

C/R variant	Transparency	Upstream Inclusion	Implementation Architecture
BLCR	pre-load re-compilation	no	kernel module
DMTCP	pre-load	N/A	user-space
Kernel-space based	yes	N/A	kernel-space
User-space based	yes	yes	kernel-space user-space

Table 3.1: Checkpoint/Restart implementations overview

3.5 Post-Copy Migration vs. Pre-Copy Migration

To optimize process migration and especially the time during which the process is not running, different memory transfer methods (see 3.2 (page 37)) have been studied. As previously mentioned, process migration is similar to virtual machine migration, but when trying to omit the hypervisor overhead it is important to understand existing virtual machine migration approaches. The following gives an overview of post-copy migration and pre-copy migration.

- **Pre-copy migration** is at least implemented for virtual machines which are running on VMware's hypervisor [1] as well as on KVM based virtualization [2]. Pre-copy migration works in such a way (see Figure 3.5 (page 40)) that after the migration has been initiated, the memory of the virtual machine is transferred from the source system to the destination system. This is done with limited speed to reduce the impact on the performance of the virtual machine. During this phase dirty memory pages are traced and after the initial transfer of the whole memory, dirty pages are iteratively transferred. After most memory pages have been transferred the virtual machine is quiesced for a short moment and the remaining dirty memory pages and the content of the virtual CPU registers are transferred as quickly as possible. After migrating the network and storage connections, the virtual machine can be resumed on the destination system. To work with minimal downtime it is required that some kind of shared storage backend is in use.

- **Post-copy migration** has been implemented as a prototype for at least KVM based virtualization and is subject to research [28]. Post-copy migration works in such a way that after the migration has been initiated, the virtual machine is quiesced and the content of the virtual CPU registers is copied to the destination system. The virtual machine is resumed as soon as possible without any memory pages transferred to the destination system.

On each page fault accessing non-transferred memory pages, the virtual machine is momentarily quiesced and resumed after those pages have been transferred.

Although there are research results which indicate that post-copy migration might be more efficient [28], virtual machines running on VMware's hypervisor or on KVM based virtualization still use pre-copy live migration. Especially for virtual machines with many memory changes during runtime, pre-copy live migration can take much longer because after every cycle of transferring dirty pages, many pages have to be re-transferred. With post-copy migration every page is transferred exactly once on demand. The drawback of post-copy migration is that transferring pages requires more communication between the destination and source system which makes the migration setup more complicated and more error prone.

In the case of process migration it becomes even more complicated as the page fault needs to be trapped in the host operating system at the destination, which then needs to wait until the missing page has been transferred from the source system to the destination system instead of the page fault having to be trapped in the hypervisor, as is the case with virtual machine migration. In KVM based virtualization each virtual machine runs in a user-space process which simplifies post-copy migration as it does not require changes to the host operating system.

3.6 Process Migration

This chapter presented which parts define a process (see 3.1 (page 33)), with which methods these parts can be transfered (see 3.2 (page 37)), which technology is available as a basis to migrate those processes (see 3.3 (page 41) and 3.4 (page 43)) and which migration approach virtual machines are using (see 3.5 (page 53)).

Process migration in this work will not be based on the existing preemptive multitasking many of today's operating systems are performing. The operating

system used in the scope of this work is Linux (see 3.4 (page 43)) and the license and source-code availability of Linux would provide the opportunity to extend the process scheduler to schedule processes to non-local CPUs. The decision, however, to provide a C/R based process migration is motivated by the desire to minimize the risk of introducing instabilities in the operating system (see 3.3 (page 41). Making changes to such a fundamental part of the operating system as the process schedulers, introduces higher risks of instabilities and race conditions which are difficult to detect and would therefore impact all processes even if process migration was not used. Basing process migration on C/R has the additional advantage that there are multiple existing C/R implementations which (depending on the implementation, see 3.4 (page 43)) have a limited impact on the operating system. Another reason to base process migration on C/R and not on preemptive multitasking are the virtual machine migration mechanisms. Post-copy migration has been studied only in research projects (see 3.5 (page 53)) and until now production level hypervisors continue to use pre-copy migration.

From the multiple available C/R implementations and the ones discussed above, the most transparent implementations were used. At first, process migration was implemented based on the kernel-based approach (see 3.4 (page 43)). Seeing that the future of the kernel-space-based implementation was unclear, the C/R implementation used in this work was then changed to the user-space-based implementation (see Chapter 5 (page 65)).

Chapter 4

Parallel Process Migration

To achieve faster results, increase complexity and granularity of computer simulations it is common to use many nodes of a compute cluster in parallel. Multiple processes running in parallel usually mean that there are data dependencies between the processes and communication to exchange data between the processes is required. In many cases the parallelization is not provided by a SSIs and have been therefore not further studied (see 3.3.1 (page 43)).

A common approach to provide parallelization over multiple nodes is MPI[29] and to make process migration useful in an HPC environment it has to support MPI parallelized jobs.

4.1 Related Work

There are different implementations of the MPI standard. Among the popular free software implementations are Open MPI[30] and MPICH[31][32]. In combination with these implementations or their predecessor, many different studies haven been conducted on the subject of C/R and process migration.

Traditionally C/R is used to improve fault tolerance and has been a research subject for many years. There have been predictions that with the increasing

size of compute clusters and the corresponding increase of components (nodes, interconnect, power, cooling) the combined system mean time between failure (SMTBF) will be in the range of hours[33][34]. The prediction was that calculations using such a large number of cores will therefore be aborted, due to failing components after only a few hours runtime and the data will be lost.

C/R and its integration into the MPI implementations have been a possible solution to avoid data loss due to component failures. There have been efforts to create a special MPI implementation targeted on fault tolerance (FT-MPI[35]) and BLCR[19] has been integrated into LAM/MPI[36] to support C/R for parallel applications.

In addition to the then newly created Open MPI, C/R has been a topic and has also been implemented as a generic fault tolerance framework[37] which also supports BLCR.

To successfully use C/R for fault tolerance in a parallel application, either coordinated checkpointing or message logging is required[38]. Coordinated checkpointing tries to synchronize all applications to checkpoint at the same moment which requires a high level of coordination and requires lots of resources. A combination of message logging and coordinated checkpointing can reduce the overhead[39].

One drawback of C/R of parallel applications is that for a large number of nodes more than half of the computation time can be wasted waiting for the checkpoints to be written[40]. One solution to avoid waiting for I/O to finish is instead of writing the checkpoint images to a storage system is pro-active fault tolerance by migrating the processes directly from one node to another[41].

One drawback of migrating processes for fault tolerance is that it is not always possible to precisely predict failures. Such pro-active fault tolerance can be improved by analyzing different environmental data which can then be used to predict failures, however not all failures can be safely predicted. This means additional methods of fault tolerance still have to be employed. In combination with no standard C/R provider in Linux (see 3.4 (page 43)) neither system level

C/R nor process migration is available in the default configuration of many MPI implementations.

In addition to the problems concerning C/R and process migration mentioned, the prediction for SMTBFs in the range of hours has not become reality. Due to the increased reliability of the used components, the SMTBF of larger systems is much better than predicted (HLRS, personal communication, July 2014).

4.2 Parallel Process Migration

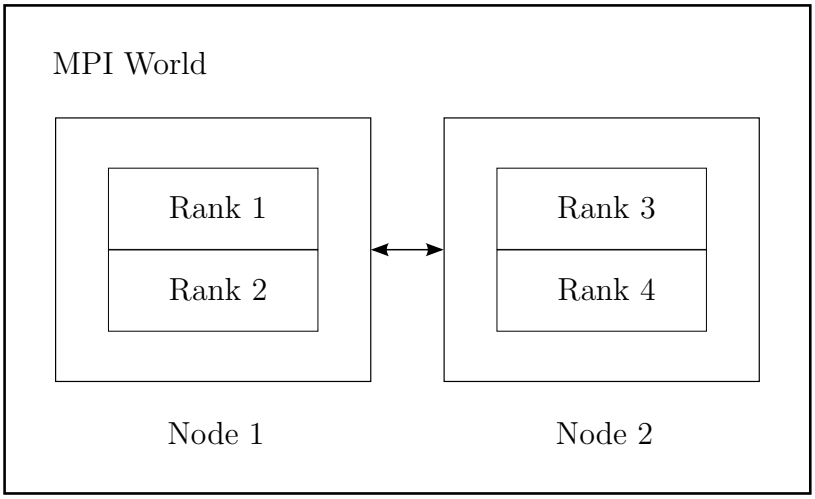

Figure 4.1: MPI migration starting point

To make process migration useful in an HPC environment it is necessary to enable process migration for an MPI parallelized application. With the possibility of moving one MPI rank from one node to another or moving all MPI ranks to another node during the application's runtime, process migration becomes useful in an HPC environment. This way it is actually possible to migrate parts of a MPI application to another node to distribute the load or to migrate all ranks from one node to another node to free the node for upcoming system management tasks.

After starting a multi-node MPI application, the runtime on each node starts the local ranks which form together the *MPI_COMM_WORLD* (see Figure 4.1 (page 59)). Each rank can now communicate with another rank using point-to-point communication or collective operations.

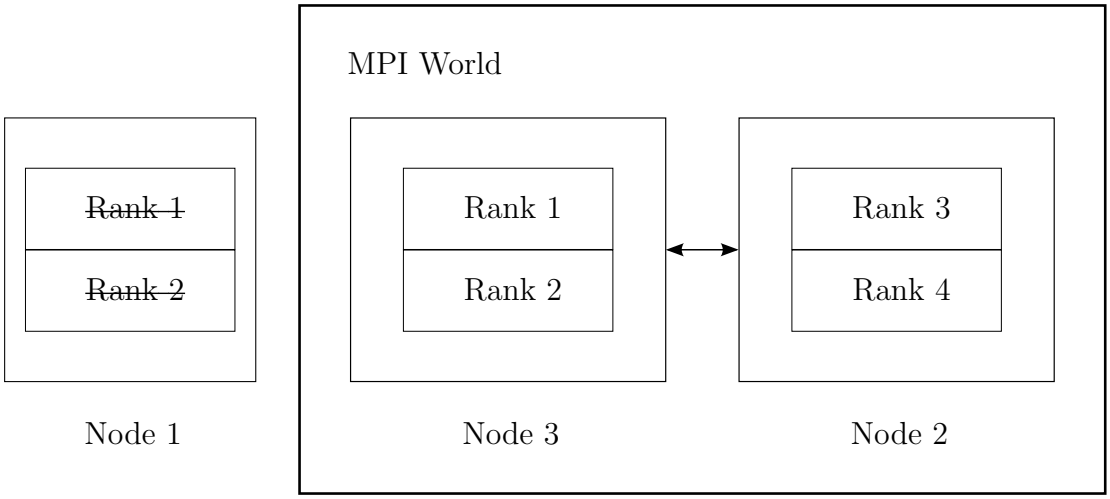

Figure 4.2: MPI migration complete node

In the case of system management tasks which require a certain node which is still in use, all the ranks from one node can be migrated to another node (see Figure 4.2 (page 60)). To start the migration, the first step is to add the new node (*Node 3* in Figure 4.2 (page 60)) to the MPI world so that all ranks are aware that a new node has joined. After the new node has been integrated into the world, the ranks can be migrated from the old node, which is about to be maintained (*Node 1*), to the new node (*Node 3*). At the start of the migration of each rank, the communication between this rank and other ranks has to be quiesced. Messages which are still in-flight have to be delivered but no new communication should be initiated. For point-to-point communication this means that any rank communicating with the rank being migrated has to wait until the migration has finished. The same is valid for collective operations; every rank included in the collective operation is stalled until the migration has finished and the rank currently migrating starts to communicate again with other ranks in the MPI world. This means that during the migration all ranks communicating with the rank being migrated have to wait until the migration has finished. The

time required for the migration is thus not only lost as computation time for the rank being migrated, but it is also multiplied by the number of ranks involved in communication with the migrated rank. After all the ranks have been migrated (off *Node 1*) the node has to be removed from the *MPI_COMM_WORLD* and is then free to be updated or rebooted.

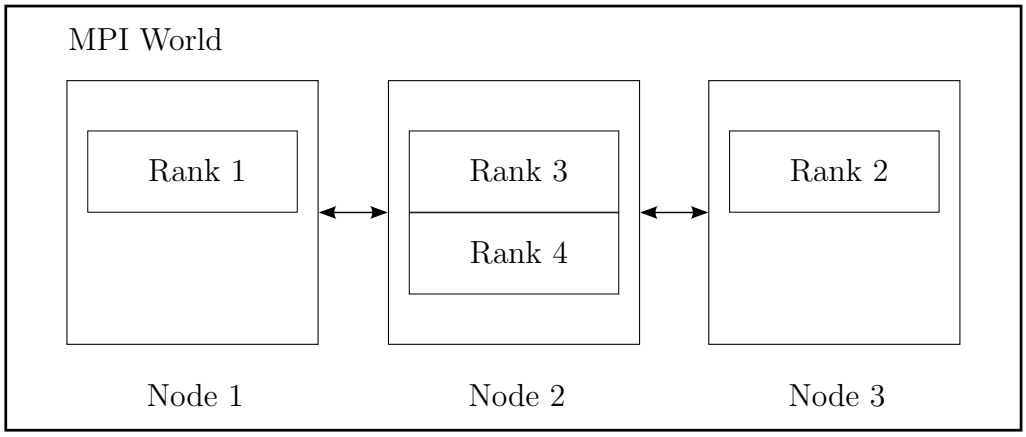

Figure 4.3: MPI migration load balancing

In the case of migration for load balancing, the steps are similar to the system management scenario. Instead of migrating all ranks from one node to another, only certain ranks are migrated to distribute the load on more nodes (see Figure 4.3 (page 61)). The *MPI_COMM_WORLD* would then be extended to include new nodes (*Node 3*) which can then be used to migrate ranks from an overloaded node (*Node 1*) to the node newly integrated in the world. Again, as mentioned in the previous system management scenario, all ranks communicating with the rank being migrated have to wait during the migration time. At the end of the migration process all ranks should continue to run with the benefit of having more resources available without interrupting the job.

Load balancing is a scenario which is particularly mentioned in the MPI Standard[29, 374]: "*MPI_COMM_SPAWN* starts MPI processes and establishes communication with them, returning an intercommunicator." According to [29, 376] up to *MPI_UNIVERSE_SIZE* processes can be started and when "a process spawns a child process, it may optionally use an `info` argument to tell the runtime environment where or how to start the process."

4.3 Open MPI

To verify the migration scenarios described in section 4.2 (page 59), Open MPI[30] is used as a sample implementation of the MPI standard. The decision to use Open MPI is based on multiple aspects of Open MPI. The openness of the development model in combination with its open license make it very easy to enhance Open MPI. Additionally, earlier versions of Open MPI included a framework to support C/R[37] with different C/R implementations.

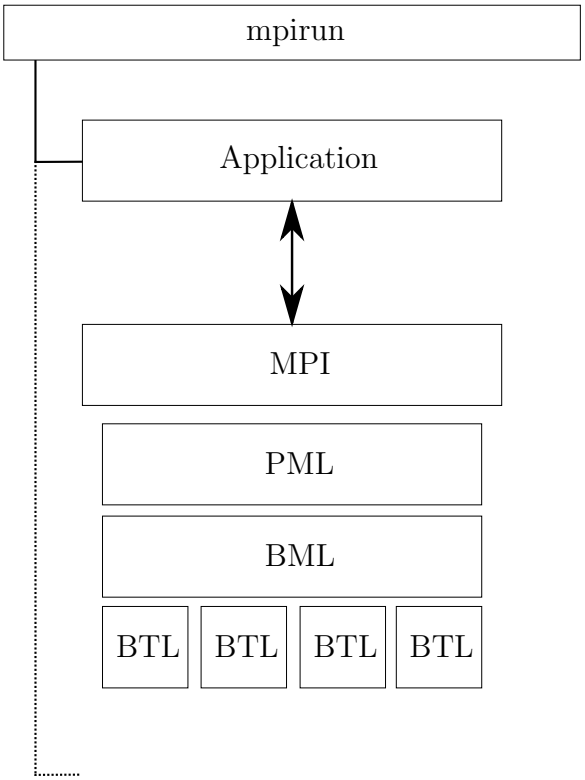

Figure 4.4: Open MPI layers

In Open MPI the Open Run-Time Environment (ORTE) starts up the number of processes (or ranks) which the user has requested (see Figure 4.5 (page 63)) and the applications which have been developed against the MPI layer as implemented by Open MPI are unaware of the actual communication method between the ranks (see Figure 4.4 (page 62)).

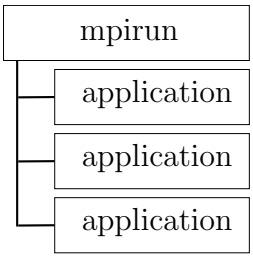

Figure 4.5: Open MPI process tree

Open MPI tries to select the best communication method available and the applications never know which Byte Transport Layer (BTL) is actually used. With this layering in place, ranks can be migrated to another node even if different communication hardware is used for future communication. The existing layering makes it possible for process migration inside a MPI process to be handled completely transparently for the application as the application only uses the MPI layer. This also means that the communication library (in this case Open MPI) needs to handle the migration of the communication channel, which means that the used C/R implementation does not need to handle inter-node communication.

Chapter 5

Results

This chapter presents the implementation specific details for migrating processes and demonstrates different programs which are migrated from one node to another. Some of the test cases are synthetic and especially developed in the scope of this work to demonstrate a specific feature and/or behavior while other test cases are based on programs actually used in an HPC environment. The following test cases are discussed in this chapter:

- UDP Ping Pong - A synthetic client/server test case using User Datagram Protocol (UDP) which is used to show the possibilities of continuing network communication with processes which have been migrated (see 5.3 (page 80)).

- memhog - Another synthetic test case used to find and demonstrate the minimal time required to migrate a process. This test cases allocates the desired amount of memory without changing the allocated memory (see 5.4 (page 81)).

- FENFLOSS - An application to compute laminar and turbulent, steady and unsteady incompressible flows. This application is in contrast to the other two test cases not synthetic test case. Process migration has to be useful not only in synthetic test cases but also with real workloads (see 5.5 (page 89)).

5.1 Approaches and Implementation

Before the actual implementation details are described, the first step is to define the requirements and constraints of the actual process migration implemented and used in the scope of this work.

5.1.1 Requirements - Constraints - Limitation

The requirement for the process migration is to be as transparent as possible and the decision to base process migration on C/R (see Chapter 3 (page 33)) means that the used C/R implementation needs to be as transparent as possible. During the process of deciding which C/R implementation process migration should be based on, the additional requirements upstream inclusion and not being too invasive emerged (see Chapter 3, section 3.4 (page 43)). This means the previously defined requirements are also used in the actual implementation.

To be able to focus on the essential parts of process migration and to avoid the necessity of having to study every corner case of process migration, which would be beyond the scope of this work, the following constraints are defined.

The first constraint is that process migration is only supported on systems sharing the same ISA. Different ISAs would not support the goal of being as transparent as possible. The only way to support process migration over ISA boundaries is to insert an additional layer between the process to be migrated and the actual hardware. This would lead to a kind of virtual machine which would at least require re-compilation of the source code and would be far from being as

transparent as possible. Following the constraint that the ISA has to be the same on all systems included in the process migration, comes the requirement that the version of the operating system has to be exactly the same on all involved systems. This includes all executables and libraries which are part of the process migration. Especially important is the availability and exact same version of shared libraries, because they will not be migrated but are expected to be on the destination system of the process migration. In addition to the same version, the executables and libraries have to be available under the same path.

It is also required that input or output files which are read or written are on a shared file system and available to all systems involved in the process migration. This way file descriptors do not need to be modified during the migration.

All these constraints seem to be contradictory to the previously stated goal of being as transparent as possible (see 3.4.4 (page 50)). For the targeted HPC environment all these constraints to not pose a real problem as the environment in a compute cluster is usually pretty homogeneous. Either due to the fact that a compute cluster usually consists of a large number of similar machines which provide the user with the same environment on all nodes of the compute cluster or the fact that, from an administrator's standpoint, a homogeneous environment is desirable to make it possible to manage such a large number of nodes. This means that there is a shared file system available on all nodes which provides executables, libraries and storage space for input and output files. The constraint of the same ISA is usually even desired as the compiler optimization for a certain CPU type can have negative effects on other similar CPUs ranging from degraded performance to not working at all (e.g., crashing).

In addition to the requirements and constraints mentioned the actual implementations have the limitation that the migrated process is not directly transferred from the source system's memory to the destination system's memory (direct migration see Figure 5.1 (page 68)). To simplify the implementation the process' data, which is extracted via C/R from the operating system, is not directly transferred to the destination system but first stored locally (indirect migration

see Figure 5.1 (page 68)). This limits the minimum migration time as it requires multiple copies of the process' data from the source system's memory directly to the destination system's memory, instead of a single copy.

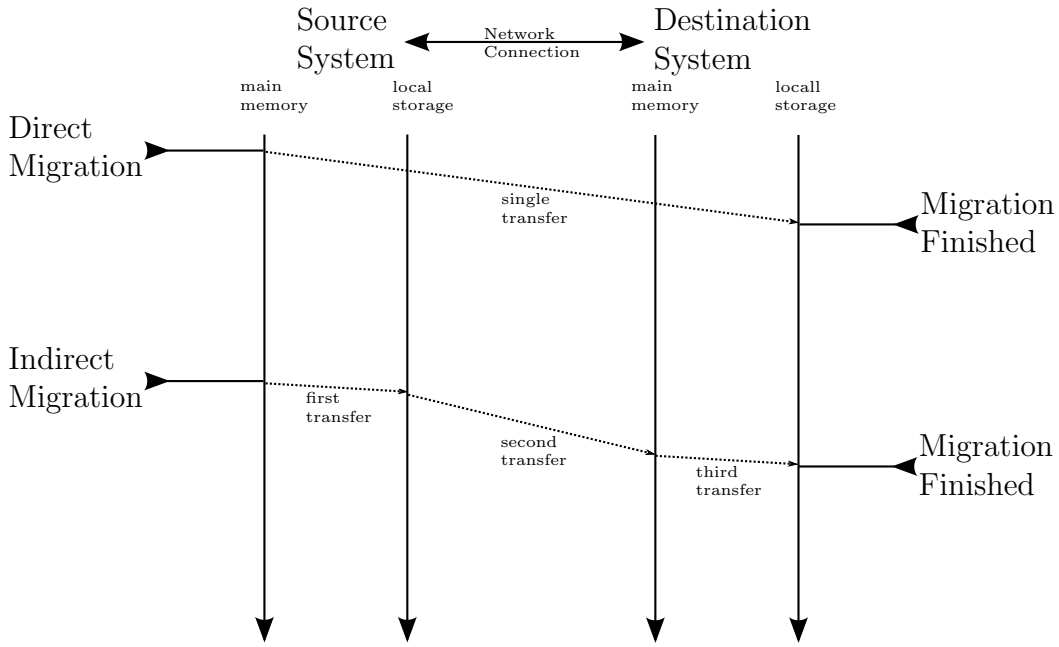

Figure 5.1: Direct vs. Indirect Migration

To reduce the time necessary to perform the required multiple copies for process migration, the process' data is stored on main memory based storage (e.g., a RAM drive based on *tmpfs*[42]) and on fast local Solid-state drives (SSDs). This way the implementation can be simplified by omitting the direct transfer from main memory to main memory and thus focusing more on the results than on an overly complex implementation. By using fast local storage, the limitations of the actual implementation are minimized.

Within the scope of this work process migration is bound by the following requirements, constraints and limitations:

- transparent - to use process migration with as many programs as possible
- upstream inclusion - to use process migration on many Linux distributions without additional requirements or software installation

- not too invasive - upstream inclusion is only possible with an approach which is not too invasive and thus easily accepted by the Linux community
- same ISA - to support process migration with different ISAs on the source and destination system would require an additional layer which would be able to intercept and translate instructions which do not exist on the destination system. Running in an HPC environment usually means similar systems and CPUs and thus systems with the same ISA.
- same operating system, binaries, libraries - again a constraint which is easy to accomplish in an HPC environment and most of the time already given.
- shared file system for input and output - see operating system
- RAM drive or SSD as fast local storage - this limits the time required to migrate a process but simplifies the implementation considerably.

5.1.2 Kernel-Space-Based Process Migration

The first attempt[1] to implement process migration was using the kernel-space-based (see 3.4.3 (page 48)) C/R. At that point in time the kernel-space-based C/R implementation had already not been maintained for over a year. It had seen its last update for the Linux kernel version 2.6.37 and was therefore, in the scope of this work, ported to Linux kernel version 3.2 which was the latest release at that time[2]. After porting the kernel-space-based C/R implementation to Linux kernel version 3.2 it was enhanced to support process migration[43]. For the proof-of-concept, a process doing calculations using the Floating Point Unit (FPU) as well as writing the results to a file was migrated.

The reason for these two functions (FPU and writing to a file) was to make sure that not only the registers of the CPU are migrated but also the registers of the FPU. Writing to a file was carried out for two reasons: first to make sure that

[1] late 2011, early 2012
[2] Linux kernel version 3.2 was released on 2012-01-04

the file-descriptors were correctly opened and re-opened and secondly to have an easy way to verify the results of the calculation after the migration.

This process migration proof-of-concept was writing the checkpoint image to a network socket instead of writing it to a local file. On the receiving side, the part to restore the checkpointed process was able to listen on a network socket.

With these changes it was possible to migrate a process from one system to another without using any storage system in between (direct migration see Figure 5.1 (page 68)). It proved that processes can be migrated by transferring the process directly from the source system's memory to the destination system's memory. It also proved that the concepts developed in this work up to this point can actually be implemented and that basing process migration on C/R is not only a theoretical possibility.

The proof-of-concept based on kernel-space-based C/R did not include any optimizations to reduce the time required to transfer the memory from the source to the destination system (as described in 3.2 (page 37)).

Unfortunately during the time of the successful proof-of-concept it became clear that, although the kernel-space-based C/R approach was feasible and could be enhanced to also support process migration, it would not be accepted by the Linux kernel community. The number of required changes had become too large and too complex to be accepted by the Linux kernel community (see 3.4.3 (page 48)).

The unclear future of the kernel-space-based process migration approach was the reason why this approach was not further followed and was replaced by the user-space-based process migration approach.

5.1.3 User-Space-Based Process Migration

As mentioned in Chapter 3 (page 33) the user-space-based C/R implementation CRIU has been selected as the most promising implementation. After the first successful attempts with kernel-space-based process migration early in this work,

it became clear that kernel-space-based process migration can be implemented, however this would have meant investing further effort in an approach which has not much chance of being further developed. At that point in time another C/R approach was discussed in the Linux kernel community (see 3.4.4 (page 50)).

CRIU had, at that point in time, a good chance of being accepted by the Linux kernel community and parts of CRIU have already been accepted upstream. This was the reason that the C/R implementation used as the basis for process migration in the scope of this work switched from the kernel-space-based C/R to the new user-space-based approach. In the context of this work CRIU was also enhanced to support process migration just like with the kernel-based C/R approach.

After an initial attempt to directly transfer the process image from the source system's memory to the destination system's memory, a less complex memory transfer method was selected. A working process migration was deemed more important than an early optimization. So instead of transferring the process image directly from the source system's memory to the destination system's memory, local storage of some kind is required to temporarily save the process image before it is transferred to its destination (indirect migration see Figure 5.1 (page 68)). Depending on the size of the process to be migrated and the memory available this could be a memory based storage (e.g., a RAM drive based on *tmpfs*[42]). For processes which require more memory so that using a RAM drive is not feasible (e.g., a process requires more than half of the available memory) another local storage is required. This work's implementation was performed using a fast local SSD as well as a RAM drive to temporarily store the process image before transferring it to the destination system.

CRIU based process migration makes it possible to use pre-copy (as discussed in Chapter 3, section 3.2.3 (page 40) and section 3.5 (page 53)). The Linux kernel offers an interface to mark all pages of a process as clean and the Linux kernel tracks which pages have been modified[44]. On subsequent runs of CRIU only dirty pages need to be transferred.

5.1.4 Process Identifier

Another problem with checkpointing and restoring processes or process groups is the PID. Restoring a process requires the checkpointed process to have the same PID that it had when checkpointed. This is necessary for processes which are part of a process group with parent-child relationships. The child processes are usually not aware of the PID of the parent process. In the case of starting child processes with `fork()` the child PID is returned to the parent process from `fork()`. It can then either ignore it or it can store it in any program structure it wants. This makes it impossible to restore the processes with a different PID if the goal of transparent C/R is to be fulfilled. If the program code of the application to be checkpointed were instrumented and recompiled, it should be possible to intercept everything related to the PID and restore the process or process group with a different PID. As the goal of this work is to offer a C/R implementation which is as transparent as possible, changing the PID during restore is out of scope. C/R based on DMTCP (see 3.4.2 (page 47)) for example uses a concept called virtual PID[21] which intercepts child creation by a pre-loaded library which is then used to resolve PID conflicts on restore.

Restoring a process on the same machine just after it has been checkpointed will probably work most of the time as the default PID space on Linux is 32768. Most of the time the required PID will be available and the restore will work flawlessly. For the case of migrating a process from one system to another the probability is still pretty low that a PID collision will occur.

If, however, the goal is to migrate hundreds of processes from one set of systems to another set of systems, the probability of one of the many migrations failing due to a PID collision increases. To make sure this kind of migration does not fail in the middle of the migration it needs to be verified that all PIDs needed are available on the target systems.

Another way to decrease the probability of a PID collision on Linux is to increase the number of available PIDs[3].

[3]The default value of 32768 can currently be increased by the factor of 128.

A simple solution is to reboot the destination system of the desired migration which frees all previously used PIDs. In a homogeneous environment as is often found in HPC, each system will use the same PIDs during boot. This means that all newly started applications will use the PIDs above the ones required to boot a system and these PIDs will therefore be free on a newly booted system.

Another option is to pre-allocate a certain range of PIDs for each application. The Linux kernel offers an interface with which it is possible to specify which value the next PID should have[45]. With the help of this interface the resource manager can then influence each node which is part of the application currently running, with which PID all processes on all related nodes will be started. In the case of a migration, this kind of pre-allocation can then also be applied to the destination node of the migration.

Seeing that PID collisions could present a serious limitation to migrating processes in a production environment, there are still multiple options which can be used to resolve this problem:

- Increase number of available PIDs
- Reboot destination system to bring PID usage to a well defined status
- Pre-allocation with the help of the resource manager to guarantee same start PID on all related systems

5.1.5 Environment Variables

Every process started has certain environment variables which define the environment a process is running in. These variables are defined by the parent process creating the target process. In most cases the parent process creating the target process, is a command shell providing a command-line interface (CLI) to start a process. This shell defines a certain set of variables which can be queried by the process started to obtain information about the environment it is running in. Depending on the shell used, the environment variables define information like:

- HOME - path of the user's home directory
- USER - the user name
- PATH - defines a list of directories used to search for commands to execute
- MAIL - path to the user's mail

Looking at environment variables especially in an HPC environment it is expected that most of the variables on all systems belonging to a compute cluster are the same (see 5.1.1 (page 66)). Therefore most of the existing environment variables do not pose a problem. If there are, however, variables which are host specific like the variable `HOSTNAME`, it poses a problem similar as with the PID. The environment variables of the process to be migrated could be changed by the tool executing the migration, but, just as with the PID (see 5.1.4 (page 72)), the target process could have read the variable and stored it anywhere in its memory. This makes changing host specific variables redundant as it cannot be ensured that the variable is not already stored somewhere in the target process' memory and therefore it will have no consequences whether or not the variable is changed.

To provide a clear solution to the problem of host specific environment variables this work requires the process or process groups to be migrated to not use host specific environment variables. In the case of `HOSTNAME` this can easily be solved programmatically by using `gethostname(2)`.

5.1.6 Security

An important but easily overlooked subject concerning process migration is security. On systems which have process migration enabled this could easily be misused to migrate an unwanted process to such a system. In addition it is also important that not only processes from authorized systems are accepted, but that the process's data can not be intercepted during the transfer. It should not be possible to modify the process's data during the migration and it also must

be guaranteed that the process's data can only be read by an authorized entity (e.g., the destination system).

If, in the scope of this work, process migration were to have been implemented using direct migration from the source's memory to the destination's memory (see Figure 5.1 (page 68)) it would have required the provision of authentication/authorization. For the indirect migration used it was possible to fall back on existing technology.

By using Secure Shell (SSH)[46] to transfer the migration data, a well known service is used which also has a good reputation concerning security. SSH provides a well audited authentication/authorization framework which is widely used and therefore well suited for a production environment where it is important that security issues are fixed in a timely manner without exposing the environment to known vulnerabilities. Another advantage of SSH is that it is usually already available in an HPC environment and used for most authentication/authorization tasks.

5.1.7 Implementation within Open MPI

The parallel process migration implementation is based on Open MPI (see 4.3 (page 62)). Open MPI has C/R mechanisms which are provided by the Modular Component Architecture (MCA) component Checkpoint/Restart Service (CRS)[47]. CRS provides interfaces for different C/R implementations to be used as the basis for fault tolerance.

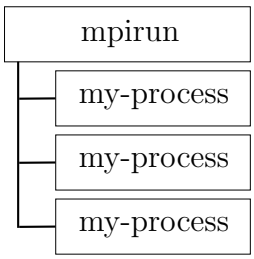

Figure 5.2: Open MPI process tree

Using CRS it is possible so signal an Open MPI process tree (see Figure 5.2 (page 75)) with *orte-checkpoint* that its processes should be checkpointed (see Figure 5.3 (page 76)).

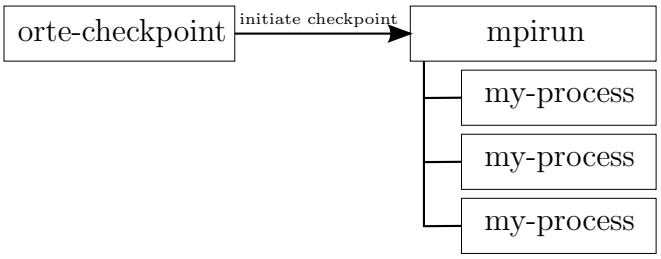

Figure 5.3: Open MPI initiate checkpoint

Depending on the configuration, the processes running under the control of *mpirun* are paused, checkpointed and will then continue or the processes will abort after being checkpointed. The counterpart to *orte-checkpoint* is *orte-restart* which can then be used to restart the processes under the control of *mpirun* later from one of the previously written checkpoints.

Open MPI's fault tolerance efforts were started in 2007[37]. Unfortunately there have been no development activities, according to the revision control system, concerning fault tolerance since 2010[48].

Starting in late 2013, in the scope of this work, the fault tolerance code paths in Open MPI were re-enabled and converted to use the new and changed interfaces of Open MPI which have not been adapted in the fault tolerance code paths. The most significant changes required were due to complete removal of blocking I/O operations in Open MPI. After switching to non-blocking I/O operations at all necessary locations and updating the fault tolerance code paths to compile and function again, a new CRS component was added which supports fault tolerance mechanisms using CRIU.

Based on CRIU it is now possible to implement process migration in a parallel environment using Open MPI.

5.1.8 Re-Parenting

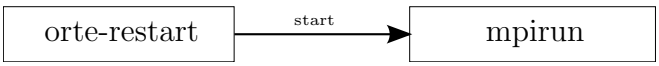

Figure 5.4: Open MPI initiate restart

Restarting a process with Open MPI is done with the command *orte-restart* (see Figure 5.4 (page 77)) which reads the checkpoint metadata. Using the metadata, *orte-restart* will start a new ORTE using *mpirun* which will start the corresponding number of processes previously dumped using *orte-checkpoint*. For each process checkpointed *mpirun* will spawn an *opal-restart* (see Figure 5.5 (page 77)) which sets up the environment and should then be replaced with the restarted process (like `exec()`).

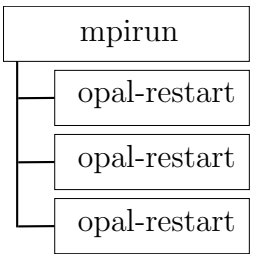

Figure 5.5: Open MPI spawn opal-restart

The problem with CRIU is that although it provides a library which has been integrated into Open MPI, this library is only a wrapper for Remote Procedure Call (RPC) to the CRIU daemon. This means that the restore will be performed from a process which is completely detached from the process initiating the restore and it also does not replace *opal-restart*. The newly restored process should be a child of the *mpirun* initiating the restore, however it is detached. Unfortunately there is, right now, no way to re-parent a process to another process in Linux which makes the CRIU restore functionality through the library call `criu_restore()` unsuitable for use in Open MPI.

The solution is to use the command line version of CRIU instead of the library to restore the checkpointed process. *orte-restart* starts the correct number of *opal-restart* child processes. These child processes restore the checkpointed process

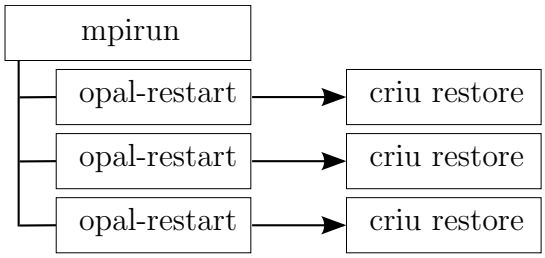

Figure 5.6: Open MPI calls CRIU for restore

using `exec()` to start CRIU which restores the processes with the desired PID (see Figure 5.6 (page 78)). Thus the restored process are child processes to the *mpirun* under whose control the restored process should be running.

Now the restored process tree should be the same as during checkpointing (see Figure 5.7 (page 78)).

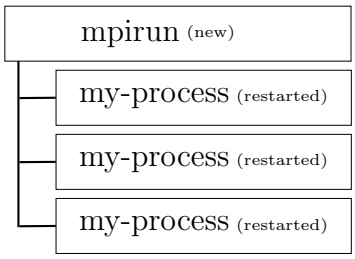

Figure 5.7: Open MPI process tree after restore

5.1.9 Output Redirection

CRIU tries to restore checkpointed process as close to the original state as possible. This also includes the treatment of the input and output channels. Running a process under Open MPI's control, however, requires special treatment of the input and output channels.

Open MPI redirects all output (and input) to the Head Node Process (HNP) so that the user sees all output from all involved processes. This is done by replacing the file descriptors for *stdin*, *stdout* and *stderr* with pipes and this way all output is proxied through *mpirun* so that the user has a single point to

get all the output from all involved processes. CRIU checkpoints those pipes in the exact state in which they were set up by Open MPI and expects that those pipes also exist during restore. The newly started *mpirun* (see Figure 5.4 (page 77)), however, will create new pipes to its spawned child processes which will have different identifiers to those in the checkpointed processes. This means that CRIU will fail during restart. To resolve the problems with output and input redirection, the information on the *mpirun*'s pipes is exported and CRIU can use this information with the help of a plugin which supports restore in an Open MPI environment.

5.2 Test and Validation Methods

Many of the following tests were performed on systems which are part of the compute cluster at the University of Applied Sciences in Esslingen. The systems boot via the network without a disk. The filesystem is provided via Network File System (NFS) (read-only) and a local SSD is available as fast local storage. The systems are equipped with 64GB of RAM and connected to a Gigabit Ethernet as well as to a Quad Data Rate (QDR) InfiniBand network.

The systems are equipped with Intel Xeon E5-2650 CPUs which have a memory bandwidth of 51.2 GB/s[49]. See Table 5.1 (page 79) for the memory bandwidth actually measured using the STREAM benchmark[50].

Number of Cores	Memory Bandwidth	Used Command
1	10 GB/s	`OMP_NUM_THREADS=1 ./stream`
2	20 GB/s	`OMP_NUM_THREADS=2 ./stream`
4	31 GB/s	`OMP_NUM_THREADS=4 ./stream`
8	48 GB/s	`OMP_NUM_THREADS=8 ./stream`
16	55 GB/s	`OMP_NUM_THREADS=16 ./stream`
32	50 GB/s	`OMP_NUM_THREADS=32 ./stream`

Table 5.1: Memory bandwidth measured using the STREAM benchmark

For reference the possible transfer rates, using IP, via Gigabit Ethernet and InfiniBand have been measured:

- Gigabit Ethernet: 942 Mbits/s (117.75 MB/s)
- InfiniBand: 15.2 Gbits/s (1.9 GB/s)

This means that all migrations will be bound by network transfer rates and not by memory bandwidth:

$time_{migration} = max(time_{memory}, time_{network})$

With the implementation used in the scope of this work (see 5.1.1 (page 66) and Figure 5.1 (page 68)) the time required for the migration is higher as it requires multiple copies:

$time_{migration} = time_{memory} + time_{network} + time_{memory}$

5.3 UDP Ping Pong - udpp

The first attempts to migrate a process were done using a test program doing a simple UDP communication. The goal of this test case was to make sure that process migration actually works even with a process doing network communication during the migration.

The test program is called *UDP Ping Pong - udpp* and does nothing more than send a UDP message to the specified host on which *udpp* has to be running in server mode. *udpp* in server mode writes the IP address of the *udpp* client on `stdout` and sends an answer back to the *udpp* client which also prints out the information about the communication with the *udpp* server (see listing 5.1 (page 81)).

The test setup is to run the *udpp* server on one system and start *udpp* in client mode on a second system (see Figure 5.8 (page 81) - step 1). During the communication between the server and the client system, the client is migrated to a third system (see Figure 5.8 (page 81) - step 2). After the client process has

```
Sending ping packet 1

Received pong packet from 172.30.200.251:34792
Data: This is pong packet 1
```

Listing 5.1: *udpp* client output

been migrated the client process on the third host continues to communicate with the *udpp* server (see Figure 5.8 (page 81) - step 3).

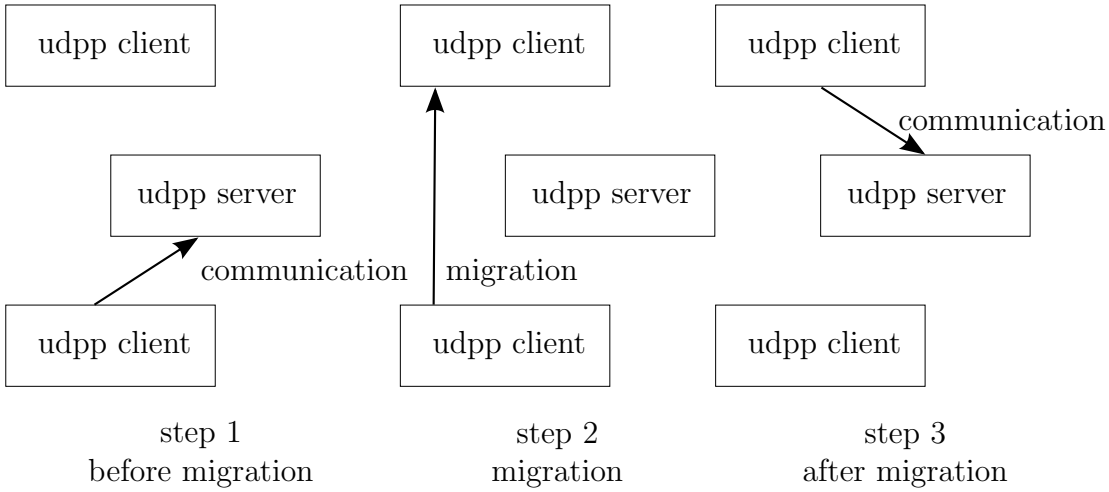

Figure 5.8: *udpp* migration

On the system where *udpp* is running in server mode the client's change of the IP address can be seen in the server's output (see listing 5.2 (page 82)).

Thus this simple test case demonstrates that it is possible to migrate a process during its UDP communication with another system to a third system without disrupting the communication.

5.4 memhog

Another simple test case for process migration was the program *memhog*. The program's only function is to acquire a certain amount of memory using `malloc()` and locking it using `mlock()`. This test case was designed with benchmarking

```
        Received ping packet from 172.30.200.252:58286
        Data: This is ping packet 6


        Sending pong packet 6
<--
-->

        Received ping packet from 172.30.200.205:58286
        Data: This is ping packet 7


        Sending pong packet 7
```

Listing 5.2: *udpp* server output

migration time in mind. It only allocates a certain amount of memory and does nothing else. Using a simple program like *memhog* makes it possible to study the required time to migrate a process of a certain size without any additional influences on the benchmark result. Using *memhog* the time to migrate processes with different memory usages was measured:

5.4.1 Via Ethernet with a local SSD

The first test setup was using *memhog* which was migrated from one system to another. This test setup was using following memory sizes on a system with 64GB RAM: 1GB, 2GB, 4GB, 8GB, 16GB, 24GB, 48GB.

The systems were both connected with Gigabit Ethernet to the same switch. The data to be migrated was temporarily stored on a local SSD and transferred using Gigabit Ethernet from the SSD of the first system to the SSD of the second system. From the SSD of the second system the data was then read to restore the *memhog* process. In addition to the pure time required for the whole migration the time required for a pre-copy (see 3.5 (page 53)) migration was also measured. Figure 5.9 (page 84) and Table 5.2 (page 83) show the time required to migrate a process (6 measurements). Two different times were measured:

- Migration time without pre-copy - this is the complete time required for migrating a process from the source to the destination system.

- Migration time with pre-copy - this is the time the process is quiesced during pre-copy migration and not actually running.

All following figures include a line called *Theoretical optimum* and *Implementation optimum* as a comparison to the measured values with the following meaning:

- The *Theoretical optimum* is the time required to migrate a process using direct migration: $time_{migration} = max(time_{memory}, time_{network})$
- The *Implementation optimum* is the time required using the implemented indirect migration: $time_{migration} = time_{memory} + time_{network} + time_{memory}$

Both *optimums* do not take any optimization (e.g., pre-copy) into account and will therefore always be slower than the pre-copy migration method also presented. They are valuable as they provide a source of comparison for the results and it is also interesting to see how close those two *optimums* are, which is related to the fact that the migration time is always bound by the network bandwidth.

Test Setup / GB	1	2	4	8	16	24	48
Migration time w/o precopy (s)	11	23	44	90	181	278	702
Migration time w/ precopy (s)	1	2	2	3	6	8	37
Theoretical optimum (s)	8.49	16.99	33.97	67.94	135.88	203.82	407.64
Implementation optimum (s)	8.69	17.39	34.77	69.54	139.08	208.62	417.24

Table 5.2: Comparison of migration time via Ethernet using SSDs with and without pre-copy

Figure 5.9 (page 84) shows that the pre-copy migration is much faster than migration without pre-copy. Especially for programs requiring a large amount of memory (24GB and 48GB) the pre-copy migration is over 20 times faster. It is important to mention that the *memhog* test case is only a synthetic test case which does not modify its memory, but it demonstrates the benefits of pre-copy migration. The gradient change in the migration time for processes larger than

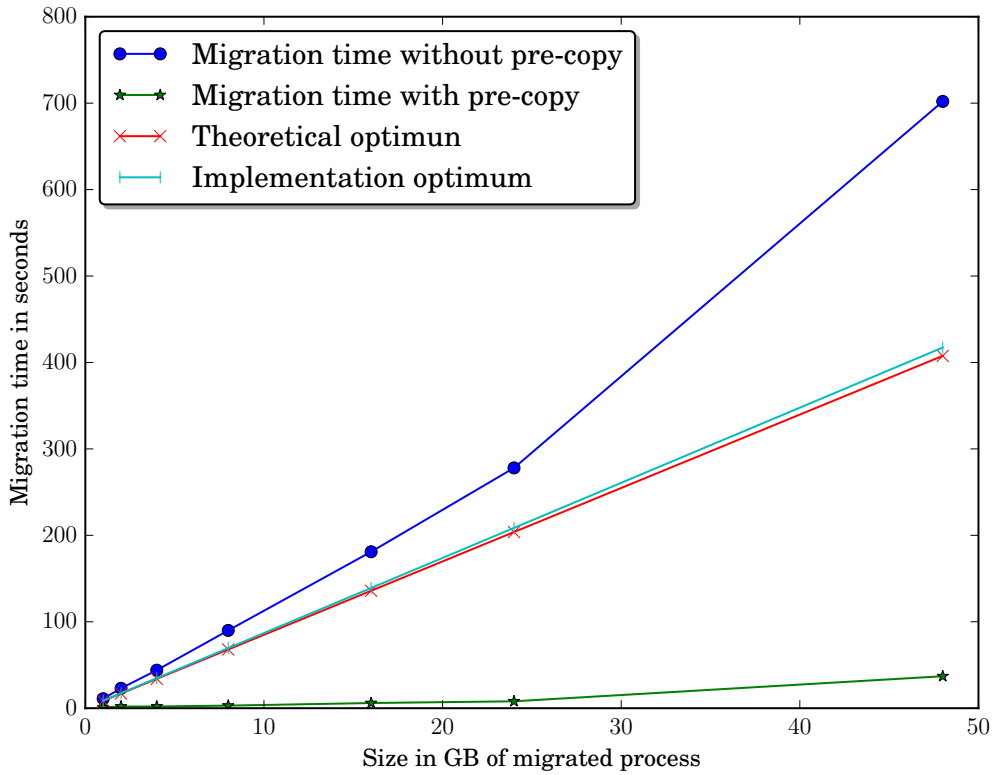

Figure 5.9: Comparison of migration time via Ethernet using SSDs with and without pre-copy

24GB is due to file caching effects in the operating system. Although the data to restore the process is stored on a local SSD, the operating system caches the file accesses, and for processes up to 24GB there is enough memory to cache all related files.

5.4.2 Via InfiniBand with a local SSD

This is the same test setup as in 5.4.1 (page 82). Instead of using Gigabit Ethernet to transfer the data, InfiniBand is used. The different data rates, as described in 5.2 (page 79), are the main reason for lower migration times with and without pre-copy (see Figure 5.10 (page 85) and Table 5.3 (page 86)). The pre-copy migration is faster by the same order of magnitude as the migration

without pre-copy.

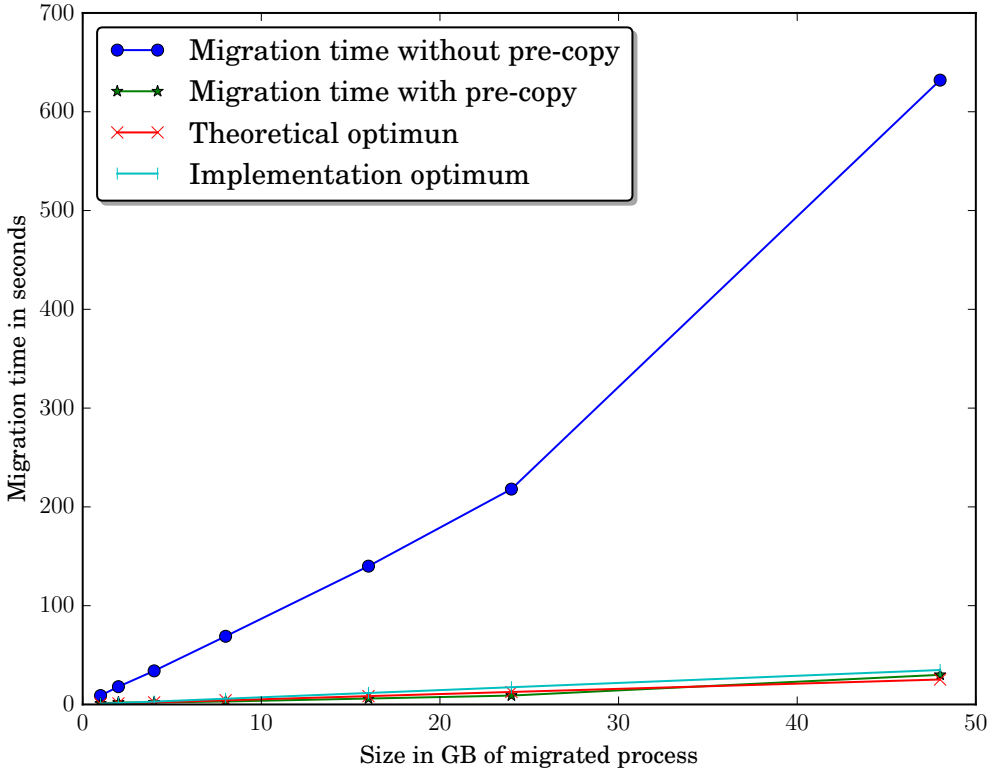

Figure 5.10: Comparison of migration time via InfiniBand using SSDs with and without pre-copy

5.4.3 Via Ethernet with a local RAM drive

This test setup using `memhog` is also communicating via Gigabit Ethernet but instead of a locally connected SSD, the migration data is stored on a RAM drive. The system has the same amount of memory as in the previous test setup (64GB) and using a RAM drive reduces the possible test case size to: 1GB, 2GB, 4GB, 8GB, 16GB, 24GB. For each test case, 6 measurements were made.

The results of the measurements can be seen in Figure 5.11 (page 87) and Table 5.4 (page 86). Just like in the previous test setups the pre-copy migration for this kind of application is many times faster.

Test Setup / GB	1	2	4	8	16	24	48
Migration time w/o precopy (s)	9	18	34	69	140	218	632
Migration time w/ precopy (s)	1	2	2	3	6	9	30
Theoretical optimum (s)	0.53	1.05	2.11	4.21	8.42	12.63	25.26
Implementation optimum (s)	0.73	1.45	2.91	5.81	11.62	17.43	34.86

Table 5.3: Comparison of migration time via InfiniBand using SSDs with and without pre-copy

Test Setup / GB	1	2	4	8	16	24
Migration time w/o precopy (s)	11	21	41	82	164	249
Migration time w/ precopy (s)	1	2	2	2	5	8
Theoretical optimum (s)	8.49	16.99	33.97	67.94	135.88	203.82
Implementation optimum (s)	8.69	17.39	34.77	69.54	139.08	208.62

Table 5.4: Comparison of migration time via Ethernet using a RAM drive with and without pre-copy

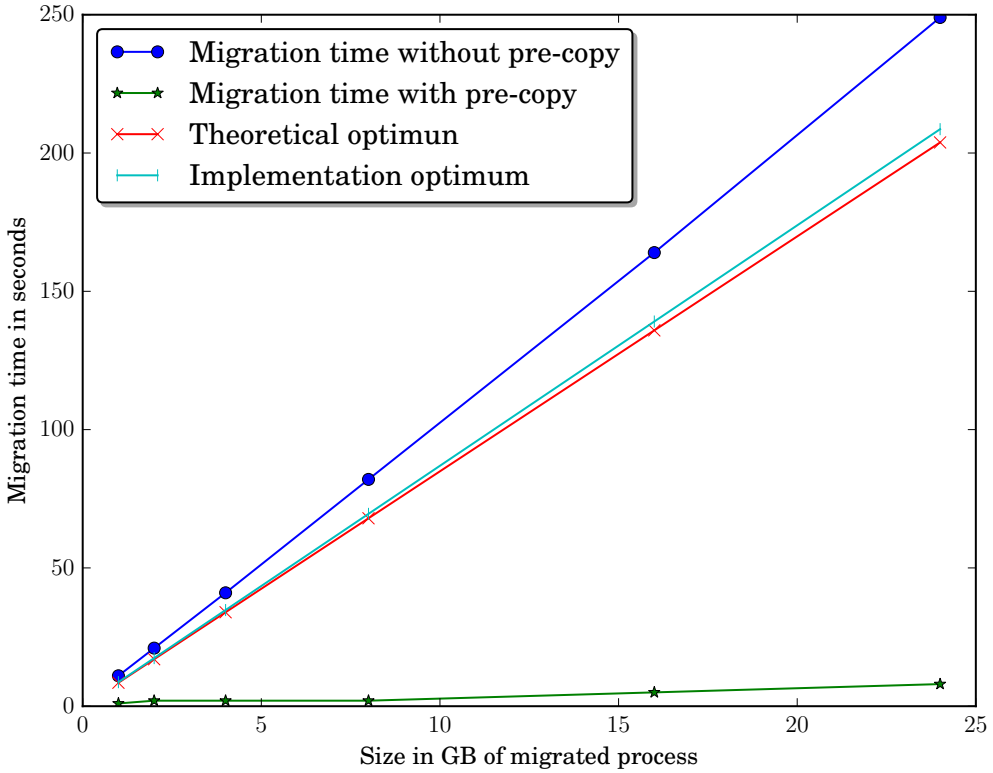

Figure 5.11: Comparison of migration time via Ethernet using a RAM drive with and without pre-copy

5.4.4 Via InfiniBand with a local RAM drive

The only difference between this test setup and the previous test setup is that now the data is transmitted via InfiniBand instead of Gigabit Ethernet. The test setup has used following test case sizes: 1GB, 2GB, 4GB, 8GB, 16GB, 24GB. For each test case, 6 measurements were made.

Using InfiniBand to transfer the migration data makes the migration even faster.

5.4.5 Test Case Summary with *memhog*

It is important to remember that *memhog* is a synthetic test case as it only allocates memory and the memory does not change during the program's life-

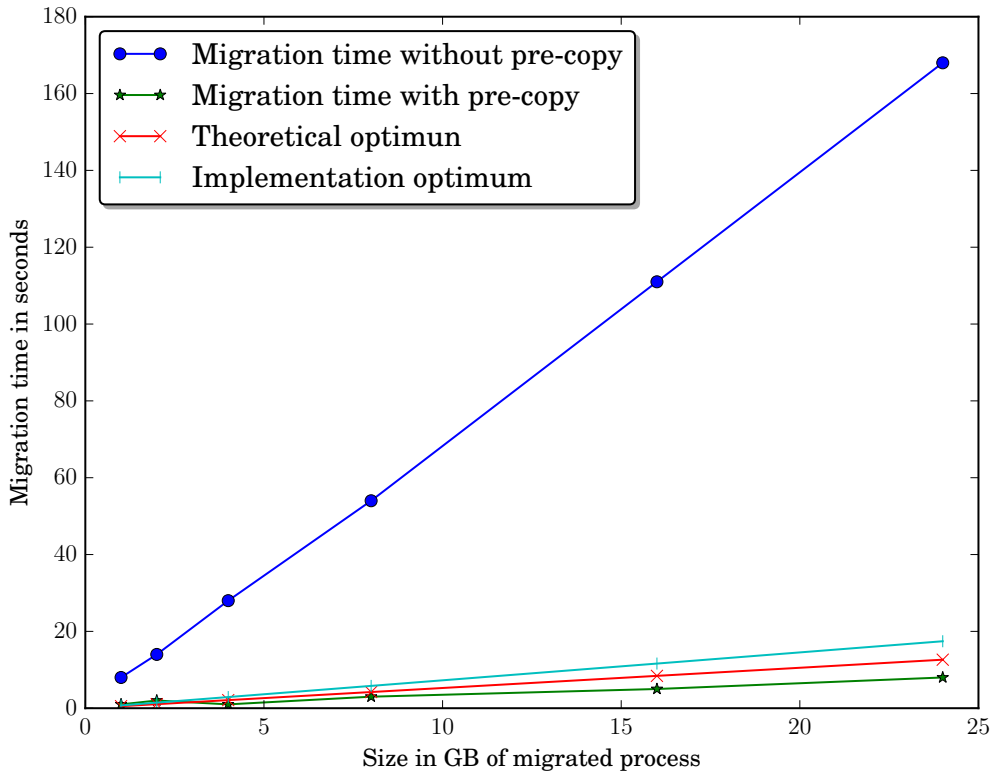

Figure 5.12: Comparison of migration time via InfiniBand using a RAM drive with and without pre-copy

time. On the other hand it is a good test case for demonstrating the best values possible for migrating a process and for comparing these values to the theoretical limits. Knowing these values helps to interpret the migration times for real applications.

In Figure 5.13 (page 90) the different measurements for pre-copy migrations are combined in one figure and it can be seen that the underlying storage and network technology make no big difference in the time required to migrate a process. Even supposedly faster storage and network technology make no noticeable differences which is due to the coarse resolution of the time measurement (seconds). For the process migrated with 48GB the difference between InfiniBand and Gigabit Ethernet is more clearly visible and exactly what has been expected (see 5.2 (page 79)).

Test Setup / GB	1	2	4	8	16	24
Migration time w/o precopy (s)	8	14	28	54	111	168
Migration time w/ precopy (s)	1	2	1	3	5	8
Theoretical optimum (s)	0.53	1.05	2.11	4.21	8.42	12.63
Implementation optimum (s)	0.73	1.45	2.91	5.81	11.62	17.43

Table 5.5: Comparison of migration time via InfiniBand using a RAM drive with and without pre-copy

In Figure 5.14 (page 91) the results from all migrations without pre-copy are displayed. All the results scale linear up to a migration size of 24GB. The time required to migrate 48GB, using a SSD as temporary storage, does not scale linear, compared to previous results, because of file system cache effects.

Both results, with and without pre-copy, are valuable as the minimum time required to migrate a process in comparison to the theoretical values (see 5.2 (page 79)).

5.5 FENFLOSS

To demonstrate the usefulness of process migration not only in synthetic test cases like `udpp` (see 5.3 (page 80)) and `memhog` (see 5.4 (page 81)) but also in a real world scenario the application Finite Element based Numerical Flow Simulation System (FENFLOSS)[51] was used.

"The numerical flow simulation software FENFLOSS (Finite Element based Numerical Flow Simulation System) is being developed at the IHS[52] since the early 80s. It is used to compute laminar and turbulent, steady and unsteady incompressible flows. Complex geometries may be meshed easily with unstructured meshes due to the highly flexible Finite Element approach. Scale and mesh adaptive turbulence models enable it to reproduce unsteady turbulent flow behaviour and associated pressure fluctuations very accurately. [...]

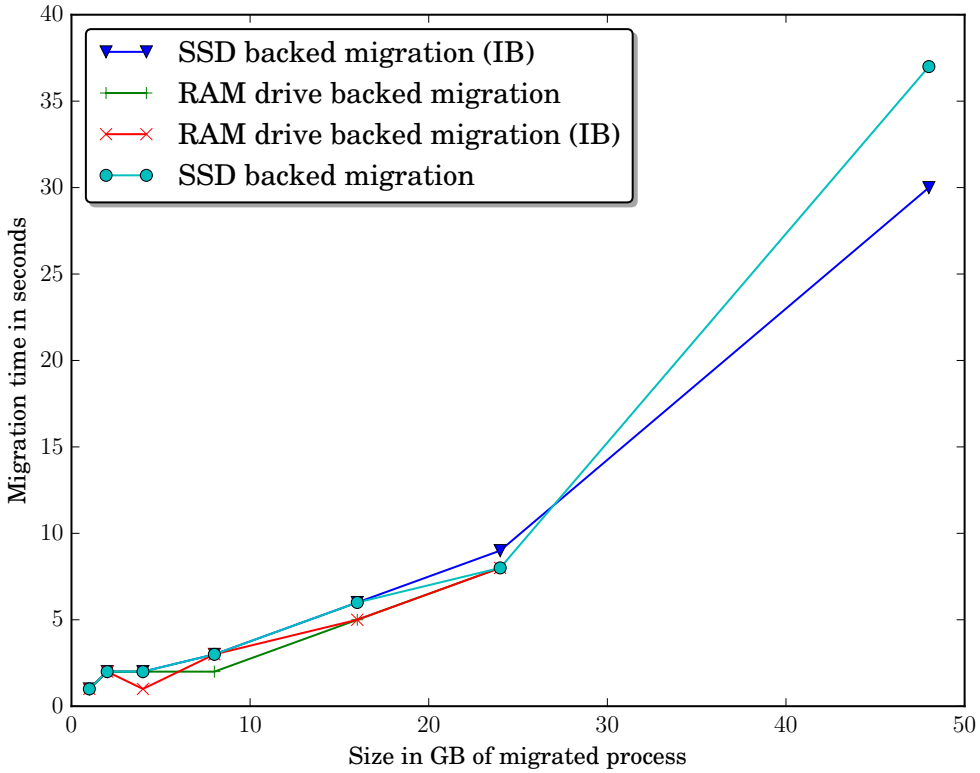

Figure 5.13: Comparison of migration time using pre-copy

FENFLOSS is used to simulate any kind of incompressible flows, especially in hydraulic machinery."[53]

FENFLOSS can be run in a serial mode using only one core and with the help of MPI also in a parallel mode using multiple systems and cores.

The following results are based on FENFLOSS running in serial mode. FENFLOSS was migrated, using a RAM drive and InfiniBand, at different states of its runtime. These different states can be classified as:

- **Initialization** - During application startup FENFLOSS reads the simulation's geometry from its configuration files and sets up the internal data structures in the memory.These memory structures will later be used for the actual simulation.

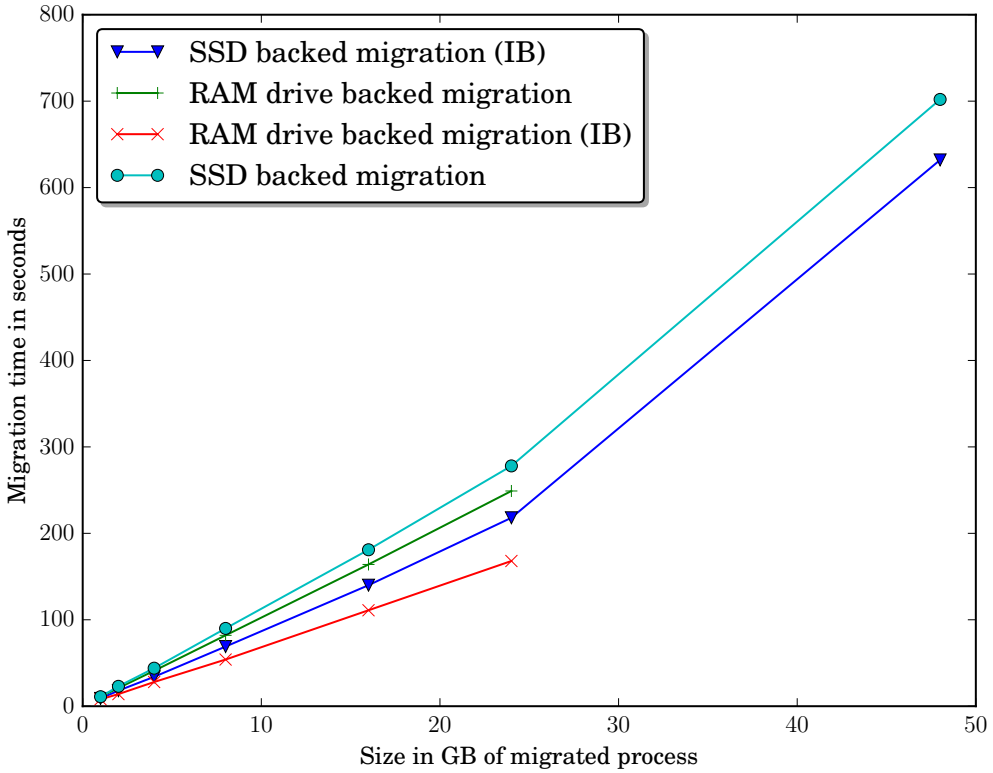

Figure 5.14: Comparison of migration time without pre-copy

During this phase of the application's runtime the content of the memory changes heavily which means that optimizations like pre-copy increase the time required to migrate the application.

- **Stabilization** - The next state in the application's runtime is after the initial creation of the data structures in the memory. These data structures are now optimized for the actual simulation. During this phase the content of the memory changes moderately and process migration with or without optimizations requires approximately the same time.

- **Calculation** - This is the last state which has been identified. After the initial setup and optimization of the internal data structures the actual calculation is running. During this phase the content of the memory changes only lightly and process migration with optimization requires noticeable less time than unoptimized process migration.

To clearly show that the optimization at the wrong point in time can lead to worse results FENFLOSS was migrated at each of the mentioned application states (Initialization, Stabilization, Calculation).

Figure 5.15 (page 92) and Table5.6 (page 93) display the different amounts of memory which have been transferred during process migration for each application state previously described. On the left there is the amount of data transferred without pre-copy optimization (*Transfer size without pre-copy*). On the right, for each application state, is the amount of data transferred using pre-copy optimization (*Second transfer*).

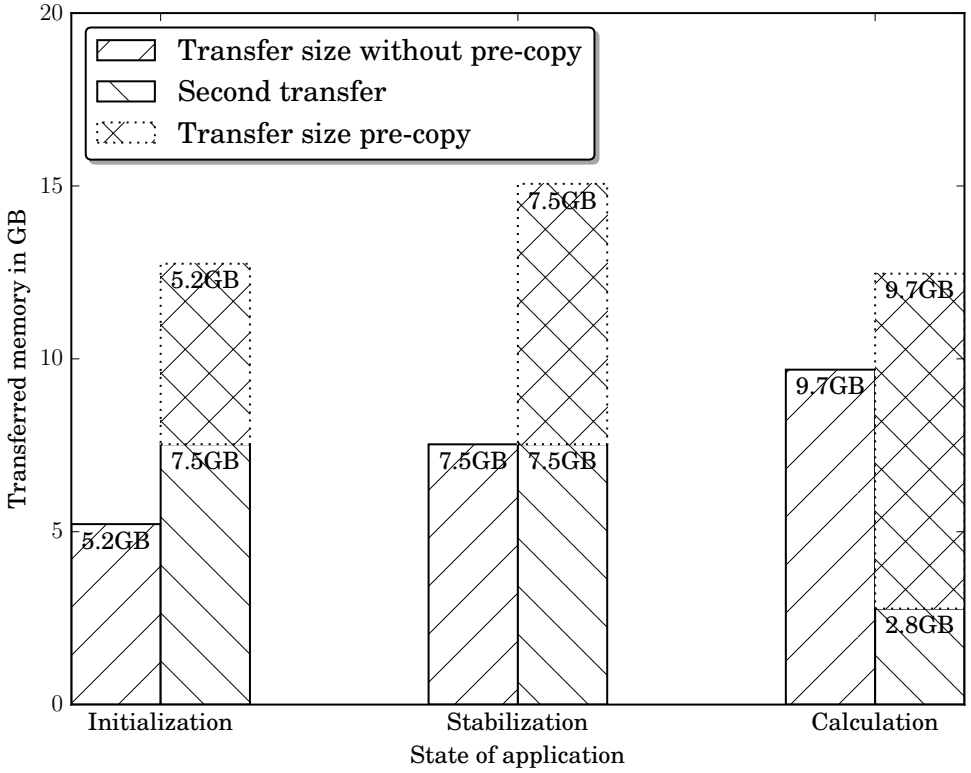

Figure 5.15: FENFLOSS memory transferred during migration with and without pre-copy

These two representations of the transferred memory are displayed using solid lines as these two transfers are also relevant for the time measurement. The amount of memory displayed using the dotted lines (*Transfer size pre-copy*) is

Application State	Transfer size without pre-copy	Transfer size pre-copy	Second transfer
Initialization	5.2GB	5.2GB	7.5GB
Stabilization	7.5GB	7.5GB	7.5GB
Calculation	9.7GB	9.7GB	2.8GB

Table 5.6: FENFLOSS memory transferred during migration with and without pre-copy

only shown for comparison. The unoptimized transfer and the transfer of pre-copy data have been started at the same point in time. The *Second transfer* has been started after the pre-copy transfer has finished. During pre-copy transfer the application continues to run.

During the application state *Initialization* pre-copy optimization requires more time to migrate the application than without optimization. The initial amount of memory transferred using pre-copy optimization is the same as for the whole unoptimized migration. After the pre-copy operation the actual migration takes place and the memory has significantly changed so that more time is required using the pre-copy optimization.

In the *Stabilization* phase unoptimized and pre-copy optimized process migration takes the same time. Memory changes at a slower pace but still considerably, so that during the *Second transfer* the same amount of data has to be transferred. Migration times are the same but pre-copy optimization still requires more data to be transferred.

The last application state (*Calculation*) demonstrates the benefits of pre-copy optimization. FENFLOSS now requires almost 10GB of memory and during the *Second transfer* of the pre-copy optimization only a fraction of the process' whole memory has to be transferred and thus pre-copy optimization decreases process downtime significantly.

Corresponding to the amount of memory transferred Figure 5.16 (page 94) and Table 5.7 (page 95) as well as Table 5.8 (page 95) display the time during which the application has to be suspended. In the unoptimized case this is the same as the migration time.

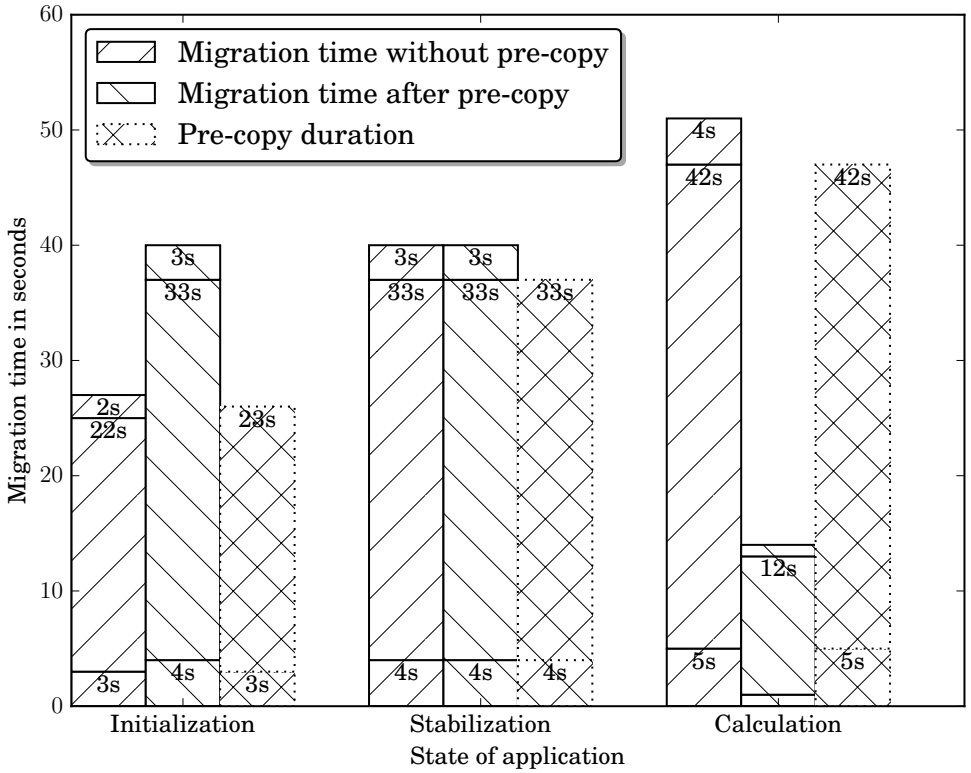

Figure 5.16: FENFLOSS migration duration with and without pre-copy

Using pre-copy optimization the application can continue to run during a certain phase of the migration.

The results for migration duration are very similar to the results of the amount of memory which had to be transferred. The figure displays the required migration time for the three identified application states (Initialization, Stabilization, Calculation). For each state the unoptimized migration duration is shown (*Migration time without pre-copy*) as well as the pre-copy optimized migration duration (*Migration time after pre-copy*). For comparison the time required for the *Pre-copy duration* is shown with a dotted line as during this time the application continues to run.

To better distinguish which phase of the migration requires which amount of time the different phases are represented in Figure 5.16 (page 94) and Table 5.7

Application State	Migration time without pre-copy			Migration time after precopy			Pre-copy duration	
	pre	dump	restart	pre	dump	restart	pre	dump
Initialization	3s	22s	2s	4s	33s	3s	3s	23s
Stabilization	4s	33s	3s	4s	33s	3s	4s	33s
Calculation	5s	42s	4s	1s	12s	1s	5s	42s

Table 5.7: FENFLOSS migration duration details with and without pre-copy

(page 95). The first time measured (bottom) is the time which is required to checkpoint (dump) the running application to the local storage. The second time measured (middle) is the time which is required to transfer the application from the source system to the destination system. The last time measured (top) is the time required to restart the application on the destination system. Even using InfiniBand this also demonstrates that the transfer from the source to the destination system requires the most time compared to the time required to copy the process' data from the kernel-space to the user-space.

Depending on the application's state pre-copy optimization can increase the time required to migrate a process as seen in the state *Initialization*. On the other hand pre-copy optimization can also have enormous advantages as seen in the state *Calculation* where the time during which FENFLOSS is suspended significantly shorter using pre-copy optimization.

Application State	Migration time without pre-copy	Migration time after precopy	Pre-copy duration
Initialization	27s	40s	26s
Stabilization	40s	40s	37s
Calculation	51s	14s	47s

Table 5.8: FENFLOSS migration duration overview with and without pre-copy

Chapter 6

Conclusion and Outlook

6.1 Conclusion

A common approach to satisfying the ever increasing demand for computational resources is to increase the number of compute nodes in a cluster. Tools to support system management tasks have unfortunately not kept pace with the ever increasing number of components and new approaches to system management are required to better support systems with such a large number of components.

Unfortunately, existing tools for the efficient handling of such a large number of components cannot be used in an HPC environment. These tools are usually based on virtualized environments and one of the main advantages of virtualization is the ability to migrate running virtual machines from one virtualization host to another without interrupting the virtual machine and the application running inside it. Despite having many advantages, virtualization is still not widely used in HPC environments. There are, especially in cloud computing environments, virtualized HPC systems but virtualization is still not very common in locally maintained compute clusters. This is also connected to the fact that virtualization in combination with specialized low latency interconnects (e.g., InfiniBand) still do not offer all the benefits which are provided by virtualization. This and the fact that there are still overheads in virtualization, motivated

studies to use thinner virtualization layers like para-virtualization and container based virtualization.

Continuing the trend towards thinner virtualization techniques leads to the complete avoidance of any hypervisor while still employing virtualization advantages. Process migration is one possible approach which provides the flexibility of virtual machine migration without the penalties. Instead of migrating a whole virtual machine with a complete operating system only the affected processes are migrated. So instead of migrating a whole operating system, only the required parts are migrated which also means that the amount of memory to be migrated is less. Being in an HPC environment also means that a parallel process usually uses existing libraries to communicate between the processes (e.g., MPI). As a result this makes it possible to handle the communication migration in this library and removes a direct dependency on the communication hardware and possible connected problems concerning communication migration.

This means that process migration should provide the advantages of virtualization without the hypervisor induced overhead. System management tasks like migrating application off a cluster node to be maintained are not limited by the application's run-time and can be performed any time necessary without interrupting the running application. Process migration also makes it possible to dynamically balance the load more efficiently in order to improve the utilization of the existing resources. This concerns not only CPU resources, but also resources like communication hardware, power and cooling. Process migration has also the benefit that the amount of data to be migrated is much less as only the parts actually affected have to be migrated.

To provide process migration in an HPC environment this work uses C/R based process migration. Different existing C/R implementations were studied and CRIU was selected as the most promising implementation. In addition to being as transparent as possible, it is also already included in the official Linux kernel. With the acceptance of the Linux kernel community it was possible to include this C/R implementation in an existing Linux distribution (Fedora 19[27]). The inclusion in a Linux distribution is important in that it provides

the opportunity to be available in Linux distributions with enterprise features (stable software and interfaces, long term support) which are actually used in many HPC production environments. This presents the prospect of C/R being available in HPC in the near future systems without the requirement to install additional core functionality like C/R, which might not be supported by the operating system vendor and which also might introduce instabilities.

With the help dirty pages tracking[44], process migration based on CRIU can use pre-copy optimization to decrease the time during which the process is suspended in order to be migrated.

The process migration used in the scope of this work is an indirect migration instead of copying the process' memory directly from the source system to the memory of the destination system. This means that rather than copying the data once, it has to copied three times (kernel-space to user-space, from the user-space via the network to destination's system user-space and once more from user-space to kernel-space). This does not mean that the migration time is tripled due to the fact that the kernel-space-to-user-space and user-space-to-kernel-space copies take much less time compared to the network transfer time. In addition, since the migration time is not tripled, a much simpler implementation is possible. This makes it possible to use existing tools to perform the actual data transfer (including authentication and encryption). Implementing direct migration would have meant integrating network transfer, authentication and encryption in kernel-space which would have meant a much more invasive change to the operating system. This would have made community acceptance much harder and could have introduced instabilities in the operating system. For this reason the simpler approach of indirect migration has been selected. The avoidance of unnecessary instabilities in the operating system was also one of the reasons why C/R based process migration has been further studied and not preemptive multitasking based process migration.

Process migration as described in this work has a few limitations. The first limitation is related to the PID. Due to the fact that not only single processes can be migrated but whole process groups with parent-child relations, the restored

processes need to have the same PID. The main reason is that the parent process can potentially store the child process PID anywhere in its memory and therefore it cannot be controlled by the C/R environment. One method (like implemented by DMTCP) is to intercept system calls like `fork()` and provide the process with a virtual PID. This would, however, contradict the goal of being as transparent as possible. Requiring the same PID on restart also means that the migration can fail if the PID is already in use on the destination system. Fortunately this limitation can be worked around by increasing the number of available PIDs or by influencing which PIDs will be used for the newly started processes (reboot, pre-allocation[45]).

Another limitation of the presented solution is related to environment variables. If an application reads an environment variable on startup and stores its value in the application's memory, the checkpointing implementation can no longer influence this value. This is especially problematic for host specific environment variables like `HOSTNAME` which will change after the process has been migrated. A requirement is therefore it is required that processes which want to be migrated do not store host specific environment variables in their local memory. In the case of the environment variables `HOSTNAME` an easy work-around is to use the function `gethostname()` instead.

In addition to the PID and the environment variables related limitations, processes can only be migrated if they are running on similar set up systems. The ISA needs to be the same and this also means that the CPUs will be very similar. Supporting different ISAs would require an abstraction layer or some kind of virtual machine which can translate or replace instructions which are not available on the CPU of migration destination. This would contradict the goal of this work to further reduce overheads and cause wasted CPU cycles to translate or replace non-existing instructions. Working with HPC systems which usually provide a homogeneous environment the requirement for the same ISA is easily fulfilled. In addition to the same ISA, the systems involved in the process migration need to have the same version of the operating system as well as the same version of all loaded libraries. Only the actual application is migrated and it still expects all loaded libraries at the same place in exactly the same version.

This again is a requirement which is fulfilled by most HPC systems. In addition to the same ISA and operating system version, a shared file system is required to provide input and output files in the same location on all systems involved in the process migration.

To support migration of parallel applications, the decision was made to support MPI parallelized applications. The MPI standard offers the required functionality to spawn additional processes and MPI parallelization is a common approach for parallelizing applications in an HPC environment. Open MPI was chosen as the most suited MPI implementation because of its open development model and license. Another advantage of Open MPI is that it used to have a modular fault tolerance framework which could be used as the basis for process migration of parallel jobs. The fault tolerance framework no longer working was re-enabled and extended to support CRIU based checkpointing and restarting. Unfortunately, due to time constraints it was not possible to provide process migration of parallel applications. However with CRIU as a C/R implementation integrated into Open MPI, important steps towards parallel process migration have been achieved. These initial steps performed in the scope of this work make it possible to support process migration in Open MPI and thus enable the integration of this functionality in resource schedulers. With the integration of process migration functionality in resource schedulers and intelligent management frameworks, system management tasks like software upgrades or hardware maintenance can be performed at any time, since the processes running on the affected systems can be migrated to another system at any time. With the integration of process migration and intelligent management frameworks[54] it should also be possible to predict failures and pro-actively migrate processes off the faulting components.

Using different application the non-parallel process migration approaches presented were implemented and tested. With two synthetic test cases it was possible to test and demonstrate concepts while using a real application (FENFLOSS) it was shown that process migration works with production level applications. In contrast to the synthetic test cases it became clear that the optimization used can lead to worse results in some cases. Furthermore it can enormously speed

up the migration of processes in other constellations.

6.2 Outlook

With the result that process migration can offer functionality in HPC environments which until now did not exist, multiple questions emerge which have not been approached in the scope of this work. One of the goals was to reduce the overhead caused by hypervisors or virtualization in general. With process migration in a homogeneous environment there is no need for an overhead but it is not clear how a parallel application reacts if one or several of its processes are suspended for a certain time and then resume on another system. Even if the downtime of the process to be migrated is relatively short it means, in most cases, that all other processes have to wait. Every communication with the suspended process is blocked until the migrated process resumes. As the application has probably not been designed to handle communication timeouts, it could be the case that the parallel application aborts if the migration takes too long. So the application and the MPI implementation have to know how to handle migration related timeouts. The effect of process migration on an application which is parallelized over thousands of cores also needs further study. If a single process of such an application is migrated this can mean that thousands of CPU cores have to wait until the migration is finished and this can mean that not only the time of the CPUs involved in the actual migration is lost, but this lost time has to be multiplied by the cores waiting for the process migration to finish. This can lead to situations where process migration leads to enormous amounts of wasted CPU cycles and this needs to be clear before initiating the migration.

Process migration as implemented in the scope of this work is indirect and C/R based. This implementation has been selected to avoid unnecessary instabilities of the operating system related to changes to very central operating system components like the process scheduler. An interesting study, however, would be how process migration based on preemptive multitasking concepts compares

to C/R based process migration. Even if the theoretical duration difference between direct and indirect process migration is minimal, it would be interesting to compare direct process migration without the additional copies between kernel-space and user-space to indirect process migration.

With the opportunity to migrate parts of a parallel process, process migration can also be used in combination with an intelligent monitoring system[54] to detect failures and pro-actively migrate processes off systems which are about to fail.

Especially in an HPC environment which often employs C/R as a means of fault tolerance, C/R has significant drawbacks. The biggest disadvantage in connection with C/R is that up to 50% of the available CPU cycles can be lost.

A checkpoint is usually written to a centralized storage system which is accessible by all cluster nodes and during the time all these nodes are storing their checkpoint image, the CPUs are idling and those CPU cycles are lost. The larger the corresponding calculation is, the more nodes try to simultaneously save their checkpoint image to the same storage system. This leads to a very high I/O load on the storage system which means that all nodes related to the calculation have to wait even longer until the checkpoint operation finishes. This is related to the fact that although the size of the storage systems has kept pace with the available memory, the bandwidth unfortunately has not[55].

With storage systems comparably slow to main memory in combination with the wrong interval at which checkpoints are taken[56], up to 50% of the available CPU cycles can be lost waiting for checkpoint operations to finish[40].

The existing drawbacks concerning C/R have been identified previously[57] and one of the commonly suggested solutions is pro-active process migration[41] instead of checkpointing with re-active restarting. Process migration, however, cannot be the only fault tolerance mechanism as it does not protect from data loss if the fault cannot be predicted.

Bibliography

[1] Inc. VMware. VMware vSphere vMotion Architecture, Performance and Best Practices in VMware vSphere 5. Technical report, 2011.

[2] KVM - Kernel-based Virtual Machine. Migration - kvm. `http://www.linux-kvm.org/page/Migration`, 2012.

[3] Todd Deshane, Zachary Shepherd, J Matthews, Muli Ben-Yehuda, Amit Shah, and Balaji Rao. Quantitative comparison of Xen and KVM. *Xen Summit, Boston, MA, USA*, pages 1–2, 2008.

[4] Jianhua Che, Yong Yu, Congcong Shi, and Weimin Lin. A synthetical performance evaluation of openvz, xen and kvm. In *Services Computing Conference (APSCC), 2010 IEEE Asia-Pacific*, pages 587–594. IEEE, 2010.

[5] Andrea Bastoni, Daniel P Bovet, Marco Cesati, and Paolo Palana. Discovering hypervisor overheads using micro and macrobenchmarks.

[6] Lamia Youseff, Rich Wolski, Brent Gorda, and Chandra Krintz. Paravirtualization for HPC systems. In *Frontiers of High Performance Computing and Networking–ISPA 2006 Workshops*, pages 474–486. Springer, 2006.

[7] Miguel G Xavier, Marcelo V Neves, Fabio D Rossi, Tiago C Ferreto, Timoteo Lange, and Cesar AF De Rose. Performance evaluation of container-based virtualization for high performance computing environments. In *Parallel, Distributed and Network-Based Processing (PDP), 2013 21st Euromicro International Conference on*, pages 233–240. IEEE, 2013.

[8] V Chaudhary, Minsuk Cha, JP Walters, S Guercio, and Steve Gallo. A comparison of virtualization technologies for HPC. In *Advanced Information Networking and Applications, 2008. AINA 2008. 22nd International Conference on*, pages 861–868. IEEE, 2008.

[9] Roberto R Expósito, Guillermo L Taboada, Sabela Ramos, Juan Touriño, and Ramón Doallo. Performance analysis of HPC applications in the cloud. *Future Generation Computer Systems*, 29(1):218–229, 2013.

[10] Abhishek Gupta, Laxmikant V Kale, Filippo Gioachin, Verdi March, Chun Hui Suen, Bu-Sung Lee, Paolo Faraboschi, Richard Kaufmann, and Dejan Milojicic. The Who, What, Why and How of High Performance Computing Applications in the Cloud. Technical report, HP Labs, Tech. Rep., 2013.[Online]. Available: `http://www.hpl.hp.com/techreports/2013/HPL-2013-49.html`, 2013.

[11] Fabian Brosig, Fabian Gorsler, Nikolaus Huber, and Samuel Kounev. Evaluating Approaches for Performance Prediction in Virtualized Environments. In *Proceedings of the 2013 IEEE 21st International Symposium on Modelling, Analysis & Simulation of Computer and Telecommunication Systems*, pages 404–408. IEEE Computer Society, 2013.

[12] Jithin Jose, Mingzhe Li, Xiaoyi Lu, Krishna Chaitanya Kandalla, Mark Daniel Arnold, and Dhabaleswar K Panda. SR-IOV Support for Virtualization on InfiniBand Clusters: Early Experience. In *Cluster, Cloud and Grid Computing (CCGrid), 2013 13th IEEE/ACM International Symposium on*, pages 385–392. IEEE, 2013.

[13] Malek Musleh, Vijay Pai, John Paul Walters, Andrew Younge, and Stephen P Crago. Bridging the Virtualization Performance Gap for HPC Using SR-IOV for InfiniBand.

[14] Andrew S. Tanenbaum. *Modern operating systems*. Prentice-Hall, Upper Saddle River, NJ, 3. ed., internat. ed. [nachdr.] edition, 2010.

[15] Renaud Lottiaux, Benoit Boissinot, Pascal Gallard, Geoffroy Vallée, Christine Morin, et al. OpenMosix, OpenSSI and Kerrighed: a comparative study. 2004.

[16] Oren Laadan, Dan Phung, and Jason Nieh. Transparent checkpoint-restart of distributed applications on commodity clusters. In *Cluster Computing, 2005. IEEE International*, pages 1–13. IEEE, 2005.

[17] Parallel Environment Runtime Edition for AIX. `http://publibfp.dhe.ibm.com/epubs/pdf/c2367811.pdf`, April 2012.

[18] TOP500 Release. `http://top500.org/`, September 2014.

[19] Jason Duell. The design and implementation of Berkeley Labs Linux Checkpoint/Restart. Technical report, 2003.

[20] Does BLCR require a kernel patch? `https://upc-bugs.lbl.gov/blcr/doc/html/FAQ.html#patch`, December 2014.

[21] Jason Ansel, Kapil Arya, and Gene Cooperman. DMTCP: Transparent Checkpointing for Cluster Computations and the Desktop. In *23rd IEEE International Parallel and Distributed Processing Symposium*, Rome, Italy, May 2009.

[22] Oren Ladaan and Serge E. Hallyn. Linux-CR: Transparent Application Checkpoint-Restart in Linux. In *The Linux Symposium 2010, Ottawa, July 2010*, 2010.

[23] Kernel based checkpoint/restart. `http://lwn.net/Articles/298887/`, December 2014.

[24] Checkpoint/restart in the userspace. `http://www.linuxplumbersconf.org/2011/ocw/sessions/831`, December 2014.

[25] Checkpoint/Restore in Userspace. `http://criu.org/`, April 2012.

[26] Fedora. `https://fedoraproject.org/`, December 2014.

[27] Features/Checkpoint Restore. `https://fedoraproject.org/wiki/Features/Checkpoint_Restore`, December 2014.

[28] T. Hirofuchi, H. Nakada, S. Itoh, and S. Sekiguchi. Enabling Instantaneous Relocation of Virtual Machines with a Lightweight VMM Extension. In *Proceedings of the 2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*, pages 73–83. IEEE Computer Society, 2010.

[29] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard, Version 3.0*, 2012.

[30] Edgar Gabriel, Graham E. Fagg, George Bosilca, Thara Angskun, Jack J. Dongarra, Jeffrey M. Squyres, Vishal Sahay, Prabhanjan Kambadur, Brian Barrett, Andrew Lumsdaine, Ralph H. Castain, David J. Daniel, Richard L. Graham, and Timothy S. Woodall. Open MPI: Goals, Concept, and Design of a Next Generation MPI Implementation. In *Proceedings, 11th European PVM/MPI Users' Group Meeting*, pages 97–104, Budapest, Hungary, September 2004.

[31] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A high-performance, portable implementation of the MPI message passing interface standard, September 1996.

[32] William D. Gropp and Ewing Lusk. *User's Guide for* `mpich`*, a Portable Implementation of MPI.* Mathematics and Computer Science Division, Argonne National Laboratory, 1996. ANL-96/6.

[33] Bianca Schroeder and Garth A Gibson. Understanding failures in petascale computers. *Journal of Physics: Conference Series*, 78(1):012022, 2007.

[34] Bianca Schroeder and Garth A. Gibson. A large-scale study of failures in high-performance computing systems. In *Proceedings of the International Conference on Dependable Systems and Networks*, pages 249–258, Washington, DC, USA, 2006. IEEE Computer Society.

[35] Graham E Fagg and Jack J Dongarra. FT-MPI: Fault tolerant MPI, supporting dynamic applications in a dynamic world. In *Recent advances in parallel virtual machine and message passing interface*, pages 346–353. Springer, 2000.

[36] Sriram Sankaran, Jeffrey M Squyres, Brian Barrett, Vishal Sahay, Andrew Lumsdaine, Jason Duell, Paul Hargrove, and Eric Roman. The LAM/MPI checkpoint/restart framework: System-initiated checkpointing. *International Journal of High Performance Computing Applications*, 19(4):479–493, 2005.

[37] Joshua Hursey, Jeffrey M. Squyres, Timothy I. Mattox, and Andrew Lumsdaine. The Design and Implementation of Checkpoint/Restart Process Fault Tolerance for Open MPI. In *Proceedings of the 21st IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE Computer Society, 03 2007.

[38] B. Bouteiller, P. Lemarinier, K. Krawezik, and F. Capello. Coordinated checkpoint versus message log for fault tolerant MPI. In *Cluster Computing, 2003. Proceedings. 2003 IEEE International Conference on*, pages 242–250, 2003.

[39] Justin CY Ho, Cho-Li Wang, and Francis CM Lau. Scalable group-based checkpoint/restart for large-scale message-passing systems. In *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, pages 1–12. IEEE, 2008.

[40] Kurt Ferreira, Rolf Riesen, Ron Oldfield, Jon Stearley, James Laros, Kevin Pedretti, Ron Brightwell, and Todd Kordenbrock. Increasing fault resiliency in a message-passing environment. Technical report SAND2009-6753, Sandia National Laboratories, October 2009.

[41] Chao Wang, Frank Mueller, Christian Engelmann, and Stephen L Scott. Proactive process-level live migration in HPC environments. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, page 43. IEEE Press, 2008.

[42] tmpfs. http://www.kernel.org/doc/Documentation/filesystems/tmpfs.txt, December 2014.

[43] Adrian Reber and Peter Väterlein. PoS (ISGC 2012) 031 Live process migration for load balancing and/or fault tolerance. In *The International Symposium on Grids and Clouds (ISGC)*, volume 2012, 2012.

[44] SOFT-DIRTY PTEs. https://www.kernel.org/doc/Documentation/vm/soft-dirty.txt, December 2014.

[45] /proc/sys/kernel/*. https://www.kernel.org/doc/Documentation/sysctl/kernel.txt, December 2014.

[46] SSH. http://en.wikipedia.org/w/index.php?title=Secure_Shell&oldid=545050666, December 2014.

[47] Joshua Hursey, Jeffrey M. Squyres, and Andrew Lumsdaine. A Checkpoint and Restart Service Specification for Open MPI. Technical Report TR635, Indiana University, Bloomington, Indiana, USA, July 2006.

[48] Open MPI CRS commits. https://github.com/open-mpi/ompi/commit/e12ca48cd9a34b1f41b11f267bddf91f05dae5be, December 2014.

[49] Processor E5-2650. http://ark.intel.com/products/64590/, December 2014.

[50] John D. McCalpin. Memory Bandwidth and Machine Balance in Current High Performance Computers. *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter*, pages 19–25, December 1995.

[51] Albert Ruprecht. Finite Elemente zur Berechnung dreidimensionaler, turbulenter Strömungen in komplexen Geometrien. Hochschulschrift, Stuttgart, 1989.

[52] Institut fr Strömungsmechanik und Hydraulische Strömungsmaschinen. http://www.ihs.uni-stuttgart.de/, December 2014.

[53] FENFLOSS. `http://www.ihs.uni-stuttgart.de/116.html`, December 2014.

[54] Eugen Volk, Jochen Buchholz, Stefan Wesner, Daniela Koudela, Matthias Schmidt, Niels Fallenbeck, Roland Schwarzkopf, Bernd Freisleben, Götz Isenmann, Jürgen Schwitalla, et al. Towards Intelligent Management of Very Large Computing Systems. *Competence in High Performance Computing 2010*, pages 191–204, 2012.

[55] Nathan DeBardeleben, James Laros, John T. Daly, Stephen L. Scott, Christian Engelmann, and Bill Harrod. High-End Computing Resilience: Analysis of Issues Facing the HEC Community and Path-Forward for Research and Development. Whitepaper, December 2009.

[56] William M. Jones, John T. Daly, and Nathan DeBardeleben. Impact of sub-optimal checkpoint intervals on application efficiency in computational clusters. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, HPDC '10, pages 276–279, New York, NY, USA, 2010. ACM.

[57] Franck Cappello, Henri Casanova, and Yves Robert. Checkpointing vs. Migration for Post-Petascale Supercomputers. In *Proceedings of the 2010 39th International Conference on Parallel Processing*, ICPP '10, pages 168–177, Washington, DC, USA, 2010. IEEE Computer Society.